# Public-key Compression in M-SIDH

Kaizhan Lin[1], Jianming Lin[1], Shiping Cai[1], Weize Wang[1], and Chang-An Zhao✉[1,2]

[1] School of Mathematics, Sun Yat-sen University,
Guangzhou 510275, P. R. China
{linkzh5,linjm28,caishp6,wangwz}@mail2.sysu.edu.cn
zhaochan3@mail.sysu.edu.cn
[2] Guangdong Key Laboratory of Information Security,
Guangzhou 510006, P. R. China

**Abstract.** Recently, SIKE was broken by the Castryck-Decru attack in polynomial time. To avoid this attack, Fouotsa proposed a SIDH-like scheme called M-SIDH, which hides the information of auxiliary points. The countermeasure also leads to huge parameter sizes, and correspondingly the public key size is relatively large.
In this paper, we present several new techniques to compress the public key of M-SIDH. Our method to compress the key is reminiscent of public-key compression in SIDH/SIKE, including torsion basis generation, pairing computation and discrete logarithm computation. We also prove that compressed M-SIDH is secure if M-SIDH is secure.
Experimental results showed that our approach fits well with compressed M-SIDH. It should be noted that most techniques proposed in this paper could be also utilized into other SIDH-like protocols.

**Keywords:** M-SIDH · Post-quantum Cryptography · Public-key Compression · SIDH

## 1 Introduction

Since supersingular isogeny Diffie-Hellman (SIDH) [20] was proposed by Jao et al., isogeny-based cryptosystems are attractive in post-quantum cryptography. As the NIST [1] round 4 finalist, supersingular isogeny key encapsulation (SIKE) [4] is famous for its small public key size.

To make SIDH/SIKE more attractive, a large variety of works targets public-key compression in SIDH/SIKE to reduce the public key size. Public-key compression in SIDH was first proposed by Azarderakhsh et al. [5]. The key was further compressed by Costello et al. [12]. There are three main procedures in public-key compression in SIDH: torsion basis generation, pairing computation and discrete logarithm computation. Zanon et al. [36] utilized several techniques to accelerate the implementation significantly. Later, Naehrig et al. [26] adapted the dual isogeny to speed up the performance of pairing computation, while Pereira et al. [28] extended the work of [36] and gave a fast method to generate binary torsion basis. However, most of the techniques require large storage for precomputation. An efficient method to compute discrete logarithms with smaller lookup tables was proposed in [18]. Lin et al. [22] improved the Miller evaluation, making the implementation faster with less storage. Several works [23,27] also managed to compress the key using other approaches.

Recently, Castryck and Decru [8] proposed an efficient attack to break SIDH and SIKE in polynomial time if the endomorphism ring of the starting curve is known. Maino et al. [24] gave a subexponential algorithm to attack SIDH with arbitrary starting curve. Inspired by these two works, Robert [31] presented a deterministic polynomial time attack on SIDH in all cases. The attacks also apply to Séta [13] and B-SIDH [11].

However, not all is lost. All the mentioned attacks entirely rely on the following information:

- the degree of the secret isogeny;
- the torsion point images.

Therefore, one could hide either of them to avoid the attack. Moriya managed to hide the degree of the secret isogenies and proposed a new SIDH-like scheme, while Fouosta proposed another scheme, called M-SIDH (Masked torsion points SIDH), to avoid this attack by masking auxiliary points [25,14,15]. However, to satisfy the desired security, both of SIDH-like schemes require relatively large parameter sizes, resulting in larger public key size compared with that of SIDH. To reduce the key size, a natural question is how to compress the public key in SIDH-like protocols.

In this paper, we give an approach to overcome this problem and propose several new techniques to compress the public key of M-SIDH, whose size is $6\log_2 p$ bits. We summarize our work as follows:

- We prove that our approach to compress the public key with size $\approx 3.5\log_2 p$ bits does not affect the security of M-SIDH. Therefore, one does not need to mask any auxiliary point by executing scalar multiplication in compressed M-SIDH.
- To compress the public key of M-SIDH, we propose a new approach which is reminiscent of public-key compression in SIDH/SIKE. Firstly, we proposed a novel way to generate torsion basis. In particular, to determine whether two points could form a torsion basis we utilized compressed pairings and Lucas sequences. Secondly, we provide a new method for discrete logarithm computation, which is to compute only three discrete logarithms efficiently, but it may fail since the order of the cyclic group involved in discrete logarithms has distinct prime factors, which is not the case in compressed SIDH/SIKE. Finally, we provide another effective method to provide more stable performance in discrete logarithm computation, although it is not as efficient as the former one.
- We give the first instantiation of compressed M-SIDH in SageMath. Experimental results verify the validity of our algorithms.

The rest of this paper is as follows. In Section 2 we recall the reduced Tate pairing, compressed pairings, Lucas sequences, M-SIDH and public-key compression in SIDH/SIKE. Section 3 sketches our approach to compress the public key of M-SIDH. In Section 4 we present several techniques to compress the public key of M-SIDH. Section 5 reports our implementation and we conclude in Section 6.

## 2 Preliminaries

In this section, we first introduce the reduced Tate pairings, compressed pairings and Lucas sequences. Next, we recall M-SIDH. Finally, we review several techniques used in public-key compression in SIDH/SIKE.

### 2.1 Reduced Tate pairings

Let $E$ be an elliptic curve over the finite field $\mathbb{F}_q$, where $q$ is a power of a prime $p$. Denote $\mu_n$ to be the cyclic group of order $n$ in $\mathbb{F}_q^*$, and $f_{n,R}$ to be a rational function on $E$ satisfying $\mathrm{div}(f_{n,R}) = n(R) - n(\mathcal{O})$, where $R$ is a point of order $n$. The reduced Tate pairing [16] is defined as:

$$\begin{aligned} e_n : E[n] \times E(\mathbb{F}_q)/nE(\mathbb{F}_q) &\to \mu_n, \\ (R, S) &\mapsto f_{n,R}(S)^{\frac{q-1}{n}}. \end{aligned}$$

Similar with the Tate pairing [34], the reduced Tate pairing has the following properties:

- Bilinearity: $\forall R, R_1, R_2 \in E[n]$, $\forall S, S_1, S_2 \in E(\mathbb{F}_q)/nE(\mathbb{F}_q)$,

$$\begin{aligned} e_n(R, S_1 + S_2) &= e_n(R, S_1) \cdot e_n(R, S_2), \\ e_n(R_1 + R_2, S) &= e_n(R_1, S) \cdot e_n(R_2, S); \end{aligned}$$

- Non-degeneracy: If $e_n(R, S) = 1$ for all $S \in E(\mathbb{F}_q)/nE(\mathbb{F}_q)$ then $R = \mathcal{O}$, and if $e_n(R, S) = 1$ for all $R \in E[n]$ then $S \in nE(\mathbb{F}_q)$.
- Compatibility with isogenies: Assume $\phi : E \to E'$ is a non-zero isogeny of degree $m$ defined over $\mathbb{F}_q$. For $R \in E[n]$, $S \in E(\mathbb{F}_q)/nE(\mathbb{F}_q)$, $R' \in E'[n]$,

$$\begin{aligned} e_n(\phi(R), \phi(S)) &= e_n(R, S)^m, \\ e_n(R', \phi(S)) &= e_n(\hat{\phi}(R'), S). \end{aligned}$$

### 2.2 Compressed pairings and Lucas sequences

Compressed pairings were first introduced by Scott et al. [32]. This kind of pairings reduces to the bandwidth of pairing values by taking the trace map. Assume that the elliptic curve is supersingular and it is defined over $\mathbb{F}_{p^2}$[3]. In this case, computing the trace of the pairing value is more efficient than computing the pairing value itself.

[3] Indeed, the techniques proposed in this section also works when the elliptic curve is defined over $\mathbb{F}_{q^2}$, where $q$ is a prime power.

The final exponentiation of pairings consists of a raising to the power of $p-1$ and a raising to the power of $(p+1)/n$. The former one is an easy part, but the latter requires relatively large computational resources. Thanks to Lucas sequences [30, Section 3.6.3], one could efficiently obtain $tr_{\mathbb{F}_{p^2}/\mathbb{F}_p}(\gamma^z)$ from $tr_{\mathbb{F}_{p^2}/\mathbb{F}_p}(\gamma)$ for any $\gamma \in \mu_{p+1}$ and $z = (z_0 z_1 \cdots z_t)_2 \in \mathbb{N}$, as shown in Algorithm 1. Therefore, this technique can improve the costly part of the final exponentiation.

