An Alternative Approach for Computing Discrete Logarithms in Compressed SIDH

Kaizhan Lin[1], Weize Wang[1], Lin Wang[2], and Chang-An Zhao✉[1,3]

[1] School of Mathematics, Sun Yat-sen University, Guangzhou 510275, P. R. China
linkzh5@mail2.sysu.edu.cn
wangwz@mail2.sysu.edu.cn
zhaochan3@mail.sysu.edu.cn

[2] Science and Technology on Communication Security Laboratory, Chengdu 610041, Sichuan, P. R. China
linwang@math.pku.edu.cn

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

Abstract. Currently, public-key compression of supersingular isogeny Diffie-Hellman (SIDH) and its variant, supersingular isogeny key encapsulation (SIKE) involve pairing computation and discrete logarithm computation. In this paper, we propose novel methods to compute only 3 discrete logarithms instead of 4, in exchange for computing a lookup table efficiently. The algorithms also allow us to make a trade-off between memory and efficiency. Our implementation shows that the efficiency of our algorithms is close to that of the previous work, and our algorithms perform better in some special cases.

Keywords: Isogeny-based Cryptography · SIDH · SIKE · Public-key Compression · Discrete Logarithms

1 Introduction

Isogeny-based cryptography has received widespread attention due to its small public key sizes in post-quantum cryptography. The most attractive isogeny-based cryptosystems are supersingular isogeny Diffie-Hellman (SIDH) [11] and its variant, supersingular isogeny key encapsulation (SIKE) [3]. The latter one was submitted to NIST, and now it still remains one of the nine key encapsulation mechanisms in Round 3 of the NIST standardization process.

Indeed, Public key sizes in SIDH/SIKE can further be compressed. Azarderakhsh et al. [4] firstly proposed a method for public-key compression, and later Costello et al. [6] proposed new techniques to further reduce the public-key size and make public-key compression practical. Zanon et al. [20,21] improved the implementation of compression and decompression by utilizing several techniques. Naehrig and Renes [14] employed the dual isogeny to increase performance of compression techniques, while the methods for efficient binary torsion basis generation were presented in [16].

However, the implementation of pairing computation and discrete logarithm computation are still bottlenecks of public-key compression of SIDH/SIKE. Lin et al. [13] saved about one third of memory for pairing computation and made it perform faster. To avoid pairing computation, Pereira and Barreto [15] compressed the public key with the help of ECDLP. As for discrete logarithms, Hutchinson et al. [10] utilized signed-digit representation and torus-based representation to reduce the size of lookup tables for computing discrete logarithms. Both of them compress discrete logarithm tables by a factor of 2, and the former one reduces without any computational cost of lookup table construction. It makes practical to construct the lookup tables without precomputation.

In the current state-of-the-art implementation, there are $four$ values to be obtained in discrete logarithm computation. Note that one of the four values must be invertible in $\mathbb{Z}_{\ell^{e_\ell}}$. One only needs to get three new values [6] by performing one inversion and three multiplications in $\mathbb{Z}_{\ell^{e_\ell}}$, and then transmit them. It is natural to ask whether one can compute the *three* transmitted values directly during discrete logarithm computation.

In this paper, we propose an alternative way to compute discrete logarithms. We summarize our work as follows:

- We propose a trick to compute only 3 discrete logarithms to compress the public key, in exchange for computing a lookup table efficiently.
- Currently, the algorithm used for discrete logarithm computation in compressed SIDH/SIKE is recursive. Inspired by [5], we present a non-recursive algorithm to compute discrete logarithms.
- We propose new algorithms to compute discrete logarithms in public-key compression of SIDH/SIKE. Our experimental results show that the efficiency of new algorithms is close to that of the previous work. Furthermore, our algorithms may perform well in storage constrained environments since we can make a memory-efficiency trade-off.

The sequel is organized as follows. In Section 2 we review the techniques that utilized for computing discrete logarithms in public-key compression. In Section 3 we propose new techniques to compute discrete logarithms without precomputation. We compare our experimental results with the previous work in Section 4 and conclude in Section 5.

2 Notations and Preliminaries

2.1 Notations

In this paper, we use $E_A : y^2 = x^3 + Ax^2 + x$ to denote a supersingular Montgomery curve defined over the field $\mathbb{F}_{p^2} = \mathbb{F}_p[i]/\langle i^2 + 1\rangle$, where $p = 2^{e_2}3^{e_3} - 1$. Let $E_6[2^{e_2}] = \langle P_2, Q_2\rangle$ and $E_6[3^{e_3}] = \langle P_3, Q_3\rangle$. We also use ϕ_2 and ϕ_3 to denote the 2^{e_2}-isogeny and 3^{e_3}-isogeny, respectively. Besides, we define μ_n to be a multiplicative subgroup of order n in $\mathbb{F}^*_{p^2}$, i.e,

$$\mu_n = \{\delta \in \mathbb{F}^*_{p^2} | \delta^n = 1\}.$$

As usual, we denote the cost of one $\mathbb{F}_{p^2}$ field multiplication and squaring by $\mathbf{M}$ and $\mathbf{S}$, respectively. We also use $\mathbf{m}$ and $\mathbf{s}$ to denote the cost of one multiplication and squaring in the field $\mathbb{F}_p$. When estimating the cost, we assume that $\mathbf{M} \approx 3\mathbf{m}$, $S \approx 2\mathbf{m}$ and $\mathbf{s} \approx 0.8\mathbf{m}$.

2.2 Public-key Compression

In this subsection, we briefly review public-key compression of SIDH/SIKE, and concentrate on computing discrete logarithms. We only consider how to compress two points of order 3^{e_3}, while the other case is similar. We refer to [11,7,8,3] for more details of SIDH and SIKE. For their security analysis, see [9,12,17,1].

Azarderakhsh et al. [4] first presented techniques to compress the public key. The main idea is to generate a 3^{e_3}-torsion basis $\langle U_3, V_3 \rangle$ by a deterministic pseudo-random number generator, and then utilize this basis to linearly represent $\phi_2(P_3)$ and $\phi_2(Q_3)$. That is,

$$\begin{bmatrix} \phi_2(P_3) \\ \phi_2(Q_3) \end{bmatrix} = \begin{bmatrix} a_0 & b_0 \\ a_1 & b_1 \end{bmatrix} \begin{bmatrix} U_3 \\ V_3 \end{bmatrix}. \tag{1}$$

Note that

$$\begin{aligned} r_0 &= \mathrm{e}_{3^{e_3}}(U_3, V_3), \\ r_1 &= \mathrm{e}_{3^{e_3}}(U_3, \phi_2(P_3)) = \mathrm{e}_{3^{e_3}}(U_3, a_0U_3 + b_0V_3) = r_0^{b_0}, \\ r_2 &= \mathrm{e}_{3^{e_3}}(U_3, \phi_2(Q_3)) = \mathrm{e}_{3^{e_3}}(U_3, a_1U_3 + b_1V_3) = r_0^{b_1}, \\ r_3 &= \mathrm{e}_{3^{e_3}}(V_3, \phi_2(P_3)) = \mathrm{e}_{3^{e_3}}(V_3, a_0U_3 + b_0V_3) = r_0^{-a_0}, \\ r_4 &= \mathrm{e}_{3^{e_3}}(V_3, \phi_2(Q_3)) = \mathrm{e}_{3^{e_3}}(V_3, a_1U_3 + b_1V_3) = r_0^{-a_1}. \end{aligned} \tag{2}$$

Therefore, with the help of bilinear pairings, one can compute a_0, a_1, b_0 and b_1 by computing four discrete logarithms in the multiplicative group $\mu_{3^{e_3}}$.

