Memory Optimization Techniques for Computing Discrete Logarithms in Compressed SIKE

Aaron Hutchinson[1], Koray Karabina[1,2], and Geovandro Pereira[1,3]

[1] University of Waterloo, Waterloo, Canada
{a5hutchinson,koray.karabina,geovandro.pereira}@uwaterloo.ca
[2] National Research Council Canada
koray.karabina@nrc-cnrc.gc.ca
[3] evolutionQ Inc., Kitchener, Canada

Abstract. The supersingular isogeny-based key encapsulation (SIKE) suite stands as an attractive post-quantum cryptosystem with its relatively small public keys. Public key sizes in SIKE can further be compressed by computing pairings and solving discrete logarithms in certain subgroups of finite fields. This comes at a cost of precomputing and storing large discrete logarithm tables. In this paper, we propose several techniques to optimize memory requirements in computing discrete logarithms in SIKE, and achive to reduce table sizes by a factor of 4. We implement our techniques and verify our theoretical findings.

Keywords: SIKE · isogeny-based cryptography · public key compression · discrete logarithms.

1 Introduction

The supersingular isogeny-based key encapsulation suite (SIKE) stands as an attractive post-quantum cryptosystem with its relatively small public keys. SIKE [4], a post-quantum key-encapsulaion mechanism (KEM) candidate based on the original supersingular isogeny-based Diffie-Hellman key exchange (SIDH) [7], was submitted to the NIST standardization process on post-quantum cryptography, and has recently advanced as one of the eight alternate candidates in public-key encryption/KEMs.

Public key sizes in SIKE can further be compressed by computing pairings and solving discrete logarithms in certain subgroups of finite fields. This comes at a cost of precomputing and storing large discrete logarithm tables. The process involves evaluating pairings over elliptic curves, and using Pohlig-Hellman algorithm to solve discrete logarithms in order-ℓ^e multiplicative subgroups of $\mathbb{F}_{p^2}^*$; [5,6,8,10]. Computing discrete logarithms is one of the main key compression bottlenecks as shown in [6,8] and the SIKE submission to NIST Round 2 [3], which make use of large precomputed tables (megabytes in some cases) for speeding up such calculation. Alternatively, a recent approach is to instead of

computing pairings and computing discrete logarithms over $\mathbb{F}_{p^2}^*$, project a point P onto particular subgroups $\langle R_0 \rangle, \langle S_0 \rangle \in E_0(\mathbb{F}_p)[\ell^e]$ of a curve defined over $\mathbb{F}_p$ and solve the elliptic curve discrete logarithm directly via Pohlig-Hellman [11]. This approach can also be optimized by using possibly large precomputed tables, but in that case the technique seems to loose its advantage over the previously known techniques according to [11].

We should also note that a related work allowed for completely discarding precomputed tables during the *Decapsulation* operation by eliminating the need of computing discrete logarithms [12]. On the other hand, large tables still appear in both *KeyGen* and *Encapsulation* of SIKE, and a constrained-memory device running more than just decapsulation would still benefit from reducing table sizes. A great progress has been made towards improving computation time in compressed SIKE, but they all come with a non-trivial overhead of static memory due to precomputed tables. In this paper, we propose several techniques to optimize memory requirements in computing discrete logarithms in SIKE.

1.1 The Organization and Contributions:

The contributions of this paper can be summarized as follows.

- First, we exploit signed digit representation of exponents in radix-ℓ^w and show how to compress discrete logarithm tables by a factor of 2 [1].
- Second, we utilize torus-based representation of cyclotomic subgroup elements and compress tables by an additional factor 2. Unfortunately, this representation introduces extra $\mathbb{F}_p$ multiplications during the Pohlig-Hellman algorithm when computing small discrete logarithms at each leaf. We devise new algorithms to overcome this challenge and keep torus-based representations competitive (actually faster for the case where $\ell = 2$).
- Finally, we implement our techniques in C for the SIKE parameters as proposed to NIST. We verified that our factor-4 table reduction does not add computational time overhead for the case where $\ell = 2$, and add an overall 4-9% overhead to SIKE *KeyGen* for $\ell = 3$ as analyzed in Section 5. Our code signed-digits-based optimizations can be found on the NIST submission package of SIKE Round 3, and the torus-based optimizations were also made available online [2].

The rest of this paper is organized as follows. Section 2 serves as an overview of some well-known techniques used for computing discrete logarithms and motivates the problem of reducing table sizes. Section 3 shows how to compress discrete logarithm tables by a factor of 2. Section 4 utilizes torus-based representations to compress tables by another factor of 2. We present our implementation results in Section 5 and conclude in Section 6. All algorithms in this work can be found in Appendix B.

[1] This technique was originally introduced by the authors of this paper; implemented in SIKE round 3 [2]; and recently reused in [11].

[2] `https://github.com/microsoft/PQCrypto-SIDH/commit/e990bc6784c68426f69ac11ada3dd5fbfed8b714`

1.2 Notation

We let p denote a large prime number with $p \equiv 3 \pmod 4$, $\mathbb{F}_p$ denote a field with p elements, and $\mathbb{F}_{p^2} := \mathbb{F}_p[i]/\langle i^2+1\rangle$. For $a+bi \in \mathbb{F}_{p^2}$, we let $\overline{a+bi} = a-bi$ denote conjugation. We let $\mathbb{G}$ denote the cyclotomic subgroup of $\mathbb{F}_{p^2}^*$ of order $p+1$. We let ℓ denote a small prime number, often taken to be 2 or 3, which divides $p+1$, and we let e be the largest integer such that ℓ^e divides $p+1$. For k an integer such that ℓ^k divides $p+1$, we let $\mathbb{G}_{\ell,k}$ denote the (unique) cyclic subgroup of $\mathbb{G}$ of order ℓ^k. When ℓ and e are fixed, we let g be a fixed generator of $\mathbb{G}_{\ell,e}$ so that $\mathbb{G}_{\ell,e} = \langle g\rangle$. We let w be a small positive integer determining the size of various tables, and define $L = \ell^w$ and $m = \lceil e/w \rceil$. We let $\rho = g^{\ell^{e-w}}$ be the generator of $\mathbb{G}_{\ell,w}$. We use T to denote the table of $\mathbb{F}_{p^2}$ elements defined by $T[u][d] = g^{-d\cdot L^u}$ for $0 \le u \le m-1$ and $0 \le d < L$. Throughout the paper, additional tables T^{sgn}, CT, and T^{exp} will be defined as they are introduced. We let $\mathbf{m}$ and $\mathbf{s}$ denote the costs of multiplication and squaring in $\mathbb{F}_p$.

2 The Pohlig-Hellman Algorithm with width-w Windows

Given an instance $\mathbb{G}_{\ell,e}$, g, and $h \in \mathbb{G}$ of a discrete logarithm problem (DLP) in $\mathbb{G}_{\ell,e}$, *Pohlig-Hellman algorithm* (PH) [13] uses width-w windows and computes $d = \log_g h$ as follows. Let w be a positive integer with $w \mid e$ [3], define $m = \lceil e/w \rceil$, and write the exponent d in base $L = \ell^w$ as $d = \sum_{i=0}^{m-1} D_i L^i$ with $D_i \in [0, L)$. Define the sequence $h_k = g^{\sum_{i=k}^{m-1} D_i L^i}$, which satisfies

$$h_0 = h, \quad h_{k+1} = h_k g^{-D_k L^k}, \quad h_k^{L^{m-1-k}} = \left(g^{L^{m-1}}\right)^{D_k} \tag{1}$$

for $k = 0, \ldots, m-2$. Note that $D_k = \log_\rho h_k^{L^{m-1-k}} \in [0, L)$, where $\rho = g^{L^{m-1}}$ generates a group $\mathbb{G}_{\ell,w}$ of order ℓ^w. In PH, a table T of elements are precomputed and stored such that $T[k][D] = g^{-DL^k}$, for $0 \le k \le m-1$ and $0 \le D < L$. Note that the table T consists of $m \cdot L$ elements of $\mathbb{G}_{\ell,e}$. One computes $h_0^{L^{m-1}}$, and determines D_0 by looking up the last row $T[m-1]$ of T. For $k = 1, \ldots, m-1$, first $h_k = h_{k-1}T[k-1][D_{k-1}]$ is computed (one multiplication), and then $h_k^{L^{m-1-k}}$ is computed ($(m-1-k)$ exponentiations by L), and D_k is determined by looking up the row $T[m-1]$. Once $D_0, \ldots, D_{m-1}$ are known, $d = \sum_{i=0}^{m-1} D_i L^i$ can be recovered.

As previously observed in the literature [8,15], the steps of the above PH algorithm can be associated with subgraph of a directed graph $\mathcal{T}_{e,w}$. The vertices of $\mathcal{T}_{e,w}$ are labeled as $\Delta_{j,k}$ for $0 \le k \le m-1$ and $0 \le j \le m-1-k$, with $\Delta_{0,0}$ being the top-most vertex and $\Delta_{j,k}$ being the vertex lying at the end of the path starting at $\Delta_{0,0}$ and following j many left edges and k many right edges. The vertices $\Delta_{m-1-k,k}$ for $0 \le k \le m-1$ are referred to as *leaves*, and $\Delta_{0,0}$ is the *root*. We make a correspondence between vertices $\Delta_{j,k}$ of $\mathcal{T}_{e,w}$ and elements

[3] The case when $w \nmid e$ requires more attention as explained in [15].

of $\mathbb{G}_{\ell,e}$ by associating $\Delta_{j,k}$ with $h_k^{L^j}$, and it will be convenient to use "=" for this association: $\Delta_{j,k} = h_k^{L^j}$, for $0 \leq k \leq m-1$ and $0 \leq j \leq m-1-k$ In particular, the root $\Delta_{0,0}$ is associated with the input $h_0 = h$ to the DLP, and the leaves $\Delta_{m-1-k,k}$ correspond to the elements $h_k^{L^{m-1-k}}$ used to determine D_k. The outgoing edges of non-leaf vertices $\Delta_{j,k}$ can then be interpreted as group operations in $\mathbb{G}_{\ell,e}$ as follows:

$$\textit{left traversal: } \Delta_{j,k} \to \Delta_{j+1,k} = \Delta_{j,k}^{L};$$
$$\textit{right traversal: } \Delta_{j,k} \to \Delta_{j,k+1} = \Delta_{j,k} \cdot g^{-D_k L^{j+k}}.$$

Note that an edge with a positive slope in $\mathcal{T}_{e,w}$ (a *left traversal*) corresponds to exponentiation by L, and an edge with a negative slope in $\mathcal{T}_{e,w}$ (a *right traversal*) corresponds to a multiplication by a group element, assuming access to the lookup table T, as previously defined.

One can notice that the computational steps in the generalized PH algorithm correspond to traversing a spanning subgraph S of $\mathcal{T}_{e,w}$, where the edge set of S consists of all the positive slope edges of $\mathcal{T}_{e,w}$, and all negative slope edges of the form $\{\Delta_{0,k}, \Delta_{0,k+1}\}$ for $k = 0, \ldots, m-1$. One can do better by assigning weights $\mathfrak{p}$ (the cost of exponentiation by ℓ^w in $\mathbb{G}_{\ell,e}$) and $\mathfrak{q}$ (the cost of multiplication in $\mathbb{G}_{\ell,e}$) to the edges of $\mathcal{T}_{e,w}$ with positive and negative slopes, respectively, and determining an *optimal strategy* (originally introduced in the context of isogeny computation [7] and then extended to discrete logarithms [8]) to minimize the cost of solving DLP. Such strategies are typically represented in *linear form* as a list of positive integers of length m. This yields a recursive algorithm to solve discrete logarithms; see Algorithm 6.3 in [15].

3 Optimization 1: Signed Digits in the Exponent

As described in Section 2, large tables are used in SIKE for solving discrete logarithms using the Pohlig-Hellman (PH) algorithm. Here we detail a simple technique which we use to reduce table sizes by a factor of 2. The main idea is to switch to a signed representation of the digits D_k of d and use the fact that inversion in $\mathbb{G}$ has negligible cost to save from storing half of the entries in each row of the table. We give the details below.

