Introduction to Isogeny Based Cryptography

Tommy He

Abstract

As quantum computing advances, many of our current cryptosystems are under grave danger. As such, researching and developing quantum resistant cryptographic schemes is vital now, before it's too late. Notably, the NIST has started a competition to find suitable candidates in a post-quantum world. In this paper, we will focus on SIDH/SIKE, which is a cryptosystem based on supersingular isogenies. We first present a brief overview and go over the necessary mathematics for understanding SIDH. We then present the SIDH protocol and algorithms used in it, as well as implementing an efficient algorithm for arbitrary degree isogenies in python. Finally, we give some cryptanalysis and state of the art attacks on SIDH/SIKE.

1 Introduction

With the recent improvements of quantum computing, the security of many present day cryptographic schemes such as RSA or ECC are threatened by quantum algorithms such as Shor's algorithm. As a result, there has been an increase in activity for developing post-quantum cryptosystems; notably, the NIST has started a competition to find new standards for a post-quantum world. We provide a table of all the current finalists and alternative schemes in round 3. For for information and future updates, visit https://csrc.nist.gov/Projects/post-quantum-cryptography

	Finalists	Alternatives
PKE/KEM	Classic McEliece, CRYSTALS-KYBER NTRU, SABER	BIKE, FrodoKEM HQC, NTRU PRIME, SIKE
Digital Signatures	CRYSTALS-DILITHIUM, FALCON, Rainbow	GeMSS, Picnic, SPHINCS+

Among these algorithms, we can broadly classify them into five different approaches, namely ones based on lattices (NTRU), multivariate (Rainbow), hashes (SPHINCS+), codes (McEliece), and, elliptic curve isogenies (SIKE). We will be focusing on SIKE/SIDH (Supersingular Isogeny Key Encapsulation/Supersingular Isogeny Diffie-Hellman), key encapsulation mechanism and a Diffie-Hellman like key exchange based on supersingular isogenies. We note that the main advantage of SIKE is its smaller key sizes for comparable security, while its main disadvantage is its rather slower speed. Later on, we will expand on the distinction between SIKE and SIDH, but briefly it is because SIDH is only secure for ephemeral keys whereas SIKE is secure for long term keys, as required by the NIST specification. As an overview, SIDH/SIKE is based on walks in a supersingular isogeny graph, which is thought to be quantum resistant. This roughly means it is based on morphisms between elliptic curves where the parties walk through a graph of elliptic curves connected by isogenies and end up with isomorphic elliptic curve as the private key.

2 Preliminaries

We first present some mathematics needed to understand how and why SIDH works. For the rest of the paper, let K be a field and $\overline{K}$ its algebraic closure.

2.1 Elliptic Curves

Definition 2.1. The n-dimensional projective space over K is

$$\mathbb{P}^n(K) = \{(x_1, \cdots, x_{n+1}) \mid (x_1, \cdots, x_{n+1}) \in K^{n+1}, (x_1, \cdots, x_{n+1}) \neq 0\}/\sim$$

over the equivalence class relation defined by $A \sim B$ if and only if $\exists \lambda \neq 0 \in K : A = \lambda B$, where we denote the equivalence class of $(x_1, \cdots, x_{n+1})$ to be $(x_1 : \cdots : x_{n+1}$. The n-dimensional affine plane over K is $\mathbb{A}^n(K) = K^n$.

Remark 2.2. Suppose we are working in $\mathbb{P}^n(K)$. If $x_{n+1} \neq 0$, then we have the same points as $\mathbb{A}^n(K)$ since $(x_1 : \cdots : x_n : x_{n+1}) = (\frac{x_1}{x_{n+1}} : \cdots : \frac{x_n}{x_{n+1}} : 1)$, which we call the finite points in $\mathbb{P}^n(K)$. If $x_{n+1} = 0$, we call the points $(x_1 : \cdots : x_n : 0)$ the points at infinity. Notice we then have a natural embedding $\mathbb{A}^n(K) \hookrightarrow \mathbb{P}^n(K)$ by $(x_1, \cdots, x_n) \mapsto (x_1 : \cdots : x_n : 1)$. Additionally, we can view the projective space as a disjoint union of the affine plane with the points at infinity. This is will be useful for converting between the two spaces as we can naturally transform a locus in the projective space into one in the affine space with additional points at infinity.

Definition 2.3. An elliptic curve E defined over K is the locus in $\mathbb{P}^2(\overline{K})$ of

$$E/K : Y^2Z = X^3 + aXZ^2 + bZ^3$$

for $a, b \in K$ and $\Delta = 4a^3 + 27b^2 \neq 0$. This elliptic curve is in Weierstrass form, with the point $(0 : 1 : 0)$ as the point at infinity. In affine form, the usual transformation into $\mathbb{A}^2(K)$ by $x = X/Z$ and $y = Y/Z$ gives

$$y^2 = x^3 + ax + b$$

with an additional point at infinity denoted $\mathcal{O}$.

Definition 2.4. An elliptic curve in the projective form is the locus in $\mathbb{P}^2(\overline{K})$

$$bY^2Z = X^3 + aX^2Z + XZ^2$$

for $a, b \in K$ and $\Delta = b(a^2 - 4) \neq 0$ is in Montgomery form, with the point $(0 : 1 : 0)$ as the point at infinity. In affine form, using the usual transformation into $\mathbb{A}^2(K)$, we have

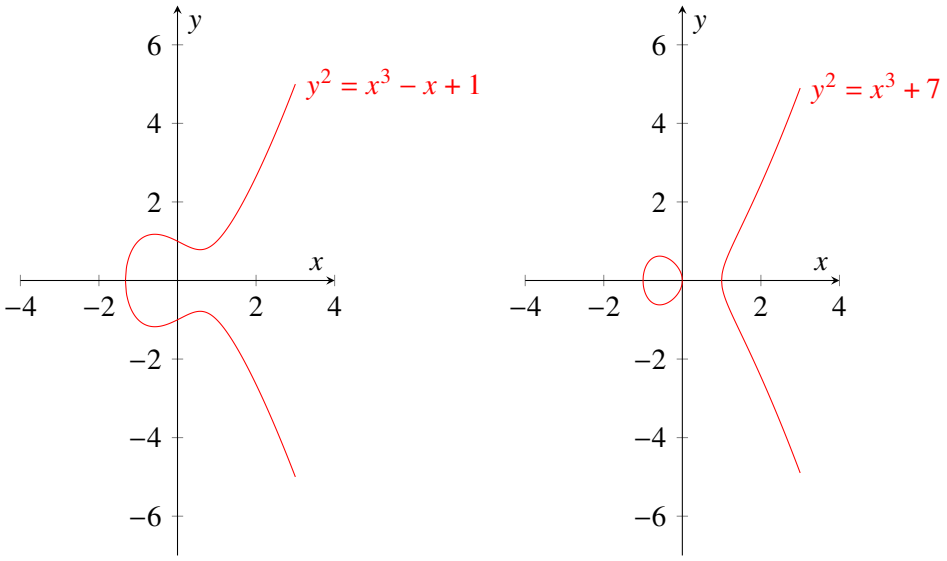
$$by^2 = x^3 + ax^2 + x$$

with an additional point at infinity $\mathcal{O}$.

Despite less general, elliptic curves in Montgomery form allow for x-coordinate only arithmetic, which prove to be simpler and faster to work with in practice.

Remark 2.5. Any elliptic curve in Montgomery form $by^2 = x^3 + ax^2 + x$ can be converted to Weierstrass form $y^2 = x^3 + a'x + b'$ by the map $(x, y) \mapsto (\frac{x}{b} + \frac{a}{3b}, \frac{y}{b})$ and letting $a' = \frac{3-a^2}{3b^2}$, $b' = \frac{2a^3-9a}{27b^3}$.

Example 2.6. Here are some examples of elliptic curves over $\mathbb{R}$ on the affine plane. Note that for cryptographic uses, we will often take finite fields, so those graphs would appear as discrete points.



Definition 2.7. For elliptic curve $E : y = x^3 + ax + b$, its j-invariant is

$$j(E) = 1728 \cdot \frac{4a^3}{4a^3 + 27b^2}$$

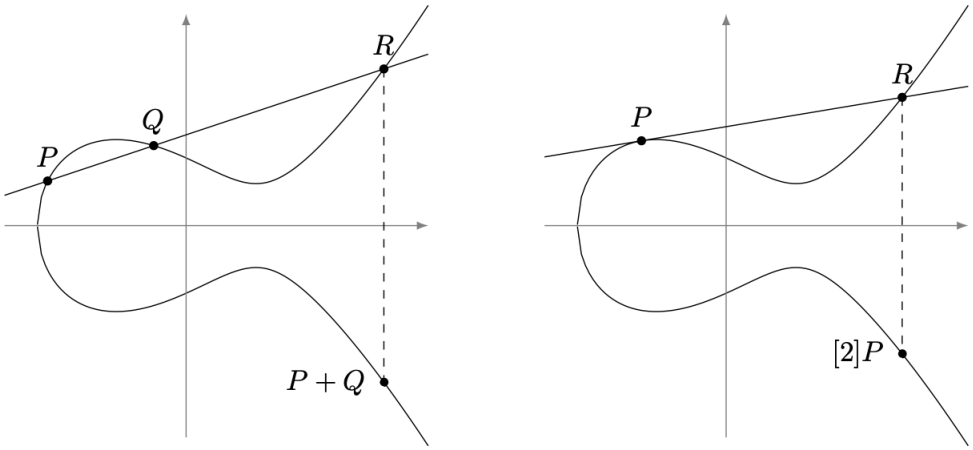
Two elliptic curves are isomorphic if and only if they have the same j-invariant.

