ADVANCED SIDH PROTOCOLS

DAVID JAO

1. Key Compression

In these notes we present the idea of *key compression* and the mathematics behind it. Key compression is a technique used in SIDH to reduce the size of public keys. It was introduced in Azarderakhsh et al. [1] and further refined by Costello et al. [9], Zanon et al. [3], Naehrig and Renes [2], and Pereira et al. [4].

We recall the basic operation of SIDH. In what follows, we use the notation $\langle g_1, g_2, \ldots \rangle$ to denote the subgroup generated by $g_1, g_2, \ldots$ in an abelian group. Two parties Alice and Bob agree on a set of public parameters consisting of:

- A prime $p = 2^e 3^f - 1$, balanced so that $2^e \approx 3^f$,
- A trace 0 supersingular elliptic curve $E/\mathbb{F}_{p^2} : y^2 = x^3 + Ax^2 + x$, in Montgomery form,
- Points $P_a, Q_a \in E[2^e]$ which generate $E[2^e]$ as an abelian group (so that $E[2^e] = \langle P_a, Q_a \rangle$),
- Points $P_b, Q_b \in E[3^f]$ which generate $E[3^f]$ as an abelian group (so that $E[3^f] = \langle P_b, Q_b \rangle$).

Under the above conditions, P_a, Q_a, P_b, Q_b are all defined over $\mathbb{F}_{p^2}$. To perform key exchange, Alice:

- Selects a secret key $n_a \leftarrow \mathbb{Z}/2^e\mathbb{Z}$,
- Computes the unique (up to isomorphism) isogeny $\phi_a \colon E \to E_a$ having kernel $\langle P_a + n_a Q_a \rangle$,
- Constructs the public key $(A_a, \phi_a(P_b), \phi_a(Q_b))$, where A_a is the Montgomery coefficient of the curve $E_a : y^2 = x^3 + A_a x^2 + x$ computed previously, and sends it to Bob.

Similarly, Bob selects a secret key $n_b \leftarrow \mathbb{Z}/3^f\mathbb{Z}$, constructs the public key $(A_b, \phi_b(P_a), \phi_b(Q_a))$, and sends it to Alice. We now consider the two isogenies

$$\phi'_a \colon E_b \to E'_b$$
$$\phi'_b \colon E_a \to E'_a$$

wherein

$$\ker \phi'_a = \langle \phi_b(P_a) + n_a \phi_b(Q_a) \rangle = \langle \phi_b(P_a + n_a Q_a) \rangle$$
$$\ker \phi'_b = \langle \phi_a(P_b) + n_b \phi_a(Q_b) \rangle = \langle \phi_a(P_b + n_b Q_b) \rangle$$

and observe that

$$\ker \phi'_b \circ \phi_a = \langle P_a + n_a Q_A, P_b + n_b Q_b \rangle = \ker \phi'_a \circ \phi_b,$$

so that $E'_a \cong E'_b$. Note that Alice (and only Alice) has the information needed to compute ϕ'_a, and Bob (and only Bob) has the information needed to compute ϕ'_b, which explains our choice of subscripts for these isogenies. Normalizing curves so that $E'_a = E'_b$, we now have $\phi'_b \circ \phi_a = \phi'_a \circ \phi_b$, so we thus obtain a commutative diagram

$$\begin{tikzcd} E \arrow[r, "\phi_a"] \arrow[d, "\phi_b"'] & E_a \arrow[d, "\phi'_b"] \\ E_b \arrow[r, "\phi'_a"'] & E_{ab} \end{tikzcd}$$

where E_{ab} denotes the common curve $E'_a = E'_b$. We remark that if Vélu's formulas [5] are used to construct the above isogenies, then the resulting curves automatically satisfy $E'_a = E'_b$, because Vélu's formulas preserve the invariant differential. Regardless of whether $E'_a = E'_b$, we always have $j(E'_a) = j(E'_b) = j(E_{ab})$, and this common value can be used to derive a shared secret for Alice and Bob (for example, by hashing it).

Date: July 30, 2021.