Instead of $(\phi_2(P_3), \phi_2(Q_3))$, one could transmit the tuple (a_0, b_0, a_1, b_1, A). Costello et al. [6] observed either $a_0 \in \mathbb{Z}^*_{3^{e_3}}$ or $b_0 \in \mathbb{Z}^*_{3^{e_3}}$ since the order of $\phi_A(P_B)$ is 3^{e_3}, and concluded that the public key could be compressed to the tuple

$$\left(a_0^{-1}b_0, a_0^{-1}a_1, a_0^{-1}b_1, 0, A\right), \text{ or } \left(b_0^{-1}a_0, b_0^{-1}a_1, b_0^{-1}b_1, 1, A\right) \text{ if } a_0 \notin \mathbb{Z}^*_{3^{e_3}}.$$

Zanon et al. [21] proposed another new technique, called *reverse basis decomposition*, to speed up the performance of computing discrete logarithms. Note that $\langle \phi_2(P_3), \phi_2(Q_3) \rangle$ is also a 3^{e_3}-torsion basis of E_A. The coefficient matrix in Equation (1) is invertible, i.e.,

$$\begin{bmatrix} U_3 \\ V_3 \end{bmatrix} = \begin{bmatrix} c_0 & d_0 \\ c_1 & d_1 \end{bmatrix} \begin{bmatrix} \phi_2(P_3) \\ \phi_2(Q_3) \end{bmatrix}, \text{ where } \begin{bmatrix} c_0 & d_0 \\ c_1 & d_1 \end{bmatrix} = \begin{bmatrix} a_0 & b_0 \\ a_1 & b_1 \end{bmatrix}^{-1}.$$

Correspondingly, the following pairing computation substitutes for Equation (2):

$$
\begin{aligned}
r_0 &= \mathrm{e}_{3^{e_3}}\left(\phi_2\left(P_3\right), \phi_2\left(Q_3\right)\right) = \mathrm{e}_{3^{e_3}}\left(P_3, Q_3\right)^{2^{e_2}}, \\
r_1 &= \mathrm{e}_{3^{e_3}}\left(\phi_2\left(P_3\right), U_3\right) = \mathrm{e}_{3^{e_3}}\left(\phi_2\left(P_3\right), c_0\phi_2\left(P_3\right) + d_0\phi_2\left(Q_3\right)\right) = r_0^{d_0}, \\
r_2 &= \mathrm{e}_{3^{e_3}}\left(\phi_2\left(P_3\right), V_3\right) = \mathrm{e}_{3^{e_3}}\left(\phi_2\left(P_3\right), c_1\phi_2\left(P_3\right) + d_1\phi_2\left(Q_3\right)\right) = r_0^{d_1}, \\
r_3 &= \mathrm{e}_{3^{e_3}}\left(\phi_2\left(Q_3\right), U_3\right) = \mathrm{e}_{3^{e_3}}\left(\phi_2\left(Q_3\right), c_0\phi_2\left(P_3\right) + d_0\phi_2\left(Q_3\right)\right) = r_0^{-c_0}, \\
r_4 &= \mathrm{e}_{3^{e_3}}\left(\phi_2\left(Q_3\right), V_3\right) = \mathrm{e}_{3^{e_3}}\left(\phi_2\left(Q_3\right), c_1\phi_2\left(P_3\right) + d_1\phi_2\left(Q_3\right)\right) = r_0^{-c_1}.
\end{aligned} \tag{3}
$$

In this situation one needs to transmit

$$(-d_1^{-1}d_0, -d_1^{-1}c_1, d_1^{-1}c_0, 0, A), \text{ or } (-d_0^{-1}d_1, d_0^{-1}c_1, -d_0^{-1}c_0, 1, A) \text{ if } d_1 \notin \mathbb{Z}_{3^{e_3}}^*.$$

Since the value r_0 only depends on public parameters, the arbitrary order of r_0 could be precomputed to improve the implementation of computing discrete logarithms. In addition, note that the order of the group $\mu_{3^{e_3}}$ is smooth. Therefore, four discrete logarithms could be computed by using Pohlig-Hellman algorithm [18] , as we will describe in the following subsection.

2.3 Pohlig-Hellman algorithm

Pohlig-Hellman algorithm is an algorithm which is used to efficiently compute discrete logarithms in a group whose order is smooth. For a discrete logarithm $h = g^x \in \mu_{\ell^{e_\ell}}$, one could simplify it to e_ℓ discrete logarithms in a multiplicative group of order ℓ.

Algorithm 1 Pohlig-Hellman Algorithm

Ensure: $\langle g \rangle$: multiplicative group of order ℓ^{e_ℓ}; h: challenge.
Require: x: integer $x \in [0, \ell^{e_\ell})$ such that $h = g^x$.

```
s ← g^(ℓ^(e_ℓ−1)), x ← 0, h_0 ← h;
for i from 0 to e_ℓ − 1 do
    t_i ← h_i^(ℓ^(e_ℓ−1−i));
    find x_i ∈ {0, 1, ··· , ℓ − 1} such that t_i = s^(x_i);
    x ← x + x_i · ℓ^i, h_(i+1) ← h_i · g^(−x_i ℓ^i);
end for
return x.
```

As we can see in Algorithm 1, a lookup table

$$T_1[i][j] = g^{-j\ell^i}, i = 0, 1, \cdots, e_\ell - 1, j = 0, 1, \cdots, \ell - 1,$$

can be precomputed to save the computational cost. Besides, one can also use a windowed version of Pohlig-Hellman algorithm to simplify the discrete logarithm to $\frac{e_\ell}{w}$ discrete logarithms in a group of order $L = \ell^w$, where $w|e_\ell$. The windowed version of Pohlig-Hellman algorithm reduces the loop length, but it consumes more storage.

When w does not divide e_ℓ the procedure needs some modifications. Zanon et al. handled this situation by storing two tables [21, Section 6.2]:

$$\begin{aligned} &T_1[i][j]{=}g^{-j\ell^{wi}}, i = 0, 1, \cdots, e_\ell - 1; \\ &T_2[i][j] = \begin{cases} g^{-j}, \text{if } i = 0, \\ g^{-j\ell^{w(i-1)+e_\ell \bmod w}}, \text{otherwise}; \end{cases} \end{aligned} \tag{4}$$

where $j = 0, 1, \cdots, \ell^{e_\ell} - 1$. This doubles the storage compared to the situation when w divides e_ℓ.

2.4 Optimal Strategy

The time complexity of Algorithm 1 is $O(e_\ell^2)$. However, this strategy is far from optimal [19]. Inspired by the optimal strategy of computing isogenies [11], Zanon et al. [21] claimed that one can also adapt the optimal strategy into Pohlig-Hellman algorithm, reducing the time complexity to $O(e_\ell \log e_\ell)$ in the end.