Definition 2.8. For elliptic curve $E : y = x^3 + ax + b$ and points $P_1 = (x_1, y_1), P_2 = (x_2, y_2) \in E$ not both O, define point composition $\oplus : E \times E \to E$ as $P_1 \oplus P_2$ the third intersection between the line formed by $\overline{P_1P_2}$ and E or the tangent line to E at P_1 if $P_1 = P_2$ and E. By Bezout's Theorem, $P_1 \oplus P_2$ always exists in the projective plane. One can show $\oplus$ forms an abelian group, so we will denote this group operation by point addition $+$, and let $[l]P$ be $\underbrace{P + \cdots + P}_{l \text{ times}}$, where $[l]$ is the multiplcation-by-l map.

Proposition 2.9. *We can define the group law algebraically as well for elliptic curve $E : y = x^3 + ax + b$ and points $P_1 = (x_1, y_1), P_2 = (x_2, y_2) \in E$ not both O by*

$$+ : E \times E \to E, \begin{cases} P_1 + O = P_1 \text{ If } P_2 = O \\ P_1 + P_2 = O \text{ If } x_1 = x_2, y_1 \neq y_2 \\ P_1 + P_2 = (\lambda^2 - x_1 - x_2, -\lambda x_3 - y_1 + \lambda x_1) \text{ Else where } \lambda \text{ is defined by} \lambda = \begin{cases} \frac{y_2 - y_1}{x_2 - x_1} \text{ if } P_1 \neq P_2 \\ \frac{3x_1^2 + a}{2y_1} \text{ if } P_1 = P_2 \end{cases} \end{cases}$$

Example 2.10. We have a geometric representation of the group operation for elliptic curves from [3]. Note that if the points form a vertical line, then they will intersect at O just as in the projective plane.



2.2 Isogenies

Definition 2.11. For elliptic curves E_1, E_2, a surjective morphism $\phi : E_1 \to E_2$ is an isogeny. If there exists such an isogeny, then E_1 and E_2 are isogenous. If ϕ is separable, then the degree d of ϕ is the order of $\ker \phi$, in which case we call ϕ a d-isogeny.

Remark 2.12. Isogenies can be separable or inseparable; however, that distinction will not be important here. We are only interested in separable isogenies, where they are in 1-to-1 correspondence with finite subgroups: every subgroup $G \leq E$ gives rise to a unique isogeny $\phi : E \to E' = E/G$ [11]. Velu's formula [12] explicitly gives a formula to calculate the constants defining E' given E and the points of G as well as the explicit map for ϕ. We will not need it in its general form, but we will give it for 2-isogenies and 3-isogenies as they will be used later. It is worth noting that the time complexity of Velu's formula scales linearly with the size of the subgroup G; so, if we have a subgroup of size l^e, we would not want to apply Velu's directly to the subgroup but rather decompose the isogeny into e isogenies of degree l as in Section 4.1

Definition 2.13. The l-torsion group of elliptic curve E is the kernel of the multiplication-by-l map $\ker[l]$, denoted $E[l]$.

Example 2.14 (2-isogenies)**.** We will consider the multiplication-by-2 map on the Montgomery form elliptic curve $E_a : y^2 = x^3 + ax^2 + x$ given by

$$[2] : E_a \to E_a, x \mapsto \frac{(x^2-1)^2}{4x(x^2+ax+1)}$$

The points that map to the identity are precisely the ones ones that satisfy $4x(x^2 + ax + 1) = 0$ or $(0,0), (\alpha, 0), (\frac{1}{\alpha}, 0)$ together with the identity. Notice this forms the 2-torsion subgroup with the structure

$$E[2] \cong \mathbb{Z}_2 \times \mathbb{Z}_2$$

Now, let's take a cyclic subgroup of order 2 such as $G = \{O, (\alpha, 0)\}$ and input this with E_a into Velu's formula to get the map

$$E_a \to E_{a'}, x \mapsto \frac{x(\alpha x - 1)}{x - \alpha}$$

where

$$E_{a'} : y^2 = x^3 + a'x^2 + x, a' = 2(1 - 2\alpha^2)$$

Example 2.15 (3-isogenies)**.** Consider the multiplication-by-3 map on $E_a : y^2 = x^3 + ax^2 + x$ given by

$$[3] : E_a \to E_a, x \mapsto \frac{x(x^4 - 6x^2 - 4ax^3 - 3)^2}{(3x^4 + 4ax^3 + 6x^2 - 1)^2}$$

The kernel is the ones that satisfy $(3x^4 + 4ax^3 + 6x^2 - 1)^2 = 0$ with O. Let its roots be $\beta, \delta, \zeta, \theta$. In this case, we would get a group with 9 elements $\{(\beta, \pm\gamma), (\delta, \pm\varepsilon), (\zeta, \pm\eta), (\theta, \pm\iota), O\}$. Notice the 3-torsion subgroup has the structure

$$E[3] \cong \mathbb{Z}_3 \times \mathbb{Z}_3$$

Now, take the cyclic subgroup $G = \{O, (\beta, \gamma), (\beta, -\gamma)\}$ of order 3 and input it with E_a into Velu's to get the map

$$E_a \mapsto E_{a'}, x \mapsto \frac{x(\beta x - 1)^2}{(x - \beta)^2}$$

where

$$E_{a'} : y^2 = x^3 + a'x^2 + x, a' = (a\beta - 6\beta^2 + 6)\beta$$

Remark 2.16. Note that we mentioned that elliptic curves E_1, E_2 are isogenous if there exists an isogeny $\phi : E_1 \to E_2$; however, it's not immediately obvious whether the existence of an isogeny $E_1 \to E_2$ implies an isogeny in the other direction $E_2 \to E_1$. In actuality, there always exists a unique isogeny $\widehat{\phi} : E_2 \to E_1$ called the dual isogeny. If ϕ has degree d, then the dual isogeny is such that $\widehat{\phi} \circ \phi = [d]$ on E_1. Specifically how the dual isogeny is constructed is not important to us, but its existence is for the isogeny graph later.

Proposition 2.17. *In the case that* $\operatorname{char} K = p$ *for some prime* p*,*

- *If* p *does not divide* l*, then* $E[l] \cong (\mathbb{Z}/l\mathbb{Z})^2$
- *Else,*

$$E[p^i] \cong \begin{cases} \mathbb{Z}/p^i\mathbb{Z} & \forall i \geq 0 \text{ called ordinary elliptic curves} \\ \{O\} & \forall i \geq 0 \text{ called supersingular elliptic curves} \end{cases}$$