For any $a+bi \in \mathbb{G} \subset \mathbb{F}_{p^2}$, we use the facts that $p \equiv 3 \pmod 4$ and $|\mathbb{G}| = p+1$ to derive the equality

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

from which it immediately follows that $(a+bi)^{-1} = \overline{a+bi}$. If $\mathbb{G}_{\ell,e} = \langle g \rangle$ is an order ℓ^e subgroup of $\mathbb{G}$ and $h \in \mathbb{G}_{\ell,e}$ is the input to a DLP, we may instead represent $d = \log_g h$ in base $L = \ell^w$ using signed digits [1] as $d = \sum_{k=0}^{m-1} D'_k L^k$, where $D'_k \in [-\lceil (L-1)/2 \rceil, \lceil (L-1)/2 \rceil]$. We modify the table size by defining a new table T^{sgn} as

$$T^{\mathsf{sgn}}[k][D] = g^{-DL^k}, \text{ for } 0 \leq k \leq m-1,\ 1 \leq D \leq \lceil (L-1)/2 \rceil.$$

In addition to the upper bound on D decreasing, notice that we've also discarded the entries corresponding to $D = 0$ since they are all the identity element.

The PH algorithm then proceeds nearly identically to Section 2: define the sequence $h_k = g^{\sum_{i=k}^{m-1} D'_i L^i}$ and determine D'_k by solving $D'_k = \log_s h_k^{L^{m-1-k}}$, where $s = g^{L^{m-1}}$. The latter equality can be solved by iterating through $T^{\mathsf{sgn}}[m-1]$ and performing two equality checks per entry:

1. If $T^{\mathsf{sgn}}[m-1][D] \stackrel{?}{=} h_k^{L^{m-1-k}}$ succeeds, then set $D'_k \leftarrow D$.
2. If $\overline{T^{\mathsf{sgn}}[m-1][D]} \stackrel{?}{=} h_k^{L^{m-1-k}}$ succeeds, then set $D'_k \leftarrow D - L$.

Using the notation from Section 2, when a right traversal $\Delta_{j,k} \to \Delta_{j,k} \cdot g^{-D'_k L^{j+k}}$ is to be performed, there are three cases: $D'_k = 0, D'_k > 0$, or $D'_k < 0$. If $D'_k = 0$, then no action needs to be taken since $g^{-D'_k L^{j+k}} = 1$. If $D'_k > 0$, then we perform $\Delta_{j,k} \to \Delta_{j,k} \cdot T^{\mathsf{sgn}}[j+k][D'_k]$ as before. When $D'_k < 0$, then $\Delta_{j,k} \cdot g^{-D'_k L^{j+k}}$ is computed as $\Delta_{j,k} \cdot \overline{T^{\mathsf{sgn}}[j+k][-D'_k]}$.

One can easily replace the table T in SIKE with T^{sgn} with minor modifications, which reduces the table sizes by a factor of 2. The only computational overhead of this approach is that of performing a conjugation at each equality check and right traversal, the cost of which is quite negligible.

4 Optimization 2: Torus-based Representation and Arithmetic in Cyclotomic Subgroups

This section details the second optimization we make, which offers an additional factor 2 compression of table sizes. In particular, we take advantage of the torus representation $a + bi \mapsto [a + 1 : b]$ of elements of $\mathbb{G}$ detailed in [14] to store elements of $\mathbb{G} \subset \mathbb{F}_{p^2}$ using only a single element of $\mathbb{F}_p$. This trade-off comes at additional computational effort introduced from the torus-based projective representation, in which checking for equality between two elements becomes a nontrivial expense. Compressing table sizes using torus-based representations are detailed in Section 4.1, and the resulting computational overheads are addressed in Sections 4.2-4.4.

4.1 Torus-based Representations

We summarize the torus-based compressed representation of elements of $\mathbb{G}$ as detailed in [14]. Elements of $\mathbb{G}$ written in the form $a+bi$ are said to be in *standard representation*, and we must have $a^2 + b^2 = 1$ (see Section 3). When $b \neq 0$, one can write $a + bi = (\alpha + i)/(\alpha - i)$, where $\alpha := (a + 1)/b$. Since taking $\alpha = 0$ produces $a + bi = -1 + 0i$, we can represent cyclotomic subgroup elements with a single element in $\mathbb{F}_p$ as follows:

$$\mathbb{G} = \{1\} \cup \left\{ \frac{\alpha + i}{\alpha - i} : \ \alpha \in \mathbb{F}_p \right\}. \tag{2}$$

Projective Squaring (**2m**):	$[x:y]^2 = [(x+y)(x-y) : 2xy]$
Projective Cubing (**2m+2s**):	$[x:y]^3 = [x(x^2-3y^2) : y(3x^2-y^2)]$
Projective Multiplication (**3m**):	$[x:y][z:t] = [xz-yt : (x+y)(z+t)-xz-yt]$
Mixed Multiplication (**2m**):	$[x:y][\alpha,1] = [x\alpha - y : x+y\alpha]$
Inversion (**0m**):	$[x:y]^{-1} = [-x:y]$
Projective Equality check (**2m**):	$[x:y] = [z:t] \iff xt - yz = 0$
Mixed Equality check (**1m**):	$[x:y] = [\alpha:1] \iff x - y\alpha = 0$

Table 1: Summary of operations and their costs for elements of $\mathbb{G}$ in projective coordinates. Here, $x, y, z, t, \alpha, a, b \in \mathbb{F}_p$ with x and y not both 0, z and t not both 0, and $a^2 + b^2 = 1$.

Under this correspondence, we define the *compression function* $C : \mathbb{G}\backslash\{1,-1\} \to \mathbb{F}_p$ as $C(a+bi) := (a+1)/b$. Group operations respect compressed representation of elements in $\mathbb{G}$ in the following sense: If $C(a+bi) = \alpha$ and $C(c+di) = \beta$, then we have $C((a+bi)^{-1}) = -\alpha$, and

$$C((a+bi)\cdot(c+di)) = \begin{cases} (\alpha\beta - 1)/(\alpha+\beta) & \alpha+\beta \neq 0 \\ 1 & \alpha+\beta = 0 \end{cases}$$

Compressed representations inherit a *projective representation* as follows. For $x, y \in \mathbb{F}_p$ not both zero, if we define $[x:y] \coloneqq \left(\frac{x+yi}{x-yi}\right)$, then we can write the identity element as $1 = [1:0]$ (*the point at infinity*), and for any $[x:y]$ with $y \neq 0$ we have $[x:y] = [x/y:1]$. In other words, we can rewrite (2) as

$$\mathbb{G} = \{[1:0]\} \cup \{[\alpha:1] :\ \alpha \in \mathbb{F}_p\} = \{[x:y] :\ x, y \in \mathbb{F}_p \text{ not both } 0\}. \quad (3)$$

Note that the compression function C is undefined for $1, -1 \in \mathbb{F}_p$, but these elements are represented in projective coordinates as $[1:0]$ and $[0:1]$, respectively. Passing from regular to projective representation is easy, but the reverse direction requires at least an inversion in $\mathbb{F}_p$: for $a + bi \neq -1$ and $x^2 + y^2 \neq 0$ we have

$$a + bi \longmapsto [a+1:b], \quad [x:y] \longmapsto \frac{x^2-y^2}{x^2+y^2} + \frac{2xy}{x^2+y^2}i$$

We summarize the group operations of $\mathbb{G}$ in projective coordinates in Table 1. Each formula can be directly verified by converting to the regular representations of the involved elements using the above mappings.

Table Compression via Torus Representation The table T^{sgn} from Section 3 used in generalized PH can be compressed by a factor of 2 (giving a net factor

4 compression over the original T) into a compressed table CT by applying the compression function $C(a + bi) = (a + 1)/b$ to each entry in T^{sgn} as follows, taking $C(-1) = 0$ when necessary:

$$CT[k][D] = C(T^{\mathsf{sgn}}[k][D]) = C(g^{-DL^k}), \text{ for } 0 \leq k \leq m-2,\ 1 \leq D \leq \left\lceil \frac{L-1}{2} \right\rceil.$$

Despite the number of table entries appearing the same when compared to T^{sgn}, factor 2 compression is achieved since each entry of CT is a single element of $\mathbb{F}_p$ whereas T^{sgn} stores elements of $\mathbb{F}_{p^2}$. Entries of CT are interpreted as elements of $\mathbb{G}$ by taking $CT[k][D]$ to be the projective element $[CT[k][D] : 1]$.

Computational Overheads Although the table CT has very appealing storage space over the original table T and its factor-2 compressed form T^{sgn}, the savings comes at some nontrivial computational cost. When using projective representation and CT throughout the PH algorithm, right traversals become mixed multiplications (see Table 1), left traversals become projective exponentiations by L, and the discrete logarithms in $\mathbb{G}_{\ell,w}$ computed at the leaf nodes require mixed equality checks between the DLP input and table elements of the form $[\alpha : 1]$. If a sequential search through the table is still used, the incurred cost for this DLP in $\mathbb{G}_{\ell,w}$ is at most $\lceil (L-1)/2 \rceil \mathbf{E}$ at each leaf, where $\mathbf{E}$ is the cost of an equality check; across the entire PH algorithm, this cost quickly accumulates to a significant slow down in run time. The remainder of this section is devoted to developing alternatives to an exhaustive search approach designed to mitigate the overhead incurred by projective equality checks.

In the remainder, we will let $\mathbf{E}$, $\mathbf{L}$, and $\mathbf{M}$ denote the costs of an equality check, exponentiation by ℓ, and mixed multiplication, respectively, in projective coordinates.

4.2 Linear time algorithms

The previous subsection showed that nontrivial overhead is introduced during the DLP solved in the leaf nodes of generalized PH when using projective representation of $\mathbb{G}$ with the compressed table CT. We therefore find great interest in algorithms which can more efficiently solve the following DLP: given $\rho = g^{\ell^{e-w}}$ so that $\mathbb{G}_{\ell,w} = \langle \rho \rangle$, some $[x : y] \in \mathbb{G}_{\ell,w}$, and access to the table $CT[m-1]$, find $D \in [-\lceil (L-1)/2 \rceil, \lceil (L-1)/2 \rceil]$ such that $\rho^D = [x : y]$. This subsection will detail algorithms for solving this problem which run in time that is linear in w.

The steps of the algorithm we will define correspond to traversing an almost-full ℓ-ary tree $\mathcal{G}_{\ell,w}$ of depth-w; we describe this tree now. Intuitively, the tree $\mathcal{G}_{\ell,w}$ represents the exponentiation-by-ℓ structure of the cyclic group $\mathbb{G}_{\ell,w}$, which has order ℓ^w. Specifically, the vertex set $V = V(\mathcal{G}_{\ell,w}) = \bigcup_{j=0}^{w} V_j$ of $\mathcal{G}_{\ell,w}$ is the disjoint union of the vertex sets V_j at depth j for $j = 0, 1, \ldots, w$, where

$$V_0 = \{v_{0,0}\}, \quad V_1 = \{v_{1,i} :\ i = 0, \ldots, (\ell-2)\},$$

$$V_j = \{v_{j,i} :\ j = 2, \ldots, w,\ i = 0, \ldots, \ell^{j-1}(\ell-1) - 1\}.$$

The edge set $E = E(\mathcal{G}_{\ell,w})$ of $\mathcal{G}_{\ell,w}$ is $E = \bigcup_{j=1}^{w} E_j$, where

$$E_1 = \{vw : \ v = v_{0,0}, \ w \in V_1\},$$
$$E_j = \{vw : \ v = v_{j-1,\lfloor i/\ell \rfloor} \in V_{j-1}, \ w = v_{j,i} \in V_j, \ i \in [0, \ell^{j-1}(\ell-1))\}.$$

Note that $|V| = |\mathbb{G}_{\ell,w}| = \ell^w$ and $|E| = \ell^w - 1$. We assign integer values to each vertex in V as follows:

$$v_{0,0} = \ell^w, \tag{4}$$
$$v_{1,i} = \frac{v_{0,\lfloor i/\ell \rfloor}}{\ell} + \ell^{w-1}(i \bmod \ell) = (i+1)\ell^{w-1}, \text{ for } i \in [0, \ell-2] \tag{5}$$
$$v_{j,i} = \frac{v_{j-1,\lfloor i/\ell \rfloor}}{\ell} + \ell^{w-1}(i \bmod \ell), \text{ for } j \in [2, w], \ i \in [0, \ell^{j-1}(\ell-1)). \tag{6}$$

The vertex $v_{0,0}$ represents the identity of $\mathbb{G}_{\ell,w}$, and any two vertices are connected with an edge if and only if one of them is the ℓ-power of the other. We make this precise in Theorem 1, where we relate the vertices of $\mathcal{G}_{\ell,w}$ to the elements of $\mathbb{G}_{\ell,w}$.