In the above description, a public key (such as Alice's) consists of the data $(A_a, \phi_a(P_b), \phi_a(Q_b))$ for

$$\begin{aligned} A_a &\in \mathbb{F}_{p^2} \\ \phi_a(P_b) &\in E_a(\mathbb{F}_{p^2}) \\ \phi_a(Q_b) &\in E_a(\mathbb{F}_{p^2}) \end{aligned}$$

Letting k denote $\lceil \log_2 p \rceil$, we see that A_a, $\phi_a(P_b)$, and $\phi_a(Q_b)$ require $2k$, $4k$, and $4k$ bits respectively in their binary representations. Standard elliptic curve point compression methods (e.g. omitting the y-coordinate) can reduce the storage requirements for $\phi_a(P_b)$ and $\phi_a(Q_b)$ to $2k$ bits each, leading to a key size of $6k$ bits in total. In practice, for efficiency reasons, a SIKE [6] key is actually represented as $(x(\phi_a(P_b)), x(\phi_a(Q_b)), x(\phi_a(P_b - Q_b))$, but the storage cost of this representation is still about $6k$ bits.

The basic idea behind key compression is that each of the values A_a, $\phi_a(P_b)$, and $\phi_a(Q_b)$ actually has much less than $2k$ bits of information-theoretic entropy, so by finding alternative representations of these values, we can reduce the storage and transmission cost of a public key. In fact, each of A_a, $\phi_a(P_b)$, and $\phi_a(Q_b)$ could in principle be represented using only k bits:

- A_a defines an isomorphism class of supersingular elliptic curves in characteristic p, and the number of such isomorphism classes of curves is almost exactly $\frac{p+1}{12} \approx O(p)$.
- $\phi_a(P_b)$ and $\phi_a(Q_b)$ are elements of $E[3^f] \cong (\mathbb{Z}/3^f\mathbb{Z})^2$, which has cardinality 3^{2f}. Given $p = 2^e 3^f - 1$, and assuming $2^e \approx 3^f$, we have $3^{2f} \approx O(p)$.

In reality, techniques for key compression have focused exclusively on finding smaller and more efficiently computable representations of $\phi_a(P_b)$ and $\phi_a(Q_b)$, since it is not known at this time how to represent a supersingular isomorphism class over $\mathbb{F}_{p^2}$ efficiently using anything non-negligibly fewer than $2k$ bits. (The latter problem has been an open problem for at least several decades, and any progress in that direction would be welcome. Note that it is not necessary to get all the way down to the theoretical minimum of k bits; even a small amount of partial progress would be an improvement.)

2. Prelude: Optimal strategies

A large portion of the literature on key compression is concerned with finding mathematical algorithms to speed up implementation of compression techniques. As a warm-up for this topic, and also because optimal strategies are an important part of SIDH implementation that to the best of my knowledge has not yet been covered in this school, we devote some attention to the topic of optimal strategies in this section. Note that optimal strategies are also used in key compression (cf. Section 5.2). Most of this material is taken from the SIKE spec [6].

The core computational task in SIDH can be phrased as follows: Given

- $p = 2^e 3^f - 1$,
- A trace 0 supersingular elliptic curve $E/\mathbb{F}_{p^2}$,
- A point S of order dividing ℓ^e, where ℓ is some small integer, either 2 (for Alice) or 3 (for Bob),
- A (possibly empty) list of points $P_1, P_2, \ldots \in E$,

find the codomain $E/\langle S \rangle$ of an isogeny $\psi \colon E \to E/\langle S \rangle$, along with the values $\psi(P_1), \psi(P_2), \ldots$ if required. Vélu's formulas do indeed produce the rational equations for an isogeny $E \to E/\langle S \rangle$, but applying the formulas directly to S yields an enormous cost of over ℓ^e operations. Recent breakthroughs in isogeny computation [8] improve this cost to $O(\ell^{e/2})$, but this cost is still too large to allow for direct computation. Therefore, instead of using Vélu's formulas directly, we compose a sequence of isogenies

$$E \xrightarrow{\psi_1} E_1 = E/\langle \ell^{e-1} S \rangle \xrightarrow{\psi_2} E_2 = E_1/\langle \psi_1(\ell^{e-2} S) \rangle \xrightarrow{\psi_3} E_3 = E_2/\langle \psi_2(\psi_1(\ell^{e-3} S)) \rangle \longrightarrow \ldots$$

where each isogeny ψ_i has degree ℓ (or less, in the case where S does not have full order); the composition $\psi_e \circ \psi_{e-1} \circ \cdots \circ \psi_2 \circ \psi_1$ is then equal to the desired isogeny $\psi \colon E \to E/\langle S \rangle$, while being much easier to compute. Algorithm 1 depicts this method in pseudocode.