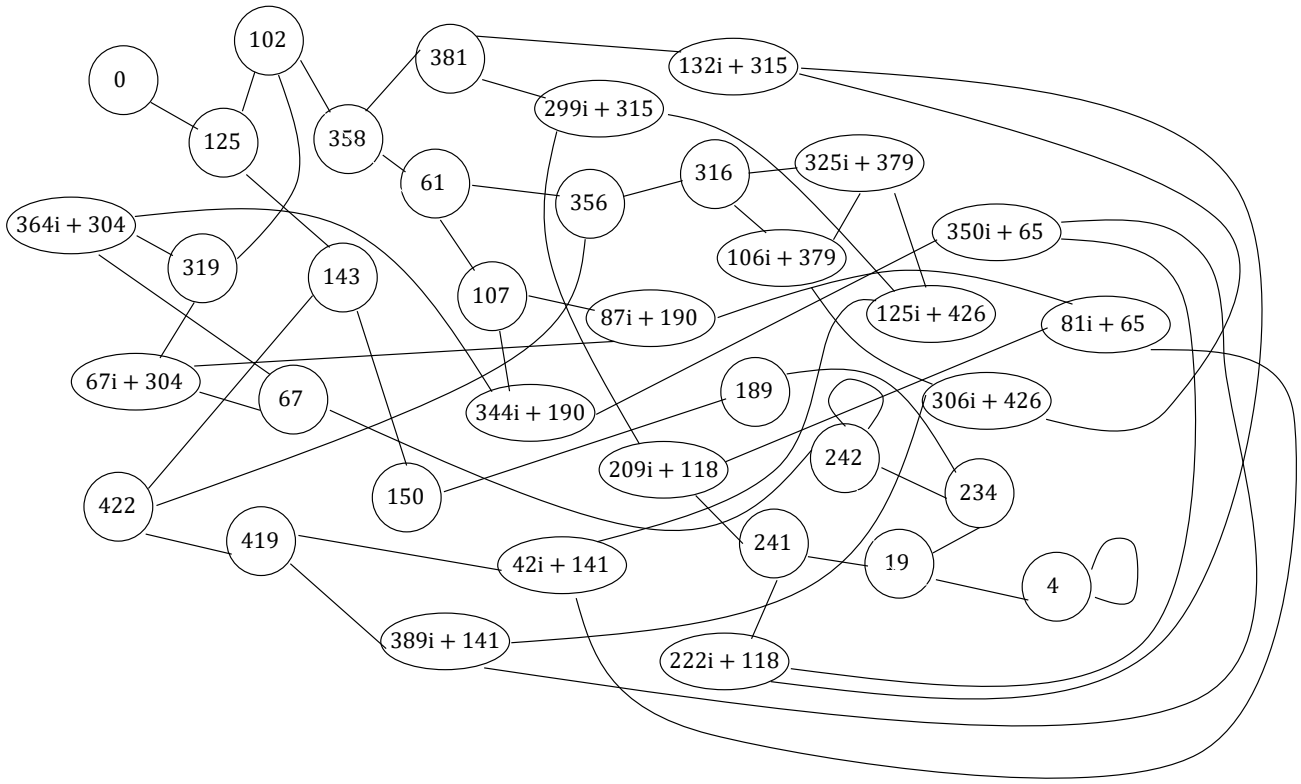
2.3 Isogeny Graphs

Definition 2.18. The l-isogeny graph is the graph formed by the j-invariants of the elliptic curves as vertices and l-isogenies as edges between them.

Proposition 2.19. *Supersingular* l*-isogeny graphs are*

- *undirected, by the dual isogenies*
- *connected, meaning there is a path between any two nodes*
- $(l+1)$*-regular, meaning every vertex has exactly* $l + 1$ *neighboring vertices*

Example 2.20. We have the 2-isogeny graph from [1] for $p = 431$ where the 37 nodes are the supersingular j-invariants and the edges are the 2-isogenies between them.



Notice that this is indeed connected and 3-regular. It may seem that j-invariants of $\{0, 4, 242\}$ do not follow this pattern; however, they can be explained by multiplicities arising from isogenies degenerating to isomorphisms or different subgroups producing the same image j-invariants.

2.4 Elliptic Curves in SIDH

In SIDH, we will typically take elliptic curves over a finite field, specifically ones of the form $\mathbb{F}_{p^2}$ for $p \equiv 3 \pmod 4$ by the construction $\mathbb{F}_{p^2} = \mathbb{F}_p(i) = \mathbb{F}_p[X]/(X^2 + 1)$. The reason for this is that of the p^2 elements in $\mathbb{F}_{p^2}$, there are $\lfloor \frac{p}{12} \rfloor + z$ for a $z \in \{0, 1, 2\}$ possible distinct supersingular j-invariants in $\mathbb{F}_{p^2}$ [11]. Precisely what determines z is not important for our purposes, but it is important to note that as p grows exponentially so does the set of j-invariants and the size of the isogeny graph. Specifically, taking supersingular elliptic curves is important for an $(l + 1)$-regular l-isogeny graph and the exponential growth in the isogeny graph size with p.

3 The Protocol

3.1 An overview

SIDH, standing for Supersingular Isogeny Diffie-Hellman, is a public key crytosystem similar to Diffie-Hellman key exchange, but it is based on walks on a supersingular isogeny graph rather than cyclic groups. We work in fields $\mathbb{F}_{p^2}$ for prime p. At its core, it relies on the commutative diagram

$$\begin{array}{ccc} E & \xrightarrow{\phi_A} & E_A \\ {\scriptstyle \phi_B}\downarrow & & \downarrow{\scriptstyle \phi'_B} \\ E_B & \xrightarrow{\phi'_A} & E_{AB} \end{array}$$

where E is part of the initial public parameter. Alice chooses a secret subgroup $G_A \leq E$ with order n_A to calculate $E_A = E/G_A$, which is part of Alice's public key. Bob similarly chooses secret subgroup $G_B \leq E$ with order n_B to calculate $E_B = E/G_B$, which is part of Bob's public key. They are then both able to calculate the shared secret elliptic curve $E_{AB} = E/\langle G_A, G_B \rangle$, where the trapdoor relies on the difficulty on calculating E_{AB} from E_A, E_B.

3.2 Parameter Selection

In general, our prime p can be of the form $p = f \cdot n_A n_B \pm 1$ where $\gcd(n_A, n_B) = 1$ and f is a cofactor. In this case, the isogeny class of supersingular elliptic curves over $\mathbb{F}_{p^2}$ will have cardinality $(p \mp 1)^2 = (f \cdot n_A n_B)^2$ [2]. Perhaps the most straightforward way to select primes of this form is to take primes of the form $p = f \cdot l_A^{e_A} l_B^{e_B} - 1$ for primes l_A, l_B. In practice, we currently take our prime p to be of the form $p = 2^{e_A} 3^{e_B} - 1$ for natural numbers e_A, e_B such that $2^{e_A} \approx 3^{e_B}$, as per the SIKE standards. We do this for efficient instantiations, since 2-isogenies and 3-isogenies are the most efficient to calculate, but we will give an algorithm for efficient computations for arbitrary numbers n_A, n_B in Section 4.2.

3.3 The Exchange

For describing the protocol, we will take primes of the form $p = l_A^{e_A} l_B^{e_B} - 1$. Then, the parameters of SIDH consist of the initial elliptic curve E, primes l_A, l_B, integers e_A, e_B, and public basis points $P_A, Q_A, P_B, Q_B \in E$ that generate the subgroup, which will be described below. We follow the one described as in [1]

3.3.1 Secret Generation

In order for Alice to find a secret subgroup of order $l_A^{e_A}$, she will start with public generators P_A, Q_A such that

$$\langle P_A, Q_A \rangle = E[l_A^{e_A}] \cong \mathbb{Z}_{l_A^{e_A}} \times \mathbb{Z}_{l_A^{e_A}}.$$

Similarly Bob, to find a secret subgroup of order $l_A^{e_B}$, will start with public generators P_B, Q_B with

$$\langle P_B, Q_B \rangle = E[l_B^{e_B}] \cong \mathbb{Z}_{l_B^{e_B}} \times \mathbb{Z}_{l_B^{e_B}}.$$

This selection will mean P_A, Q_A both have order $l_A^{e_A}$ and P_B, Q_B both have order $l_B^{e_B}$. To find their secret generators S_A, S_B, they will then pick secret integers $k_A \in [0, l_A^{e_A}), k_B \in [0, l_B^{e_B})$ to get

$$S_A = P_A + [k_A]Q_A$$

$$S_B = P_B + [k_B]Q_B$$

Now, Alice can compute her secret isogeny $\phi_A : E \to E_A = E/\langle S_A \rangle$, and Bob can compute his secret isogeny $\phi_B : E \to E_B = E/\langle S_B \rangle$ (the specifics of the isogeny computation will be described below).

3.3.2 Public Key Generation

Now, Alice will have public key

$$PK_A = (E_A, P'_B, Q'_B) = (\phi_A(E), \phi_A(P_B), \phi_A(Q_B))$$

where E_A is Alice's image curve and P'_B, Q'_B are images of Bob's public basis points under ϕ_A. Bob will have public key

$$PK_B = (E_B, P'_A, Q'_A) = (\phi_B(E), \phi_B(P_A), \phi_B(Q_A))$$

where, similarly, E_B is Bob's image curve and P'_A, Q'_A are images of Alice's public basis points under ϕ_B. These image points will be necessary for the shared secret computation below.

3.3.3 Shared Secret Computation

With k_A and PK_B, Alice computes

$$S'_A = P'_A + [k_A]Q'_A$$

on E_B and another isogeny $\phi'_A : E_B \to E_{AB} = E_B/\langle S'_A \rangle$. Then, she gets the shared secret $j_{AB} = j(E_{AB})$. Bob does similarly with k_B and PK_A to find

$$S'_B = P'_B + [k_B]Q'_B$$

on E_A and isogeny $\phi'_B : E_A \to E_{BA} = E_A/\langle S'_B \rangle$ (specifics of isogeny computation will be below). Then, he gets the same shared secret $j_{AB=}j_{BA} = j(E_{AB})$.