Theorem 1. *Let $\mathbb{G}_{\ell,w} = \langle \rho \rangle$ and $\mathcal{G}_{\ell,w} = (V, E)$ as above. Define $g_{j,i} = \rho^{v_{j,i}}$ for all $v_{j,i} \in V$. Then*

$$\mathbb{G}_{\ell,w} = \{g_{j,i} = \rho^{v_{j,i}} : \ v_{j,i} \in V\}.$$

Furthermore, $g_{0,0}$ is the identity element; we have $g_{j,i}^{\ell} = g_{j-1,\lfloor i/\ell \rfloor}$ for $j = 1, \ldots, w$, $i = 0, \ldots, \ell^{j-1}(\ell-1)$; and the order of $g_{j,i}$ is exactly ℓ^j for $j = 0, \ldots, w$.

Proof. Clearly, $g_{0,0} = \rho^{v_{0,0}} = \rho^{\ell^w} = 1$. For $j \geq 1$, we have $g_{j,i}^{\ell} = (\rho^{v_{j,i}})^{\ell} = \rho^{v_{j-1,\lfloor i/\ell \rfloor}} = g_{j-1,\lfloor i/\ell \rfloor}$. Observing that $1 \leq v_{j,i} \leq \ell^w$ are pairwise distinct integers, and that the highest power of ℓ that divides $v_{j,i}$ is $(w-j)$ proves the rest of the claims.

Theorem 2. *Let $h \in \mathbb{G}_{\ell,w}$ be an element of order ℓ^k for some arbitrary $k \in \{1, \ldots, w\}$. Define a sequence $H = [h_0, \ldots, h_k]$ such that $h_k = h$ and $h_{j-1} = h_j^{\ell}$ for $j = 1, \ldots, k$. Then, there exists a unique path $P_{0,k} = v_{0,0}, v_{1,i_1}, \ldots, v_{k,i_k}$ in $\mathcal{G}_{\ell,w}$ such that $v_{j,i_j} \in V_j$ and $h_j = g_{j,i_j}$ for $j = 0, \ldots, k$.*

Proof. We prove this by induction on k. For the base case $k = 1$, we have

$$h_1 = h = \rho^{(i_1+1)\ell^{w-1}} = \rho^{v_{1,i_1}} = g_{1,i_1},$$

for some $i_1 \in \{0, \ldots, (\ell-2)\}$ because the order of h_1 is ℓ. Also,

$$h_0 = h_1^{\ell} = 1 = \rho^{\ell^w} = \rho^{v_{0,0}} = g_{0,0}.$$

This proves the base case because $v_{0,0}v_{1,i_1}$ is an edge in $\mathcal{G}_{\ell,w}$. Now, suppose that $k \geq 2$. By Theorem 1, $h_k = g_{k,i_k}$ for some $i_k \in \{0, \ldots, \ell^{k-1}(\ell-1)\}$ because h_k is of order ℓ^k. Next, using Theorem 1, we write

$$h_{k-1} = h_k^{\ell} = g_{k,i_k}^{\ell} = g_{k-1,\lfloor i_k/\ell \rfloor} = g_{k-1,i_{k-1}},$$

where $i_{k-1} = \lfloor i_k/\ell \rfloor$. Note that $e_k = v_{k-1,i_{k-1}} v_{k,i_k} \in E_k$ is an edge of $\mathcal{G}_{\ell,w}$. By the induction hypothesis, there exists a unique path $P_{0,k-1} = v_{0,0}, v_{1,i_1}, \ldots, v_{k-1,i_{k-1}}$ in $\mathcal{G}_{\ell,w}$ such that $v_{j,i_j} \in V_j$ and $h_j = g_{j,i_j}$ for $j = 0, \ldots, (k-1)$. Concatenating $P_{0,k-1}$ and e_k yields the desired path $P_{0,k}$.

Theorem 3. *Let $1 \neq h \in \mathbb{G}_{\ell,w} = \langle \rho \rangle$. Given h and ρ, one can determine*

1. *k, $i_1, i_2, \ldots, i_k$, that corresponds to the path $P_{0,k} = v_{0,0}, v_{1,i_1}, \ldots, v_{j,i_k}$ as in Theorem 2;*
2. *$s_1, s_2, \ldots, s_k$ such that , $s_1 = i_1 + 1$, $s_j \in \{0, \ldots, (\ell-1)\}$, and $i_j = \ell \cdot i_{j-1} + s_k$ for $j = 2, \ldots, k$.*

Moreover, $h = \rho^d$, where $d = \ell^{w-k} \sum_{j=1}^{k} s_j \ell^{j-1}$.

Proof. Given $1 \neq h$, we first determine the least positive integer k such that $h^{\ell^k} = 1$, by performing k exponentiations by ℓ in $\mathbb{G}_{\ell,w}$. We also store the intermediate values $h_i = h^{\ell^{k-i}}$, $i = 0, \ldots, k$, in an array $H = [h_0 = 1, h_1, \ldots, h_k = h]$. By Theorem 2, there is a unique path $P_{0,k} = v_{0,0}, v_{1,i_1}, \ldots, v_{j,i_k}$ in $\mathcal{G}_{\ell,w}$ such that $v_{j,i_j} \in V_j$ and $h_j = g_{j,i_j}$ for $j = 0, \ldots, k$. By the proof of Theorem 2, $i_1 \in \{0, \ldots, (\ell-2)\}$ can be determined as the integer satisfying $h_1 = g_{1,i_1}$, and we set $s_1 = i_1 + 1$. Next, we inductively assume that i_{j-1} and s_{j-1} are already known for some $2 \leq j \leq w$. By the definition of E_j, and the fact that $v_{j-1,i_{j-1}} v_{j,i_j} \in E_j$, $i_j = \ell \cdot i_{j-1} + s_j$ for some $s_j \in \{0, \ldots, (\ell-1)\}$. Since i_{j-1} is already known, the value of s_j (and i_j) can be determined as the integer satisfying $h_j = g_{j,i_j} = g_{j,\ell \cdot i_{j-1} + s_j}$. As a result, we can recover all i_j and s_j for all $j = 1, \ldots, k$, where $s_1 = i_1 + 1$ and $i_j = \ell \cdot i_{j-1} + s_j$. To prove the last claim of our theorem, we need to show $d = v_{k,i_k}$, because $h = h_k = g_{k,i_k} = \rho^{v_{k,i_k}}$. For $k = 1$, Theorem 3 yields $d = s_1 \ell^{w-1}$, and it follows from (5) that $v_{1,i_1} = (i_1 + 1)\ell^{w-1} = s_1 \ell^{w-1} = d$, as required. In the following, we assume that $k \geq 2$. Using (4)-(6), we write

$$\begin{aligned} v_{k,i_k} &= \frac{v_{k-1,i_{k-1}}}{\ell} + s_k \ell^{w-1} = \frac{v_{k-2,i_{k-2}}}{\ell^2} + s_{k-1}\ell^{w-2} + s_k \ell^{w-1} \\ &= \frac{v_{1,i_1}}{\ell^{k-1}} + s_2 \ell^{w-k+1} + \cdots + s_k \ell^{w-1} = s_1 \ell^{w-k} + s_2 \ell^{w-k+1} + \cdots + s_k \ell^{w-1} \\ &= \ell^{w-k} \sum_{j=1}^{k} s_j \ell^{j-1} = d, \text{ which finishes the proof.} \end{aligned}$$

Algorithm 1 uses the graph $\mathcal{G}_{\ell,w}$ and its properties to solve the DLP in $\mathbb{G}_{\ell,w}$. The algorithm applies to solving the DLP in any cyclic group of order ℓ^w, but is stated specifically for $\mathbb{G}_{\ell,w}$. The computational and storage complexities of Algorithm 1 are completely described in Theorem 4.

Theorem 4. *For $h \in \mathbb{G}_{\ell,w}$ let $Cost_1(h)$ denote the total cost of running Algorithm 1 with input h in terms of $\mathbf{L}$ and $\mathbf{E}$. By Theorem 1, $h = \rho^{v_{j',i'}}$ for some j' and i'.*

- *If $j' = 0$, then $Cost_1(h) = 0$.*

– *If* $1 \leq j' \leq w$ *and* $0 \leq i' \leq \ell^{j'-1}(\ell-1)-1$, *then* $Cost_1(h) = j'\mathbf{L} + \left(j' + \sum_{j=0}^{j'-1} (\lfloor i'/\ell^j \rfloor \mod \ell)\right)\mathbf{E}$.

Consequently, the average case complexity of Algorithm 1 on a uniformly random input is exactly

$$\left(w - \frac{1}{\ell-1} + \frac{1}{(\ell-1)\ell^w}\right)\mathbf{L} + \left(\frac{w(\ell+1)}{2} - \frac{\ell}{\ell-1} + \frac{1}{\ell^{w-1}(\ell-1)}\right)\mathbf{E},$$

and Algorithm 1 has a worst case cost of $Cost_1(\rho^{v_{w,\ell^{w-1}(\ell-1)-1}}) = w\mathbf{L}+(w\ell-1)\mathbf{E}$. *Furthermore, the total storage space required by Algorithm 1 is at most* ℓ^w+w-2 *elements of* $\mathbb{G}_{\ell,w}$.

Proof. The proof of Theorem 4 is given in Appendix A. The proof also provides some correctness verifications for the algorithm.

Refinements of Algorithm 1 If the cost of finding the inverse of an element in $\mathbb{G}_{\ell,w}$ is negligible, then Algorithm 1 can be improved. Intuitively, one can eliminate half of the paths $P_{0,k}$ in Theorem 3 at a cost of inverting elements in H in the beginning of the algorithm. This reduces the number of elements to be stored in Algorithm 1 by a factor of 2. One can further improve on the storage requirement and the number of equality checks by a factor of $(\ell-1)/\ell$ because, in practice, we do not need to run the comparison $H[j] = G[j][i_j + s]$ for $s = \ell - 1$ (one out of ℓ comparisons) in Algorithm 1. Implementing these optimizations differs slightly when $\ell = 2$ and $\ell > 2$ because of the unique situation of the root vertex of $\mathcal{G}_{\ell,w}$ having only one child when $\ell = 2$. Therefore, we present these cases separately in Algorithm 2 and Algorithm 3, and Theorems 5 and 6 (for $\ell = 3$) give the time and storage complexities for these algorithms.

Theorem 5. *For* $h \in \mathbb{G}_{2,w}$, *let* $Cost_2(h)$ *denote the total cost of running Algorithm 2 with input* h *in terms of* $\mathbf{L}$ *and* $\mathbf{E}$. *By Theorem 1,* $h = \rho^{v_{j',i'}}$ *for some* j' *and* i'.

– *If* $j' = 0$ *(so* $h = 1$*), then* $Cost_2(h) = 0$.
– *If* $1 \leq j' \leq w$ *(so* $h \neq 1$*), then* $Cost_2(h) = j'\mathbf{L} + (j'-1)\mathbf{E}$.

Consequently, the average case complexity of Algorithm 2 on a uniformly random input is exactly

$$\left(w - 1 + \frac{1}{2^w}\right)\mathbf{L} + \left(w - 2 + \frac{1}{2^{w-1}}\right)\mathbf{E},$$

and Algorithm 2 has a worst case cost of $Cost_2(\rho^{v_{w,i'}}) = w\mathbf{L}+(w-1)\mathbf{E}$ *occurring at any* $i' \in [0, 2^{w-1})$. *Furthermore, the total storage space required by Algorithm 2 is at most* $2^{w-2} + w + 2$ *elements of* $\mathbb{G}_{2,w}$.