**Algorithm 1** LS: Lucas sequences

**Require:** $tr_{\mathbb{F}_{p^2}/\mathbb{F}_p}(\gamma)$ with $\gamma \in \mu_{p+1}$, $z = (z_0 z_1 \cdots z_t)_2 \in \mathbb{N}$;
**Ensure:** $tr_{\mathbb{F}_{p^2}/\mathbb{F}_p}(\gamma^z)$.
$v_0 \leftarrow 2$, $v_1 \leftarrow tr_{\mathbb{F}_{p^2}/\mathbb{F}_p}(\gamma)$, $tmp \leftarrow v_1$;
**for** each $j \in \{0, 1, \cdots, t\}$ **do**
  **if** $z_j = 1$ **then**
    $v_0 \leftarrow v_0 \cdot v_1$, $v_0 \leftarrow v_0 - tmp$, $v_1 \leftarrow v_1^2$, $v_1 \leftarrow v_1 - 2$;
  **else**
    $v_0 \leftarrow v_0^2$, $v_0 \leftarrow v_0 - 2$, $v_1 \leftarrow v_0 \cdot v_1$, $v_1 \leftarrow v_1 - tmp$;
  **end if**
**end for**
**return** $v_0$.

Lucas sequences have potential to improve the exponentiation in the group $\mu_{p+1}$ as well. According to the observation in [32], for any element $\gamma = \gamma_1 + \gamma_2 \cdot i \in \mu_{p+1}$ and $z \in \mathbb{N}$,

$$(\gamma_1 + \gamma_2 \cdot i)^z = \frac{\mathrm{LS}(\gamma, z)}{2} + \frac{\gamma_1 \cdot \mathrm{LS}(\gamma, z) - \mathrm{LS}(\gamma, z-1)}{2\gamma_1^2 - 2} \cdot \gamma_2 \cdot i.$$

Note that when computing LS($\gamma$,$z-1$), the explicit value of LS($\gamma$,$z$) is also obtained. When the inverse operation is not costly (for instance one can adapt the the binary GCD algorithm) and $z$ is large, utilizing Lucas sequences will improve the performance significantly. The main idea is summarized in Algorithm 2.

**Algorithm 2** ELS: Exponentiation using Lucas sequences

**Require:** $\gamma = \gamma_1 + \gamma_2 \cdot i \in \mu_{p+1}$, $z \in \mathbb{N}$;
**Ensure:** $\gamma^z$.
$tmp_1 \leftarrow \mathrm{LS}(\gamma, z)$, $tmp_1 \leftarrow \mathrm{LS}(\gamma, z-1)$;
  //when computing LS($\gamma$,$z-1$), LS($\gamma$,$z$) is also obtained
$tmp_1 \leftarrow tmp_1/2$, $tmp_2 \leftarrow tmp_2/2$;
$tmp_2 \leftarrow \gamma_1 \cdot tmp_1 - tmp_2$, $tmp_2 \leftarrow tmp_2/(\gamma_1^2 - 1)$, $tmp_2 \leftarrow tmp_2 \cdot \gamma_2$;
**return** $tmp_1 + tmp_2 \cdot i$.

### 2.3 M-SIDH

Let $p = 4 \cdot f \cdot \ell_1 \cdot \ell_2 \cdots \ell_t - 1$, where the primes $\ell_1, \ell_2, \cdots, \ell_t$ are the first $t$ odd primes and $f$ is a small cofactor such that $p$ is a prime. Denote $\ell_0 = 2$, $N_A = \ell_0 \cdot \ell_2 \cdots \ell_{t-1}$ and $N_B = \ell_1 \cdot \ell_3 \cdots \ell_t$. Define $E_0$ to be a supersingular curve over $\mathbb{F}_{p^2}$ together with $E_0[N_A] = \langle P_A, Q_A \rangle$ and $E_0[N_B] = \langle P_B, Q_B \rangle$. Similar to the SIDH protocol, M-SIDH proceeds as follows:

- Key Generation: Alice chooses a random integer $s_A \in \mathbb{Z}_{N_A}$ as her secret key. She computes the point $P_A + [s_A]Q_A$ and constructs the $N_A$-isogeny $\phi_A$ with kernel $\langle P_A + [s_A]Q_A \rangle$. Then she evaluates two torsion point images $\phi_A(P_B)$, $\phi_A(Q_B)$ and the image curve $E_A$. Finally, she transmits the tuple $(E_A, [a]\phi_A(P_B), [a]\phi_A(Q_B))$ to Bob, where $a \in \mathbb{Z}/N_B\mathbb{Z}^*$. Similar to Alice, Bob selects a random integer $s_B \in \mathbb{Z}_{N_B}$ to compute $P_B + [s_B]Q_B$ as the kernel generator of the $N_B$-isogeny $\phi_B$. His public key is $(E_B, [b]\phi_B(P_A), [b]\phi_B(Q_A))$ with $b \in \mathbb{Z}/N_A\mathbb{Z}^*$.
- Key Agreement: Alice begins her key agreement phase after receiving Bob's public key. She first checks the existence of $U$ s.t. $e_{N_A}(\phi_B(P_A), \phi_B(Q_A)) = e_{N_A}(P_A, Q_A)^U$ and $U/N_B (\bmod N_A)$ is a square, if not she aborts. Then she computes $\phi_B(P_A) + [s_A]\phi_B(Q_A)$ to construct the $N_A$-isogeny $\phi'_A$ and regards the $j$-invariant of the image curve $j_{E_{BA}}$ as her shared key. Analogously, Bob checks the existence of $V$ such that $e_{N_B}(\phi_A(P_B), \phi_A(Q_B)) = e_{N_B}(P_B, Q_B)^V$ and $V/N_A (\bmod N_B)$ is a square, if not he aborts. He computes the image curve $E_{AB}$ of the $N_B$-isogeny $\phi'_B$ and the shared key $j(E_{AB})$.

### 2.4 Public-key compression in SIDH/SIKE

In this subsection, we briefly review the main techniques utilized in public-key compression in SIDH/SIKE. For simplicity, we only consider how to compress the key $(E_B, \phi_B(P_A), \phi_B(Q_A))$.

The main idea of public-key compression is to implement a deterministic pseudorandom number generator to generate a basis of the $N_A$-torsion group, and use this basis to linearly represent $\phi_B(P_A)$ and $\phi_B(Q_A)$, i.e.,

$$\begin{bmatrix} \phi_B(P_A) \\ \phi_B(Q_A) \end{bmatrix} = \begin{bmatrix} a_0 & b_0 \\ a_1 & b_1 \end{bmatrix} \begin{bmatrix} U_A \\ V_A \end{bmatrix}. \tag{1}$$

After revealing $a_0, a_1, b_0$ and $b_1$, Bob checks whether $a_0$ is invertible in $\mathbb{Z}^*_{N_A}$. If so, Bob sends $(E_B, 0, a_0^{-1}b_0, a_0^{-1}a_1, a_0^{-1}b_1)$ to Alice. Otherwise, the element $b_0$ must be invertible in $\mathbb{Z}^*_{N_A}$ and Bob transmits $(E_B, 1, b_0^{-1}a_0, b_0^{-1}a_1, b_0^{-1}b_1)$ instead.

Assume that $a_0 \in \mathbb{Z}^*_{N_A}$, while the other case is similar. After receiving Bob's public key, Alice could compute the kernel of the isogeny $\phi'_A$ [12]:

$$\begin{aligned} \langle \phi_B(P_A) + [s_A]\phi_B(Q_A) \rangle &= \langle [a_0]U_A + [b_0]V_A + [s_A a_1]U_A + [s_A b_0]V_A \rangle \\ &= \langle U_A + [a_0^{-1}b_0]V_A + [s_A a_0^{-1}a_1]U_A + [s_A a_0^{-1}b_0]V_A \rangle \\ &= \langle [1 + s_A(a_0^{-1}a_1)]U_A + [(a_0^{-1}b_0) + s_A(a_0^{-1}b_0)]V_A \rangle. \end{aligned}$$

Therefore, Alice could complete the key agreement phase, although she does not recover $\phi_A(P_B)$ and $\phi_A(Q_B)$.