4 Algorithms for Computing Isogenies

The major computational roadblock in SIDH is the computation of the isogeny $E \to E' = E/S$ for subgroup $S \leq E$. Here, we will present 2 algorithms for achieving this and a speedup on a subroutine at the end. For this section, we will be working with Montgomery elliptic curves of the form

$$E_a/K : y^2 = x^3 + ax^2 + x$$

to take advantage of the faster and simpler x only arithmetic.

4.1 A First Algorithm

The first algorithm proposed in [6] is the most straightforward using primes of the form $p = 2^{e_A}3^{e_B} - 1$. In order to compute an l^e-isogeny, we loop e times with the following operations:

- Take generator $P_i \in E_i$ with order l^{e-i}
- Find $[l^{e-i-1}]P_i \in E_i$ with order l
- By Velu's formula, compute the l-isogeny $\phi_i : E_i \to E_{i+1}$ with $\ker \phi_i = \langle [l^{e-i-1}]P_i \rangle$
- Evaluate $\phi_i(P_i) = P_{i+1} \in E_{i+1}$

} FOR $i = 0$ to $e - 1$

Essentially we find e l-isogenies and compose them all together. We find the final l^e-isogeny by $\phi_{e-1}\phi_{e-2}\cdots\phi_0 : E_0 \to E_e$. Specifically for primes of the form $p = 2^{e_A}3^{e_B}$, we can use the maps given by Velu's formula for 2-isogenies and 3-isogenies as calculated in Example 2.14 and Example 2.15

4.2 Efficient Algorithm For Arbitrary Degree Isogenies

In this section, we will present an efficient formula and algorithm for computing arbitrary odd degree isogenies due to Costello and Hisil [2] as well as a unification between isogenies on elliptic curves and points. Note that any separable isogeny can be separated into a composition of prime degree isogenies, so even degree isogenies can be separated into a power of the 2-isogeny from Example 2.14 with some odd isogenies, which can be calculated using this algorithm.

4.2.1 Coordinate Maps

Montgomery [8] gave his formulas for arithmetic in the x coordinates for point addition and point doubling by

$$x_{P+Q} = \frac{(x_P x_Q - 1)^2}{x_{P-Q}(x_P - x_Q)^2}$$

$$x_{[2]P} = \frac{(x_P^2 - 1)^2}{4x_P(x_P^2 + ax_P + 1)}$$

Define the $\mathbf{x}$ map as the one that drops the Y coordinate in projective coordinates by

$$\mathbf{x} : E \setminus \{O\} \to \mathbb{P}^1(K), (X : Y : Z) \mapsto (X : Z)$$

With the above maps, we can find Montgomery's algorithms for projective arithmetic in $\mathbb{P}^1(K)$ given by

$$\text{xDBL} : (\mathbf{x}(P), (\widehat{A} : \widehat{C})) \mapsto \mathbf{x}([2]P)$$
$$\text{xADD} : (\mathbf{x}(P), \mathbf{x}(Q), \mathbf{x}(P - Q)) \mapsto \mathbf{x}(P + Q).$$

Let $\mathbf{x}(P) = (X_P : Z_P), \mathbf{x}(Q) = (X_Q : Z_Q), \mathbf{x}(P + Q) = (X_+ : Z_+), \mathbf{x}(P - Q) = (X_- : Z_-), \mathbf{x}([2]P) = (X_{[2]P}, Z_{[2]P})$. Then, Montgomery [8] projectivizes his formulas as

$$\begin{cases} X_+ = Z_-((X_P - Z_P)(X_Q + Z_Q) + (X_P + Z_P)(X_Q - Z_Q))^2 \\ Z_+ = X_-((X_P - Z_P)(X_Q + Z_Q) - (X_P + Z_P)(X_Q - Z_Q))^2 \end{cases}$$

$$\begin{cases} X_{[2]P} = (X_P + Z_P)^2(X_P - Z_P)^2 \\ Z_{[2]P} = (4X_P Z_P)((X_P - Z_P)^2 + ((a + 2)/4)(4X_P Z_P)) \end{cases}$$

Note that in the algorithm, we will use the identity $4X_P Z_P = (X_P + Z_P)^2 - (X_P - Z_P)^2$ for calculating $\mathbf{x}([2]P)$to speed it up. Additionally for xDBL, we will pass in the constant $(\widehat{A} : \widehat{C}) = (a - 2 : 4)$ as we can compute it for use elsewhere (this will be explained later more). With these, we have our first 2 algorithms

Algorithm 1 xADD: $(\mathbb{P}^1)^2 \to \mathbb{P}^1$

Input: $(X_P : Z_P), (X_Q : Z_Q), (X_- : Z_-) \in \mathbb{P}^1$
Output: $(X_+ : Z_+) \in \mathbb{P}^1$

1: **procedure** xADD ▷ Finds $\mathbf{x}(P + Q)$ with $\mathbf{x}(P), \mathbf{x}(Q), \mathbf{x}(P - Q)$ projectively
2: $V_0 \leftarrow X_P - Z_P$
3: $V_1 \leftarrow X_Q + Z_Q$
4: $V_2 \leftarrow X_P + Z_P$
5: $V_3 \leftarrow X_Q - Z_Q$
6: $V_4 \leftarrow V_0 V_1$
7: $V_5 \leftarrow V_2 V_3$
8: **return** $(Z_-(V_4 + V_5)^2 : X_-(V_4 - V_5)^2)$

Algorithm 2 xDBL: $\mathbb{P}^1 \times \mathbb{P}^1 \to \mathbb{P}^1$

Input: $(X_P : Z_P), (\widehat{A} : \widehat{C}) \in \mathbb{P}^1$
Output: $(X_{[2]P} : Z_{[2]P}) \in \mathbb{P}^1$

1: **procedure** xDBL ▷ Finds $\mathbf{x}([2]P)$ with $\mathbf{x}(P)$ projectively
2: $V_0 \leftarrow Z_P + Z_P$
3: $V_1 \leftarrow X_P - Z_P$
4: $V_2 \leftarrow V_0^2$
5: $V_3 \leftarrow V_1^2$
6: $V_4 \leftarrow V_2 - V_3$
7: **return** $(V_2V_3 : V_4(V_3 + \widehat{A}V_4/\widehat{C}))$

The core of this isogeny algorithm relies on Theorem 4.1, which is essentially a simplified formula for Velu's formula for odd degree isogenies on Montgomery curves.

Theorem 4.1. *For field K with* $\operatorname{char} K \neq 0$ *with $P \in E(\overline{K})$ of order $l = 2d + 1$ on $E/K : by^2 = x^3 + ax^2 + x$, the codomain E' of the l-isogeny $\phi : E \to E'$ with* $\ker \phi = \langle P \rangle$ *is*

$$\phi : (x, y) \mapsto (f(x), yf'(x))$$

where f is defined by

$$f : x \mapsto x \cdot \prod_{i=1}^{d} \left(\frac{x \cdot x_{[i]P} - 1}{x - x_{[i]P}} \right)^2$$

The above coordinate map is enough for us to find E' explicitly as we will see in the next Section 4.2.2 (an explicit formula for E' is also provided in [2], though it turns out to be slower with the results of the following section).

4.2.2 Finding Isogenies Using 2-Torsion and Unification of Two Operations

Suppose we have $\phi : E_a \to E_{a'}$ as an odd degree isogeny of Montgomery curves. All current supersingular isogeny cryptosystems use separate functions for computing isogenous curves and isogenous points i.e. for $E \mapsto \phi(E)$ and $P \mapsto \phi(P)$.

As per Example 2.14, the 3 affine points of order 2 on E_a are

$$P_0 = (0, 0), P_\alpha = (\alpha, 0), P_{1/\alpha} = \left(\frac{1}{\alpha}, 0 \right)$$

where $\alpha^2 + a\alpha + 1 = 0 \implies a = -\frac{\alpha^2+1}{\alpha}$. This relation between a and α will be used to parse between them later. In $\mathbb{P}^1$, we have

$$\mathbf{x}(P_0) = (0 : 1), \mathbf{x}(P_\alpha) = (X_\alpha : Z_\alpha), \mathbf{x}(P_{1/\alpha}) = (Z_\alpha : X_\alpha)$$

with

$$(a : 1) = (X_\alpha^2 + Z_\alpha^2 : -X_\alpha Z_\alpha) \tag{1}$$

Observe that from Theorem 4.1, we must have $\phi : 0 \mapsto 0$. Since 2-torsion points preserve their order under odd isogenies, we will have

$$\mathbf{x}(\phi(P)) = (0 : 1), \mathbf{x}(\phi(P_\alpha)) = (X'_\alpha : Z'_\alpha), \mathbf{x}(\phi(P_{1/\alpha})) = (Z'_\alpha : X'_\alpha)$$

Since, they still have order 2, we have the same relation

$$(a' : 1) = (X'^2_\alpha + Z'^2_\alpha : -X'_\alpha Z'_\alpha)$$

on the new curve $E_{a'}$. So, instead of viewing the Montgomery curve as represented by $(a : 1) = (A : C)$, we can represent it with the 2-torsion point $X_\alpha : Z_\alpha$ and find the curve coefficient through Equation 1. In fact, the mulitplication-by-l maps and xDBL use the constant $(a - 2 : 4) = ((A - 2C)/4 : C) = (\widehat{A} : \widehat{C})$, which is actually faster to calculate with $(\widehat{A} : \widehat{C}) = ((A - 2C)/4 : C) = ((X_\alpha + Z_\alpha)^2 : (X_\alpha - Z_\alpha)^2 - (X_\alpha + Z_\alpha)^2)$. This parsing from α to $(a : 1)$, or rather $(\widehat{A} : \widehat{C}) = (a - 2 : 4)$, is used often, so we will write this in the subroutine aHatFromAlpha.