Proof. Algorithm 2 performs exactly j' many squarings in the first loop and sets $k = j'$. If $j' = 0$, then no squarings are performed and the algorithm immediately returns 0 at no cost. If $j' = 1$, one squaring is performed and 2^{w-1} is returned at a cost of $1\mathbf{L} = j'\mathbf{L} + (j'-1)\mathbf{E}$. Otherwise, the second loop is entered and exactly $j'-1$ equality checks are performed, giving a total cost of $j'\mathbf{L} + (j'-1)\mathbf{E}$. This proves the first two claims in the theorem. The worst case complexity follows immediately from the second claim. The average case complexity can now be computed as

$$\sum_{h \in \mathbb{G}_{2,w}} \Pr[h] \cdot Cost_2(h) = \frac{1}{2^w} \left(1\mathbf{L} + \sum_{j'=2}^{w} \sum_{i'=0}^{2^{j'-1}-1} Cost_2(\rho^{v_{j',i'}}) \right)$$

which one can check simplifies to the expression given in the theorem. The table G stores exactly $|V_j|/4 = 2^{j-3}$ elements in row $G[j]$ for $j \geq 3$ and one element in $G[2]$, and so G stores $2^{w-2}+2$ elements of $\mathbb{G}$. Accounting for at most w elements stored in the list H, the total storage is then $2^{w-2} + w + 2$. This completes the proof.

Theorem 6. *The average case complexity of Algorithm 3 for $\ell = 3$ on a uniformly random input is exactly*

$$\left(\frac{79}{15} w - \frac{33}{10} + \frac{33}{10 \cdot 3^w} \right) \mathbf{m},$$

assuming $\mathbf{L} = 4\mathbf{m}$ and $\mathbf{E} = 1\mathbf{m}$. Furthermore, the total storage space required by Algorithm 3 is at most $3^{w-1} + w - 1$ elements of $\mathbb{G}_{3,w}$.

Remark 1. In Theorems 5 and 6, we assume the algorithms use only a single multiplication when checking if some element $H[j]$ is equal to some table entry $G[j][k]$ or its inverse $G[j][k]^{-1}$. If projective coordinates are used with $H[j] = [x : y]$ and $G[j][k] = [\alpha : 1]$, then both equalities can be tested at cost $1\mathbf{m}$ by checking $x \stackrel{?}{=} y \cdot \alpha$ and $x \stackrel{?}{=} -y \cdot \alpha$.

4.3 An Exponential Time Algorithm for $\ell = 2$

The algorithm described in the previous section has asymptotically linear time in w for $\ell = 2$, but performs relatively poorly for small values of w. In this section we describe an algorithm specific to projective coordinates and $\ell = 2$, which uses exponentially many $\mathbb{F}_p$ multiplications but performs very well for small values of w. In Section 4.4, we will combine both of these algorithms together to achieve an algorithm with better performance than Algorithm 2 for $\ell = 2$. We move toward describing this exponential algorithm now.

The projective points of order 2^k in $\mathbb{G}_{2,e}$ for $k \geq 1$ have a quite simple form and can be solved for explicitly. There is a single point of order 2 which is found to be $[0 : 1]$ by solving the equation $[x : 1]^2 = [1 : 0]$ for x. To find points of order 2^k for $k > 1$ we solve the equation $[x : 1]^2 = [a : 1]$ for x, which leads to the

quadratic equation $x^2 - 2ax - 1 = 0$ with solution $x = a \pm \sqrt{a^2+1}$. We apply this formula iteratively starting with the point $[0:1]$ to generate all points of order 2^k for any $k \geq 1$. To study these points, we define the following recursive sequence: let $a_1 = 0$ and for $j \geq 1$ and $0 \leq i < 2^{j-1}$ define

$$a_{2^j+2i} = a_{2^{j-1}+i} + \sqrt{a_{2^{j-1}+i}^2 + 1}, \quad a_{2^j+2i+1} = a_{2^{j-1}+i} - \sqrt{a_{2^{j-1}+i}^2 + 1}. \tag{7}$$

The first 7 terms of this sequence are: $a_1 = 0$, $a_2 = 1$, $a_3 = -1$, $a_4 = 1+\sqrt{2}$, $a_5 = 1-\sqrt{2}$, $a_6 = -1+\sqrt{2}$, $a_7 = -1-\sqrt{2}$. One can use straight-forward induction on j to show that $a_{2^j+2i} = -a_{2^{j+1}-2i-1}$ and that $[a_{2^j+i} : 1]$ has order 2^{j+1} for $j \geq 0$ and $0 \leq i < 2^j$.

As may be apparent, terms in the sequence a are linear $\{-1,0,1\}$-combinations of a smaller set of terms. We get a hold on this sequence explicitly by defining a new sequence b as

$$b_{2^j+i} = a_{2^{j+2}+2i} - a_{2^{j+1}+i} = \sqrt{a_{2^{j+1}+i}^2 + 1}$$

for $j \geq 0$ and $0 \leq i < 2^j$. The first 3 terms of this sequence are:

$$b_1 = \sqrt{2}, \quad b_2 = \sqrt{2}\sqrt{2+\sqrt{2}}, \quad b_3 = \sqrt{2}\sqrt{2-\sqrt{2}}.$$

Notice that terms $a_1, \ldots, a_{15}$ are $\{-1,0,1\}$-combinations of $1, b_1, b_2$, and b_3. As expected, the following theorem shows that we can write any term of the sequence a as a combination of terms from the sequence b.

Theorem 7. *Let $j \geq 2$ and $0 \leq i < 2^{j-1}$. Let $(\beta_{j-2}\cdots\beta_0)_2$ be the binary representation of i so that $i = \beta_{j-2}2^{j-2} + \cdots + \beta_1 2 + \beta_0$. Then*

$$\begin{aligned} a_{2^j+i} &= 1 + (-1)^{\beta_{j-2}} b_1 + (-1)^{\beta_{j-3}} b_{2^1+(\beta_{j-2})_2} + \cdots + (-1)^{\beta_0} b_{2^{j-2}+(\beta_{j-2}\cdots\beta_1)_2} \\ &= 1 + \sum_{k=0}^{j-2} (-1)^{\beta_k} b_{2^{j-k-2}+(\beta_{j-2}\cdots\beta_{k+1})_2}. \end{aligned}$$

Proof. We use induction on j. When $j = 2$ we have $i \in \{0,1\}$, and

$$a_{2^2+0} = 1+\sqrt{2} = 1 + (-1)^0 b_1, \qquad a_{2^2+1} = 1-\sqrt{2} = 1 + (-1)^1 b_1.$$

Now let $j > 2$ and assume that the equality holds for any $a_{2^{j'}+i'}$ with $j' < j$ and $0 \leq i' < 2^{j'-1}$. Let $0 \leq i < 2^{j-1}$ with $i = (\beta_{j-2}\cdots\beta_0)_2$. By the formula which defines the sequence a (Equations (7)) and the inductive hypothesis, we have

$$\begin{aligned} a_{2^j+(\beta_{j-2}\cdots\beta_0)_2} &= a_{2^{j-1}+(\beta_{j-2}\cdots\beta_1)_2} + (-1)^{\beta_0}\sqrt{(a_{2^{j-1}+(\beta_{j-2}\cdots\beta_1)_2})^2 + 1} \\ &= a_{2^{j-1}+(\beta_{j-2}\cdots\beta_1)_2} + (-1)^{\beta_{j-1}} b_{2^{j-2}+(\beta_{j-2}\cdots\beta_1)_2} \\ &= \Big(1 + (-1)^{\beta_{j-2}} b_1 + (-1)^{\beta_{j-3}} b_{2^1+(\beta_{j-2})_2} + \cdots + (-1)^{\beta_1} b_{2^{j-3}+(\beta_{j-2}\cdots\beta_2)_2}\Big) \\ &\qquad + (-1)^{\beta_{j-1}} b_{2^{j-2}+(\beta_{j-2}\cdots\beta_1)_2} \end{aligned}$$

where we've used the inductive hypothesis on $j' = j-1$ and $i' = (\beta_{j-2}\cdots\beta_1)_2 < 2^{j-2} = 2^{j'-1}$.

Note that the theorem gives a representation of all a_k with $k \in \mathbb{N}$ as a linear combination of terms from b, since if $2^{j-1} \leq i < 2^j$ we have $a_{2^j+i} = -a_{2^j+(2^j-i-1)}$, and $0 \leq 2^j - i - 1 < 2^{j-1}$.

Algorithm 4 can be used to compute discrete logarithms by means of Theorem 7 using a table T^{exp} defined by $T^{\mathsf{exp}}[j][i] = b_{2^j+i}$ for $j \in [0, w-3]$ and $i \in [0, 2^j)$. Given some $[x : y]$ such that $[x : y] = [a_k : 1]$ for some a_k with $k \in [1, 2^w)$, Algorithm 4 first finds k. A precomputed conversion table Log can then be used to translate points $[a_k : 1]$ into the corresponding discrete logarithm in the desired base ρ; i.e. a table Log is defined by $Log[k] = D$ where $\rho^D = [a_k : 1]$ for $1 \leq k < 2^w$ and $D \in [-2^{w-1}, 2^{w-1}]$.

Theorem 8. *Let $Cost_4(h)$ denote the total number of $\mathbb{F}_p$ multiplications used by Algorithm 4 when ran with input h with $w \geq 2$. Then for $j' \in [2, w)$ and $i' \in [0, 2^{j'-1})$, we have*

$$Cost_4([a_{2^{j'}+i'} : 1]) = Cost_4([a_{2^{j'}+(2^{j'}-i'-1)} : 1]) = 2^{j'-2} + \lfloor i'/2 \rfloor,$$

and consequently the average number of $\mathbb{F}_p$ multiplications used by Algorithm 4 on a uniformly random input is exactly $2^{w-3} - \frac{1}{2}$. Furthermore, the total storage required by Algorithm 4 is at most $2^{w-1} - 1$ many elements of $\mathbb{F}_p$.

Proof. By Theorem 7, Algorithm 4 achieves a successful equality check on iteration $j = j'$ of loop 6 and iteration $i = i'$ of loop 7, after which it immediately terminates. The only $\mathbb{F}_p$ multiplication in the algorithm occurs on line 11, and so we count how many times this line is executed. When $j < j'$ the loop on variable i executes in full, whereas when $j = j'$ only iterations $0 \leq i \leq \lfloor i'/2 \rfloor$ are performed before the algorithm returns. Line 11 is executed exactly on iterations when i is even, and so we have

$$Cost_4([a_{2^{j'}+i'} : 1]) = \lfloor i'/2 \rfloor + 1 + \sum_{j=2}^{j'-1} 2^{j-2} = 2^{j'-2} + \lfloor i'/2 \rfloor.$$

Since $a_{2^{j'}+2i'} = -a_{2^{j'+1}-2i'-1}$, it follows that Algorithm 4 also terminates on iteration $j = j'$ and $i = i'$ when ran with input $[a_{2^{j'+1}-2i'-1} : 1]$, and so $Cost_4([a_{2^{j'}+i'} : 1]) = Cost_4([a_{2^{j'}+(2^{j'}-i'-1)} : 1])$.

Treating the identity, a_1, a_2, and a_3 as special cases with 0 cost, the sum of the costs of Algorithm 4 on all inputs is computed as

$$\begin{aligned}\sum_{k=4}^{2^w-1} Cost_4([a_k : 1]) &= \sum_{j'=2}^{w-1} \sum_{i'=0}^{2^{j'}-1} Cost_4([a_{2^{j'}+i'} : 1]) \\ &= 2 \sum_{j'=2}^{w-1} \sum_{i'=0}^{2^{j'-1}-1} 2^{j'-2} + \lfloor i'/2 \rfloor = 2^{2w-3} - 2^{w-1}.\end{aligned}$$

Dividing by 2^w gives $2^{w-3} - \frac{1}{2}$ as the average cost of Algorithm 4 on a uniformly random input, as claimed.