Let $\triangle_n$ be a graph containing the vertices $\{\triangle_{j,k} | j + k \leq n - 1, j \geq 0, k \geq 0\}$, satisfying the following properties:

- Each vertex $\triangle_{j,k}(j + k < n - 1, j > 0, k > 0)$ has either two outgoing edges $\triangle_{j,k} \to \triangle_{j+1,k}$ and $\triangle_{j,k} \to \triangle_{j,k+1}$, or no edges at all;
- Each vertex $\triangle_{j,0}(0 < j < n-1)$ has only one outgoing edge $\triangle_{j,0} \to \triangle_{j+1,0}$, and $\triangle_{0,k}(0 < k < n - 1)$ has only one outgoing edge $\triangle_{0,k} \to \triangle_{0,k+1}$;
- Each vertex $\triangle_{j,k}(j + k = n - 1)$ has no edges, called leaves; We also call the vertex $\triangle_{0,0}$ the root.

A subgraph is called a strategy if it contains a given vertex $\triangle_{j,k}$ such that all leaves and vertices can be reached from $\triangle_{j,k}$. A strategy $\triangle'_n$ of $\triangle_n$ is full if it contains the root $\triangle_{0,0}$ and all leaves $\triangle_{j,k}(j + k = n - 1)$. Assigning the weights $p, q > 0$ to the left edges and the right edges, respectively, [4] we can define the cost of an optimal strategy $\triangle'_n$ by

$$C_{p,q}(n) = \begin{cases} 0, \text{if } n = 1, \\ \min\left\{C_{p,q}(i) + C_{p,q}(n-i) + (n-i)p + iq \mid 0 \leq i \leq n\right\}, \text{if } n > 1. \end{cases} \tag{5}$$

By utilizing Equation (5), the optimal strategy could be attained by [21, Algorithm 6.2].

2.5 Signed-digit Representation

Hutchinson et al. [10] reduced the memory for computing discrete logarithms by utilizing signed-digit representation [2]. Here we only introduce the situation

[4] In this case, they are the cost of raising an element in μ_{p+1} to ℓ^w-power and one multiplication in $\mathbb{F}_{p^2}$, respectively.

when w divides e_ℓ, while the other situation when w does not divide e_ℓ one needs to store an additional table, but the handling is similar.

Instead of limiting $x = \log_g h \in \{0, 1, \cdots, \ell^{e_\ell} - 1\}$, we represent it by

$$x = \sum_{k=0}^{e_\ell/w-1} D'_k L^k,$$

where $L = \ell^w$ and $D'_k \in [-\frac{\lceil L-1 \rceil}{2}, \frac{\lceil L-1 \rceil}{2}]$. It seems that in this case we need to store

$$T_1^{sgn}[i][j] = g^{jL^i}, i = 0, 1, \cdots, \frac{e_\ell}{w} - 1, j \in [-\lceil \frac{L-1}{2} \rceil, \lceil \frac{L-1}{2} \rceil].$$

Since for any element $a + bi \in \mu_{p+1}$ $(a, b \in \mathbb{F}_p)$ and $p \equiv 3 \mod 4$,

$$(a + bi)^{p+1} = 1 = (a + bi)(a + bi)^p = (a + bi)(a^p + b^p i^p) = (a + bi)(a - bi).$$

Hence, one inversion of an arbitrary element in μ_{p+1} is equal to its conjugate. This property guarantees one can reduce the table size by a factor of 2, i.e.,

$$T_1^{sgn}[i][j] = g^{jL^i}, i = 0, 1, \cdots, \frac{e_\ell}{w} - 1, j \in [1, \lceil \frac{L-1}{2} \rceil].$$

Remark 1. All the values in Column 0, i.e., $T_1^{sgn}[i][0]$, are equal to $g^0 = 1$. This is the reason why we do not need to precompute and store them.

In fact Hutchinson et al. took advantages of torus-based representation of cyclotomic subgroup elements to further reduce the table size by a factor of 2. Since this technique is difficult to be utilized into this work, we do not review here and refer the interested reader to [10] for more details.

2.6 Section Summary

The implementation of computing discrete logarithms in public-key compression of SIDH/SIKE has been optimized in recent years. However, it is still one of the main bottlenecks of key compression.

To summarize, we propose Algorithm 2 to compute discrete logarithms by utilizing the techniques mentioned above.

3 Computing Discrete Logarithms Without Precomputed Tables

As mentioned in Section 2.2, one needs to compute four discrete logarithms in the multiplicative group $\langle r_0 \rangle$ during public-key compression. Since r_0 is fixed, the techniques mentioned above are put to good use. In this section, we present another method to compute discrete logarithms, offering a time-memory trade-off as well.

Algorithm 2 Traverse(r, j, k, z, S, T_1^{sgn}, L, D): Improved Pohlig-Hellman Algorithm [21] [10]

Ensure: h: value of root vertex $\triangle_{j,k}$ (i.e., challenge); j, k: coordinates of root vertex $\triangle_{j,k}$; z: number of leaves in subtree rooted at vertex $\triangle_{j,k}$; S: optimal strategy; T_1^{sgn}: lookup table; L: ℓ^w.
Require: D: Array such that $h = g^{\left(D[\frac{e_\ell}{w}-1]\cdots D[1]D[0]\right)_L}$.

```
if z > 1 then
    t ← S[z];
    h' ← h^{L^{z-t}};
    Traverse(h', j + (z − t), k, t, S, T_1^{sgn}, L, D);
    h' ← h · ∏_{l=k}^{k+t−1} (T_1^{sgn}[j + l][|D[k]| − 1])^{−sign(D[k])};
    Traverse(h', j, k + t, z − t, S, T_1^{sgn}, L, D);
else
    if h = 1 then
        D[k] ← 0.
    else
        find x_k ∈ {0, ··· , ⌊(ℓ^w − 1)/2⌋} such that h = T_1^{sgn}[e_ℓ/w − 1][x_k + 1] or h = overline(T_1^{sgn}[e_ℓ/w − 1][x_k + 1]);
        if h = T_1^{sgn}[e_ℓ/w − 1][x_k + 1] then
            D[k] ← x_k + 1;
        else
            D[k] ← −x_k − 1;
        end if
    end if
end if
return D.
```

3.1 Three Discrete Logarithms

Note that the main purpose of computing discrete logarithms is to compute three values $(-d_1^{-1}d_0, -d_1^{-1}c_1, d_1^{-1}c_0)$ (or $(-d_0^{-1}d_1, d_0^{-1}c_1, -d_0^{-1}c_0)$ when d_1 is not invertible in $\mathbb{Z}_{\ell^{e_\ell}}$). For simplicity, we assume that d_1 is invertible and aim to compute $(-d_1^{-1}d_0, -d_1^{-1}c_1, d_1^{-1}c_0)$.

Since d_1 is invertible in $\mathbb{Z}_{\ell^{e_\ell}}$, we can deduce that $r_2 = r_0^{d_1}$ is a generator of the multiplicative group $\langle r_0 \rangle$. Hence, instead of computing four discrete logarithms of r_1, r_2, r_3, r_4 to the base r_0 (defined in Equation (3)), we consider three discrete logarithms of r_1, r_3, r_4 to the base r_2. It is clear that

$$
\begin{aligned}
r_1 &= r_0^{d_0} = r_0^{d_1 \cdot d_1^{-1} \cdot d_0} = r_2^{d_1^{-1} d_0}, \\
r_3 &= r_0^{c_0} = r_0^{d_1 \cdot d_1^{-1} \cdot c_0} = r_2^{d_1^{-1} c_0}, \\
r_4 &= r_0^{c_1} = r_0^{d_1 \cdot d_1^{-1} \cdot c_1} = r_2^{d_1^{-1} c_1}.
\end{aligned}
$$

In other words, we only need to compute three discrete logarithms to compress the public key. Since it is unnecessary to compute d_1^{-1} and multiply it by d_0, c_0 and c_1, we also save one inversion and three multiplications in $\mathbb{Z}_{\ell^{e_\ell}}$.