It remains how to obtain $a_0^{-1}b_0$, $a_0^{-1}a_1$ and $a_0^{-1}b_1$. Zanon et al. [36] proposed a new technique to speed up the performance. Since $\phi_B(P_A)$ and $\phi_B(Q_A)$ also form a basis of $E_B[N_A]$, they can also linearly represent $U_A$ and $V_A$, i.e.,

$$\begin{bmatrix} U_A \\ V_A \end{bmatrix} = \begin{bmatrix} c_0 & d_0 \\ c_1 & d_1 \end{bmatrix} \begin{bmatrix} \phi_B\left(P_A\right) \\ \phi_B\left(Q_A\right) \end{bmatrix}. \tag{2}$$

It is easy to verify that $(a_0^{-1}b_0, a_0^{-1}a_1, a_0^{-1}b_1) = (-d_1^{-1}d_0, -d_1^{-1}c_1, d_1^{-1}c_0)$. With the help of bilinear pairings,

$$\begin{aligned}
h_0 &= \mathrm{e}_{N_A}\left(\phi_B\left(P_A\right), \phi_B\left(Q_A\right)\right) = \mathrm{e}_{N_A}\left(P_A, Q_A\right)^{N_B}, \\
h_1 &= \mathrm{e}_{N_A}\left(\phi_B\left(P_A\right), U_A\right) = \mathrm{e}_{N_A}\left(\phi_B\left(P_A\right), c_0\phi_B\left(P_A\right) + d_0\phi_B\left(Q_A\right)\right) = h_0^{d_0}, \\
h_2 &= \mathrm{e}_{N_A}\left(\phi_B\left(P_A\right), V_A\right) = \mathrm{e}_{N_A}\left(\phi_B\left(P_A\right), c_1\phi_B\left(P_A\right) + d_1\phi_B\left(Q_A\right)\right) = h_0^{d_1}, \\
h_3 &= \mathrm{e}_{N_A}\left(\phi_B\left(Q_A\right), U_A\right) = \mathrm{e}_{N_A}\left(\phi_B\left(Q_A\right), c_0\phi_B\left(P_A\right) + d_0\phi_B\left(Q_A\right)\right) = h_0^{-c_0}, \\
h_4 &= \mathrm{e}_{N_A}\left(\phi_B\left(Q_A\right), V_A\right) = \mathrm{e}_{N_A}\left(\phi_B\left(Q_A\right), c_1\phi_B\left(P_A\right) + d_1\phi_B\left(Q_A\right)\right) = h_0^{-c_1}.
\end{aligned} \tag{3}$$

Note that $h_0$ only depends on public parameters. Therefore, one can recover $c_0$, $c_1$, $d_0$, $d_1$ by computing 4 discrete logarithms of $h_1$, $h_2$, $h_3$, $h_4$ to the base $h_0$ efficiently with precomputed lookup tables [36,18,26,22]. Another approach is to compute only 3 discrete logarithms of $h_1$, $h_3$, $h_4$ (resp. $h_2$, $h_3$, $h_4$) to the base $h_2$ (resp. $h_1$) without precomputation for $h_2$ (resp. $h_1$) could not be computed in advance [23].

## 3 Public-key Compression in M-SIDH

In this section, we sketch our approach to compress the public key of M-SIDH and give Proposition 2 to show that compressed M-SIDH is secure if M-SIDH is secure.

### 3.1 Our approach to compress the key

Our approach to compress the public key of M-SIDH is reminiscent of public-key compression in SIDH/SIKE. Given a secret $N_B$-isogeny from $E_0$ to $E_B$, the main procedures are as follows:

1. Torsion basis generation: Generate $\{U_A, V_A\}$ such that $\langle U_A, V_A\rangle = E_B[N_A]$;
2. Pairing computation: Compute the following four pairings:

$$\begin{aligned}
h_1 = e_{N_A}\left(\phi_B(P_A), U_A\right),\ h_2 = e_{N_A}\left(\phi_B(P_A), V_A\right), \\
h_3 = e_{N_A}\left(\phi_B(Q_A), U_A\right),\ h_4 = e_{N_A}\left(\phi_B(Q_A), V_A\right);
\end{aligned} \tag{4}$$

3. Discrete logarithm computation: Check whether each $h_i$, $i = 1, 2, 3, 4$ could be a generator of the multiplicative group $\mu_{N_A}$. If there exists an element of order $N_A$, set it as the base and compute discrete logarithms of other three elements to the base. Otherwise, select another secret key $sk_A$ and repeat the key generation phase and public-key compression step.

The public key is $(B, s_1, s_2, s_3, label)$, where $B$ is the image curve coefficient of the isogeny $\phi_B$, $s_1$, $s_2$ and $s_3$ are the solutions of three discrete logarithms in Procedure 3, and *label* is used to mark which one is the base of discrete logarithms. Similar to public-key compression in SIDH, it is easy to see that our method to compress the key is valid.

**Proposition 1.** *One could compress the public key successfully by performing the above procedures.*

*Remark 1.* In the compressed SIDH protocol, it is impossible that none of $h_i$ is a generator. However, it happens in compressed M-SIDH with small possibility. For example, in Equation (2), $\ell_0$ may divide $c_0$ and $d_1$, while $\ell_2$ may divide $d_0$ and $c_1$. To minimize the public key size, one can select another secret key. Another alternative approach to overcome this issue is proposed in Section 4.3.

*Remark 2.* As mentioned in Section 2.4, one could utilize dual isogenies to optimize pairing computation [26,22] in compressed SIDH. However, the dual isogeny construction in compressed M-SIDH is much more costly compared to that of compressed SIDH. According to our experiments, directly computing $h_1, h_2, h_3$ and $h_4$ in Equation (4) without the dual isogeny technique is more efficient. This is the reason why we do not utilize the dual isogeny technique.

Now we prove that compressed M-SIDH is secure whenever M-SIDH is secure.

**Proposition 2.** *Compressed M-SIDH is secure if M-SIDH is secure.*

*Proof.* Without loss of generality, we only consider Bob's case.

If Bob transmits $(E_B, \phi_B(P_A), \phi_B(Q_A))$ to Alice without compression, then the adversary Eve could apply the Castryck-Decru attack to recover the secret key. Therefore, M-SIDH requires masking auxiliary points by executing two scalar multiplications. In compressed M-SIDH, Bob compresses the key and transmits $(E_B, a_0^{-1}b_0, a_0^{-1}a_1, a_0^{-1}b_1, label)$ to Alice (assume that $a_0 \in \mathbb{Z}_{N_A}^*$). In the following, we will prove that it is hard for Eve to recover $\phi_B$ as long as M-SIDH is secure.

After generating $U_A$ and $V_A$, Eve can compute

$$[a_0^{-1}]\phi_B(P_A) = U_A + [a_0^{-1}b_0]Q_A,$$
$$[a_0^{-1}]\phi_B(Q_A) = [a_0^{-1}a_1]U_A + [a_0^{-1}b_1]Q_A.$$

Hence,

$$e_{N_A}([a_0^{-1}]\phi_B(P_A), [a_0^{-1}]\phi_B(Q_A)) = e_{N_A}(P_A, Q_A)^{N_B a_0^{-2}}.$$

To recover $a_0^{-2} \bmod N_A$, Eve computes a discrete logarithm of $e_{N_A}(P_A, Q_A)^{N_B a_0^{-2}}$ to the base $e_{N_A}(P_A, Q_A)$. Select $a_0'$ such that $(a')^2 \equiv (a_0)^2 \bmod N_A$. Set $\alpha = a' a_0^{-1} \in \mu_2(N_A)$, where

$$\mu_2(N) = \{x \in \mathbb{Z}/N\mathbb{Z} | x^2 \equiv 1 \bmod N\}.$$

Therefore, Eve has the information that

$$[\alpha]\phi_B(P_A) = [a'][a_0^{-1}]\phi_B(P_A),$$
$$[\alpha]\phi_B(Q_A) = [a'][a_0^{-1}]\phi_B(Q_A).$$

However, as Fouotsa et al. claimed [15], in the M-SIDH setting, it is hard to recover $\alpha$, which is equivalent to recover $a_0^{-1}$. Therefore, if M-SIDH is secure, compressed M-SIDH is also secure. This completes the proof. ■

# 4 Optimizations on Compressed M-SIDH

Proposition 2 induces that compressed M-SIDH saves two large scalar multiplications of length $\approx \sqrt{p}$ as M-SIDH does. However, it should be noted that the performance of compressed M-SIDH is still not as efficient as that of M-SIDH because of torsion basis generation, pairing computation and discrete logarithm computation. In this section we will optimize the performance of key compression to close the gap. As before, we only handle Bob's case and Alice could adapt all the techniques to accelerate the performance.

## 4.1 Torsion basis generation

Since $N_A$ and $N_B$ are not the power of 2 and 3, the torsion basis generation of compressed M-SIDH could not benefit from several techniques such as shared Elligator [36] and 3-descent of elliptic curves [12]. In this subsection we propose a new method to generate $\{U_A, V_A\}$ such that $\langle U_A, V_A \rangle = E_B[N_A]$, while the torsion basis generation of the $N_B$-torsion group of $E_A$ is similar. For simplicity, we abbreviate $U_A$ and $V_A$ to $U$ and $V$, respectively.

Generating one of the torsion points is relatively easy: we can choose a point of order $N_A$ and then set it as $U$. After $U$ is successfully generated, we generate another point $V$ such that $\langle U, V \rangle = E_B[N_A]$.

As for the first torsion point, a naive way is to sample a random point $R \in E_B(\mathbb{F}_{p^2})$, and then check whether the order of $[2fN_B]R$ is $N_A$. Here we propose Algorithm 3 to generate $U$, which is more efficient than the naive approach. We also output $\{U_j | j \in I\}$, which are useful for the generation of the second torsion point $V$.