Algorithm 3 aHatFromAlpha: $\mathbb{P}^1 \to \mathbb{P}^1$

Input: $(\alpha : 1) \in \mathbb{P}^1$
Output: $(\widehat{A} : \widehat{C}) \in \mathbb{P}^1$

1: **procedure** aFromAlpha ▷ Finds $(\widehat{A} : \widehat{C})$ from $(\alpha : 1)$
2: $a \leftarrow (\alpha^2 + 1)/\alpha$
3: **return** $(a - 2 : 4)$

4.2.3 Algorithm for Arbitrary Odd Degree Isogenies

Our algorithm will find the map

$$x \mapsto x \left(\prod_{i=1}^{d} \frac{x \cdot x_{[i]d} - 1}{x - x_{[i]d}} \right)^2,$$

as per Theorem 4.1, which is the only equation we need, and it will be done in projective coordinates in $\mathbb{P}^1$. For notation, let $(X_i : Z_i) = (x_{[i]P} : 1)$, $(X : Z) = (x : 1)$, $(X' : Z') = \mathbf{x}(\phi(x, y))$. Then, we find

$$X' = X \left(\prod_{i=1}^{d} (X \cdot X_i - Z_i \cdot Z) \right)^2$$

$$Z' = Z \left(\prod_{i=1}^{d} (X \cdot Z_i - X_i \cdot Z) \right)^2,$$

which one can verify by calculating $X'/Z' = \mathbf{x}(\phi(x, y)) = f(x)$. Then, by following Montgomery [8], we reduce the operations needed further with

$$\begin{aligned} X' &= X \left(\prod_{i=1}^{d} ((X - Z)(X_i + Z_i) + (X + Z)(X_i - Z_i)) \right)^2 \\ Z' &= Z \left(\prod_{i=1}^{d} ((X - Z)(X_i + Z_i) - (X + Z)(X_i - Z_i)) \right)^2 \end{aligned} \tag{2}$$

Notice that for implementing this algorithm, we can reuse $X - Z, X + Z$ over all the d iterations, which must compute the sum and difference between pairwise products multiple times, so we dub this subroutine crissCross.

Algorithm 4 CRISSCROSS: $K^4 \to K^2$

Input: $\alpha, \beta, \gamma, \delta \in K$
Output: $\alpha\delta + \beta\gamma, \alpha\delta - \beta\gamma \in K$

procedure CRISSCROSS ▷ Finds the sum and difference between pairwise products
  $V_0 \leftarrow \alpha\delta$
  $V_1 \leftarrow \beta\gamma$
  return V_0, V_1

Now, with the generator $\mathbf{x}(P) = (X_1 : Z_1)$, the first step is to generate the $d-1$ additional points in the kernel $\mathbf{x}(P), \cdots, \mathbf{x}([d]P) = (X_1 : Z_1), \cdots, (X_d : Z_d)$, which we do in KERNELPOINTS.

Algorithm 5 KERNELPOINTS: $\mathbb{P}^1 \times \mathbb{P}^1 \to (\mathbb{P}^1)^d$

Input: $(X_1 : Z_1), (\widehat{A} : \widehat{C}) \in \mathbb{P}^1$
Output: $(X_1 : Z_1), \cdots, (X_d : Z_d) = \mathbf{x}(P), \cdots, \mathbf{x}([d]P) \in \mathbb{P}^1$

procedure KERNELPOINTS ▷ Computes the d multiples of P
  if $d \geq 2$ **then**
    $(X_2 : Z_2) \leftarrow \text{xDBL}((X_1 : Z_1), (\widehat{A} : \widehat{C}))$
  for $i = 3$ **to** d **do**
    $(X_i : Z_i) \leftarrow \text{xADD}((X_{i-1} : Z_{i-1}), (X_1 : Z_1), (X_{i-2} : Z_{i-2}))$
  return $(X_1 : Z_1), \cdots, (X_d : Z_d)$

Looking back to Equation 2 with the d kernel points, we can now calculate $(\widehat{X}_i, \widehat{Z}_i) \leftarrow (X_i + Z_i, X_i - Z_i)$ and $(\widehat{X}, \widehat{Y}) \leftarrow (X + Z, X - Z)$ in preparation for CRISSCROSS. Supposing we already have $(\widehat{X}_i, \widehat{Z}_i)$, we put this algorithm in ODDISOGENY, which gives us the desired $(X' : Z')$.

Algorithm 6 ODDISOGENY: $(K^2)^d \times \mathbb{P} \to \mathbb{P}$

Input: $(\widehat{X}_1, \widehat{Z}_1), \cdots, (\widehat{X}_d, \widehat{Z}_d) \in K^2, (X : Z) \in \mathbb{P}$
Output: $(X' : Z') \in \mathbb{P}^1$

procedure ODDISOGENY ▷ Computes l-isogeny point
  $(\widehat{X}, \widehat{Z}) \leftarrow (X + Z, X - Z)$
  $X', Z' \leftarrow \text{CRISSCROSS}(\widehat{X}_1, \widehat{Z}_1, \widehat{X}, \widehat{Z})$
  for $i = 2$ **to** d **do**
    $V_0, V_1 \leftarrow \text{CRISSCROSS}(\widehat{X}_i, \widehat{Z}_i, \widehat{X}, \widehat{Z})$
    $(X', Z') \leftarrow V_0 X', V_1 Z'$
  $(X', Z') \leftarrow (X(X')^2, Z(Z')^2)$
  return $(X' : Z')$

Putting everything together as well as using $(\widehat{A} : \widehat{C}) = (a - 2 : 4)$, SIMULTANEOUSODDISOGENY calculates multiple l-isogenies for n points $\mathbf{x}(P_1), \cdots, \mathbf{x}(P_n) = (\mathbb{X}_1 : \mathbb{Z}_1), \cdots, (\mathbb{X}_n, \mathbb{Z}_n) \in \mathbb{P}^1$ where $P_i \in E$ are such that $P_i \notin \langle P \rangle$ to be the points $(\mathbb{X}'_1 : \mathbb{Z}'_1), \cdots, (\mathbb{X}'_n : \mathbb{Z}'_n)$

Algorithm 7 simultaneousOddIsogeny: $\mathbb{P}^1 \times \mathbb{P}^1 \times (\mathbb{P}^1)^n \to (\mathbb{P}^1)^n$

Input: $(X_1 : Z_1), (\widehat{A} : \widehat{C}), (\mathbb{X}_1, \mathbb{Z}_1), \cdots, (\mathbb{X}_n, \mathbb{Z}_n) \in \mathbb{P}^1$
Output: $(\mathbb{X}'_1 : \mathbb{Z}'_1), \cdots, (\mathbb{X}'_n : \mathbb{Z}'_n) \in \mathbb{P}^1$

procedure simultaneousOddIsogeny ▷ Computes multiple l-isogeny points
  $(X_1 : Z_1), \cdots, (X_d : Z_d) \leftarrow$ KernelPoints$((X_1 : Z_1), (\widehat{A} : \widehat{C}))$
  $(\widehat{X}_1, \widehat{Z}_1), \cdots, (\widehat{X}_d, \widehat{Z}_d) \leftarrow (X_1 + Z_1, X_1 - Z_1), \cdots, (X_d + Z_d, X_d - Z_d)$
  for $i = 1$ **to** n **do**
    $(\mathbb{X}'_i : \mathbb{Z}'_i) \leftarrow$ OddIsogeny$((\widehat{X}_1, \widehat{Z}_1), \cdots, (\widehat{X}_d, \widehat{Z}_d), (\mathbb{X}_i : \mathbb{Z}_i))$
    return $(\mathbb{X}'_1 : \mathbb{Z}'_1), \cdots, (\mathbb{X}'_n : \mathbb{Z}'_n)$

With simultaneousOddIsogeny, the only other algorithm needed for calculating l^e-isogenies is one for computing the multiplication-by-k map for $k \in \mathbb{Z}$ where we will call simultenousOddIsogeny e times and combine them, similar to in Section 4.1. In order to compute the multiplication-by-k map, we will use a ladder algorithm. There are multiple ways to accomplish this, but we will use the Montgomery ladder introduced by Montgomery in [8]. Let $\mathbf{P} = \mathbf{x}(P), \mathbf{P}' = \mathbf{x}([k]P)$ for ease of notation, and note that we use $(k_0 k_1 \cdots k_m)_2$ to denote the base 2 representation of k.