Unfortunately, computing discrete logarithms to the base r_0 when lookup tables are available are much more efficient than computing discrete logarithms to the base r_2. Furthermore, it is impossible to precompute values to improve the performance due to the fact that the base r_2 depends on d_1. Hence, compared to the previous work in the case where $w|e_\ell$, one needs to efficiently construct the lookup table

$$T_1^{sgn}[i][j] = (r_2)^{(j+1)L^i}, i = 0, 1, \cdots, \frac{e_\ell}{w} - 1, j = 0, 1, \cdots, \lceil \frac{L-1}{2} \rceil - 1. \quad (6)$$

Zanon et al. handled the situation when $w \nmid e_\ell$ to precompute an extra lookup table, as described in Equation (4). Inspired by the method proposed by Pereira et al. when handling ECDLP [15, Section 4.4], we present a similar approach for computing discrete logarithms when $w \nmid e_\ell$. That is, instead of discrete logarithms of r_1, r_3, r_4 to the base r_2, we compute discrete logarithms of $(r_1)^{\ell^m}$, $(r_3)^{\ell^m}$, $(r_4)^{\ell^m}$ to the base r_2, where $m = e_\ell \bmod w$. Correspondingly, the lookup table should be modified by the following:

$$T_1^{sgn}[i][j] = r_2^{(j+1)L^i + \ell^m}, i = 0, 1, \cdots, \lfloor \frac{e_\ell}{w} \rfloor - 1, j = 0, 1, \cdots, \lceil \frac{L-1}{2} \rceil - 1.$$

In this situation, we recover the values $d_1^{-1}d_0(\bmod\ \ell^{e_\ell - m})$, $d_1^{-1}c_0(\bmod\ \ell^{e_\ell - m})$ and $d_1^{-1}c_1(\bmod\ \ell^{e_\ell - m})$. Afterwards, we compute the three values as follows:

$$\begin{aligned} r_1 \cdot (r_2)^{-d_1^{-1}d_0 \bmod \ell^{e_\ell - m}} &= (r_2)^{d_1^{-1}d_0 - \left(d_1^{-1}d_0 \bmod \ell^{e_\ell - m}\right)}, \\ r_3 \cdot (r_2)^{-d_1^{-1}c_0 \bmod \ell^{e_\ell - m}} &= (r_2)^{d_1^{-1}c_0 - \left(d_1^{-1}c_0 \bmod \ell^{e_\ell - m}\right)}, \\ r_4 \cdot (r_2)^{-d_1^{-1}c_1 \bmod \ell^{e_\ell - m}} &= (r_2)^{d_1^{-1}c_1 - \left(d_1^{-1}c_1 \bmod \ell^{e_\ell - m}\right)}. \end{aligned} \quad (7)$$

Finally, we compute three discrete logarithms of the above values to the base $(r_2)^{\ell^{e_\ell - m}}$ to recover the full digits of three values $-d_1^{-1}d_0$, $-d_1^{-1}c_1$ and $d_1^{-1}c_0$.

Since $\langle (r_2)^{\ell^{e_\ell - m}} \rangle$ is a multiplicative subgroup of $\langle (r_2)^{\ell^{e_\ell - w}} \rangle$, we can regard the last three discrete logarithms as the discrete logarithms to the base $(r_2)^{\ell^{e_\ell - w}}$, which are computed efficiently with the help of the lookup table. However, the computation in Equation (7) is not an easy task. Therefore, except the construction of the lookup table, we also take into account how to obtain the three values mentioned in Equation (7) with high efficiency when $w \nmid e_\ell$.

3.2 Base Choosing

Before constructing the lookup table, it is necessary to check whether r_2 is a generator of the multiplicative group $\langle r_0 \rangle$. If not, we choose r_1 to be the base of discrete logarithms and construct the corresponding lookup table.

Note that in this case, d_1 is unknown. So we can not determine the order of r_2 by computing the greatest common divisor of d_1 and ℓ^{e_ℓ}. Instead, we compute

$(r_2)^{\ell^{e_\ell - 1}}$ to check whether it is equal to 1. For any element $\delta = u + vi \in \mu_{p+1}$, we have

$$\begin{aligned}
\delta^2 =& (u + vi)^2 \\
=& u^2 - v^2 + 2uvi \\
=& u^2 - v^2 + (1 - u^2 - v^2)i, \\
\delta^3 =& (u + vi)^3 \\
=& u^3 + 3u^2 vi - 3uv^2 - v^3 i \\
=& u^3 + 3u^2 \cdot vi - 3u(1 - u^2) - (1 - u^2) \cdot vi \\
=& -3u + 4u^2 \cdot u + (4u^2 - 1) \cdot vi.
\end{aligned} \tag{8}$$

Hence, we can efficiently compute $(r_2)^{\ell^{e_\ell - 1}}$ by squaring or cubing $e_\ell - 1$ times with respect to ℓ and check whether it is equal to 1. Another advantage is that we also compute the values in the first column of the lookup table when r_2 is a generator of $\langle r_0 \rangle$. Furthermore, when r_2 is a generator, the intermediate values

$$C[i] = (r_2)^{\ell^i}, i = 0, 1, \cdots, e_\ell - m, \tag{9}$$

could be utilized to speed up the performance when w does not divide e_ℓ. When r_1 is a generator, one can also construct the array

$$C[i] = (r_1)^{\ell^i}, i = 0, 1, \cdots, e_\ell - m,$$

with a few additional square or cube operations. We will explain the reason why we also require these values in Section 3.4.

We present Algorithm 3 for determining the base of discrete logarithms and computing the values in the first column of the lookup table. We also output the intermediate values that are used to improve the performance of discrete logarithms when $w \nmid e_\ell$.

Algorithm 3 choose_base(ℓ, e_ℓ, w, r_1, r_2)

Ensure: w : base power; r_1, r_2: elements defined in Equation (3); *label*: sign bit used to mark the choice of the generator.

Require: A: values in the first column of the lookup table; C: intermediate values used to improve the performance of discrete logarithms when $w \nmid e_\ell$.

```
label ← 1, A[0] ← r_2, C[0] ← r_2, j ← 0;
for i from 0 to (e_ℓ mod w) − 1 do
    A[0] ← (A[0])^ℓ, j ← j + 1, C[j] ← A[0];
end for
for i from 1 to ⌊e_ℓ / w⌋ − 1 do
    A[i] ← A[i − 1];
    for k from 0 to w − 1 do
        A[i] ← (A[i])^ℓ, j ← j + 1, C[j] ← A[i];
        if A[i] = 1 then
            label ← 0, break.
```

end if
end for
end for
if $label = 1$ **then**
$t \leftarrow A[\lfloor \frac{e_\ell}{w} \rfloor - 1]$;
for i from 0 to $w-2$ **do**
$t \leftarrow t^\ell$, $j \leftarrow j+1$, $C[j] \leftarrow t$;
if $t = 1$ **then**
$label \leftarrow 0$, **break**.
end if
end for
end if
if $label = 0$ **then**
$A[0] \leftarrow r_1$, $C[0] \leftarrow r_1$, $j \leftarrow 0$;
for i from 0 to $(e_\ell \mod w) - 1$ **do**
$A[0] \leftarrow (A[0])^\ell$, $j \leftarrow j+1$, $C[j] \leftarrow A[0]$;
end for
for i from 1 to $\lfloor \frac{e_\ell}{w} \rfloor - 1$ **do**
$A[i] \leftarrow A[i-1]$;
for k from 0 to $w-1$ **do**
$A[i] \leftarrow (A[i])^\ell$, $j \leftarrow j+1$, $C[j] \leftarrow A[i]$;
end for
end for
$t \leftarrow A[\lfloor \frac{e_\ell}{w} \rfloor - 1]$;
for i from 0 to $w-m-1$ **do**
$t \leftarrow t^\ell$, $j \leftarrow j+1$, $C[j] \leftarrow t$;
end for
end if
return $label, A, C$.