This approach, though polynomial-time, is relatively inefficient in practice. The most expensive part is the computation of the point $[\ell^{e-i}]S$ in step 4. Indeed, each such computation requires (up to) $e - 1$ scalar multiplications by ℓ, and the computation is repeated e times, for a total of $O(e^2)$ *elementary operations.*

Algorithm 1 Naive isogeny computation

```
Given: $p, E, S, (P_1, P_2, \dots)$
1: $(x_1, x_2, \dots) \leftarrow (P_1, P_2, \dots)$
2: $E_0 \leftarrow E$
3: for $i = 1$ to $e$ do
4:     compute $\psi_i \colon E_{i-1} \to E'$ having kernel $\langle \ell^{e-i} S \rangle$
5:     $E_i \leftarrow E'$
6:     $S \leftarrow \psi_i(S)$
7:     $(x_1, x_2, \dots) \leftarrow (\psi_i(x_1), \psi_i(x_2), \dots)$
8: return $E_e, (x_1, x_2, \dots)$
```

In optimized implementations, following [7], it is recommended to replace Algorithm 1 by a recursive decomposition requiring only $O(e \log e)$ elementary multiplications, and a similar number of evaluations of ℓ-isogenies. We call such a decomposition a *computational strategy*, and we describe it by a full binary tree on $e - 1$ nodes[1]. If we draw such trees so that all nodes lie within a triangular region of a hexagonal lattice, with all leaves on one border, then the path length of the tree is proportional to the computational effort required by the strategy. See Figure 1 for an example, and [7, §4] for a more formal definition.

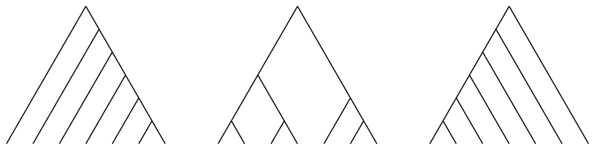
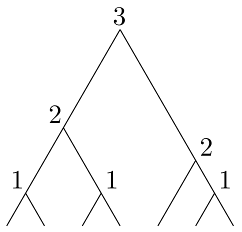


FIGURE 1. Three computational strategies of size $e - 1 = n$. The simple approach used in Algorithm 1 corresponds to the leftmost strategy.

In practice, we represent any full binary tree on $e-1$ nodes in the following way: associate to any internal node the number of leaves to its right, then walk the tree in depth-first left-first order and output the labels as they are encountered. See Figure 2 for an example.



Linearization: $(3, 2, 1, 1, 2, 1)$

FIGURE 2. Linear representation of a strategy on 6 nodes.

Note that the length of the linearlization is one less than the number of leaves in the tree. Given any full binary tree represented linearly as a list $(s_1, s_2, \dots, s_{t-1})$, the computation in Algorithm 1 can be replaced by the recursive procedure of Algorithm 2.

Exercise 1. Download Craig Costello's SAGE implementation of SIDH (`https://github.com/microsoft/SIKE-challenges`), which by default uses Algorithm 1, and modify it to perform "`SIKEp434`"[2] (the default parameter choice) using Algorithm 2, and using the following strategies given in the SIKE specification [6]:

- S_4 = [48, 28, 16, 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 13, 7, 4, 2, 1, 1, 2, 1, 1, 3, 2, 1, 1, 1, 1, 5, 4, 2, 1, 1, 2, 1, 1, 2, 1, 1, 1, 21, 12, 7, 4, 2, 1, 1, 2, 1, 1, 3, 2, 1, 1, 1, 1, 5, 3, 2, 1, 1, 1, 1, 2, 1, 1, 1, 9, 5, 3, 2, 1, 1, 1, 1, 2, 1, 1, 1, 4, 2, 1, 1, 1, 2, 1, 1]

[1]Recall a *full* binary tree on n nodes is a binary tree with exactly n nodes of degree 2 and $n + 1$ nodes (leaves) of degree 0.

[2]Note that the name `SIKEp434` is a misnomer in this setting, since the script actually performs SIDH, not SIKE.