Algorithm 8 ladder: $\mathbb{P}^1 \times \mathbb{Z} \times \mathbb{P}^1 \to \mathbb{P}^1$

Input: $\mathbf{P} \in \mathbb{P}^1, k \in \mathbb{Z}, (\widehat{A} : \widehat{C}) \in \mathbb{P}^1$
Output: $\mathbf{P}' \in \mathbb{P}^1$

procedure ladder ▷ Computes multiple of projective x coordinate
  $V_0 \leftarrow O, V_1 \leftarrow \mathbf{P}$
  $k_0 + 2k_1 + \cdots + 2^m k_m = k$
  for $i = m$ **to** 0 **do**
    if $k_i = 0$ **then**
      $V_0 \leftarrow \text{xDBL}(V_0, (\widehat{A} : \widehat{C}))$
      $V_1 \leftarrow \text{xADD}(V_0, V_1, \mathbf{P})$
    else
      $V_0 \leftarrow \text{xADD}(V_0, V_1, \mathbf{P})$
      $V_1 \leftarrow \text{xDBL}(V_1, (\widehat{A} : \widehat{C}))$

With these, we can introduce our final algorithm SIDHIsogeny, which combines the previous subroutines to compute multiple l^e-isogenies. This can now be used for arbitrary degree isogeny computations in SIDH.

Algorithm 9 SIDHIsogeny: $\mathbb{P}^1 \times \mathbb{Z}^2 \times \mathbb{P}^1 \times (\mathbb{P}^1)^k \to \mathbb{P}^1 \times (\mathbb{P}^1)^k$

Input: $(X_1 : Z_1) \in \mathbb{P}^1, (l, e) \in \mathbb{Z}^2, (\alpha : 1) \in \mathbb{P}^1, (\mathbb{X}_1 : \mathbb{Z}_1), \cdots, (\mathbb{X}_k : \mathbb{Z}_k) \in \mathbb{P}^1$
Output: $(X_{\alpha'} : Z_{\alpha'}) \in \mathbb{P}^1, (\mathbb{X}'_1 : \mathbb{Z}'_1), \cdots, (\mathbb{X}'_k : \mathbb{Z}'_k) \in \mathbb{P}^1$

procedure SIDHIsogeny ▷ Computes multiple l^e-isogeny points
  $(X_{\alpha'} : Z_{\alpha'}), (\mathbb{X}'_1 : \mathbb{Z}'_1), \cdots, (\mathbb{X}'_k : \mathbb{Z}'_k) \leftarrow (\alpha : 1), (\mathbb{X}_1 : \mathbb{Z}_1), \cdots, (\mathbb{X}_k : \mathbb{Z}_k)$
  $(X_R : Z_R) \leftarrow (X_1 : Z_1)$
  for $i = e - 1$ **to** 0 **do**
    $(\widehat{A} : \widehat{C}) \leftarrow$ aHatFromAlpha$((X_{\alpha'} : Z_{\alpha'}))$
    $(X_S : Z_S) \leftarrow$ ladder$((X_R : Z_R), l^z, (\widehat{A} : \widehat{C}))$
    $(X_R : Z_R), (X_{\alpha'} : Z_{\alpha'}), (\mathbb{X}'_1 : \mathbb{Z}'_1) \cdots, (\mathbb{X}'_k \mathbb{Z}'_k) \leftarrow$ simultaneousOddIsogeny$((X_S : Z_S), (\widehat{A} : \widehat{C}), ((X_R : Z_R), (X_{\alpha'} : Z_{\alpha'}), (\mathbb{X}'_1 : \mathbb{Z}'_1), \cdots, (\mathbb{X}'_k : \mathbb{Z}'_k))$
  return $X_{\alpha'} : Z_{\alpha'}, ((\mathbb{X}'_1 : \mathbb{Z}'_1), \cdots, (\mathbb{X}'_n : \mathbb{Z}'_n))$

4.3 Efficient Point Operation Algorithms

The only computational difficulty that remains is computing $P + [k]Q$ for points $P, Q \in E$. Note that this differs from only using the Montgomery ladder due to adding P to it. This comes up in in every step of SIDH for computing the generator point S of subgroups. For instance, Alice must compute $S_A = P_A + [k_A]Q_A$ in her secret generation. We have the following 3 ways

(1) Using the Montgomery ladder [8] to compute $\mathbf{x}([k]Q)$ and then recovering $\mathbf{y}([k]Q)$ by the Okeya-Sakurai formula (which recovers the y coordinate given the x coordinate in a Montgomery elliptic curve [9]) and finally projectively adding $P + (\mathbf{x}([k]Q) : \mathbf{y}([k]Q) : 1)$ to get $\mathbf{x}(P + [k]Q)$

(2) Using a three-point ladder to directly compute $\mathbf{x}(P + [k]Q)$ given $\mathbf{x}(P), \mathbf{x}(Q), \mathbf{x}(P - Q)$ as a left to right ladder in terms of the bits of k as proposed in [6]. This is used in most SOTA implementations currently.

(3) A novel right to left algorithm to find $\mathbf{x}(P + [k]Q)$ given $\mathbf{x}(P), \mathbf{x}(Q), \mathbf{x}(P - Q)$ via a right to left ladder in terms of the bits of k given in [4]. One main advantage of this that Faz-Hernández et al. [4] discusses is that for static Q, this algorithm can precompute multiples of Q to further speed it up when Q is known.

5 Security

Using the same notation as before, we present the 3 major computational problems underlying isogeny-based cryptography.

Problem 5.1 (General Isogeny Problem)**.** Given $j, j' \in \mathbb{F}_p$, find isogeny $\phi : E \to E'$, if it exists, such that $j(E) = j, j(E') = j'$.

Problem 5.2 (SIDH Isogeny Problem)**.** Given public key $PK_A = (E_A, P'_B, Q'_B)$, find the isogeny ϕ_A corresponding to it.

Problem 5.3 (Decisional SIDH Isogeny Problem)**.** For any elliptic curve E', points $P'_B, Q'_B \in E'[l_2^{e_2}]$, integer $n : 0 < n \leq e$, determine whether $\exists \phi : E \to E'$ with degree l_1^n such that $P'_B = \phi(P_B), Q'_B = \phi(Q_B)$.

Notice that solving any of these efficiently poses a threat to SIDH. If we can find ϕ_A somehow, then we can recover S_A by computing the kernel and equivalently the secret key k_A. With respect to the general

isogeny problem, if given $j(E), j(E_A)$, we can find $\phi_A : E \to E_A$. The SIDH Isogeny problem is similar, though one has more information such as P'_B, Q'_B. For the decisional SIDH Isogeny problem, take some l_1-isogeny $\psi : E_A \to E'$ and $u \in \mathbb{Z} : ul_1 \equiv 1 \pmod{l_2^{e_2}}$. Then, consider the problem with parameters elliptic curve E', points $[u]\psi(P_B)$, $[e]\psi(P_Q)$ and $n = e - 1$. If we have an isogeny satisfying the conditions, then we have the first $e_1 - 1$ steps in the path between $E - E_A$ in the isogeny graph. We can repeat this for e_1 total times to get the full path corresponding to the isogeny.

5.1 Attack on Static Keys in SIDH

In this section, we will show an attack on static keys in SIDH due to Galbraith et al. [5], which will show us why we must use ephemeral keys for SIDH or SIKE. Suppose we have a neutral user Alice and an active, malicious user Bob where Alice reuses a static key. Bob, instead of sending (E_B, P'_A, Q'_A), will send (E_B, P'_A, Q''_A) where $Q''_A = Q'_A + T_2$, and T_2 is a point of order 2 on E_B. After exchange, SIDH will succeed if and only if k_A's final bit is even due to Lemma 5.4

Lemma 5.4. *For linearly independent $R, S \in E[2^n]$ with order 2^n and $a_1, a_2 \in \mathbb{Z}$ with*

$$\left\langle [a_1]R + [a_2](S + [2^{n-1}]R) \right\rangle = \langle [a_1]R + [a_2]S \rangle$$

if and only if a_2 is even

With this, Bob can learn the final bit of Alice's k_A depending on whether the SIDH exchange succeeds or not. By changing the selection of T_2, Bob can actually target each bit in k_A to learn all of Alice's static secret bit by bit. There is no current way for Alice to verify the non-maliciousness of Bob's public key, and so they must use ephemeral secret keys or SIKE, where we encapsulate the keys.