	Restriction	Average Complexity	Storage
Alg. 1	$\ell = 2$	$\left(\frac{7}{2}w - 4 + \frac{4}{2^w}\right)\mathbf{m}$	$2^w + w - 2$
Alg. 2	$\ell = 2$	$\left(3w - 4 + \frac{1}{2^{w-2}}\right)\mathbf{m}$	$2^{w-2} + w + 2$
Alg. 4	$\ell = 2$	$\left(2^{w-3} - \frac{1}{2}\right)\mathbf{m}$	$2^{w-1} + 1$
Alg. 1	$\ell = 3$	$\left(\frac{28}{5}w - \frac{33}{10} + \frac{33}{10 \cdot 3^w}\right)\mathbf{m}$	$3^w + w - 2$
Alg. 3	$\ell = 3$	$\left(\frac{79}{15}w - \frac{33}{10} + \frac{33}{10 \cdot 3^w}\right)\mathbf{m}$	$3^{w-1} + w - 1$

Table 2: Summary of algorithm computational and storage complexities. Storage is the total number of $\mathbb{F}_p$ elements stored.

There are $2^{w-2} - 1$ elements of $\mathbb{F}_p$ stored in the table T^{exp}, and line 11 potentially populates all of *prods* with a copy of T^{exp} multiplied by y. Accounting for the variable *sum* gives a total storage of at most $2^{w-1} - 1$. This completes the proof.

A summary of the average complexities and storage costs of the algorithms discussed so far is given in Table 2.

4.4 A Hybrid Algorithm for $\ell = 2$

In this section we formulate a hybrid of Algorithms 2 and 4, resulting in our most efficient algorithm for computing discrete logarithms in $\mathbb{G}_{\ell,w}$ when using projective representation. Algorithm 2 performs very well asymptotically, but Algorithm 4 has much better performance for small values of w. Intuitively, the hybrid algorithm will square the input element until the order of the element is guaranteed to be small, pushing it toward the root of $\mathcal{G}_{\ell,w}$; then Algorithm 4 is ran on this small order element to compute its discrete logarithm, and finally the path which was taken through $\mathcal{G}_{\ell,w}$ is traced backwards as in Algorithm 2 to compute the full discrete logarithm.

Specifically, let $w_1, w_2 \in \mathbb{N}$ such that $w = w_1 + w_2$. The hybrid algorithm, detailed in Algorithm 5, first performs (at most) w_2 many squarings on the input element h to produce a list H of length k satisfying $H[j] = h^{2^{k-j}}$ for $j \in [0, k)$. The element $H[0]$ is then guaranteed to be in $\mathbb{G}_{2,w_1}$, and Algorithm 4 is then called with $H[0]$ as input. It could be the case that $H[0]$ is the identity element, in which case we instead call Algorithm 4 with input $H[w_1]$. Once Algorithm 4 completes and returns some logarithm d, the values ord and i_j are computed such that $d = v_{ord,i_j}$. If necessary, the list H is inverted so that the path through $\mathcal{G}_{2,w}$ lies in the region corresponding to the elements stored in the table G. The remainder of the algorithm then proceeds as in Algorithm 2 by backtracing the path through $\mathcal{G}_{2,w}$. To simplify the indexing on the computation of d in the latter part of the algorithm, a bit-shift and bit-set approach is used instead.

In the event that $H[0]$ is the identity and the number of squarings performed was no more than w_1, Algorithm 4 is instead called on h and Algorithm 5 immediately returns the proper logarithm.

To compute the value of i_j from d in Algorithm 5, the following lemma is used. The statement for $j \in \{1, 2\}$ can be easily verified by hand, after which the general proof follows easily with induction on j using the definition of $v_{j,i}$.

Lemma 1. *Let $j \in [2, w]$ and $i \in [0, 2^{j-1})$. Write*

$$i = \beta_{j-2}2^{j-2} + \beta_{j-3}2^{j-3} + \cdots + \beta_1 2 + \beta_0 \qquad \text{for } \beta_k \in \{0, 1\}.$$

Then $v_{j,i} = (2(\beta_0 2^{j-2} + \beta_1 2^{j-3} + \cdots + \beta_{j-3}2 + \beta_{j-2}) + 1)2^{w-j}$. When $j = 1$, we have $v_{1,0} = (2 \cdot 0 + 1)2^{w-1}$.

Theorem 9. *Let $2 \leq w_1 < w$ and let $w_2 = w - w_1$. Let $Cost_5(h)$ denote the total number of $\mathbb{F}_p$ multiplications used by Algorithm 5 when ran with input h, using w_1 as the parameter for Algorithm 4.*

1. *If $j \leq w_1$ and $j \leq w_2$, then $Cost_5(\rho^{v_{j,i}}) = j\mathbf{L} + Cost_4(\rho^{v_{j,i}})$.*
2. *If $w_1 < j \leq w_2$, then $Cost_5(\rho^{v_{j,i}}) = j\mathbf{L} + (j - w_1)\mathbf{E} + Cost_4(\rho^{2^{j-w_1}v_{j,i}})$.*
3. *If $j > w_2$, then $Cost_5(\rho^{v_{j,i}}) = w_2\mathbf{L} + w_2\mathbf{E} + Cost_4(\rho^{2^{w_2}v_{j,i}})$.*

When $\mathbf{L} = 2\mathbf{m}$ and $\mathbf{E} = 1\mathbf{m}$, the average number of $\mathbb{F}_p$ multiplications used by Algorithm 5 on a uniformly random input is given by the following table:

if $w_2 \leq 2$ and $w_2 \leq w_1$	$3w_2 + 2^{w_1-3} - \frac{1}{2} - \frac{w_2+2}{2^{w_1}} + \frac{2}{2^w}$
if $2 < w_2$ and $w_2 \leq w_1$	$3w_2 + 2^{w_1-3} - \frac{1}{2} + \frac{2^{w_2-3}-w_2-\frac{5}{2}}{2^{w_1}} + \frac{2}{2^w}$
if $w_1 = 2$ and $2 < w_2$	$3w_2 - \frac{5}{4} + \frac{6}{2^w}$
if $2 < w_1$ and $w_1 < w_2$	$3w_2 + 2^{w_1-3} - 2^{w_1-w_2-4} - \frac{5}{16} - \frac{w_1+\frac{7}{2}}{2^{w_1}} + \frac{1}{2^{w_2}} + \frac{2}{2^w}$

Proof. Recall that the element $\rho^{v_{j,i}}$ has order 2^j. If $j \leq w_1$ and $j \leq w_2$, then Algorithm 5 performs j many squarings in the initial loop 2 and terminates on line 10 after calling Algorithm 4 on $\rho^{v_{j,i}}$. When $w_1 < j \leq w_2$, the algorithm performs j many squarings, calls Algorithm 4 on the element $H[w_1] = \rho^{2^{j-w_1}v_{j,i}}$, and then performs $j - w_1$ equality checks in the loop on line 33. Finally if $j > w_2$, the maximal number w_2 of squarings is performed in loop 2, Algorithm 4 is called on $H[0] = \rho^{2^{w_2}v_{j,i}}$, and w_2 many equality checks are performed in loop 33.

To compute the average case complexity, we first sum over all j and i values in each of the three cases. We take $\mathbf{L} = 2\mathbf{m}$ and $\mathbf{E} = 1\mathbf{m}$. Letting $m = \min\{w_1, w_2\}$ for the first case, we have

$$\begin{aligned}\sum_{j=0}^{m}\sum_{i=0}^{2^{j-1}-1} Cost_5(\rho^{v_{j,i}}) &= \sum_{j=0}^{m}\sum_{i=0}^{2^{j-1}-1} 2j + \sum_{j=0}^{m}\sum_{i=0}^{2^{j-1}-1} Cost_4(\rho^{v_{j,i}}) \\ &= (2(m-1)2^m + 2)\mathbf{m} + \sum_{j=0}^{m}\sum_{i=0}^{2^{j-1}-1} Cost_4(\rho^{v_{j,i}}).\end{aligned}$$

When $m \leq 2$ the final summation above is 0. Otherwise, by Theorem 7 we get $2(m-1)2^m + 2 + 2^{m-3} - \frac{1}{2}$ multiplications. For the second case, we have that

$\sum_{j=w_1+1}^{w_2}\sum_{i=0}^{2^{j-1}-1} j\mathbf{L} + (j-w_1)\mathbf{E} = (2^{w_1}(3-2w_1) - 2^{w_2}(w_1 - 3w_2 + 3))\mathbf{m}$, and by repeated use of Theorem 1 we get

$$\sum_{j=w_1+1}^{w_2}\sum_{i=0}^{2^{j-1}-1} Cost_4(\rho^{2^{j-w_1}v_{j,i}}) = \sum_{j=w_1+1}^{w_2}\sum_{i=0}^{2^{j-1}-1} Cost_4(\rho^{v_{w_1,\lfloor i/2^{j-w_1}\rfloor}})$$
$$= \sum_{j=1}^{w_2-w_1}\sum_{i=0}^{2^{j+w_1-1}-1} Cost_4(\rho^{v_{w_1,\lfloor i/2^{j}\rfloor}}) = \sum_{j=1}^{w_2-w_1}\sum_{i=0}^{2^{w_1-1}-1} 2^j Cost_4(\rho^{v_{w_1,i}}).$$

When $w_1 \leq 2$, the sum above is 0. Otherwise, the inner summation totals the cost of all order w_1 elements under Algorithm 4. By Theorem 7, we get

$$\sum_{j=w_1+1}^{w_2}\sum_{i=0}^{2^{j-1}-1} Cost_4(\rho^{2^{j-w_1}v_{j,i}}) = \sum_{j=1}^{w_2-w_1}\sum_{i=0}^{2^{w_1-1}-1} 2^j(2^{w_1-3} + \lfloor i/2\rfloor)$$
$$= \frac{1}{16}(3\cdot 2^{w_1} - 8)(2^{w_2} - 2^{w_1}).$$

Finally we sum all j and i values in the third case. We have $\sum_{j=w_2+1}^{w}\sum_{i=0}^{2^{j-1}-1} w_2\mathbf{L} + w_2\mathbf{E} = 3w_2(2^w - 2^{w_2})\mathbf{m}$, and

$$\sum_{j=w_2+1}^{w}\sum_{i=0}^{2^{j-1}-1} Cost_4(\rho^{2^{w_2}v_{j,i}}) = \sum_{j=1}^{w_1}\sum_{i=0}^{2^{j+w_2-1}-1} Cost_4(\rho^{2^{w_2}v_{j+w_2,i}})$$
$$= \sum_{j=1}^{w_1}\sum_{i=0}^{2^{j+w_2-1}-1} Cost_4(\rho^{v_{j,\lfloor i/2^{w_2}\rfloor}}) = \sum_{j=1}^{w_1}\sum_{i=0}^{2^{j-1}-1} 2^{w_2} Cost_4(\rho^{v_{j,i}})$$

If $w_1 \leq 2$, the sum is 0. Otherwise, by Theorem 7 it equals $2^{w_2}(2^{w_1-3} - \frac{1}{2})\mathbf{m} = (2^{w-3} - 2^{w_2-1})\mathbf{m}$.

The above calculations yield all the necessary components to derive the average case complexity of Algorithm 5. To reach the expressions given in the table in the theorem statement, one takes the sum over the components corresponding to each case and divides the result by 2^w. We omit the details. This completes the proof.

5 Implementation results and comparisons

We have implemented our proposal in C language on top of the Microsoft SIDH library [9]. Our contributions were incrementally included in the SIDH library [9]. The signed-digits technique (Section 3) was integrated during the SIKE submission to Round 3 of the NIST process and can be found in the submitted package

Dlog in $\mathbb{G}_{2,e}$	source	$w=3$		$w=4$		$w=5$	
		time	size	time	size	time	size
$e=216$ (SIKEp434)	previous [8]	1944	70	1600	105	1415	342
	ours	1818	18	1542	27	1408	86
$e=250$ (SIKEp503)	previous [8]	2340	186	1937	279	1662	221
	ours	2184	47	1859	70	1649	56
$e=305$ (SIKEp610)	previous [8]	2973	268	2482	405	2126	321
	ours	2761	67	2368	102	2095	81
$e=372$ (SIKEp751)	previous [8]	3765	197	3126	296	2748	954
	ours	3476	49	2964	74	2688	240

Dlog in $\mathbb{G}_{3,e}$	source	$w=2$		$w=3$		$w=4$	
		time	size	time	size	time	size
$e=137$ (SIKEp434)	previous [8]	1845	151	1407	301	1185	688
	ours	2371	34	2029	73	1868	172
$e=159$ (SIKEp503)	previous [8]	2208	199	1680	198	1407	895
	ours	2828	44	2397	48	2185	221
$e=192$ (SIKEp610)	previous [8]	2757	142	2121	284	1767	639
	ours	3496	32	2994	68	2703	158
$e=239$ (SIKEp751)	previous [8]	3621	429	2793	859	2337	1932
	ours	4542	95	3886	207	3507	477

Table 3: Comparative results of the average cost (in $\mathbb{F}_p$ multiplications) and table sizes (in KiB) to compute logarithms in $\mathbb{G}_{2,e}$ and $\mathbb{G}_{3,e}$ using Pohlig-Hellman with torus-based representations, and using Algorithm 5 (or Algorithm 3) *vs.* previous Pohlig-Hellman with standard $\mathbb{F}_{p^2}$ representations [8,15]. We set $w_1 = w - 1$ in Algorithm 5 as it gives the best speed.