Algorithm 2 Optimized isogeny computation

```
Given: $p, E, S, (P_1, P_2, \ldots), (s_1, s_2, \ldots, s_{t-1})$
 1: if $t = 1$ (i.e. the list is empty) then
 2:     compute an $\ell$-isogeny $\psi\colon E \to E'$ of kernel $\langle S \rangle$
 3:     return $(E', (\phi(P_1), \phi(P_2), \ldots))$
 4: $n \leftarrow s_1$
 5: $L \leftarrow (s_2, \ldots, s_{t-n})$
 6: $R \leftarrow (s_{t-n+1}, \ldots, s_{t-1})$
 7: $T \leftarrow \ell^n S$
 8: $(E, (U, P_1, P_2, \ldots)) \leftarrow$ Recurse on input $(p, E, T, (S, P_1, P_2, \ldots), L)$
 9: $(E, (P_1, P_2, \ldots)) \leftarrow$ Recurse on input $(p, E, U, (P_1, P_2, \ldots), R)$
10: return $(E, (P_1, P_2, \ldots))$
```

- $S_3 =$ [66, 33, 17, 9, 5, 3, 2, 1, 1, 1, 1, 2, 1, 1, 1, 4, 2, 1, 1, 1, 2, 1, 1, 8, 4, 2, 1, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 16, 8, 4, 2, 1, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 32, 16, 8, 4, 3, 1, 1, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 16, 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1]

valid for $\ell = 4$ and $\ell = 3$ respectively. Note that $\ell = 4$ is obviously not a prime, but the methods described here still work in that setting; in the specific case of `SIKEp434`, we can simply express the prime p in the form $p = 4^{108}3^{137} - 1 = 2^{216}3^{137} - 1$ depending on if we want $\ell = 4$ or $\ell = 2$. Nevertheless, for simplicity, you may want to do the $\ell = 3$ case first.

Time how long the original implementation takes to execute, and how long your own implementation with optimal strategies takes.

3. Torsion basis representations

Key compression is based on the observation that the points $\phi_a(P_b)$ and $\phi_a(Q_b)$ in Alice's public key lie in $E_a[3^f]$ (and similarly Bob's public key points lie in $E_b[2^e]$). Given a point $P \in E_a[3^f]$ and a fixed generating set $\{R_1, R_2\}$ such that $E_a[3^f] = \langle R_1, R_2 \rangle$, there exist integers α_P and β_P such that $P = \alpha_R R_1 + \beta_R R_2$ and $0 \leq \alpha_P, \beta_P < 3^f$. Therefore, provided that both parties can agree upon the same values of R_1 and R_2, the point P can be stored and communicated in the form (α_P, β_P), at a cost of $2 \cdot \log_2(3^f) \approx k$ bits, where k denotes $\lceil \log_2 p \rceil$ as before. To go back from (α_P, β_P) to P is easy, provided that R_1 and R_2 can be determined; simply set $P \leftarrow \alpha_P R_1 + \beta_P R_2$. To go in the other direction involves not only determining R_1 and R_2, but also the task of computing α_P and β_P.

The initial paper on key compression [1] uses the following naive algorithm. Letting h denote the cofactor of P (so that $h = 2^e$ if P has order 3^f, and vice-versa), we produce R_1 and R_2 as follows:

1. Choose a pseudorandom point $R \in E(\mathbb{F}_{p^2})$.
2. Set $R_1 \leftarrow hR$.
3. We check whether R_1 has the correct order, by computing $3^{f-1}R_1$ (for Alice) or $2^{e-1}R_1$ (for Bob). If the result of the computation is not the identity, continue; otherwise return to step 1.
4. Repeat steps 1-3 to produce a second point R_2 of the correct order.
5. Compute the Weil pairing $e(R_1, R_2)$, and check whether it has the correct order as in step 3. If it has the correct order, then output (R_1, R_2); otherwise, return to step 1. (See also Exercise 2.)

Since both Alice and Bob need to derive the same values of R_1 and R_2 in order for key compression to work, the sequence of pseudorandom points sampled in step 1 needs to be seeded from a seed value that depends only on E_a. How exactly to do this is left as an exercise for the reader (cf. Exercise 3).