5.2 The van Oorschot-Wiener Collision Finding Algorithm

Our goal is still to find Alice's secret by finding her isogeny $\phi_A : E \to E_A$ with degree 2^{e_A}. First, we can try a generic meeting in the middle attack that can be done by building a table with all the curves that are $2^{e_A/2}$-isogenous to E and $2^{e_A/2}$-isogenous to E_A. Then, we can linearly search for a match in the table in $O(p^{\frac{1}{4}})$ time and memory, which would correspond to a 2^{e_A} isogeny $E \to E_A$ when put together.

However, the issue with the generic meet in the middle attack is the exponential memory requirement, which makes it irrelevant, as stated by the NIST. So, we will fix an upper memory bound w and give the van Oorschot-Wiener (vOW) parallel collision finding algorithm [10] for ϕ_A. The general idea behind vOW is to use a probabilistic modified version of meet in the middle. We need to trim down out the size of the table from $2 \cdot (2^{e_A/2} + 2^{e_A/2-1})$ to w, which we can do with pseudorandom walks. Afterwards, we can see if we have a collision when the memory is full; if there is none, we can discard our previous table and fill in a new one with a new pseudorandom function.

In order to cut down the size of the table, we can use a distinguishing property, which can be almost anything, as long as it gives the right fraction of elements (for example one distinguishing property we can take is to hash the numbers and then check for leading $e_A - \log_2(w)$ zeros). Now, we get to the algorithm. First, let $S \coloneqq \{\text{j-invariants } 2^{e_A/2}\text{-isogenous to } E \text{ or } E_A\}$ where $|S| = 2 \cdot (2^{e_A/2} + 2^{e_A/2-1})$. Basically, S is the original table that we would have built. Now, we will not be building S in memory but instead just be walking through it and storing only a fraction of it.

We will have a pseudorandom function $f : S \to S$ create a pseudorandom walk on S and then send the elements that satisfy the distinguishing property during the walk to memory. f takes in a j-invariant, hashes it to a string, uses one bit of this string to decide whether the isogeny is from E or E_A, and then use the rest

of the string to choose a subgroup/isogeny. Essentially, we have pseudorandom

$$f : s \xmapsto{\text{hash}} \text{string} \begin{cases} \_ \text{ (1 bit)} & : E \text{ or } E_A \\ \_\ldots\_ \text{ (rest)} & : \text{subgroup/isogeny} \end{cases}$$

To accomplish the pseudorandom walk, we start with an initial $x_0 \in S$ randomly and apply $f : x_i \mapsto x_{i+1}$ repeatedly until we reach an x_n that satisfies the distinguishing property, at which point we store (x_n, x_0) in memory (we must store initial elements to redo the walk later). We continue performing these walks and picking x_0 randomly until either memory is full or we find a repeated element. Notice that each iteration of walks are independent and so we can parallelize these processes.

Let $x \in S, y \in S$ be such that $x \neq y, f(x) = f(y) = z$, x was mapped via a a subgroup of E, and y was mapped via a subgroup of E_A. Then, z is precisely the middle j-invariant that we are looking for. It's unlikely that z will be distinguished, but as long as our walk finds x, y and continues, then we will find a repeated element in memory. Since f is deterministic, we can redo both the walks to then find x, y and recover the secret. Basically, we have

$$x_0 \mapsto x_1 \mapsto \cdots \mapsto x \mapsto z \mapsto z_1 \mapsto \cdots$$

$$y_0 \mapsto y_1 \mapsto \cdots \mapsto y \mapsto z \mapsto z_1 \mapsto \cdots .$$

Even if z is not distinguished, as long as the walk reaches both x, y, and both x, y are not distinguished, then we will eventually hit a common z_j that is distinguished, which both the walks will store into memory. Once we find a repeated element z_j in memory, we can use their initial elements x_0, y_0 to redo the same walks from the start and find the elements x, y before the first common element z, since f is deterministic.

This idea does solve the exponential memory issue; however, it creates some new problems. Since f is pseudorandom, it's possible that our choice of f does not have any x, y that satisfy what we are looking for. Even if there are such x, y, it's possible that one of them do not have preimages under f or that either one of them are distinguished. Though not a problem, it is worth noting that we may have introduced several valid pairs x, y such that $x \neq y, f(x) = f(y)$ and x mapped through E, y through E_A. Ideally, we would have like to find an f for which x, y both exist (perhaps even several) and for which many walks get to x, y under f. However, we can not actually tell how good each f is, and so after filling and refilling our memory under one f and coming up empty handed, we must find a new pseudorandom function to restart the attack. Recent analysis of the vOW collision finding algorithm gives a runtime of

$$\left(\frac{2.5}{m} \sqrt{\frac{|S^3|}{w}} \right) \cdot t$$

with m processors, where the available memory can hold w elements from the set of j-invariants S, and t is the time required for one function iteration (mostly one $l^{e_A/2}$ computation). In terms of Big-O, we know the size of $S \approx p^{1/4}$ and so the runtime is in $O(p^{3/8})$ whereas the memory is constant in $O(w)$. We can see that this is slower than the generic meet in the middle, but it is the current best known concrete algorithm for attacking SIDH. It is worth noting that Jacques and Schanck [7] researched quantum attacks via Tani's algorithm and Grover's algorithm, though noted that they provide minimal improvements, if any at all.

6 Acknowledgments

I wrote this report as part of COMP400: Honors Project in Computer Science during Spring 2022. I would like to thank Prof. Claude Crepeau for his consistent help and guidance in selecting research directions and with understanding of material.

Appendix

Here I will provide code for the efficient algorithm for arbitrary odd degree isogenies from Section 4.2 written up with SageMath in Jupyter Notebook as well as a demonstration of its correctness by comparing with isogeny computations in SageMath.

[129]:
```
## NOTE: all descriptions of arrays are 1-indexed but implementations are␣
 ↪0-indexed
```

[11]:
```
def Sage_Isogeny(S, le, a, P, d):
    E = Mont(a)
    phi = E.isogeny(S)
    res = []
    for i in P:
        res.append(phi(i))
    return (phi.codomain().j_invariant(), res)
```

[1]:
```
# Input: P.x, a
# Output: [2]P.x
def xDBL(x_P, a):
    if x_P == infinity:
        return x_P
    return (x_P ** 2 - 1) ** 2 / (4 * x_P * (x_P ** 2 + a * x_P + 1))
```

[2]:
```
# Input: P.x, Q.x, (P - Q).x
# Output: (P + Q).x
def xADD(x_P, x_Q, x_m):
    if x_P == infinity:
        return x_Q
    elif x_Q == infinity:
        return x_P
    return (x_P * x_Q - 1) ** 2 / ((x_P - x_Q) ** 2 * x_m)
```

[3]:
```
# Input: $a, b, g, d \in K$
# Output: returns the "CrissCross"; used as subroutine
def CrissCross(a, b, g, d):
    t_1, t_2 = a * d, b * g
    return t_1 + t_2, t_1 - t_2
```

[4]:
```
# Input: (X_1 : Z_2) = x(P) \in \bP^1, (A : C) = (a - 2: 4) \in \bP^1
# Output: ((X_1 : Z_2),..., (X_d : Z_d)) = (x(P), x([2]P),..., x([d]P)) \in␣
 ↪(\bP^1)^d
def KernelPoints(x_P, A, d):
    P = [None] * d
    P[0] = x_P
    if d >= 2:
        P[1] = xDBL(P[0], A)
    for i in range(2, d):
        P[i] = xADD(P[i - 1], P_1, P[i - 2])
    return P
```

[5]:
```
# Input: ((X_1, Z_1),..., (X_d, Z_d)) \in (K^2)^d, (X: Z) \in \bP^1
# Output: (X' : Z') \in \bP^1
def OddIsogeny(P, x_I, d):
    (X_hat, Z_hat) = (x_I[0] + x_I[1], x_I[0] - x_I[1])
    (Xt, Zt) = CrissCross(P[0][0], P[0][1], X_hat, Z_hat)
    for i in range(1, d):
        (t_0, t_1) = CrissCross(P[i][0], P[i][1], X_hat, Z_hat)
        (Xt, Zt) = (t_0 * Xt, t_1 * Zt)
    (Xt, Zt) = (X * Xt ** 2, Z * Zt ** 2)
    return (Xt, Zt)
```

[6]:
```
# Input: (X_1 : Z_1) \in \bP^1, (A : C) \in \bP^1, ((X_1 : Z_1),..., (X_n :␣
 ↪_n)) \in (\bP^1)^d
# Output: ((X_1t : Z_1t),..., (X_nt : Z_nt)) \in (\bP^1)^n
def SimultaneousOddIsogeny(x_P, A, P, d):
    P_kernels = KernelPoints(x_P, A, d)
    P_hat = [(x + z, x - z) for (x, z) in P_kernels]
    print(P_hat, P[0])
    Pt = [OddIsogeny(P_hat, P[i], d) for i in range(0, n)]
    return Pt
```