3.3 Lookup Table Construction

Algorithm 3 outputs the values in the first column of the lookup table. As we can see in Equation (6), all the values in the lookup table are the small powers of the values in the corresponding row. More precisely,

$$T_1^{sgn}[i][j] = (T_1^{sgn}[i][1])^{j+1}, i = 0, 1, \cdots, \frac{e_\ell}{w} - 1, j = 1, 2, \cdots, \lceil \frac{L-1}{2} \rceil - 1.$$

Therefore, one can raise the powers of the values in the first column to generate all the values in the lookup table. As mentioned in Equation (8), the costs of squaring and cubing in the multiplicative group μ_{p+1} are approximately $2\mathbf{s} \approx 1.6\mathbf{m}$ and $1\mathbf{s} + 2\mathbf{m} \approx 2.8\mathbf{m}$, respectively. Both of them are more efficient than operating one multiplication in $\mathbb{F}_{p^2}$, which costs approximately $3\mathbf{m}$. Note that all the values are in the group μ_{p+1}. One can utilize squaring and cubing operations, as we summarized in Algorithm 4.

Algorithm 4 T_DLP(ℓ, e_ℓ, w, A)

Ensure: w : base power; A: values in the first column of the lookup table T_1^{sgn}.
Require: T_1^{sgn}: entire lookup table.

for i from 0 to $\lfloor \frac{e_\ell}{w} \rfloor - 1$ **do**
  $T_1^{sgn}[i][0] \leftarrow A[i]$;
end for
for i from 0 to $\lfloor \frac{e_\ell}{w} \rfloor - 1$ **do**
  for j from 1 to $\lfloor \frac{\ell^w - 1}{2} \rfloor$ **do**
    if $j \bmod 2 = 1$ **then**
      $T_1^{sgn}[i][j] \leftarrow \left(T_1^{sgn}[i][\frac{j-1}{2}]\right)^2$;
    else
      if $j \bmod 3 = 2$ **then**
        $T_1^{sgn}[i][j] \leftarrow \left(T_1^{sgn}[i][\frac{j-2}{3}]\right)^3$;
      else
        $T_1^{sgn}[i][j] \leftarrow \left(T_1^{sgn}[i][\frac{j-1}{2}]\right) \cdot T_1^{sgn}[i][0]$;
      end if
    end if
  end for
end for
return T_1^{sgn}.

The bigger the base power w, the larger the size of the lookup table T_1^{sgn}, i.e., the higher the computational cost of lookup table construction, but the less discrete logarithms to be computed. Hence, just like efficiency-memory trade-offs provided by the previous work, we also explore the optimal base power w to minimize the whole computational cost. We leave this exploration in Section 4.

3.4 Discrete Logarithm Computation

For ease of exposition, in this subsection we assume that we have chosen r_2 as the base of discrete logarithms. By utilizing Pohlig-Hellman algorithm, three discrete logarithms to the base r_2 could be simplified into discrete logarithms to the base $(r_2)^{\ell^{e_\ell - w}}$ or $(r_2)^{\ell^{e_\ell - m}}$. Indeed, the discrete logarithms to the base $(r_2)^{\ell^{e_\ell - m}}$ can also be regarded as discrete logarithms to the base $(r_2)^{\ell^{e_\ell - w}}$ since $(r_2)^{\ell^{e_\ell - m}}$ is an element in the multiplicative group $\langle (r_2)^{\ell^{e_\ell - w}} \rangle$. Thus, we consider how to compute discrete logarithms to the base $(r_2)^{\ell^{e_\ell - w}}$ first.

Note that all the entries in the last row of the lookup table T_1^{sgn} are of the form

$$T_1^{sgn}[\lfloor \frac{e_\ell}{w} \rfloor - 1][j] = (r_2)^{(j+1)\ell^{e_\ell - w}}, j = 0, 1, \cdots, \lceil \frac{L-1}{2} \rceil - 1.$$

Thanks to signed-digit representation, all the entries in the last row of the lookup table and their conjugates consist of all nontrivial elements in the multiplicative group $\langle (r_2)^{\ell^{e_\ell - w}} \rangle$. Therefore, computing discrete logarithms to the base $(r_2)^{\ell^{e_\ell - w}}$ is relatively easy with the help of $T_1^{sgn}[\lfloor \frac{e_\ell}{w} \rfloor - 1][j], j = 0, 1, \cdots, \lceil \frac{L-1}{2} \rceil - 1$.

Algorithm 5 small_DLP(ℓ, w, h, B')

Ensure: w: base power; h: challenge; B': last row of the lookup table T_1^{sgn};
Require: x, sgn: integers such that $h = (B'[0])^{sgn \cdot x}$.

if $h = 1$ **then**
  $x \leftarrow 0$, $sgn \leftarrow 1$;
else
  find $x \in \{0, \cdots, \lfloor \frac{L-1}{2} \rfloor\}$ such that $h = B'[x]$ or $h = \overline{B'[x]}$;
  if $h = B'[x]$ **then**
    $x \leftarrow -x - 1$;
  else
    $x \leftarrow x + 1$;
  end if
end if
return x, sgn.

Remark 2. When handling discrete logarithms to the base $(r_2)^{\ell^{e_\ell - m}}$, the output of Algorithm 5 is ℓ^{w-m} times of the correct answer. Therefore, we should modify the output by dividing it by ℓ^{w-m}.

As we have pointed out in Section 3.1, when the base power w does not divide e_ℓ, one efficiency issue to be solved is how to compute the values in Equation (7). We propose a method to deal with this issue by utilizing the intermediate values C from Algorithm 3.

In Algorithm 3, we repeat squaring or cubing operations and store the intermediate values, as described in Equation (9). On the other hand, after computing discrete logarithms of $(r_1)^{\ell^m}$, $(r_3)^{\ell^m}$, $(r_4)^{\ell^m}$ to the base r_2, we recover $d_1^{-1}d_0 (\bmod \ell^{e_\ell - m})$, $d_1^{-1}c_0 (\bmod \ell^{e_\ell - m})$ and $d_1^{-1}c_1 (\bmod \ell^{e_\ell - m})$. Therefore, similar to the Double-and-Add algorithm, one can compute $r_2^{-d_1^{-1}d_0 \bmod \ell^{e_\ell - m}}$ (the other two are similar) according to the binary/ternary expansion of the value $-d_1^{-1}d_0 (\bmod \ell^{e_\ell - m})$, with respect to ℓ. Afterwards, it just needs to perform three multiplications in $\mathbb{F}_{p^2}$ to obtain all the values.