The main idea of Algorithm 3 is as follows:

- Firstly, we randomly generate a point $R$ using Elligator [7] and set $U = [2fN_B]R$.

**Algorithm 3** GenerationU: generate a point of order $N_A$

**Require:** $E_B/\mathbb{F}_{p^2}$: A supersingular curve, $I : \{j|\ell_j \text{ divides } N_A\}$;
**Ensure:** A point $U \in E_B(\mathbb{F}_{p^2})$ of order $N_A$, $\{U_j|j \in I\}$.

Generate a point $R \in E_B(\mathbb{F}_{p^2})$ using Elligator;
$U \leftarrow [2fN_B]R$;
$\{U_j\} \leftarrow$BCM$(U, I)$; // Algorithm 4
$I_U \leftarrow \{j|U_j = \mathcal{O}\}$;
**while** $I_U \neq \emptyset$ **do**
    Generate a point $R \in E_B(\mathbb{F}_{p^2})$ using Elligator;
    $U' \leftarrow [2fN_B]R$;
    $U' \leftarrow \prod_{j \in I \setminus I_U} [\ell_j]U'$;
    $\{U'_j\} \leftarrow$BCM$(U', I_U)$; // Algorithm 4
    **for** each $j \in \{k|U'_k \neq \mathcal{O}\}$ **do**
        $U \leftarrow U + U'_j$, $U_j \leftarrow U'_j$;
    **end for**
    $I_U \leftarrow \{j|U'_j = \mathcal{O}\}$;
**end while**
**return** $U$, $\{U_j|j \in I\}$.

– Next, we compute $U_j = [N_A/\ell_j]U$, where $I = \{j|\ell_j \text{ divides } N_A\}$. It is easy to see that $U_j$ is a point of order $\ell_j$ if $\ell_j$ divides the order of $U$. Otherwise, $U_j$ is at infinity.
– Denote $I_U = \{j|U_j = \mathcal{O}\}$. If $I_U$ is not empty, we randomly sample another point $R$ and compute $U' = [2fN_B]R$. According to $I_U$, we compute $U'_j = [N_A/\ell_j]U'$ where $j \in I_U$. If $U'_j$ is not at infinity, set $U = U + U'_j$. Finally, let $I_U = \{j|U'_j = \mathcal{O}\}$. We repeat the above progress to generate $U'$ until $I_U$ is empty.

As a result, for each $j \in I$, $U_j = [N_A/\ell_j]U \neq \mathcal{O}$. Therefore, $U$ is a point of order $N_A$.

*Remark 3.* The approach to compute $U_j$ is inspired by the public-key validation of CSIDH [9]. The authors check the key by generating a point and then check the order of the point using a divide-and-conquer approach [33]. Although this approach consumes slightly larger memory, it performs more efficient than directly computing each $U_j$.

In the following we focus on how to generate another point $V$ such that $\langle U, V\rangle = E_B[N_A]$. A naive approach is to generate $V$ with respect to the above method, and then check if $U$ and $V$ can generate the $N_A$-torsion group. However, this method is not so practical because the success probability is relatively small. Here we present a more efficient method to generate $V$ thanks to Proposition 3.

**Proposition 3.** *Assume that $U$ is a point of order $N_A = \ell_0\ell_2 \cdots \ell_{t-1}$ on $E_B$, and $V$ a random point on $E_B(\mathbb{F}_{p^2})/N_A E_B(\mathbb{F}_{p^2})$. Let $I = \{j|\ell_j \text{ divides } N_A\}$,*

**Algorithm 4** BCM: Batch cofactor multiplication

**Require:** A point $U$, $I_U$: a subset of $I = \{j | \ell_j \text{ divides } N_A\}$;
**Ensure:** Points $\{U_1, U_2, \cdots, U_{n'}\}$, where $U_k = \prod_{j \in I_U \setminus \{k\}} [\ell_j]U$ and $n' = \# I_U$.

**if** $n' = 1$ **then**
  **return** $\{U\}$;
**end if**
$m' \leftarrow \lfloor n'/2 \rfloor$;
$L_1 \leftarrow \prod_{i=0}^{m'-1} \ell_{I_U[i]}$, $L_2 \leftarrow \prod_{i=m'}^{n'-1} \ell_{I_U[i]}$;
$left \leftarrow [L_1]U$;
$right \leftarrow [L_2]U$;
Divide $I_U$ into two subsets $I_1$, $I_2$ such that $\# I_1 = n' - m'$ and $\# I_2 = m'$;
$r_1 \leftarrow$BCM(left,$I_1$);
$r_2 \leftarrow$BCM(right,$I_2$);
**return** $r_1 \cup r_2$.

$U_k = \prod_{j \in I \setminus \{k\}} [\ell_j]U$. *Denote by* $ord(\gamma)$ *the order of* $\gamma$ *in* $\mu_{N_A}$. *Then*

$$ord(e_{N_A}(U, V)) = \prod_{\substack{j \in I \\ e_{\ell_j}(U_j, V) \neq 1}} \ell_j. \tag{5}$$

*In particular,* $e_{N_A}(U, V)$ *is a generator of* $\mu_{N_A}$ *iff* $\langle U, V \rangle = E_B[N_A]$.

*Proof.* Let $s_k = \prod_{j \in I \setminus \{k\}} \ell_j$ and $s'_k = s_k^{-1} \bmod \ell_k$. From $U_k = \prod_{j \in I \setminus \{k\}} [\ell_j]U$ we have $U = \sum_{k \in I} [s'_k] U_k$. Utilizing the bilinearity of the reduced Tate pairing,

$$\begin{aligned} & e_{N_A}(U, V) \\ = & e_{N_A}([s'_0]U_0, V) \cdot e_{N_A}([s'_2]U_2, V) \cdots e_{N_A}([s'_{t-1}]U_{t-1}, V) \\ = & e_{N_A}(U_0, V)^{s'_0} \cdot e_{N_A}(U_2, V)^{s'_2} \cdots e_{N_A}(U_{t-1}, V)^{s'_{t-1}}. \end{aligned} \tag{6}$$

With the help of the linearity of the reduced Tate pairings,

$$e_{N_A}(U_k, V) = e_{\ell_k}(U_k, V).$$

Let $V_k = \prod_{j \in I \setminus \{k\}} [\ell_j]V$. Obviously, $e_{\ell_k}(U_k, V) = 1$ *iff* $e_{\ell_k}(U_k, V_k) = 1$.

In the following, we will prove that $V_k$ and $U_k$ are linearly dependent *iff* $e_{\ell_k}(U_k, V_k) = 1$, i.e., $e_{N_A}(U_k, V) = 1$.

We first assume that $V_k$ and $U_k$ are linearly dependent. Then we have

- $V_k = \mathcal{O}$, or
- $V_k \neq \mathcal{O}$, but $V_k \in \langle U_k \rangle$,

and *vice versa*. Since that the curve $E : y^2 = x^3 + x$ is supersingular, there exists an isogeny from $E$ to $E_A$. Applying the KLPT algorithm [21], one could find an isogeny $\phi : E \to E_B$ of degree $\ell_1^{\bullet}$. Since $\gcd(N_A, \ell_1) = 1$, the isogeny $\phi$ is a one-to-one correspondence between $E[N_A]$ and $E_B[N_A]$. Therefore, there exist $U'_k, V'_k \in E[N_A]$ such that $U_k = \phi(U'_k)$ and $V_k = \phi(V'_k)$. According to [17, Theorem IX.9(4.)], we have

$$e_{\ell_k}(U_k, V_k) = e_{\ell_k}(\phi_B(U'_k), \phi_B(V'_k)) = e_{\ell_k}(U'_k, V'_k)^{\ell_1^{\bullet}}.$$

Note that the original curve $E_0$ is defined over the base field and now the embedding degree corresponding to $p$ and $\ell_k$ is 2. By [17, Lemma IX.13], we deduce that $e_{\ell_k}(U_k, V_k) = 1$.

Conversely, from $V_k$ and $U_k$ are linearly independent, we can easily deduce that $e_{N_A}(U_k, V) \neq 1$ . In this case, $e_{N_A}(U_k, V)$ is a generator of the group $\mu_{\ell_k}$.

It is clear that $e_{N_A}(U_k, V) \neq 1$ *iff* $e_{N_A}(U_k, V)^{s'_k} \neq 1$. According to Equation (6), the order of $e_{N_A}(U, V)$ depends on the order of each $e_{N_A}(U_k, V)$:

$$ord\left(e_{N_A}(U, V)\right) = \prod_{k \in I} ord\left(e_{N_A}(U_k, V)^{s'_k}\right) = \prod_{k \in I} ord(e_{N_A}(U_k, V)).$$

If $e_{N_A}(U_k, V)$ is not equal to 1, then $e_{N_A}(U, V)$ has order $\ell_k$. Otherwise, we know that $\ell_k$ does not divide the order of $e_{N_A}(U, V)$. Consequently, we have Equation (5).