[7]:
```
# Input: x(P), (A : C), l^z \in \bP^1, l, z
# Output: x([l^z]P)
def LADDER(x_P, a, l, z):
    n = l ** z
    m = int((log(n) / log(2)).n())
    R0 = infinity
    R1 = x_P
    for i in range(m, 0, -1):
        xi = (n // 2 ** i) % 2
        if xi == 0:
            (R0, R1) = (xDBL(R0, a), xADD(R0, R1, x_P))
        else:
            (R0, R1) = (xADD(R0, R1, x_P), xDBL(R1, a))
    return R0
```

[8]:
```
# Input: \alpha
# Output: a s.t. (a : 1) = (1 + \alpha^2 + : -\alpha)
def a_from_alpha(alpha):
    return P1((1 + alpha[0] ** 2) / (-1 * alpha[0]), 1)
```

[9]:
```
# Input: \alpha
# Ouput: a s.t. \alpha^2 + a\alpha + 1 = 0 (we take plus in quad form.)
def alpha_from_a(a):
    return ((-1) * a + sqrt(a ** 2 - 4)) / 2
```

[10]:
```
# Input: x(P) = (X_1 : Z_1) \in \bP^1, (l, e) \in \Z^2 with |<P>| = l^e,
# Input: (\alpha : 1) \in \bP^1 with order 2, (x_(Q_1),..., x(Q_k)) \in␣
 ↪(\bP^1)^k
```

```
# Output: (X_\alpha': Z_\alpha') \in \bP^1 with order 2,
# Output: (x(\phi(Q_1),..., x(\phi(Q_k)))) \in (\bp^1)^k
def SIDH_Isogeny(x_P, le, alpha, P, d):
    (x_alpha, Pt) = (alpha, P)
    x_R = x_P
    for z in range(le[1] - 1, -1, -1):
        A_hat = a_from_alpha(x_alpha)
        x_S = P1(LADDER(x_R[0]/x_R[1], A_hat[0]/A_hat[1], le[0], z), 1)
        (x_R, x_alpha, Pt) = SimultaneousOddIsogeny(x_S, A_hat, (x_R, x_alpha,␣
 ↪Pt), d)
    return x_alpha, Pt
```

[ ]:
```
# p = 2 ** 216 * 3 ** 137 - 1
p = 2 ** 4 * 3 ** 3 - 1
_.<I> = GF(p)[]
K.<i> = GF(p ** 2, modulus=I ** 2+1)
K(-1).nth_root(2, all=True)
```

[13]:
```
def Mont(a):
    return EllipticCurve(K, [0, a, 0, 1, 0])
a = 329 * i + 423
E = Mont(a)
E.j_invariant()
```

[13]:
```
87*i + 190
```

[14]:
```
P_A = E((100 * i + 248, 304 * i + 199))
Q_A = E((426 * i + 394, 51 * i + 79))
P_B = E((358 * i + 275, 410 * i + 104))
Q_B = E((20 * i + 185, 281 * i + 239))
(l_A, e_A) = (2, 4)
(l_B, e_B) = (3, 3)
type(P_A)
```

[14]:
```
<class 'sage.schemes.elliptic_curves.ell_point.EllipticCurvePoint_finite_field'>
```

[15]:
```
P1.<X, Z> = ProjectiveSpace(1, K)
```

[16]:
```
# Input: point P \in \bP^2
# Output: point x(P) = P_x \in bP^1
def x(P):
    return P1(P[0], P[2])
```

[17]:
```
S_A = P_A[0] + LADDER(Q_A[0], a, 11, 1)
S_A
```

[17]:
```
126*i + 112
```

[21]:
```
k_A = 11
k_B = 2
```

[31]:
```
S_A = P_A + k_A * Q_A
phi_A = E.isogeny(S_A)
PK_A = (phi_A.codomain(), phi_A(P_B), phi_A(Q_B))
phi_A.codomain().j_invariant()
```

[31]: 222*i + 118

[33]:
```
S_B = P_B + k_B * Q_B
phi_B = E.isogeny(S_B)
PK_B = (phi_B.codomain(), phi_B(P_A), phi_B(Q_A))
print(PK_B)
phi_B.codomain().j_invariant()
```

```
(Elliptic Curve defined by y^2 = x^3 + (329*i+423)*x^2 + (215*i+415)*x +
(419*i+369) over Finite Field in i of size 431^2, (402*i + 424 : 191*i + 157 :
1), (198*i + 336 : 29*i + 287 : 1))
```

[33]: 344*i + 190

[34]:
```
S_At = PK_B[1] + k_A * PK_B[2]
phi_AB = PK_B[0].isogeny(S_At)
print(phi_AB.codomain())
phi_AB.codomain().j_invariant()
```

```
Elliptic Curve defined by y^2 = x^3 + (329*i+423)*x^2 + (358*i+326)*x +
(237*i+2) over Finite Field in i of size 431^2
```

[34]: 234

[35]:
```
S_Bt = PK_A[1] + k_B * PK_A[2]
phi_BA = PK_A[0].isogeny(S_Bt)
print(phi_BA.codomain())
phi_BA.codomain().j_invariant()
```

```
Elliptic Curve defined by y^2 = x^3 + (329*i+423)*x^2 + (358*i+326)*x +
(237*i+2) over Finite Field in i of size 431^2
```

[35]: 234

[36]:
```
res_B = SIDH_Isogeny(S_B, [3, 3], a, (P_A, Q_A), 1)
print(res_B)
```

```
(344*i + 190, [(402*i + 424 : 191*i + 157 : 1), (198*i + 336 : 29*i + 287 : 1)])
```

References

[1] Craig Costello. *Supersingular Isogeny Key Exchange for Beginners*. Jan. 2020, pp. 21–50. ISBN: 978-3-030-38470-8. DOI: 10.1007/978-3-030-38471-5_2.

[2] Craig Costello and Huseyin Hisil. "A simple and compact algorithm for SIDH with arbitrary degree isogenies". In: *International Conference on the Theory and Application of Cryptology and Information Security*. Springer. 2017, pp. 303–329.

[3] Luca De Feo. "Mathematics of isogeny based cryptography". In: *arXiv preprint arXiv:1711.04062* 12 (2017).

[4] Armando Faz-Hernández et al. "A Faster Software Implementation of the Supersingular Isogeny Diffie-Hellman Key Exchange Protocol". In: *IEEE Transactions on Computers* 67.11 (2018), pp. 1622–1636. DOI: 10.1109/TC.2017.2771535.

[5] Steven D. Galbraith et al. "On the Security of Supersingular Isogeny Cryptosystems". In: *Advances in Cryptology – ASIACRYPT 2016*. Ed. by Jung Hee Cheon and Tsuyoshi Takagi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 63–91. ISBN: 978-3-662-53887-6.

[6] David Jao and Luca Feo. "Towards Quantum-Resistant Cryptosystems from Supersingular Elliptic Curve Isogenies Post-Quantum Cryptography". In: vol. 2011. Nov. 2011, pp. 19–34. ISBN: 978-3-642-25404-8. DOI: 10.1007/978-3-642-25405-5_2.

[7] Samuel Jaques and John Schanck. "Quantum Cryptanalysis in the RAM Model: Claw-Finding Attacks on SIKE". In: Aug. 2019, pp. 32–61. ISBN: 978-3-030-26947-0. DOI: 10.1007/978-3-030-26948-7_2.

[8] Peter L. Montgomery. "Speeding the Pollard and Elliptic Curve Methods of Factorization". In: *Mathematics of Computation* 48.177 (1987), pp. 243–264. ISSN: 00255718, 10886842. URL: http://www.jstor.org/stable/2007888 (visited on 04/22/2022).

[9] Katsuyuki Okeya and Kouichi Sakurai. "Efficient Elliptic Curve Cryptosystems from a Scalar Multiplication Algorithm with Recovery of the y-Coordinate on a Montgomery-Form Elliptic Curve". In: *Cryptographic Hardware and Embedded Systems - CHES 2001, Third International Workshop, Paris, France, May 14-16, 2001, Proceedings*. Ed. by Çetin Kaya Koç, David Naccache, and Christof Paar. Vol. 2162. Lecture Notes in Computer Science. Springer, 2001, pp. 126–141. DOI: 10.1007/3-540-44709-1_12. URL: https://doi.org/10.1007/3-540-44709-1%5C_12.

[10] Paul C. van Oorschot and Michael J. Wiener. "Parallel Collision Search with Cryptanalytic Applications". In: *J. Cryptology* 12 (1999), pp. 1–28. DOI: 10.1007/PL00003816.

[11] Joseph H. Silverman. *The Arithmetic of Elliptic Curves*. 2nd ed. 2009.

[12] Jacques Vélu. "Isogénies entre courbes elliptiques". In: *CR Acad. Sci. Paris Sér. AB* 273 (1971), A238–A241.