In Algorithm 6, we present pseudocode for computing $r_2^{-d_1^{-1}d_0 \bmod \ell^{e_\ell - m}}$, $r_2^{-d_1^{-1}c_0 \bmod \ell^{e_\ell - m}}$ and $r_2^{-d_1^{-1}c_1 \bmod \ell^{e_\ell - m}}$. Note that squaring in μ_{p+1} can also benefit from Equation (8). Hence, for Line 18 of Algorithm 6, it would be efficient if we square $C[i]$ (or its conjugate $\overline{C[i]}$) first, and then perform a multiplication.

Algorithm 6 fast_power(ℓ, w, D, C)

Ensure: D: array in base $L = \ell^w$ with signed digits; C: array from Algorithm 3;
Require: h: $(C[0])^{\left(D[\lfloor \frac{e_\ell}{w} \rfloor - 2] \cdots D[1]D[0]\right)_L}$.

1: $h \leftarrow 1$, $i_1 \leftarrow 0$, $i_2 \leftarrow 0$;
2: **for** i from 0 to $\lfloor \frac{e_\ell}{w} \rfloor - 2$ **do**
3:   $t \leftarrow D[i]$, $s \leftarrow 1$;
4:   **if** $D[i] < 0$ **then**
5:     $t \leftarrow -t$, $s \leftarrow -1$;

```
    end if
    while t > 0 do
        if ℓ = 2 then
            if t mod 2 = 1 then
                h ← h · (C[i_2])^s;
            end if
            i_2 ← i_2 + 1, t ← ⌊t/2⌋;
        else
            if t mod 3 = 1 then
                h ← h · (C[i_2])^s;
            end if
            if t mod 3 = 2 then
                h ← (C[i_2])^{2s} · h;
            end if
            i_2 ← i_2 + 1, t ← ⌊t/3⌋;
        end if
    end while
    i_1 ← i_1 + w, i_2 ← i_1;
end for
return h.
```

It remains how to compute discrete logarithms of r_1, r_3 and r_4 to the base r_2 efficiently. Cervantes-Vázquez et al. proposed a non-recursive algorithm to compute ℓ^{e_ℓ}-isogeny [5]. Inspired by their work, we present Algorithm 7 to compute discrete logarithms. Now we describe how Algorithm 7 works in detail.

Notations: The input h is the challenge of discrete logarithms, i.e, r_1, r_3 or r_4. The vector Str is the linear representation of the optimal strategy. In the algorithm, we construct a stack, denoted by $Stack$, which contains the tuples of the form (h_t, e_t, l_t), where $h_t \in \mu_{p+1}$ and $e_t, l_t \in \mathbb{N}$. Each tuple in $Stack$ represents the vertex which has been passed through (in left-first order), with the value h_t, the order $\ell^{e_\ell - e_t - m}$ and a right outgoing edge. When pushing a tuple into $Stack$, we also record the label $Str[i]$ of the previous vertex, denoted by l_t. The integers (j, k) are coordinates of the last vertex which has been passed through. The other notations, such as the lookup table T_1^{sgn}, are defined as above.

Lines 3-6: As we described in Section 3.1, we compute discrete logarithms of $(h)^{\ell^m}$ to the base r_2 when $w \nmid e_\ell$. So we first compute $(h)^{\ell^m}$ when $m \neq 0$. Afterwards, we push $((h)^{\ell^m}, 0, 0)$ into $Stack$.

Lines 7-33: This part is the core of Algorithm 7. The main idea is to traverse the optimal strategy according to a left-first ordering and construct a stack to store all the vertices that have right outgoing edges. Once a discrete logarithm is computed, all the vertices in $Stack$ are replaced by their right vertices, respectively.

Line 7 checks if k is equal to $\lfloor \frac{e_\ell}{w} \rfloor - 1$, i.e, the rightmost vertex $\triangle_{0, \lfloor \frac{e_\ell}{w} \rfloor - 1}$ has been traversed. In this case we jump out of the loop.

Line 8 aims to check whether the last vertex that has been passed through is a leaf or not. When the vertex is not a leaf, we go the left $Str[i]$ edges to enter the next split vertex and then **push** the information of this vertex into *Stack* until the vertex is a leaf (Lines 10-13). When the vertex is a leaf, there are no edges to traverse left or right, and the values of the vertex is an element of order ℓ^w in the multiplicative group $\mu_{\ell^{e_\ell}}$. Hence, we **pop** the tuple from *Stack* and then execute the algorithm **small_DLP** in Lines 16-17. Then we store the result into the array D in Lines 18-22.

Note that in this case, there are no left edges to be traversed. But all the right edges of the vertices in *Stack* can be traversed since we have recovered $D[k]$. For each tuple (h_t, e_t, l_t) in *Stack*, we execute

$$h_t \leftarrow h_t \cdot \overline{T_1^{sgn}[e_t][x_t - 1]} \text{ or } h_t \leftarrow h_t \cdot T_1^{sgn}[e_t][x_t - 1],$$

with respect to sgn_t (Lines 23-31).

The rest is to modify the position of the last vertex, as described in Line 32.

Lines 34-40: Now we have passed through the whole optimal strategy and in this case *Stack* remains one tuple, i.e., it remains the vertex $\triangle_{0,\lfloor \frac{e_\ell}{w} \rfloor - 1}$ that needed to be handled. Therefore, we **pop** the tuple from *Stack* and execute the algorithm **small_DLP** again. Finally, we store the answer into $D[k]$ (Note that $k = \lfloor \frac{e_\ell}{w} \rfloor - 1$).

Lines 41-50: Line 41 checks whether the base power w divides e_ℓ. When w divides e_ℓ, we are done. If not, we need to compute the values in Equation (7) and an extra discrete logarithm to the base $r_2^{\ell^{e_\ell - m}}$. Hence, when $m \neq 0$, we execute the algorithm **fast_power** to compute $(r_2)^{\left(D[\lfloor \frac{e_\ell}{w} \rfloor - 2] \cdots D[1] D[0]\right)_L}$ with the help of the array C and the efficiency of squaring and cubing in μ_{p+1}. After that, we perform a multiplication in $\mathbb{F}_{p^2}$ and finally execute the algorithm **small_DLP**. As we mentioned in Remark 2, the output of **small_DLP** is ℓ^{w-m} times of the correct answer. Therefore, we divide ℓ^{w-m} into the output.

Line 51: Return the array D.

Now we give a toy example to show how Algorithm 7 computes the discrete logarithm h to the base g. For simplicity, we assume that $m = 0$, and there are three leaves in the strategy $Str = (1, 1)$, as illustrated in Figure (a). We first **push** the tuple $(h, 0, 0)$ into *Stack*. Now Lines 7-8 check that the vertex $\triangle_{0,0}$ is not a leaf, and therefore we are able to traverse left by squaring or cubing w times and then **push** the tuple $(h^{\ell^w}, 1, 1)$ into *Stack*, as described in Lines 10-13. Again, Line 8 checks that $\triangle_{0,1}$ is not a leaf as well, so we continue traversing left and **push** the tuple $(h^{\ell^w}, 2, 1)$ into *Stack* (Figure (c)). Note that $\triangle_{2,0}$ is a leaf of order ℓ^w. We **pop** the tuple and then execute the algorithm **small_DLP** to compute the discrete logarithm, and then we recover $D[0]$. Afterwards, Lines 23-31 handle all the vertices in *Stack* by performing two multiplications in $\mathbb{F}_{p^2}$, as shown in Figures (d) and (e). In this case, we check that $\triangle_{1,1}$ is a leaf, so we **pop** the top tuple from *Stack* and then execute **small_DLP** again to recover $D[1]$. We traverse right from $\triangle_{0,1}$ to enter the rightmost vertex with the help of

$D[1]$ (Figure (f)). Finally, Lines 34-40 **pop** the tuple and execute **small_DLP** once again to recover $D[2]$.