If $e_{N_A}(U, V)$ is a generator of $\mu_{N_A}$, for each $k$ we have $e_{\ell_k}(U_k, V_k) \neq 1$ and thus they are linearly independent. Hence, $\langle U_k, V_k \rangle = E_B[\ell_k]$ for each $k$. It should be noted that

$$E_B[N_A] \cong E_B[\ell_0] \oplus E_B[\ell_2] \oplus \cdots \oplus E_B[\ell_{t-1}]. \tag{7}$$

Therefore, $\langle U, V \rangle = E_B[N_A]$. Suppose that $\langle U, V \rangle = E_B[N_A]$, and now we are going to prove $e_{N_A}(U, V) \in \mu_{N_A}$ is of order $N_A$. Assume that $\ell_k$ does not divide the order of $e_{N_A}(U, V) \in \mu_{N_A}$. Then

$$e_{N_A}(U, V)^{N_A/\ell_k} = e_{N_A}([N_A/\ell_k]U, V) = e_{N_A}(U_k, V) = 1 = e_{\ell_k}(U_k, V_k).$$

This induces $\langle U_k, V_k \rangle \cong \mathbb{Z}_{\ell_k}$. From Equation (7), $\{U, V\}$ is not the torsion basis of $E_B[N_A]$, which completes the proof. ■

Proposition 3 gives a way to test whether two points could generate the torsion group $E_B[N_A]$ by checking the order of the pairing value in the group $\mu_{N_A}$. One can randomly generate a point $V \in E_B(\mathbb{F}_{p^2})/N_A E_B(\mathbb{F}_{p^2})$ using Elligator, and compute the order of $e_{N_A}(U, V)$ in $\mu_{N_A}$. Then we have a subset $I_V = \{j_k | e_{\ell_{j_k}}(U_{j_k}, V) \neq 1\}$ of the set $I = \{j | \ell_j \text{ divides } N_A\}$. Similar to the method to generate the point $U$, we generate another point $V' \neq V$ and compute:

$$f' = e_{\prod_{j_k \in I_V} \ell_{j_k}}\left(\sum_{j_k \in I_V} U_{j_k}, \prod_{j \in I \setminus I_V} [\ell_j] V'\right). \tag{8}$$

After that, we check whether $\ell_{j_k}$ divides the order of $f' \in \mu_{N_A}$ for each $j_k \in I_V$. If so, set $V = V + V'_{j_k}$, where $V'_{j_k} = [N_A/\ell_{j_k}]V'$. If the set $I_V = \{j_k | f'_{j_k} = 1\}$ is not empty, we generate another new point $V'$ and repeat the procedure. Finally, we have a point $V$ such that $e_{N_A}(U,V)$ is a generator of $\mu_{N_A}$, then $\langle U,V\rangle = E_B[N_A]$ according to Proposition 3.

It seems that once we would like to generate $V \in E_B(\mathbb{F}_{p^2})/N_A E_B(\mathbb{F}_{p^2})$, we need to randomly generate a point $R$ on $E(\mathbb{F}_q)$ and then perform a large scalar multiplication $V = [2fN_B]R$ such that $ord(V)|N_A$. Fortunately, this large scalar multiplication is not necessary when just computing $ord(e_{N_A}(U,V))$. It is obvious that $2fN_B$ and $N_A$ are coprime and therefore,

$$ord(e_{N_A}(U,V)) = ord\Big((e_{N_A}(U,R))^{2fN_B}\Big) = ord(e_{N_A}(U,R)).$$

It confirms that we can just randomly generate a point $R \in E(\mathbb{F}_q)$ to compute $ord(e_{N_A}(U,V)) = ord(e_{N_A}(U,R))$. For the same reason we can save the scalar multiplication of $V'$ in Equation (8) as well.

Checking the order of the pairing value is also a costly step. Indeed, the aim of the pairing computation is not to compute the precise pairing value but its order. Here we give a lemma, which allows us to compute compressed pairings to reach the goal.

**Lemma 1.** *Let $\gamma \in \mu_{p+1}$, then $\gamma = 1$ iff $tr_{\mathbb{F}_{p^2}/\mathbb{F}_p}(\gamma) = 2$.*

*Proof.* The necessity is obvious, now we show the sufficiency. Suppose that $\gamma = \gamma_1 + \gamma_2 \cdot i$. From $tr_{\mathbb{F}_{p^2}/\mathbb{F}_p}(\gamma) = 2$, we have $2\gamma_1 = 2$ and hence $\gamma_1 = 1$. Since $\gamma \in \mu_{p+1}$, $\gamma^{p+1} = \gamma_1^2 + \gamma_2^2 = 1$. It implies that $\gamma_2 = 0$. ∎

Therefore, to check the order of the pairing value $f'$, one can first compute $tr_{\mathbb{F}_{p^2}/\mathbb{F}_p}(f')$, and then utilize Lucas sequences to obtain $tr_{\mathbb{F}_{p^2}/\mathbb{F}_p}\left((f')^{N_A/j_k}\right)$ for each $j_k \in I_V$. Similar to Algorithm 4, we present Algorithm 5 to compute them efficiently.

**Algorithm 5** BCE: Batch cofactor exponentiation

**Require:** An element $f' \in tr_{\mathbb{F}_{p^2}/\mathbb{F}_p}(\mu_{N_A})$, $I_V$: a subset of $I = \{j|\ell_j \text{ divides } N_A\}$;
**Ensure:** $\{f'_1, f'_2, \cdots, f'_{n'}\}$, where $f'_k = tr_{\mathbb{F}_{p^2}/\mathbb{F}_p}\left((f'_k)^{\prod_{j\in I_V\setminus\{k\}}\ell_j}\right)$ and $n' = \#I_V$.

```
if n′ = 1 then
    return {f′};
end if
m′ ← ⌊n′/2⌋;
L1 ← ∏_{i=0}^{m′−1} ℓ_{I_V[i]}, L2 ← ∏_{i=m′}^{n′−1} ℓ_{I_V[i]};
left ← LS(f′, L1);                                  // Algorithm 1
right ← LS(f′, L2);                                 // Algorithm 1
Divide I_V into two subsets I1, I2 such that #I1 = n′ − m′ and #I2 = m′;
r1 ← BCE(left, I1);
r2 ← BCE(right, I2);
return r1 ∪ r2.
```

After that, we check if each of them is equal to 2 or not. Thanks to Lemma 1, we can deduce whether $(f')^{N_A/j_k}$ is equal to 1, and so its order could be determined.

In a nutshell, we present Algorithm 6 to generate $V$.

---

**Algorithm 6** GenerationV: generate a point of order $N_A$ such that $\langle U, V\rangle = E_B[N_A]$

---

**Require:** $E_B/\mathbb{F}_{p^2}$: A supersingular curve, $I$ : $\{j|\ell_j$ divides $N_A\}$, $U$: A point of order $N_A$ on $E_B/\mathbb{F}_{p^2}$;
**Ensure:** A point $V \in E_B(\mathbb{F}_{p^2})$ of order $N_A$ such that $\langle U, V\rangle = E_B[N_A]$.
1: Generate a point $V \in E_B(\mathbb{F}_{p^2})$ using Elligator;
2: $f' \leftarrow tr_{\mathbb{F}_{p^2}/\mathbb{F}_p}(e_{N_A}(U, V))$;
3: $\{f'_j\} \leftarrow$BCE$(f', I)$; // Algorithm 5
4: $I_V \leftarrow \{j_k|f'_{j_k} = 2\}$;
5: **while** $I_V \neq \emptyset$ **do**
6: Generate a point $V' \in E_B(\mathbb{F}_{p^2})$ using Elligator;
7: $U' \leftarrow \sum_{j_k \in I_V} U_{j_k}$, $L \leftarrow \prod_{j_k \in I_V} \ell_{j_k}$;
8: $f' \leftarrow tr_{\mathbb{F}_{p^2}/\mathbb{F}_p}(e_L(U', V'))$;
9: $\{f'_{j_k}\} \leftarrow$BCE$(f', I_V)$; // Algorithm 5
10: **if** $f'_{j_k} \neq 2$ for some $j_k$ **then**
11: $V' \leftarrow \prod_{j \in I \setminus I_V} [\ell_j]V'$;
12: $\{V'_{j_k}\} \leftarrow$BCM$(V', I_V)$; // Algorithm 4
13: **end if**
14: **for** each $j_k \in \{j_k|f'_{j_k} \neq 2\}$ **do**
15: $V \leftarrow V + V'_{j_k}$;
16: **end for**
17: $I_V \leftarrow \{j_k|f'_{j_k} = 2\}$;
18: **end while**
19: $V \leftarrow [2fN_B]V$;
20: **return** $V$.