Once we have produced a generating set $\{R_1, R_2\}$, we need an algorithm which takes in an input point $P \in E_a[3^f]$ for Alice (or $P \in E_b[2^e]$ for Bob) and returns integers α_P, β_P such that $P = \alpha_P R_1 + \beta_P R_2$. This problem is similar to the discrete log problem in a group, in that we are given a generating set for the group (which in our case has size 2, in contrast to the size 1 generating set typically seen in discrete log), and we have to find which linear combination of generators equals a given element. In principle, one could solve this problem using methods similar to those for discrete log (e.g. baby-step giant-step). In [1], an alternative approach based on the Weil pairing is used. Recall that the Weil pairing satisfies the following properties:

- Bilinearity:
  - $e(P_1 + P_2, Q) = e(P_1, Q) \cdot e(P_2, Q)$
  - $e(P, Q_1 + Q_2) = e(P, Q_1) \cdot e(P, Q_2)$
  - $e(P, Q)^\alpha = e(\alpha P, Q) = e(P, \alpha Q)$
- Anti-symmetry:
  - $e(P, Q) = e(Q, P)^{-1}$
  - $e(P, P) = 1$
- Non-degeneracy:
  - $\forall\ P,\ (\forall\ Q,\ e(P, Q) = 1) \implies P = \mathcal{O}$

Using these properties, we have

$$e(R_1, P) = e(R_1, \alpha_P R_1 + \beta_P R_2) = e(R_1, R_1)^{\alpha_P} \cdot e(R_1, R_2)^{\beta_P} = e(R_1, R_2)^{\beta_P}$$

$$e(R_2, P) = e(R_2, \alpha_P R_1 + \beta_P R_2) = e(R_2, R_1)^{\alpha_P} \cdot e(R_2, R_2)^{\beta_P} = e(R_2, R_1)^{\alpha_P} = e(R_1, R_2)^{-\alpha_P}$$

Hence one can compute α_P and β_P as follows:

1. Set $g \leftarrow e(R_1, R_2)$
2. Set $h_\alpha \leftarrow e(R_2, P)$
3. Set $h_\beta \leftarrow e(R_1, P)$
4. Set $\alpha_P = -\mathrm{DLOG}(g, h_\alpha)$
5. Set $\beta_P = \mathrm{DLOG}(g, h_\beta)$
6. Output (α_P, β_P)

Normally, discrete logarithms in elliptic curves and finite fields are hard, but in this case, since the points in question have order 2^e or 3^f, the discrete log values in steps 4 and 5 can be obtained efficiently using the Pohlig-Hellman algorithm.

Exercise 2. Let $R_1, R_2 \in E[\ell^e]$, for ℓ prime. Show that the Weil pairing value $e(R_1, R_2)$ has order ℓ^e if and only if $E[\ell^e] = \langle R_1, R_2 \rangle$.

Exercise 3. Using `https://github.com/microsoft/SIKE-challenges` (as in Exercise 1), implement the naive algorithm described in this section. Use your implementation to compress and decompress Alice and Bob's SIDH public keys. Time how long your implementation takes to run. Notes:

- SAGE has Pohlig-Hellman built in: given elements g and h in a group, `h.log(g)` yields $\mathrm{DLOG}(g, h)$.
- To compute Weil pairings in SAGE, use `P.weil_pairing(Q,n)`, where n is the order of P and Q.
- Beware: `E.lift_x` in SAGE does not always return the same output when called on the same input.

4. Entangled basis generation

In the case of 2^e-torsion points, it is possible to compute generating sets for $E[2^e]$ more quickly using special algorithms which directly produce pairs of points (R_1, R_2) such that $E[2^e] = \langle R_1, R_2 \rangle$, without the need for any order or pairing computations. A generating set produced in this manner is called an *entangled basis* [3]. In this section, we focus on the 2^e case. Finding analogous algorithms for the 3^f case remains an open problem.

A preliminary refinement of the naive algorithm is given in [9]. The key observation is that, given a curve $E : y^2 = x^3 + Ax^2 + x$, if we write the curve equation in the form $y^2 = x(x - \gamma)(x - \delta)$ for $\gamma, \delta \in \mathbb{F}_{p^2}$, then a point $R \in E(\mathbb{F}_{p^2})$ lies in $2E(\mathbb{F}_{p^2})$ if and only if x_R, $x_R - \gamma$, and $x_R - \delta$ are all squares in $\mathbb{F}_{p^2}$ [10, §1.4, Theorem 4.1]. Hence, if x_R is non-square, then this is sufficient (though not necessary) to guarantee that $R \notin 2E$, which in turn implies that $3^f R$ must have order exactly 2^e. This method allows us to produce candidate 2^e-torsion points with the assurance that the resulting points will have order exactly 2^e, so that testing for the order is unnecessary. Since all elements of $\mathbb{F}_p$ are square in $\mathbb{F}_{p^2}$, it is easy to generate lots of non-squares x_R by taking a single non-square and multiplying it by elements of $\mathbb{F}_p$. Using this method, it is still necessary to test for whether $x_R^3 + Ax_R^2 + x_R$ is square, as well as whether $e(R_1, R_2)$ has full order. We note that testing for whether or not an element in $\mathbb{F}_{p^2}$ is a square is relatively easy:

Exercise 4. Let $p \equiv 3 \pmod 4$. Let $a + bi \in \mathbb{F}_{p^2}$, where $i = \sqrt{-1}$. Show that $a + bi$ is a square in $\mathbb{F}_{p^2}$ if and only if $a^2 + b^2$ is a square in $\mathbb{F}_p$.

The full entangled basis generation algorithm, given in [3], relies on the following theorem:

Theorem 5 ([3])**.** *Suppose $A \neq 0$. Given $E/\mathbb{F}_{p^2} : y^2 = x^3 + Ax^2 + x$, let $t \in \mathbb{F}_{p^2}$, and suppose $t^2 \notin \mathbb{F}_p$, $P \in E(\mathbb{F}_{p^2})$, $x_P = -A/(1+t^2)$ and x_P is not a square in $\mathbb{F}_{p^2}$. Then*

$$x_Q = -x_P - A$$
$$Q = (x_Q, \sqrt{x_Q^3 + Ax_Q^2 + x_Q})$$

defines a point $Q \in E(\mathbb{F}_{p^2})$ such that $E[2^e] = \langle 3^f P, 3^f Q\rangle$.

Theorem 5 leads to a simple algorithm for entangled basis generation: choose suitable values of $t \in \mathbb{F}_{p^2}$ at random until $x_P = -A/(1+t^2)$ is non-square and $x_P^3 + Ax_P^2 + x_P$ is square; then compute P and Q and output $(3^f P, 3^f Q)$. With additional effort, one can produce a faster method as follows.

```
1. Pre-computation:
   (a) Set $u = 1 + i$. (Note that $u \in \mathbb{F}_{p^2} \setminus \mathbb{F}_p$ and $u^2 \in \mathbb{F}_{p^2} \setminus \mathbb{F}_p$.)
   (b) Compute $(r, 1/(1 + (ur)^2))$ for various values of $r$ (e.g. $r = 1, 2, \ldots, 100$).
       • If $v \leftarrow 1/(1 + (ur)^2)$ is a non-square in $\mathbb{F}_{p^2}$, add $(r, v)$ to table $T_1$.
       • If $v \leftarrow 1/(1 + (ur)^2)$ is a square in $\mathbb{F}_{p^2}$, add $(r, v)$ to table $T_2$.
2. If $A$ is a square in $\mathbb{F}_{p^2}$, set $T \leftarrow T_1$; otherwise set $T \leftarrow T_2$.
3. For each entry $(r, v)$ from $T$:
   (a) Set $x \leftarrow -A \cdot v$. (Note: $x$ is guaranteed to be a non-square.)
   (b) Set $z \leftarrow x^3 + Ax^2 + x$.
   (c) If $z$ is a square in $\mathbb{F}_{p^2}$:
       (i) Set $y \leftarrow \sqrt{z}$.
       (ii) Set $P = (x, y)$.
       (iii) Set $Q = (-x - A, ury)$.
       (iv) Return $(P, Q)$.
```

The return values P and Q are not themselves 2^e-torsion points, but they satisfy the property that $E[2^e] = \langle 3^f P, 3^f Q\rangle$. The pre-computation in step 1 is independent of A, and can be performed once and re-used for all inputs A. The result in Exercise 4 can be used to test for whether v, A, and z are squares.

Exercise 6. Using `https://github.com/microsoft/SIKE-challenges` (as in Exercise 1), implement the faster entangled basis generation method above. Time how long your implementation takes to run.