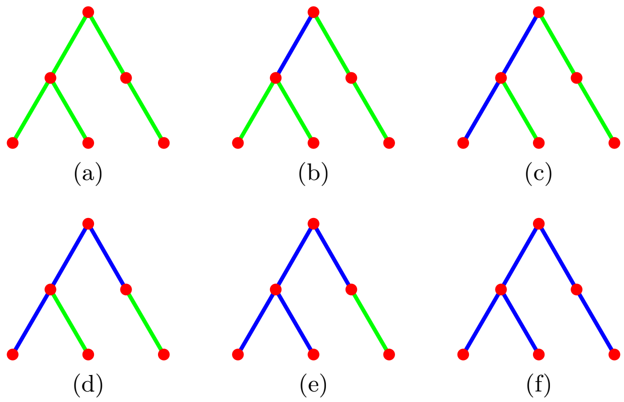

Fig. 1: A toy example of Algorithm 7

Algorithm 7 PH_DLP(ℓ, e_ℓ, w, h, Str, T_1^{sgn}, C)

Ensure: w: base power; h: challenge; Str: Optimal strategy; T_1^{sgn}: entire lookup table, C: Array from Algorithm 3;

Require: D: Array such that $h = g^{\left(D[\lfloor \frac{e_\ell}{w} \rfloor -1]\cdots D[1]D[0]\right)_{\ell^w}}$.

1: initialize a Stack $Stack$, which contains tuples of the form (h_t, e_t, l_t), where $h_t \in \mu_{p+1}$, $e_t, l_t \in \mathbb{N}$.
2: $B' \leftarrow$ last row of the lookup table T_1^{sgn}, $i \leftarrow 0$, $j \leftarrow 0$, $k \leftarrow 0$, $m \leftarrow e_\ell \bmod w$, $h_t \leftarrow h$;
3: **for** i_1 from 0 to $m-1$ **do**
4:     $h_t \leftarrow (h_t)^\ell$;
5: **end for**
6: **Push** the tuple (h_t, j, k) into $Stack$;
7: **while** $k \neq \lfloor \frac{e_\ell}{w} \rfloor - 1$ **do**
8:     **while** $j + k \neq \lfloor \frac{e_\ell}{w} \rfloor - 1$ **do**
9:         $j \leftarrow j + Str[i]$;
10:         **for** i_2 from 0 to $w \cdot Str[i] - 1$ **do**
11:             $h_t \leftarrow (h_t)^\ell$;
12:             **Push** the tuple $(h_t, j + k, Str[i])$ into $Stack$;
13:         **end for**
14:         $i \leftarrow i + 1$;
15:     **end while**
16:     **Pop** the top tuple (h_t, e_t, l_t) from $Stack$;

$(x_t, sgn_t) \leftarrow$ **small_DLP**$(\ell,\ w,\ h_t,\ B')$;
  if $sgn_t = 1$ **then**
    $D[k] \leftarrow x_t + 1$;
  else
    $D[k] \leftarrow -x_t - 1$;
  end if
  for each tuple (h_t, e_t, l_t) in *Stack* **do**
    if $x_t \neq 0$ **then**
      if $sgn_t = 1$ **then**
        $h_t \leftarrow h_t \cdot \overline{T_1^{sgn}[e_t][x_t - 1]}$;
      end if
    else
      $h_t \leftarrow h_t \cdot T_1^{sgn}[e_t][x_t - 1]$;
    end if
  end for
  $j \leftarrow j - l_t,\ k \leftarrow k + 1$;
end while
Pop the top tuple (h_t, e_t, l_t) from *Stack*;
$(x_t, sgn_t) \leftarrow$ **small_DLP**$(\ell,\ w,\ h_t,\ B')$;
if $sgn_t = 1$ **then**
  $D[k] \leftarrow x_t + 1$;
else
  $D[k] \leftarrow -x_t - 1$;
end if
if $m \neq 0$ **then**
  $h_t \leftarrow$ **fast_power**$(\ell,\ D,\ C)$;
  $h_t \leftarrow h \cdot \overline{h_t}$;
  $(x_t, sgn_t) \leftarrow$ **small_DLP**$(\ell,\ w,\ h_t,\ B')$;
  if $sgn_t = 1$ **then**
    $D[k+1] \leftarrow \frac{x_t + 1}{\ell^{w-m}}$;
  else
    $D[k+1] \leftarrow -\frac{x_t + 1}{\ell^{w-m}}$;
  end if
end if
return D.

4 Cost Estimates and Implementation Results

In this section, we estimate the computational cost of discrete logarithms and compare our work with the previous work. We also report the implementation of key generation of SIDH by utilizing our techniques.

4.1 Cost Estimates

We neglect additions and mainly take into account multiplications and squarings ($1\mathbf{s} \approx 0.8\mathbf{m}$) since they are much more expensive than additions. As shown in

Table 1, we predict that for all the Round-3 SIKE parameters, the cost of discrete logarithm computation in $\mu_{3^{e_3}}$ would be minimal when the base power w is equal to 3. When handling $\mu_{2^{e_2}}$, the base power $w = 4$ would be the best choice.

Table 1: Cost estimates of three discrete logarithms utilizing our techniques. The minimal costs in the same row, i.e, in the same setting except the base power, are reported in bold.

Setting		w=1	w=2	w=3	w=4	w=6
SIKEp434	$\mu_{3^{e_3}}$	8892.6	6904.3	**6463.3**	7603	21915
	$\mu_{2^{e_2}}$	11762.4	7516	6083.6	**5544.6**	6232.4
SIKEp503	$\mu_{3^{e_3}}$	10780.3	8223.7	**6859**	8869.8	21960
	$\mu_{2^{e_2}}$	13968.6	8902.2	8061.4	**7441.7**	8187.1
SIKEp610	$\mu_{3^{e_3}}$	13477.5	9237.5	**8552.2**	9990.5	30941.8
	$\mu_{2^{e_2}}$	17650.2	12327.2	9542.4	**9404.8**	10256.6
SIKEp751	$\mu_{3^{e_3}}$	17354.3	13265.9	**12326.8**	14076.5	39564.4
	$\mu_{2^{e_2}}$	22181.4	14334.4	11594	**10539**	11552

4.2 Implementation Results and Efficiency Comparisons

Based on the Microsoft SIDH library[1] (version 3.4), we compiled our codeby using an 11th Gen Intel(R) Core(TM) i7-1185G7 @ 3.00GHz on 64-bit Linux.

For each setting we execute 10^4 times and record the average cost of key generation, as summarized in Table 2. The implementation results show that our prediction in the previous subsection is correct.

Table 2: Implementation of key generation of compressed SIDH (expressed in millions of clock cycles). The minimal cost in the same row, i.e, in the same setting except the base power, are reported in bold.