---

*Remark 4.* During the torsion basis generation, the first batch cofactor multiplication of $U$ in Line 3 of Algorithm 3 and the first pairing computation in Line 2 Algorithm 6 consume large computational resources. To eliminate these two expensive parts for Alice, Bob could send her the initial $I_U$ (in Line 4 of Algorithm 3) and $I_V$ (in Line 4 of Algorithm 6). They can be translated into two $(t+1)/2$-bit strings. It would be a trade-off between the public key size and efficiency.

### 4.2 Discrete logarithm computation

Different from the case we handle in SIDH, one should compute discrete logarithms in the multiplicative group $\mu_{N_A}$. Since $N_A$ is smooth, one could use the Pohlig-Hellman algorithm [29] to simplify a discrete logarithm in $\mu_{N_A}$ to discrete logarithms in the groups $\mu_{\ell_j}$ with $j \in I = \{j|\ell_j \text{ divides } N_A\}$, and finally use the Chinese Remainder Theorem to recombine.

Firstly, we compute $h_i^{N_A/\ell_j}$ with $j \in I$ and $i = 1, 2, 3, 4$ using a divide-and-conquer approach. Note that this step could be also accelerated with the help of Lucas sequences [32, Section 3], as we proposed in Algorithm 7.

**Algorithm 7** BCEA: Batch cofactor exponentiation in $\mu_{N_A}$

**Require:** An element $h' \in \mu_{N_A}$, $I'$: a subset of $I = \{j|\ell_j \text{ divides } N_A\}$;
**Ensure:** $\{h'_1, h'_2, \cdots, h'_{n'}\}$, where $h'_k = \left((f'_k)^{\prod_{j\in I'\setminus\{k\}} \ell_j}\right)$ and $n' = \#I'$.

```
if n' = 1 then
    return {h'};
end if
m' ← ⌊n'/2⌋;
L_1 ← ∏_{i=0}^{m'-1} ℓ_i, L_2 ← ∏_{i=m'}^{n'-1} ℓ_i;
left ← ELS(h, L_1);                                   // Algorithm 2
right → ELS(h, L_2);                                  // Algorithm 2
Divide I' into two subsets I_1, I_2 such that #I_1 = n' − m' and #I_2 = m';
r_1 ←BCEA(left,I_1);
r_2 ←BCEA(right,I_2);
return r_1 ∪ r_2.
```

After that, we need to choose one of $h_i$, $i = 1, 2, 3, 4$ to be the base of discrete logarithms. A direct approach is to check whether the order of $h_i$ is equal to $N_A$. It is easy to be done by observing if $h_i^{N_A/\ell_j} = 1$ for some $j$. When it happens, $\ell_j$ does not divide the order of $h_i$ and thus $h_i$ is not a generator. We use *label* to mark which one to be the base.

Without loss of generality, we assume that $h_4$ is a generator. For each $j \in I$, compute the discrete logarithms of $h_1^{N_A/\ell_j}$, $h_2^{N_A/\ell_j}$ and $h_3^{N_A/\ell_j}$ to the base $h_4^{N_A/\ell_j}$, denoted by $s_1^{(j)}$, $s_2^{(j)}$ and $s_3^{(j)}$, respectively. This step is efficient since $\ell_j$ is relatively small.

Finally, from $s_1^{(j)}$, $s_2^{(j)}$ and $s_3^{(j)}$ with $j \in I$ we recover $s_1 = \log_{h_4}(h_1)$, $s_2 = \log_{h_4}(h_2)$ and $s_3 = \log_{h_4}(h_3)$, respectively. This step is fast with the help of the Chinese Remainder Theorem.

Algorithm 8 is the pseudocode summarizing our ideas to compute discrete logarithms.

### 4.3 An alternative approach to solve discrete logarithms

In this subsection we propose another method to overcome the issue mentioned in Remark 1. The method is not as efficient as the former method, but avoids repeating the procedure when none of $h_i$, $i = 1, 2, 3, 4$ could generate $\mu_{N_A}$.

Firstly, we compute four discrete logarithms of $h_i$, $i = 1, 2, 3, 4$ to the base $h_0 = e_{N_A}(\phi_B(P_A), \phi_B(Q_A))$, which is the generator of $\mu_{N_A}$. Since $P_A$ and $Q_A$

**Algorithm 8** Discrete logarithm computation

**Input**: $I$: $\{j|\ell_j$ divides $N_A\}$; $h_1$, $h_2$, $h_3$, $h_4$: the values computed in Equation (4);
**Output**: *label*: a label to mark the base is $H_b = h_{label}$, and $H_1, H_2, H_3$ are the other three elements; $s_1, s_2, s_3$: Integers in $\{0, 1, \cdots, N_A - 1\}$ such that $H_1 = h_b^{s_1}$, $H_2 = h_b^{s_2}$ and $H_3 = h_b^{s_3}$.

**for** $k \in \{1, 2, 3, 4\}$ **do**
  $\{h_k^{(j)}\} \leftarrow \mathrm{BCEA}(h_k, I)$; // Algorithm 7
**end for**
**for** $k \in \{1, 2, 3, 4\}$ **do**
  **if** $h_k^{(j)} \neq 1$ for all $j$ **then**
    *label* $\leftarrow k$, $H_b \leftarrow h_k$, $\{H_1, H_2, H_3\} \leftarrow \{h_j|j \neq k\}$; **break**;
  **end if**
**end for**
**for** each $k \in \{1, 2, 3\}$ **do**
  **for** each $j \in I$ **do**
    find $s_k^{(j)}$ such that $H_k^{(j)} = \left(H_b^{(j)}\right)^{s_k^{(j)}}$;
  **end for**
**end for**
**for** each $k \in \{1, 2, 3\}$ **do**
  Use the Chinese remainder theorem to compute $s_k \bmod N_A$ such that $s_k \equiv s_k^{(j)} \bmod \ell_j$ with $j \in I$;
**end for**
**return** $s_1, s_2, s_3, label$.

are fixed, the value $h_0 = e_{N_A}(P_A, Q_A)^{N_B}$ could be precomputed to accelerate the performance. Note that $c_i = -\log_{h_0} h_i$, $d_i = \log_{h_0} h_{i+2}$, $i = 0, 1$. For each $j \in I = \{j|\ell_j$ divides $N_A\}$, let $c_i^{(j)} = c_i \bmod \ell_j$, $d_i^{(j)} = d_i \bmod \ell_j$, $i = 0, 1$. Since $\ell_j$ is prime and $\langle U_A, V_A\rangle = E_B[N_A] = \langle \phi_B(P_A), \phi_B(Q_A)\rangle$, either $d_0^{(j)}$ or $d_1^{(j)}$ is invertible. Therefore, we have

$$\left(S_1^{(j)}, S_2^{(j)}, S_3^{(j)}, label_j\right) = \begin{cases} \left(-(d_1^{(j)})^{-1} d_0^{(j)}, -(d_1^{(j)})^{-1} c_1^{(j)}, (d_1^{(j)})^{-1} c_0^{(j)}, 1\right), & \text{if } d_1^{(j)} \neq 0, \\ \left(1, (d_0^{(j)})^{-1} c_1^{(j)}, -(d_0^{(j)})^{-1} c_0^{(j)}, 0\right), & \text{otherwise.} \end{cases} \tag{9}$$

Thanks to the Chinese Remainder Theorem, one could obtain $S_i \bmod N_A$ from $S_i^{(j)} \bmod \ell_j$ with $j \in I$.

The public key is $(B, S_1, S_2, S_3, label)$, where

$$label = label_0 + label_2 \cdot 2 + \cdots + label_{t-1} \cdot 2^{(t-1)/2}. \tag{10}$$

The pesudocode is proposed in Algorithm 9.

A question raised here is how Alice generates a kernel generator $G_A$ of the group $\langle \phi_B(P_A) + [sk_A]\phi_B(Q_A)\rangle = \langle [d_1 - c_1 \cdot sk_A]U + [-d_0 + c_0 \cdot sk_A]V\rangle$ according to $(B, S_1, S_2, S_3, label)$.