[2]. More recently the torus-based optimizations (Section 4) were included in the official Microsoft SIDH library [4].

Table 3 showcases the result of our memory optimized approach for computing discrete logarithms in $\mathbb{G}_{2,e}$ using signed-digits with torus-based representations and Algorithm 5 for leaf computations. We compare discrete logarithm computation time against previous work in terms of number of $\mathbb{F}_p$ multiplications. In this case, our approach turns out to be slightly faster in addition to the factor-4 reduction in table sizes. Despite leaf computations being more expensive than [8], savings are gained through cheaper edge traversal costs by using projective squaring and mixed multiplications (each costing $2\mathbf{m}$ instead of $3\mathbf{m}$).

Table 3 also illustrates our results for computing discrete logarithms in $\mathbb{G}_{3,e}$ using signed-digits with torus-based representations and Algorithm 3 for leaf computations. One can see that for practical values of w (i.e., tables not too large), the time overhead significantly increases ranging from $29-58\%$ for $w \in [2,4]$. Such large time overheads, which do not occur for $\ell = 2$, arise for two reasons: 1) left edge traversals for $\ell = 3$ consist of projective cubings that take

[4] `https://github.com/microsoft/PQCrypto-SIDH/commit/e990bc6784c68426f69ac11ada3dd5fbfed8b714`

4m, which is more expensive than the cost of standard cyclotomic cubing (**3m**); and 2) Algorithm 3 cannot exploit the extra structures that were present in the $\mathbb{G}_{2,w}$ subgroup and were exploited in Algorithm 5. On the other hand, this is not as bad as it may seem, since according to benchmarks we ran on the compressed SIKE code, the discrete logarithm phase takes only $\approx 15\%$ of the whole *KeyGen*. Hence, the overall computational overheads would reduce to 4-9% from an end user perspective.

For integrating our results into the official SIKE suite [2], we decided to use the torus-based method only for the $\ell = 2$ to obtain a factor-4 reduction in table sizes. For $\ell = 3$, in order not to impact the runtime negatively, we use the signed-digits technique and obtain a factor-2 reduction in table sizes with no time overhead. Thus, SIKE enjoys an average factor-3 reduction in table sizes with negligible overhead in computation time. Overall cycle count benchmarks with our C code can be seen in Table 2.1 in the SIKE specification document [2], in particular in the rows named SIKEpXXX_compressed.

6 Concluding remarks

We have proposed methods to reduce the table sizes in compressed SIKE by a factor of 4, with minimal computational overheads. For $\ell = 2$, new table sizes can range from 18KiB to 240KiB for all SIKE parameter sets and from 34KiB to 477KiB for $\ell = 3$. We have confirmed our theoretical estimates experimentally and our C implementation has been effectively integrated into the official SIKE library.

Acknowledgements The authors would like to thank the SIKE team members for their comments on the paper and support with integrating our results into the official SIKE suite. G. Pereira is supported in part by NSERC, CryptoWorks21, Canada First Research Excellence Fund, Public Works and Government Services Canada, and by the National Research Council Canada and University of Waterloo Collaboration Center (NUCC) program 927517.

References

1. Avizienis, A.: Signed-digit number representations for fast parallel arithmetic. IEEE Transactions on Electronic Computers **EC-10**, 389–400 (1961)
2. 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. Submission to the 3rd Round of the NIST Post-Quantum Standardization project (2020)
3. Azarderakhsh, R., Campagna, M., Costello, C., De Feo, L., Hess, B., Jalali, A., Jao, D., Koziel, B., LaMacchia, B., Longa, P., Naehrig, M., Pereira, G., Renes, J., Soukharev, V., Urbanik, D.: Supersingular isogeny key encapsulation. Submission to the 2nd Round of the NIST Post-Quantum Standardization project (2019)

4. Azarderakhsh, R., Campagna, M., Costello, C., DeFeo, L., Hess, B., Jalali, A., Jao, D., Koziel, B., LaMacchia, B., Longa, P., Naehrig, M., Pereira, G., Renes, J., Soukharev, V., Urbanik, D.: Supersingular Isogeny Key Encapsulation. SIKE Team, `https://sike.org/` (2020)
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. Costello, C., Jao, D., Longa, P., Naehrig, M., Renes, J., Urbanik, D.: Efficient compression of SIDH public keys. In: Advances in Cryptology – Eurocrypt 2017. pp. 679–706. No. 10210 in Lecture Notes in Computer Science, Springer, Paris, France (2017)
7. De Feo, L., Jao, D., Plût, J.: Towards Quantum-Resistant Cryptosystems from Supersingular Elliptic Curve Isogenies. Journal of Mathematical Cryptology pp. 209–247 (2014)
8. Gustavo H. M. Zanon and Marcos A. Simplicio and Geovandro C. C. F. Pereira and Javad Doliskani and Paulo S. L. M. Barreto: Faster Key Compression for Isogeny-Based Cryptosystems. IEEE Transactions on Computers **68**, 688–701 (2018)
9. Longa, P.: SIDH Library. `https://github.com/microsoft/PQCrypto-SIDH`
10. 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. Lecture Notes in Computer Science, vol. 11922, pp. 243–272. Springer (2019). https://doi.org/10.1007/978-3-030-34621-8_9, `https://doi.org/10.1007/978-3-030-34621-8_9`
11. Pereira, G., Barreto, P.: Isogeny-based key compression without pairings. Cryptology ePrint Archive, Report 2021/272 (2021), `http://eprint.iacr.org/2021/272`
12. Pereira, G., Doliskani, J., Jao, D.: x-only point addition formula and faster compressed SIKE. Journal of Cryptographic Engineering pp. 1–13 (2020). https://doi.org/10.1007/s13389-020-00245-4
13. Pohlig, S., Hellman, M.: An Improved Algorithm for Computing Logarithms over $GF(p)$ and its Cryptographic Significance. IEEE Transactions on Information Theory **24**, 106–110 (1978)
14. Rubin, K., Silverberg, A.: Compression in finite fields and torus-based cryptography. SIAM Journal on Computing **37**, 1401–1428 (2008)
15. 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: International Conference on Post-Quantum Cryptography. pp. 248–268. Springer (2018)

A Proof of Theorem 4

This section gives a proof of Theorem 4. To complete the proof, we will need the following lemma.

Lemma 2. *For* $i' \in [0, \ell^w)$*, we have* $i' = \sum_{k'=0}^{w-1} (\lfloor i'/\ell^{k'} \rfloor \mod \ell)\ell^{k'}$*, and for* $j \in [0, w)$ *we have* $\lfloor i'/\ell^j \rfloor = \sum_{k'=j}^{w-1} (\lfloor i'/\ell^{k'} \rfloor \mod \ell)\ell^{k'-j}$.

Proof. Informally, $\lfloor i'/\ell^{k'} \rfloor \mod \ell$ is simply the k'-th digit of i' when i' is written in base ℓ. The first summation is therefore just the base ℓ representation of i'.

Formally, let $i' = \sum_{k'=0}^{w-1} i'_{k'} \ell^{k'}$ be the standard base ℓ representation of i' with each $i'_{k'} \in [0, \ell)$. Then $\lfloor i'/\ell^j \rfloor = \left\lfloor \sum_{k'=0}^{w-1} i'_{k'} \ell^{k'-j} \right\rfloor = \sum_{k'=j}^{w-1} i'_{k'} \ell^{k'-j}$ for any $j \in [0, w)$, and so $\lfloor i'/\ell^j \rfloor \mod \ell = i'_j$.

We now prove Theorem 4, which we restate here for convenience.

Theorem 4. *Let* $\mathbf{L}$ *denote the cost of exponentiation by* ℓ *in* $\mathbb{G}_{\ell,w}$*, let* $\mathbf{E}$ *denote the cost of an equality check in* $\mathbb{G}_{\ell,w}$*, and for* $h \in \mathbb{G}_{\ell,w}$ *let* $Cost_1(h)$ *denote the total cost of running Algorithm 1 with input* h *in terms of* $\mathbf{L}$ *and* $\mathbf{E}$*. For any* $h \in \mathbb{G}_{\ell,w}$*, assume that checking* $h \stackrel{?}{=} 1$ *has cost* 0*.*

Let $h \in \mathbb{G}_{\ell,w}$*. By Theorem 1,* $h = \rho^{v_{j',i'}}$ *for some* j' *and* i'*.*

- *If* $j' = 0$*, then* $Cost_1(h) = 0$*.*
- *If* $1 \le j' \le w$ *and* $0 \le i' \le \ell^{j'-1}(\ell - 1) - 1$*, then* $Cost_1(h) = j'\mathbf{L} + \left(j' + \sum_{j=0}^{j'-1} (\lfloor i'/\ell^j \rfloor \mod \ell) \right) \mathbf{E}$*.*

Consequently, the average case complexity of Algorithm 1 on a uniformly random input is exactly

$$\left(w - \frac{1}{\ell - 1} + \frac{1}{(\ell-1)\ell^w} \right) \mathbf{L} + \left(\frac{w(\ell+1)}{2} - \frac{\ell}{\ell-1} + \frac{1}{\ell^{w-1}(\ell-1)} \right) \mathbf{E},$$

with a worst case cost of $Cost_1(\rho^{v_{w,\ell^{w-1}(\ell-1)-1}}) = w\mathbf{L} + (w\ell - 1)\mathbf{E}$*.*

Proof. For the storage requirement of the algorithm, note that $\sum_{j=0}^{w} |V_j| = |V| = \ell^w$, and so Algorithm 1 precomputes and stores all of the elements of $\mathbb{G}_{\ell,w}$ in the list G. Algorithm 1 also keeps track of $k \le w$ group elements in the list H. Ignoring the two identity elements in G and H, Algorithm 1 requires to store at most $\ell^w + w - 2$ group elements.

We prove the computational complexity statements now. Note that the worst case complexity statement follows from Lemma 2 by taking $j' = w$ and $i' = (\ell - 1)\ell^{w-1} - 1$ in the cost formula given for $Cost_1(\rho^{v_{j',i'}})$ in the theorem.

Theorem 1 guarantees that h can be written in the form $\rho^{v_{j',i'}}$ with either $j' = i' = 0$, or $j' = 1$ and $0 \le i' \le \ell - 2$, or $2 \le j' \le w$ and $0 \le i' \le \ell^{j'-1}(\ell-1)-1$. We examine each case individually to determine the number of operations used by Algorithm 1 when ran with input $\rho^{v_{j',i'}}$.

When $j' = i' = 0$, we have $h = \rho^{v_{0,0}} = \rho^{\ell^w} = 1$ since ρ has order ℓ^w. Algorithm 1 then skips the loop on line 3 and returns 0 on line 7 at no cost.

Suppose $j' = 1$ and $0 \le i' \le \ell - 2$. By Theorem 1, we have that $|h| = |\rho^{v_{j',i'}}| = |g_{j',i'}| = \ell^{j'} = \ell$. Algorithm 1 then executes loop 3 exactly once at a cost of $1\mathbf{L} = j'\mathbf{L}$ (we've assumed the associated equality checks have negligible cost), and we have $H = [1, h]$ and $k = 1$. Loop 9 then executes until $H[1] = G[1][i]$, if and only if $h = g_{1,i}$, if and only if $\rho^{v_{1,i'}} = \rho^{v_{1,i}}$, if and only if $i' = i$. Therefore

exactly $i' + 1 = \sum_{j=0}^{1-1}(\lceil i'/\ell^j \rceil \mod \ell)$ equality checks are used. Since $k = 1$, loop 16 is skipped and the algorithm terminates on line 27.