5. Additional topics

5.1. **Reverse basis decomposition.** Recall that key compression requires computing the values of the coefficients $\alpha_P, \beta_P, \alpha_Q, \beta_Q$ such that

$$\phi_a(P_b) = \alpha_P R_1 + \beta_P R_2$$
$$\phi_a(Q_b) = \alpha_Q R_1 + \beta_Q R_2$$

The method of Section 3 requires computing the five pairing values

$$g_0 = e(R_1, R_2)$$
$$g_1 = e(R_1, \phi_a(P_b)) = g_0^{\beta_P}$$
$$g_2 = e(R_1, \phi_a(Q_b)) = g_0^{\beta_Q}$$
$$g_3 = e(R_2, \phi_a(P_b)) = g_0^{-\alpha_P}$$
$$g_4 = e(R_2, \phi_a(Q_b)) = g_0^{-\alpha_Q}$$

from which $\alpha_P, \beta_P, \alpha_Q, \beta_Q$ can be obtained using discrete logarithms. An alternative approach is to observe that $\{\phi_a(P_b), \phi_a(Q_b\}$ *also* generates $E_a[3^f]$ (because $\{P_b, Q_b\}$ generates $E[3^f]$, and the isogeny ϕ_a has degree

Algorithm 3 Pohlig-Hellman discrete logarithms

```
Given: $g, h \in G$ such that $\mathrm{ord}(g) = \ell^e$ and $h = g^d$
1: $s \leftarrow g^{\ell^{e-1}}$
2: $d \leftarrow 0$
3: $r_0 \leftarrow h$
4: for $k = 0$ to $e - 1$ do
5:     $v_k \leftarrow r_k^{\ell^{e-1-k}}$
6:     find $d_k \in \{0, \ldots, \ell - 1\}$ such that $v_k = s^{d_k}$
7:     $d \leftarrow d + d_k \ell^k$
8:     $r_{k+1} \leftarrow r_k \cdot g^{-\ell^k d_k}$
9: return $d$
```

2^e, so as a map it is injective on 3^f-torsion). Hence we can write

$$\begin{aligned} R_1 &= = \gamma_P \phi_a(P_b) + \gamma_Q \phi_a(Q_b) \\ R_2 &= = \delta_P \phi_a(P_b) + \delta_Q \phi_a(Q_b) \end{aligned}$$

where

$$\begin{bmatrix} \alpha_p & \beta_p \\ \alpha_Q & \beta_Q \end{bmatrix} = \begin{bmatrix} \gamma_P & \gamma_Q \\ \delta_P & \delta_Q \end{bmatrix}^{-1}$$

and we can compute the five pairing values

$$\begin{aligned} h_0 &= e(\phi_a(P_b), \phi_a(Q_b)) \\ h_1 &= e(\phi_a(P_b), R_1) = h_0^{\gamma_Q} \\ h_2 &= e(\phi_a(P_b), R_2) = h_0^{\delta_Q} \\ h_3 &= e(\phi_a(Q_b), R_1) = h_0^{-\gamma_P} \\ h_4 &= e(\phi_a(Q_b), R_2) = h_0^{-\delta_P} \end{aligned}$$

Using discrete logarithms, we compute $\gamma_P, \gamma_Q, \delta_P, \delta_Q$, and invert the matrix to obtain $\alpha_P, \beta_P, \alpha_Q, \beta_Q$. The advantage of this approach is that the value of h_0 does not actually depend on ϕ_a, and hence it can be pre-computed:

$$h_0 = e(\phi_a(P_b), \phi_a(Q_b)) = e(\hat{\phi}_a \circ \phi_a(P_b), Q_b) = e(2^e P_b, Q_b) = e(P_b, Q_b)^{2^e}$$

wherein we have made use of the equivariance property of the Weil pairing: $e(P, \phi(Q)) = e(\hat{\phi}(P), Q)$, where $\hat{\phi}$ denotes the dual isogeny of ϕ [11, III.8.2].

Many more optimizations of this portion of the computation are possible, and these optimizations can be combined with entangled basis generation in a compatible way. We refer the interested reader to [2–4, 9] for further details.

5.2. **Optimal strategies for Pohlig-Hellman.** A basic implementation of Pohlig-Hellman might be something along the lines of Algorithm 3. The astute reader may notice that Algorithm 3 bears a strong similarity to Algorithm 1. Indeed, as Shoup [12, Ch. 11] observed (long before the invention of SIDH), one can speed up Pohlig-Hellman using optimal strategy traversal trees. One possible approach, adapted from [12, §11.2.3], is given in Algorithm 4.