**Algorithm 9** Another approach to compute discrete logarithms

**Input**: $I$: $\{j|\ell_j$ divides $N_A\}$; $h_0$: $e_{N_A}(P_A, Q_A)^{N_B}$; $h_1$, $h_2$, $h_3$, $h_4$: the values computed in Equation (4);

**Output**: *label*: A $(t-1)/2$-bit integer defined in Equation (10); $S_1, S_2, S_3$: Integers in $\{0, 1, \cdots, N_A - 1\}$ defined as above, which satisfies Equation (9).

```
for k ∈ {0,1,2,3,4} do
    {h_k^(j)} ← BCEA(h_k, I);                                    // Algorithm 7
end for
for each j ∈ I do
    for each k ∈ {1,2} do
        find c_k^(j) such that h_k^(j) = (h_0^(j))^(-c_k^(j));
        find d_k^(j) such that h_k^(2+j) = (h_0^(j))^(d_k^(j));
    end for
    if d_1^(j) ≠ 0 then
        S_1^(j) ← -(d_1^(j))^(-1) d_0^(j), S_2^(j) ← -(d_1^(j))^(-1) c_1^(j), S_3^(j) ← (d_1^(j))^(-1) c_0^(j),
        label_j ← 1;
    else
        S_1^(j) ← 1, S_2^(j) ← (d_0^(j))^(-1) c_1^(j), S_3^(j) ← -(d_0^(j))^(-1) c_0^(j), label_j ← 0;
    end if
end for
for each k ∈ {1,2,3} do
    Use the Chinese remainder theorem to compute S_k mod N_A such that S_k ≡
    S_k^(j) mod ℓ_j with j ∈ I;
end for
label ← Σ_{j∈I} label_j · 2^j;
return S_1, S_2, S_3, label.
```

Using Algorithms 3 and 6, Alice obtains $U$ and $V$. Besides, she could construct

$$S_4^{(j)} \equiv 1 \bmod \ell_j \text{ if } label_j = 1, \text{ or } S_4^{(j)} \equiv 0 \bmod \ell_j \text{ otherwise.} \tag{11}$$

Utilizing the Chinese Remainder Theorem, Alice could recover $S_4 \bmod N_A$ from Equation (11). Let

$$G_A = [S_4 + S_2 \cdot sk_A]U + [S_3 + S_1 \cdot sk_A]V.$$

Now we show that $G_A$ is a kernel generator of $\langle \phi_B(P_A) + [sk_A]\phi_B(Q_A) \rangle$. It is equivalent to show that for each $k \in I$,

$$\langle [N_A/\ell_k]G_A \rangle = \langle [d_1 - c_1 \cdot sk_A]U_k + [-d_0 + c_0 \cdot sk_A]V_k \rangle, \tag{12}$$

where $U_k = [N_A/\ell_k]U$ and $V_k = [N_A/\ell_k]V$. From Equation (9), we know that

$$\begin{cases} S_1^{(j)} \equiv S_1 \bmod \ell_j \\ S_2^{(j)} \equiv S_2 \bmod \ell_j \\ S_3^{(j)} \equiv S_3 \bmod \ell_j \end{cases} \text{if } label_j = 1, \text{ or } \begin{cases} S_1^{(j)} \equiv 1 \bmod \ell_j \\ S_2^{(j)} \equiv S_2 \bmod \ell_j \\ S_3^{(j)} \equiv S_3 \bmod \ell_j \end{cases} \text{otherwise.}$$

If $label_j = 1$, then $S_4' = 1 \bmod \ell_j$ and hence

$$[N_A/\ell_k]G_A = [1 + S_2 \cdot sk_A]U_k + [S_3 + S_1 \cdot sk_A]V_k.$$

Note that

$$\begin{aligned} &[1 + S_2 \cdot sk_A]U_k + [S_3 + S_1 \cdot sk_A]V_k \\ =&[1 + S_2^{(j)} \cdot sk_A]U_k + [S_1^{(j)} + S_3^{(j)} \cdot sk_A]V_k \\ =&[1 - (d_1^{(j)})^{-1}c_1^{(j)} \cdot sk_A]U_k + [-(d_1^{(j)})^{-1}d_0^{(j)} + (d_1^{(j)})^{-1}c_0^{(j)} \cdot sk_A]V_k \\ =&[(d_1^{(j)})^{-1}] \cdot \left([d_1^{(j)} - c_1^{(j)} \cdot sk_A]U_k + [-d_0^{(j)} + c_0^{(j)} \cdot sk_A]V_k\right) \\ =&[(d_1^{(j)})^{-1}] \cdot ([d_1 - c_1 \cdot sk_A]U_k + [-d_0 + c_0 \cdot sk_A]V_k)\,. \end{aligned}$$

In other words, we have

$$[N_A/\ell_k]G_A \in \langle [d_1 - c_1 \cdot sk_A]U_k + [-d_0 + c_0 \cdot sk_A]V_k \rangle$$

when $S_4^{(j)} = 1$. Similarly, we can deduce that $[N_A/\ell_k]G_A$ and $[d_1 - c_1 \cdot sk_A]U_k + [-d_0 + c_0 \cdot sk_A]V_k$ are linearly dependent when $S_1^{(j)} = 0$. Therefore, $G_A$ satisfies Equation (12).

**Proposition 4.** *After applying Algorithm 9 and modifying the public key, compressed M-SIDH is still secure whenever M-SIDH is secure.*

*Proof.* Analogous to what Bob does in the key agreement phase, Eve could recover $(S_1^{(j)}, S_2^{(j)}, S_3^{(j)}, S_4^{(j)})$, then $S_1$, $S_2$, $S_3$ and $S_4$ using the Chinese Remainder Theorem, and thus he is able to compute

$$\begin{aligned} P_A' &= [S_4]U_A + [S_3]V_A = [\alpha']\phi_B(P_A), \\ Q_A' &= [S_2]U_A + [S_1]V_A = [\alpha']\phi_B(Q_A), \end{aligned}$$

where $\alpha' \in \mathbb{Z}_{N_A}^*$ satsifies

$$\begin{cases} \alpha' \equiv d_1^{(j)} \bmod \ell_j, \text{if } label_j = 1, \\ \alpha' \equiv d_0^{(j)} \bmod \ell_j, \text{if } label_j = 0. \end{cases}$$

Similar to the proof in Proposition 2, one could compute pairings and solve one discrete logarithm to compute $(\alpha')^2 \bmod N_A$. Choose one root $\alpha_0'$ such that $(\alpha_0'\alpha')^2 \equiv 1 \bmod N_A$ and set $\alpha = \alpha_0'\alpha'$, then

$$\begin{aligned} [\alpha]\phi_B(P_A) &= [a'][a_0^{-1}]\phi_B(P_A), \\ [\alpha]\phi_B(Q_A) &= [a'][a_0^{-1}]\phi_B(Q_A). \end{aligned}$$

In the M-SIDH setting, recovering $\alpha$ is hard and so is $\alpha_0'$, which ends the proof.■

The new method is not as efficient as the method proposed in Section 4.2. The main reason is that the former requires one more execution of BCEA (Algorithm 7) and one more discrete logarithm. Furthermore, the public key size is slightly larger since a $(t+1)/2$-bit integer *label* is required. However, the new method confirms that there is no need to select another secret key when all $h_i$ $(i = 1, 2, 3, 4)$ are not of full order $N_A$. This leads to more stable performance.

## 5 Implementation Results

In this section, we implement compressed M-SIDH in SageMath (version 9.5) [2] and give our experimental results.

Isogeny computation is the most expensive part of (compressed) M-SIDH. There are mainly two ways to construct the isogeny. One is the traditional Vélu's formula [35], and the other is a more efficient formula to construct the large degree isogeny [6]. We combine both of them to implement compressed M-SIDH. For small degree isogeny computations we use traditional Vélu's formula, and use the method proposed in [6] to compute the large degree isogeny.

Based on the code[1] from [6], we give a proof-of-concept implementation of compressed M-SIDH in SageMath. We compiled our code[2] by using a 12th Gen Intel(R) Core(TM) i9-12900K 3.20 GHz on 64-bit Linux with the so-called turbo boost and hyper-threading features disabled. Table 1 reports the performance of the key generation phase.

**Table 1.** Experimental results of key generation of Alice in compressed M-SIDH for the NIST-1 level of security.

| Procedure | Alice | Bob |
|---|---|---|
| Isogeny Computation | 687.05s | 690.53s |
| Torsion Basis Generation | 16.29s | 16.37s |
| Pairing Computation | 16.24s | 16.15s |
| Discrete Logarithm Computation (Alg. 8/Alg. 9) | 5.66s/6.04s | 5.55s/6.01s |
| Total Cost (the whole key generation phase) | 725.24s/725.62s | 728.60s/729.06s |

As shown in Table 1, isogeny computation dominates the cost of key generation. One may try to utilize several techniques proposed in the literature to speed up the compressed M-SIDH implementation. There are several works on the optimizations of CSIDH [9]. For example, the approach [10] to find an optimal strategy of CSIDH could be easily extended to the isogeny computation of M-SIDH. It is also possible to improve the performance by changing the permutation of the $\ell_j$-isogeny computation [19]. The improvement of large degree isogeny computation is explored by [3].

Torsion basis generation and pairing computation are the efficiency bottlenecks of public-key compression in M-SIDH. The computational cost of discrete

[1] `https://velusqrt.isogeny.org/`
[2] `https://github.com/CompressedMSIDH/CompressedMSIDH`

logarithm computation is approximately one third of that of torsion basis generation. We leave the exploration of the faster implementation of compressed M-SIDH for future work.