Suppose now that $2 \leq j' \leq w$ and $0 \leq i' \leq \ell^{j'-1}(\ell - 1) - 1$. By Theorem 1 $|h| = \ell^{j'}$, and so loop 3 executes exactly j' times at a cost of $j'\mathbf{L}$, ending with $H = [1, h^{\ell^{j'-1}}, \ldots, h^{\ell}, h]$ and $k = j'$. By Theorem 1, we have $h^{\ell} = g_{j',i'}^{\ell} = g_{j'-1,\lfloor i'/\ell \rfloor}$, $h^{\ell^2} = g_{j'-1,\lfloor i'/\ell \rfloor}^{\ell} = g_{j'-2,\lfloor \lfloor i'/\ell \rfloor/\ell \rfloor} = g_{j'-2,\lfloor i'/\ell^2 \rfloor}$, and so on. In this manner, the elements of H can then be written as

$$H[j] = h^{\ell^{j'-j}} = g_{j,\lfloor i'/\ell^{j'-j} \rfloor} = \rho^{v_{j,\lfloor i'/\ell^{j'-j} \rfloor}}. \tag{8}$$

Next loop 9 is executed, which iterates until $H[1] = G[1][i]$, if and only if $\rho^{v_{1,\lfloor i'/\ell^{j'-1} \rfloor}} = \rho^{v_{1,i}}$, if and only if $\lfloor i'/\ell^{j'-1} \rfloor = i$. The cost incurred by loop 9 is then $(\lfloor i'/\ell^{j'-1} \rfloor + 1)\mathbf{E}$, and the variable i_j is initialized to value $i_j = \lfloor i'/\ell^{j'-1} \rfloor$.

The initial iteration $j = 2$ of loop 16 updates i_j to $i_j = \lfloor i'/\ell^{j'-1} \rfloor \ell$ and loop 18 runs until $H[2] = G[2][i_j + s]$, if and only if $\rho^{v_{2,\lfloor i'/\ell^{j'-2} \rfloor}} = \rho^{v_{2,i_j+s}}$, if and only if

$$\begin{aligned} s &= \lfloor i'/\ell^{j'-2} \rfloor - i_j = \lfloor i'/\ell^{j'-2} \rfloor - \lfloor i'/\ell^{j'-1} \rfloor \ell \\ &= \sum_{k'=j'-2}^{w-1} (\lfloor i'/\ell^{k'} \rfloor \mod \ell)\ell^{k'-(j'-2)} - \ell \sum_{k'=j'-1}^{w-1} (\lfloor i'/\ell^{k'} \rfloor \mod \ell)\ell^{k'-(j'-1)} \\ &= \lfloor i'/\ell^{j'-2} \rfloor \mod \ell, \end{aligned}$$

where we've used two applications of Lemma 2. For $j = 2$, loop 18 therefore incurs cost $(\lfloor i'/\ell^{j'-2} \rfloor \mod \ell + 1)\mathbf{E}$ and updates variable i_j to

$$\begin{aligned} i_j &= \lfloor i'/\ell^{j'-1} \rfloor \ell + s = \lfloor i'/\ell^{j'-1} \rfloor \ell + (\lfloor i'/\ell^{j'-2} \rfloor \mod \ell) \\ &= \sum_{k'=0}^{1} (\lfloor i'/\ell^{j'+k'-2} \rfloor \mod \ell)\ell^{k'}. \end{aligned}$$

For the remaining iterations of loop 16, we prove the following claim.

> **Claim**: For $2 \leq j \leq k = j'$, the j-th iteration of loop 16 incurs cost $(\lfloor i'/\ell^{j'-j} \rfloor \mod \ell + 1)\mathbf{E}$ and updates variable i_j to value $i_j = \sum_{k'=0}^{j-1} (\lfloor i'/\ell^{j'+k'-j} \rfloor \mod \ell)\ell^{k'}$

We prove the claim using induction on j, with the base case $j = 2$ already given above. Assume the claim holds after the j-th iteration. The $(j+1)$-th iteration begins by updating i_j to $i_j = \ell \sum_{k'=0}^{j-1} (\lfloor i'/\ell^{j'+k'-j} \rfloor \mod \ell)\ell^{k'}$. As before, loop 18

continues until

$$\begin{aligned}
s &= \lfloor i'/\ell^{j'-(j+1)} \rfloor - i_j = \lfloor i'/\ell^{j'-(j+1)} \rfloor - \sum_{k'=0}^{j-1} (\lfloor i'/\ell^{j'+k'-j} \rfloor \mod \ell)\ell^{k'+1} \\
&= \sum_{k'=0}^{w-1} (\lfloor i'/\ell^{j'+k'-(j+1)} \rfloor \mod \ell)\ell^{k'} - \sum_{k'=0}^{j-1} (\lfloor i'/\ell^{j'+k'-j} \rfloor \mod \ell)\ell^{k'+1} \\
&= (\lfloor i'/\ell^{j'-(j+1)} \rfloor \mod \ell) + \sum_{k'=1}^{w-1} (\lfloor i'/\ell^{j'+k'-(j+1)} \rfloor \mod \ell)\ell^{k'} - \sum_{k'=1}^{j} (\lfloor i'/\ell^{j'+k'-(j+1)} \rfloor \mod \ell)\ell^{k'} \\
&= \lfloor i'/\ell^{j'-(j+1)} \rfloor \mod \ell,
\end{aligned}$$

where we've again used Lemma 2, and since $i' \in [0, \ell^{j'})$ the last two summations cancel entirely. The iteration therefore incurs cost $(\lfloor i'/\ell^{j'-(j+1)} \rfloor \mod \ell + 1)\mathbf{E}$, and i_j updates to $i_j = \lfloor i'/\ell^{j'-(j+1)} \rfloor \mod \ell + \ell \sum_{k'=0}^{j-1} (\lfloor i'/\ell^{j'+k'-j} \rfloor \mod \ell)\ell^{k'} = \sum_{k'=0}^{j} (\lfloor i'/\ell^{j'+k'-(j+1)} \rfloor \mod \ell)\ell^{k'}$. This proves the claim.

By the claim just proved, loop 16 completes on iteration $j = k = j'$ with $i_j = \sum_{k'=0}^{j'-1} (\lfloor i'/\ell^{k'} \rfloor \mod \ell)\ell^{k'} = i'$ and the algorithm returns on line 27. Notice that by the end of the algorithm we have $(k, i_j) = (j', i')$ and the variable d has computed $v_{j',i'} = \log_\rho(h)$, and so we have a correctness verification of the algorithm. The total cost used by the algorithm is

$$\begin{aligned}
Cost_1(h) &= j'\mathbf{L} + (\lfloor i'/\ell^{j'-1} \rfloor + 1)\mathbf{E} + \sum_{j=2}^{j'} (\lfloor i'/\ell^{j'-j} \rfloor \mod \ell + 1)\mathbf{E} \\
&= j'\mathbf{L} + \left(j' + \sum_{j=0}^{j'-1} (\lfloor i'/\ell^{j} \rfloor \mod \ell) \right) \mathbf{E},
\end{aligned}$$

as claimed by the theorem.

We may now compute the average case complexity of Algorithm 1 as

$$\begin{aligned}
&\sum_{h\in\mathbb{G}_{\ell,w}} \Pr[h]\cdot Cost_1(h) = \frac{1}{\ell^w}\left(\sum_{j'=1}^{w}\sum_{i'=0}^{\ell^{j'-1}(\ell-1)-1} Cost_1(\rho^{v_{j',i'}})\right)\\
&= \frac{1}{\ell^w}\left(\sum_{j'=1}^{w}\sum_{i'=0}^{\ell^{j'-1}(\ell-1)-1} j'\mathbf{L} + \left(j' + \sum_{j=0}^{j'-1}(\lfloor i'/\ell^j\rfloor \mod \ell)\right)\mathbf{E}\right)\\
&= \frac{1}{\ell^w}\left(\sum_{j'=1}^{w}\sum_{i'=0}^{\ell^{j'-1}(\ell-1)-1} j'(\mathbf{L}+\mathbf{E}) + \sum_{j'=1}^{w}\sum_{i'=0}^{\ell^{j'-1}(\ell-1)-1}\sum_{j=0}^{j'-1}(\lfloor i'/\ell^j\rfloor \mod \ell)\mathbf{E}\right)\\
&= \frac{1}{\ell^w}\left(\frac{w\ell^{w+1}-(w+1)\ell^w+1}{\ell-1}(\mathbf{L}+\mathbf{E}) + \sum_{j'=1}^{w}\sum_{i'=0}^{\ell^{j'-1}(\ell-1)-1}\sum_{j=0}^{j'-1}(\lfloor i'/\ell^j\rfloor \mod \ell)\mathbf{E}\right).
\end{aligned}$$

The value of the remaining summations above will be computed through the following lemma. After substitution, one may simplify the resulting expression to that which is stated in Theorem 4. This will conclude the proof of the theorem.

Lemma 3. *The expression* $\sum_{i'=0}^{\ell^{j'-1}-1}\sum_{j=0}^{j'-2}(\lfloor i'/\ell^j\rfloor \mod \ell)$ *computes the sum of the digit sum of all numbers* $i' \in [0, \ell^{j'-1})$ *written in base* ℓ*, and equals* $(\ell-1)(j'-1)\ell^{j'-1}/2$*. Consequently, we have*

$$\sum_{i'=0}^{(\ell-1)\ell^{j'-1}-1}\sum_{j=0}^{j'-1}(\lfloor i'/\ell^j\rfloor \mod \ell) = \frac{(\ell-1)(j'(\ell-1)-1)\ell^{j'-1}}{2}.$$

Proof. That the expression computes the sum of the digit sums follows from Lemma 2; we compute the value of this expression using this fact. Suppose that ℓ is even and for simplicity write the base ℓ digits of any i' as $i_{j'-2}\cdots i_0$, using leading 0's if necessary. Then the number with base ℓ digits $(\ell-1-i_{j'-2})\cdots(\ell-1-i_0)$ is another distinct integer in $[0,\ell^{j'-1})$, and the sum of the digits of these two integers is $(\ell-1)(j'-1)$. In this way, every integer has a unique corresponding complement, and so the sum of the digit sums is $(\ell-1)(j'-1)\ell^{j'-1}/2$. When ℓ is odd the same argument works, except that the integer with digits $((\ell-1)/2)\cdots((\ell-1)/2)$ is its own complement and is the only number with this property. The sum of this number's digits is $(\ell-1)(j'-1)/2$, and so the sum of the digit sums is then $(\ell-1)(j'-1)(\ell^{j'-1}-1)/2+(\ell-1)(j'-1)/2 = (\ell-1)(j'-1)\ell^{j'-1}/2$ as before.