Exercise 7. Implement Algorithms 3 and 4 for `SIKEp434`. Test Algorithm 4 using the following `SIKEp434` parameter sets:

- $\ell = 2, e = 216$, strategy $= S_4$ from Exercise 1.
- $\ell = 3, e = 137$, strategy $= S_3$ from Exercise 1.
- $\ell = 16, e = 54$, strategy = [19, 13, 8, 5, 3, 2, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 3, 2, 1, 1, 1, 1, 1, 4, 3, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 6, 4, 3, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1] from the SIKE spec [6].

Compare the speed of your implementations to each other (and to SAGE's built-in `log()` function!).

Algorithm 4 Optimized Pohlig-Hellman

```
Given: $g, h, e, (s_1, s_2, \ldots, s_{t-1})$
1: if $t = 1$ (i.e. the list is empty) then
2:     return $d \in \{0, \ldots, \ell - 1\}$ such that $h = g^d$
3: $n \leftarrow s_1$
4: $L \leftarrow (s_2, \ldots, s_{t-n})$
5: $R \leftarrow (s_{t-n+1}, \ldots, s_{t-1})$
6: $u \leftarrow$ Recurse on input $(g^{\ell^{e-n}}, h^{\ell^{e-n}}, n, R)$
7: $v \leftarrow$ Recurse on input $(g^{\ell^n}, h/g^u, e - n, L)$
8: return $u + \ell^n \cdot v$
```

References

[1] Reza Azarderakhsh, David Jao, Kassem Kalach, Brian Koziel, and Christopher Leonardi, *Key compression for isogeny-based cryptosystems*, Asia Public Key Cryptography—AsiaPKC 2016, pp. 1–10.

[2] Michael Naehrig and Joost Renes, *Dual isogenies and their applications to public-key compression for isogeny-based cryptography*, Advances in cryptology—Asiacrypt 2019, Lecture Notes in Comput. Sci., vol. 11922, Springer, Cham, pp. 243–272.

[3] Gustavo H. M. Zanon, Marcos A. Simplicio Jr., Geovandro C. C. F. Pereira, Javad Doliskani, and Paulo S. L. M. Barreto, *Faster key compression for isogeny-based cryptosystems*, IEEE Trans. Comput. **68** (2019), no. 5, 688–701.

[4] Geovandro Pereira, Javad Doliskani, and David Jao, *x-only point addition formula and faster compressed SIKE*, J. Cryptographic Engineering **11** (2021), no. 1, 57–69.

[5] Jacques Vélu, *Isogénies entre courbes elliptiques*, C. R. Acad. Sci. Paris Sér. A-B **273** (1971), A238–A241.

[6] David Jao, Reza Azarderakhsh, Matthew Campagna, Craig Costello, Luca De Feo, Basil Hess, Aaron Hutchinson, Amir Jalali, Koray Karabina, Brian Koziel, Brian LaMacchia, Patrick Longa, Michael Naehrig, Geovandro Pereira, Joost Renes, Vladimir Soukharev, and David Urbanik, *Supersingular Isogeny Key Encapsulation*, NIST post-quantum cryptography standardization process, 2017. `https://sike.org/`.

[7] Luca De Feo, David Jao, and Jérôme Plût, *Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies*, J. Math. Cryptol. **8** (2014), no. 3, 209–247.

[8] Daniel J. Bernstein, Luca De Feo, Antonin Leroux, and Benjamin Smith, *Faster computation of isogenies of large prime degree*, ANTS XIV—Proceedings of the Fourteenth Algorithmic Number Theory Symposium, Open Book Ser., vol. 4, Math. Sci. Publ., Berkeley, CA, 2020, pp. 39–55.

[9] Craig Costello, David Jao, Patrick Longa, Michael Naehrig, Joost Renes, and David Urbanik, *Efficient compression of SIDH public keys*, Advances in cryptology—EUROCRYPT 2017. Part I, Lecture Notes in Comput. Sci., vol. 10210, Springer, Cham, 2017, pp. 679–706.

[10] Dale Husemöller, *Elliptic curves*, 2nd ed., Graduate Texts in Mathematics, vol. 111, Springer-Verlag, New York, 2004. With appendices by Otto Forster, Ruth Lawrence and Stefan Theisen.

[11] Joseph H. Silverman, *The arithmetic of elliptic curves*, 2nd ed., Graduate Texts in Mathematics, vol. 106, Springer, Dordrecht, 2009.

[12] Victor Shoup, *A computational introduction to number theory and algebra*, 2nd ed., Cambridge University Press, Cambridge, 2009.