Setting		w=1	w=2	w=3	w=4	w=6
SIKEp434	$\mu_{3^{e_3}}$	6.41	6.16	**6.07**	6.27	8.68
	$\mu_{2^{e_2}}$	6.35	6.00	5.96	**5.82**	5.97
SIKEp503	$\mu_{3^{e_3}}$	8.50	8.27	**7.93**	8.54	11.93
	$\mu_{2^{e_2}}$	8.51	8.25	8.10	**8.01**	8.12
SIKEp610	$\mu_{3^{e_3}}$	17.08	16.54	**16.52**	16.88	22.69
	$\mu_{2^{e_2}}$	16.31	15.89	15.65	**15.59**	15.71
SIKEp751	$\mu_{3^{e_3}}$	26.46	25.81	**25.69**	26.38	35.71
	$\mu_{2^{e_2}}$	27.26	26.36	25.84	**25.32**	26.09

[1] `https://github.com/Microsoft/PQCrypto-SIDH`

On memory-constrained devices, our algorithms would be attractive for their relatively efficient performance even though we set small w. Table 3 reports RAM requirements for the different parameters.

Table 3: RAM requirements (in KiB) for the different parameters.

Setting		w=1	w=2	w=3	w=4	w=6
SIKEp434	$\mu_{3^{e_3}}$	14.98	29.75	63.98	148.75	875.88
	$\mu_{2^{e_2}}$	23.63	23.63	31.50	47.25	126.00
SIKEp503	$\mu_{3^{e_3}}$	19.88	39.50	86.13	195.00	1183.00
	$\mu_{2^{e_2}}$	31.25	31.25	41.50	62.00	164.00
SIKEp610	$\mu_{3^{e_3}}$	30.00	60.00	130.00	300.00	1820.00
	$\mu_{2^{e_2}}$	47.66	47.50	63.13	95.00	250.00
SIKEp751	$\mu_{3^{e_3}}$	44.81	89.25	192.56	442.50	2661.75
	$\mu_{2^{e_2}}$	69.75	69.75	93.00	139.50	372.00

Table 4 shows the comparison of efficiency between the previous work with ours. We can see that the efficiency of our algorithms is close to that of the previous work. When solving discrete logarithms in $\mu_{2^{e_2}}$, our algorithms are more efficient than the previous work when we set SIKEp434 or SIKEp751 as parameters. In addition, when the base power w divides e_ℓ, our algorithms perform better because there is no need to compute three values in Equation 7 and execute three additive discrete logarithms.

Table 4: Key generation performance of the previous work and ours (expressed in millions of clock cycles). In the last column we report the ratio of the cost of the previous work to ours. In the same situation, we emphasize the lower cost in bold.

Setting		Previous work [3]	This work	$w \mid e_\ell$?	Ratio
SIKEp434	$\mu_{3^{e_3}}$	**5.96**	6.07	No	98.2%
	$\mu_{2^{e_2}}$	5.90	**5.82**	Yes	101.4%
SIKEp503	$\mu_{3^{e_3}}$	**8.07**	8.14	Yes	99.14%
	$\mu_{2^{e_2}}$	**7.93**	8.01	No	99.00%
SIKEp610	$\mu_{3^{e_3}}$	**16.34**	16.52	Yes	98.91%
	$\mu_{2^{e_2}}$	**15.25**	15.59	No	97.82%
SIKEp751	$\mu_{3^{e_3}}$	**25.20**	25.69	No	98.09%
	$\mu_{2^{e_2}}$	25.61	**25.32**	Yes	101.15%

5 Conclusion

In this paper, we presented new techniques to compute discrete logarithms in public-key compression of SIDH/SIKE with no pre-computed tables. We analyze cost estimates of discrete logarithm computation with our techniques, and predict the best choices of w in different situations. The implementation confirmed our deduction, and our algorithms to compute discrete logarithms in $\mu_{2^{e_2}}$ performed better in the situation when w divides e_2. We believe that this work would be also attractive in storage restrained environments, for the reason that we can make a trade-off between memory and efficiency.

Note that Algorithm 7 is a non-recursive algorithm. Hence, it would be more efficient in parallel environments. We leave those further explorations for future research.

Acknowledgments

The authors thank the anonymous reviewers for their useful and valuable comments. The work of Chang-An Zhao is partially supported by NSFC under Grant No. 61972428 and by the Major Program of Guangdong Basic and Applied Research under Grant No. 2019B030302008.

References

1. Adj, G., Cervantes-Vázquez, D., Chi-Domínguez, J.J., Menezes, A., Rodríguez-Henríquez, F.: On the Cost of Computing Isogenies Between Supersingular Elliptic Curves. In: Cid, C., Jacobson Jr., M.J. (eds.) Selected Areas in Cryptography – SAC 2018. pp. 322–343. Springer International Publishing, Cham (2019)
2. Avizienis, A.: Signed-digit Number Representations for Fast Parallel Arithmetic. IRE Transactions on Electronic Computers **EC-10**(3), 389–400 (1961). https://doi.org/10.1109/TEC.1961.5219227
3. 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`
4. 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)
5. Cervantes-Vázquez, D., Ochoa-Jiménez, E., Rodríguez-Henríquez, F.: Parallel strategies for SIDH: Towards computing SIDH twice as fast. Cryptology ePrint Archive, Report 2020/383 (2020), `https://ia.cr/2020/383`, accepted by IEEE Transactions on Computers
6. 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)

7. Costello, C., Longa, P., Naehrig, M.: Efficient Algorithms for Supersingular Isogeny Diffie-Hellman. In: Robshaw, M., Katz, J. (eds.) Advances in Cryptology – CRYPTO 2016. pp. 572–601. Springer Berlin Heidelberg, Berlin, Heidelberg (2016)
8. Faz-Hernández, A., López, J., Ochoa-Jiménez, E., Rodríguez-Henríquez, F.: A Faster Software Implementation of the Supersingular Isogeny Diffie-Hellman Key Exchange Protocol. IEEE Transactions on Computers **67**(11), 1622–1636 (2018)
9. Galbraith, S.D., Petit, C., Shani, B., Ti, Y.B.: On the Security of Supersingular Isogeny Cryptosystems. In: Cheon, J.H., Takagi, T. (eds.) Advances in Cryptology – ASIACRYPT 2016. pp. 63–91. Springer Berlin Heidelberg, Berlin, Heidelberg (2016)
10. 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)
11. 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)
12. Jaques, S., Schanck, J.M.: Quantum Cryptanalysis in the RAM Model: Claw-Finding Attacks on SIKE. In: Boldyreva, A., Micciancio, D. (eds.) Advances in Cryptology – CRYPTO 2019. pp. 32–61. Springer International Publishing, Cham (2019)
13. Lin, K., Lin, J., Wang, W., an Zhao, C.: Faster Public-key Compression of SIDH with Less Memory. Cryptology ePrint Archive, Report 2021/992 (2021), `https://ia.cr/2021/992`
14. 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)
15. 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)
16. 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)
17. Petit, C.: Faster Algorithms for Isogeny Problems Using Torsion Point Images. In: Takagi, T., Peyrin, T. (eds.) Advances in Cryptology – ASIACRYPT 2017. pp. 330–353. Springer International Publishing, Cham (2017)
18. Pohlig, S., Hellman, M.: An Improved Algorithm for Computing Logarithms over GF(p) and Its Cryptographic Significance (Corresp.). IEEE Trans. Inf. Theor. **24**(1), 106–110 (2006)
19. Shoup, V.: A computational introduction to number theory and algebra. Cambridge University Press (2005)
20. Zanon, G.H.M., Simplicio, M.A., Pereira, G.C.C.F., Doliskani, J., Barreto, P.S.L.M.: Faster Isogeny-Based Compressed Key Agreement. In: Lange, T., Steinwandt, R. (eds.) Post-Quantum Cryptography. pp. 248–268. Springer International Publishing, Cham (2018)
21. 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)