Lastly, we compute the final statement of the lemma as

$$\begin{aligned}
&\sum_{i'=0}^{(\ell-1)\ell^{j'-1}-1}\sum_{j=0}^{j'-1}(\lfloor i'/\ell^j\rfloor \mod \ell) = \sum_{k=1}^{\ell-1}\sum_{i'=(k-1)\ell^{j'-1}}^{k\ell^{j'-1}-1}\sum_{j=0}^{j'-1}(\lfloor i'/\ell^j\rfloor \mod \ell)\\
&= \sum_{k=1}^{\ell-1}\sum_{i'=0}^{\ell^{j'-1}-1}\sum_{j=0}^{j'-1}\left(\left\lfloor \frac{i'+(k-1)\ell^{j'-1}}{\ell^j}\right\rfloor \mod \ell\right)\\
&= \sum_{k=1}^{\ell-1}\sum_{i'=0}^{\ell^{j'-1}-1}\sum_{j=0}^{j'-1}\left(\left(\left\lfloor \frac{i'}{\ell^j}\right\rfloor + (k-1)\ell^{j'-j-1}\right) \mod \ell\right)\\
&= \sum_{k=1}^{\ell-1}\sum_{i'=0}^{\ell^{j'-1}-1}\left(k-1+\sum_{j=0}^{j'-2}\left(\left\lfloor \frac{i'}{\ell^j}\right\rfloor\right) \mod \ell\right)\\
&= \frac{\ell^{j'-1}(\ell-2)(\ell-1)}{2} + \sum_{k=1}^{\ell-1}\sum_{i'=0}^{\ell^{j'-1}-1}\sum_{j=0}^{j'-2}\left(\left\lfloor \frac{i'}{\ell^j}\right\rfloor \mod \ell\right)\\
&= \frac{\ell^{j'-1}(\ell-2)(\ell-1)}{2} + \sum_{k=1}^{\ell-1}\frac{(\ell-1)(j'-1)\ell^{j'-1}}{2}\\
&= \frac{\ell^{j'-1}(\ell-2)(\ell-1)}{2} + \frac{(\ell-1)^2(j'-1)\ell^{j'-1}}{2} = \frac{(\ell-1)(j'(\ell-1)-1)\ell^{j'-1}}{2}.
\end{aligned}$$

B Algorithms

Algorithm 1: Discrete Logarithm Computation in $\mathbb{G}_{\ell,w}$

Parameters: ℓ (prime), $w \geq 1$, a precomputed table
$G[j][i] = g_{j,i} = \rho^{v_{j,i}} \in \mathbb{G}_{\ell,w}$ for $j \in [1, w], v_{j,i} \in V_j$.
Input : $h \in \mathbb{G}_{\ell,w}$
Output : $d \in [0, \ell^w)$ such that $h = \rho^d$

$H = [h]$
$k \leftarrow 0$
while $H[0] \neq 1$ **do** `// Exponentiate until reaching root of` $\mathcal{G}_{\ell,w}$
  $H.insert(0, H[0]^\ell)$
  $k \leftarrow k + 1$
end
if $k = 0$ **then** return 0
$d \leftarrow 1$
for $i = 0$ **to** $\ell - 2$ **do** `// Initial branch of` $\mathcal{G}_{\ell,w}$
  if $H[1] = G[1][i]$ **then**
    $i_j \leftarrow i$
    $d \leftarrow (i_j + 1)$
    break
  end
end
for $j = 2$ **to** k **do** `// Backtrace path taken through` $\mathcal{G}_{\ell,w}$
  $i_j \leftarrow \ell \cdot i_j$
  for $s = 0$ **to** $(\ell - 1)$ **do**
    if $H[j] = G[j][i_j + s]$ **then**
      $d \leftarrow d + s \cdot \ell^{j-1}$
      $i_j \leftarrow i_j + s$
      break
    end
  end
  $d \leftarrow \ell^{w-k} \cdot d$
end
return d

Algorithm 2: Discrete Logarithm Computation in $\mathbb{G}_{2,w}$ (a variant of Algorithm 1 for $\ell = 2$)

Parameters: $\ell = 2$, $w \geq 1$, a precomputed table G such that $G[0], G[1]$ are null, $G[2][0] = g_{2,0} = \rho^{v_{2,0}} = \rho^{2^{w-2}}$, and $G[j][i_j] = g_{j,2i_j} = \rho^{v_{j,2i_j}} \in \mathbb{G}_{2,w}$ for $j \in [3, w]$, $i_j = 0, \ldots, |V_j|/4 - 1$.
Input : $h \in \mathbb{G}_{2,w}$
Output : $d \in [0, 2^w)$ such that $h = \rho^d$

$H = [h]$
$k \leftarrow 0$
while $H[0] \neq 1$ **do** `// Square until reaching root of` $\mathcal{G}_{2,w}$
  $H.insert(0, H[0]^2)$
  $k \leftarrow k + 1$
end
if $k = 0$ **then** return 0
if $k = 1$ **then** return 2^{w-1} `// Initial 'branch' of` $\mathcal{G}_{2,w}$
$d \leftarrow 1$
$inv \leftarrow False$
if $H[2] \neq G[2][0]$ **then** `// Force path to lie in correct half of` $\mathcal{G}_{2,w}$
  $H[i] \leftarrow H[i]^{-1}$ for $i = 2, \ldots, k$
  $inv \leftarrow True$
end
$i_j \leftarrow 0$
for $j = 3$ **to** k **do** `// Backtrace path taken through` $\mathcal{G}_{2,w}$
  $i_j \leftarrow 2 \cdot i_j$
  if $H[j] \neq G[j][i_j/2]$ **then**
    $d \leftarrow d + 2^{j-1}$
    $i_j \leftarrow i_j + 1$
  end
end
$d \leftarrow 2^{w-k} \cdot d$
if $inv = True$ **then**
  $d \leftarrow 2^w - d$
end
return d

Algorithm 3: Discrete Logarithm Computation in $\mathbb{G}_{\ell,w}$ (a variant of Algorithm 1 for $\ell > 2$)

Parameters: $\ell > 2$, $w \geq 1$, a precomputed table G such that $G[0]$ is null,
$G[1][i] = g_{1,i} = \rho^{v_{1,i}}$ for $i = 0, \ldots, (\ell-1)/2 - 1$,
$G[j][b_j(\ell-1)+s] = g_{j,b_j\ell+s} = \rho^{v_{j,b_j\ell+s}}$, for $j \in [2, w]$,
$b_j = 0, \ldots, \ell^{j-2}(\ell-1)/2 - 1$, $s = 0, \ldots, (\ell-1) - 1$.
Input : $h \in \mathbb{G}_{\ell,w}$
Output : $d \in [0, \ell^w)$ such that $h = \rho^d$

$H = [h]$
$k \leftarrow 0$
while $H[0] \neq 1$ **do** `// Exponentiate until reaching root of` $\mathcal{G}_{\ell,w}$
  $H.insert(0, H[0]^\ell)$
  $k \leftarrow k + 1$
end
if $k = 0$ **then** return 0
$d \leftarrow 1$
$inv \leftarrow False$
for $i = 0$ **to** $(\ell-1)/2 - 1$ **do** `// Initial branch of` $\mathcal{G}_{\ell,w}$
  if $H[1] = G[1][i]$ **then**
    $i_j \leftarrow i$
    $d \leftarrow (i_j + 1)$
    break
  else if $H[1] = G[1][i]^{-1}$ **then**
    $i_j \leftarrow i$
    $d \leftarrow (i_j + 1)$
    $inv \leftarrow True$
    break
  end
end
if $inv = True$ **then** $H[i] \leftarrow H[i]^{-1}$ for $i = 2, \ldots, k$
for $j = 2$ **to** k **do** `// Backtrace path taken through` $\mathcal{G}_{\ell,w}$
  $i_j \leftarrow \ell \cdot i_j$
  $cont \leftarrow True$
  for $s = 0$ **to** $(\ell-1) - 1$ **do**
    $b \leftarrow i_j/\ell$
    if $H[j] = G[j][b(\ell-1)+s]$ **then**
      $d \leftarrow d + s \cdot \ell^{j-1}$
      $i_j \leftarrow i_j + s$
      $cont \leftarrow False$
      break
    end
  end
  if $cont = True$ **then**
    $d \leftarrow d + (\ell-1) \cdot \ell^{j-1}$
    $i_j \leftarrow i_j + (\ell-1)$
  end
  $d \leftarrow \ell^{w-k} \cdot d$
end
if $inv = True$ **then** $d \leftarrow \ell^w - d$
return d

Algorithm 4: 2^w Discrete Logarithm Computation

Parameters: $w \geq 2$; $\rho \in \mathbb{G}_{\ell,w}$ of order 2^w; precomputed table $T^{\mathsf{exp}}[j][i] = b_{2^j+i} \in \mathbb{F}_p$ for $j \in [0, w-3], i \in [0, 2^j)$; precomputed table Log of size 2^w such that $\rho^{Log[k]} = [a_k : 1]$.
Input : $[x : y]$ such that $[x : y] = [a_k : 1]$ for some $k \in [1, 2^w)$, or $[x : y] = [1 : 0]$.
Output : Digit $D \in [-2^{w-1}, 2^{w-1}]$ such that $[x : y] = \rho^D$.

if $y = 0$ **then** return $Log[0]$.
if $x = 0$ **then** return $Log[1]$.
if $x = y$ **then** return $Log[2]$.
if $x = -y$ **then** return $Log[3]$.
Initialize an empty list $prods$.
for $j = 2$ **to** $w - 1$ **do**
  for $i = 0$ **to** $2^{j-1} - 1$ **do**
    Let $(\beta_{j-2}, \ldots, \beta_0)$ be the bits such that
    $i = \beta_{j-2}2^{j-2} + \cdots + \beta_1 2 + \beta_0$.
    if $\beta_0 = 0$ **then**
      $prods[2^{j-2} + i/2] \leftarrow y \cdot T^{\mathsf{exp}}[j-2][i/2]$.
      $sum \leftarrow y$.
    end
    for $k = 0$ **to** $j - 2$ **do**
      $sum \leftarrow sum + (-1)^{\beta_{j-k-2}} prods[2^k + (\beta_{j-2} \cdots \beta_{j-k-1})_2]$.
    end
    if $x = sum$ **then** return $Log[2^j + i]$.
    if $x = -sum$ **then** return $Log[2^{j+1} - i - 1]$.
  end
end

Algorithm 5: A hybrid of Algorithms 2 and 4 for computing discrete logarithms in $\mathbb{G}_{\ell,w}$.

Parameters: $\ell = 2$, $w \geq 2$; precomputed table G such that $G[0], G[1]$ are null, $G[2][0] = \rho^{2^w-2}$, $G[j][i_j] = \rho^{v_{j,2i_j}}$ for $j \in [3, w]$, $i_j \in [0, |V_j|/4)$; w_1 with $2 \leq w_1 < w$; precomputed table $T^{\mathsf{exp}}[j][i] = b_{2^j+i}$ for $j \in [0, w_1 - 3], i \in [0, 2^j)$; precomputed table Log of size 2^{w_1} such that $\rho^{Log[k]} = [a_k : 1]$; $w_2 = w - w_1$.
Input : $h \in \mathbb{G}_{2,w}$.
Output : $d \in [0, 2^w)$ such that $h = \rho^d$.

$H \leftarrow [h]$; $k \leftarrow 0$
for $i = 1$ to w_2 **do**
  if $H[0] \neq 1$ **then**
    $H.insert(0, H[0]^2)$
    $k \leftarrow k + 1$
  else
    break
  end
end
if $H[0] = 1$ and $k \leq w_1$, **then** return $2^{w_2} \cdot$ Algorithm4(h)
if $H[0] \neq 1$ **then**
  $d \leftarrow$ Algorithm4($H[0]$) mod 2^{w_1}
  $Hindex \leftarrow 0$
else
  $d \leftarrow$ Algorithm4($H[w_1]$) mod 2^{w_1}
  $Hindex \leftarrow w_1$
end
$ord \leftarrow w_1 - t$, where $2^t \mid d$ and $2^{t+1} \nmid d$ with $t \geq 0$
$tmp \leftarrow (d/2^{w_1-t} - 1)/2$ // tmp has at most $ord - 1$ bits
Let $(\beta_{ord-2} \cdots \beta_0)$ be the bits such that $tmp = \beta_{ord-2}2^{ord-2} + \cdots + \beta_1 2 + \beta_0$
$i_j \leftarrow \beta_0 2^{ord-2} + \cdots + \beta_{ord-3}2 + \beta_{ord-2}$; $inv \leftarrow False$
if $H[0] = -1$ **then**
  if $H[1] \neq G[2][0]$ **then**
    $H \leftarrow [H[i]^{-1} : i \in [0, k]]$; $inv \leftarrow True$
  end
else
  if $i_j \geq 2^{ord-2}$ **then**
    $i_j \leftarrow 2^{ord-1} - i_j - 1$; $d \leftarrow 2^{w_1} - d$
    $H \leftarrow [H[i]^{-1} : i \in [0, k]]$; $inv \leftarrow True$
  end
end
$d \leftarrow 2^{w_2} d$
for $j = Hindex + 1$ to k **do**
  $i_j \leftarrow 2i_j$; $d \leftarrow d/2$
  if $H[j] \neq G[j - Hindex + ord + 1][i_j/2 + 1]$ **then**
    $d \leftarrow d + 2^{w-1}$; $i_j \leftarrow i_j + 1$
  end
end
if inv **then** $d \leftarrow 2^w - d$
return d