## 6 Conclusion

In this paper, we proposed a method to compress the public key of M-SIDH by utilizing several techniques. The implementation showed that public-key compression was relatively efficient, compared with isogeny computation.

It should be noted that the techniques proposed in this work could be also extended easily to other SIDH-like schemes. Although the implementation of compressed M-SIDH is not efficient now because of the huge characteristic of the base field and expensive isogeny computation, we believe that compressed SIDH-like schemes could find their positions with further research.

## References


1. The National Institute of Standards and Technology (NIST): Post-quantum cryptography standardization, `https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Post-Quantum-Cryptography-Standardization`
2. The Sage Developers: SageMath, the Sage Mathematics Software System (version 9.5) (2022), `https://sagemath.org`
3. Adj, G., Chi-Domínguez, J.J., Rodríguez-Henríquez, F.: Karatsuba-based square-root Vélu's formulas applied to two isogeny-based protocols. Journal of Cryptographic Engineering (Jul 2022)
4. Azarderakhsh, R., Campagna, M., Costello, C., De Feo, L., Hess, B., Hutchinson, A., Jalali, A., Jao, D., Karabina, K., Koziel, B., LaMacchia, B., Longa, P., Naehrig, M., Pereira, G., Renes, J., Soukharev, V., Urbanik, D.: Supersingular Isogeny Key Encapsulation (2020), `http://sike.org`
5. Azarderakhsh, R., Jao, D., Kalach, K., Koziel, B., Leonardi, C.: Key Compression for Isogeny-Based Cryptosystems. In: Proceedings of the 3rd ACM International Workshop on ASIA Public-Key Cryptography. pp. 1–10 (2016)
6. Bernstein, D., Feo, L., Leroux, A., Smith, B.: Faster computation of isogenies of large prime degree. Open Book Series **4**, 39–55 (2020)
7. Bernstein, D.J., Hamburg, M., Krasnova, A., Lange, T.: Elligator: Elliptic-curve points indistinguishable from uniform random strings. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security. p. 967980 (2013)
8. Castryck, W., Decru, T.: An efficient key recovery attack on SIDH (preliminary version). Cryptology ePrint Archive, Paper 2022/975 (2022), `https://eprint.iacr.org/2022/975`
9. Castryck, W., Lange, T., Martindale, C., Panny, L., Renes, J.: CSIDH: An Efficient Post-Quantum Commutative Group Action. In: Peyrin, T., Galbraith, S. (eds.) Advances in Cryptology – ASIACRYPT 2018. pp. 395–427. Springer International Publishing, Cham (2018)
10. Chi-Domínguez, J.J., Rodríguez-Henríquez, F.: Optimal strategies for csidh. Advances in Mathematics of Communications **16**(2), 383–411 (2022)

11. Costello, C.: B-SIDH: Supersingular Isogeny Diffie-Hellman Using Twisted Torsion. In: Moriai, S., Wang, H. (eds.) Advances in Cryptology – ASIACRYPT 2020. pp. 440–463. Springer International Publishing, Cham (2020)
12. Costello, C., Jao, D., Longa, P., Naehrig, M., Renes, J., Urbanik, D.: Efficient Compression of SIDH Public Keys. In: Coron, J.S., Nielsen, J.B. (eds.) Advances in Cryptology – EUROCRYPT 2017. pp. 679–706. Springer International Publishing, Cham (2017)
13. De Feo, L., Delpech de Saint Guilhem, C., Fouotsa, T.B., Kutas, P., Leroux, A., Petit, C., Silva, J., Wesolowski, B.: Séta: Supersingular Encryption fromăTorsion Attacks. In: Tibouchi, M., Wang, H. (eds.) Advances in Cryptology – ASIACRYPT 2021. pp. 249–278. Springer International Publishing, Cham (2021)
14. Fouotsa, T.B.: SIDH with masked torsion point images. Cryptology ePrint Archive, Paper 2022/1054 (2022), `https://eprint.iacr.org/2022/1054`
15. Fouotsa, T.B., Moriya, T., Petit, C.: M-SIDH and MD-SIDH: countering SIDH attacks by masking information. Cryptology ePrint Archive, Paper 2023/013 (2023), `https://eprint.iacr.org/2023/013`
16. Frey, G., Rück, H.G.: A Remark Concerning M-Divisibility and the Discrete Logarithm in the Divisor Class Group of Curves. Math. Comput. **62**(206), 865874 (1994)
17. Galbraith, S.: Pairings, pp. 183–214. London Mathematical Society Lecture Note Series, Cambridge University Press (2005)
18. Hutchinson, A., Karabina, K., Pereira, G.: Memory Optimization Techniques for Computing Discrete Logarithms in Compressed SIKE. In: Cheon, J.H., Tillich, J.P. (eds.) Post-Quantum Cryptography. pp. 296–315. Springer International Publishing, Cham (2021)
19. Hutchinson, A., LeGrow, J., Koziel, B., Azarderakhsh, R.: Further Optimizations of CSIDH: A Systematic Approach to Efficient Strategies, Permutations, and Bound Vectors. In: Conti, M., Zhou, J., Casalicchio, E., Spognardi, A. (eds.) Applied Cryptography and Network Security. pp. 481–501. Springer International Publishing, Cham (2020)
20. Jao, D., De Feo, L.: Towards Quantum-Resistant Cryptosystems from Supersingular Elliptic Curve Isogenies. In: Yang, B.Y. (ed.) Post-Quantum Cryptography. pp. 19–34. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
21. Kohel, D., Lauter, K., Petit, C., Tignol, J.P.: On the quaternion $\ell$-isogeny path problem. LMS Journal of Computation and Mathematics **17**(A), 418432 (2014)
22. Lin, K., Lin, J., Wang, W., Zhao, C.A.: Faster Public-key Compression of SIDH with Less Memory. Cryptology ePrint Archive, Paper 2021/992 (2021), `https://eprint.iacr.org/2021/992`
23. Lin, K., Wang, W., Wang, L., Zhao, C.A.: An Alternative Approach for Computing Discrete Logarithms in Compressed SIDH. Cryptology ePrint Archive, Paper 2021/1528 (2021), `https://eprint.iacr.org/2021/1528`
24. Maino, L., Martindale, C.: An attack on SIDH with arbitrary starting curve. Cryptology ePrint Archive, Paper 2022/1026 (2022), `https://eprint.iacr.org/2022/1026`
25. Moriya, T.: Masked-degree SIDH. Cryptology ePrint Archive, Paper 2022/1019 (2022), `https://eprint.iacr.org/2022/1019`
26. Naehrig, M., Renes, J.: Dual Isogenies and Their Application to Public-Key Compression for Isogeny-Based Cryptography. In: Galbraith, S.D., Moriai, S. (eds.) Advances in Cryptology – ASIACRYPT 2019. pp. 243–272. Springer International Publishing, Cham (2019)

27. Pereira, G.C.C.F., Barreto, P.S.L.M.: Isogeny-Based Key Compression Without Pairings. In: Garay, J.A. (ed.) Public-Key Cryptography – PKC 2021. pp. 131–154. Springer International Publishing, Cham (2021)
28. Pereira, G.C.C.F., Doliskani, J., Jao, D.: $x$-only point addition formula and faster compressed SIKE. Journal of Cryptographic Engineering **11**, 57–69 (2021)
29. Pohlig, S., Hellman, M.: An Improved Algorithm for Computing Logarithms over GF($p$) and Its Cryptographic Significance (Corresp.). IEEE Trans. Inf. Theor. **24**(1), 106110 (2006)
30. Richard Crandall, C.B.P.: Prime numbers: a computational perspective. Springer, 2nd ed edn. (2005)
31. Robert, D.: Breaking SIDH in polynomial time. Cryptology ePrint Archive, Paper 2022/1038 (2022), `https://eprint.iacr.org/2022/1038`
32. Scott, M., Barreto, P.S.L.M.: Compressed Pairings. In: Franklin, M. (ed.) Advances in Cryptology – CRYPTO 2004. pp. 140–156. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)
33. Sutherland, A.: Order computations in generic groups. PhD thesis, Massachusetts Institute of Technology (2007)
34. Tate, J.: $WC$-groups over $p$-adic fields. Exposé no. 156. In Années 1956/57 - 1957/58, exposés 137-168, volume 4 of Séminaire Bourbaki p. 265277 (1956-1958)
35. Vélu, J.: Isogénies entre courbes elliptiques. Comptes Rendus Hebdomadaires des Séances de l'Académie des Sciences, Série A **273**, 238–241 (1971)
36. Zanon, G.H.M., Simplicio, M.A., Pereira, G.C.C.F., Doliskani, J., Barreto, P.S.L.M.: Faster Key Compression for Isogeny-Based Cryptosystems. IEEE Transactions on Computers **68**(5), 688–701 (2019)