Attacking SIKE

- A survey of known attacks -

Nikolai T. Opdan

August 18, 2021

Abstract

We survey known attacks against NIST's post-quantum cryptography standardization candidate SIKE, ranging from specialized attacks on static keys, curves defined over $\mathbb{F}_p$, unbalanced keys, and general attacks such as the Tani's quantum algorithm, the classical vOW-algorithm, subgraphs, and side-channel attacks.

1 Introduction

Due to the threat of Shor's algorithm on quantum computers breaking many of todays public-key cryptosystems in polynomial time, the National Institute of Standards and Technology (NIST) is currently hosting a process of finding new post-quantum cryptographic standardizations. One of the remaining candidates is the cryptosystem SIKE (Supersingular Isogeny Key Encryption), the only candidate employing elliptic curves. In the process of establishing new standardizations, the security of the candidates is investigated by many researcher. It is the purpose of this article to collect and survey such attacks.

The attacks on SIKE range from specialized attacks on parameters and implementations of the underlying SIDH protocol, to general attacks by classical and quantum computers. By an attack we mean an algorithm that may (or may not) be implemented on a (quantum) computer in order to break (or weaken) the security of the protocol. Although specialized, it is important that an implementation of SIKE must avoid the conditions under with specialized attacks operate. In considering such attacks we therefore also discuss methods of prevention, and thereby relevance. For general attacks it is interesting to investigate the necessary resources needed to carry the attack. This provides us with an estimate on the future security of the protocol.

The article is structured as follows: In section 2 we give the basic preliminaries on elliptic curves. The protocol uses a considerable amount of mathematics, and we have in this section tried to include the necessary background material for non-specialists. In section 3 we explore the underlying SIDH-protocol, while section 4 is about SIKE, an explicit implementation of SIDH. The articles main body is section 5, in which we survey known attacks on SIKE. We have divided the attacks into specialized attacks, i.e. attacks operating under certain conditions, and generalized attacks, that is attacks assuming no extra information. We end with a short discussion in section 6 about generalizing the SIDH-protocol to curves of higher genus (hyperelliptic curves) and varieties of higher dimension (abelian varieties).

2 Preliminaries

For our purposes we are interested in elliptic curves on *Montgomery form* [1]

$$E_{a,b} : by^2 = x^3 + ax^2 + x,$$

where a, b are elements in the finite field $\mathbb{F}_{p^2}$ with p^2 elements. The points of $E_{a,b}$ over a general field k, denoted $E_{a,b}(k)$, consists of pairs $(x, y) \in k \times k$ satisfying the equation $by^2 = x^3 + ax^2 + x$, together with the point $\mathcal{O}$ at infinity. We often have $b = 0$, in which case we write $E_a := E_{a,0}$, and if the value a is understood from context, we will often just abbreviate E_a as E.

Our prime p is usually a very large prime of the form $p \equiv 3 \bmod 4$, and of the p^2 elements in $\mathbb{F}_{p^2}$ we are interested in a subset of $\lfloor p/12 \rfloor + 2$ elements [Sil09, Theorem V.4.1(c)]. These are the supersingular j-invariants over $\mathbb{F}_{p^2}$, which corresponds exactly to the *supersingular* elliptic curves over $\mathbb{F}_{p^2}$. Indeed, every elliptic curve have a unique (up to isomorphism) j-invariants over $\mathbb{F}_{p^2}$ given by

$$j(E_a) = \frac{256(a^2 - 3)^3}{a^2 - 4},$$

and every j-invariant over $\mathbb{F}_{p^2}$ corresponds to exactly one (isomorphism class of an) elliptic curve. For cryptographic purposes, singular elliptic curves have the most efficient algorithms, and all known classical and quantum attacks have exponential complexity (contrary to the case for *ordinary* curves).

Elliptic curves have the important property that their points can be given a group structure: Geometrically for $E_a(\mathbb{R})$, we define the multiplication $P * Q$ of two distinct points P and Q by first drawing the line connecting them. Such a line intersects the curve in a third point $-R$. Subject to the group law $P * Q * R = 0$, we define the point R as the reflection of $-R$ by the x-axis (see fig. 1). Points are added to themselves by instead drawing the tangent, and the point at infinity $\mathcal{O}$ acts as the identity. This creates a group structure on $E_a(\mathbb{R})$.

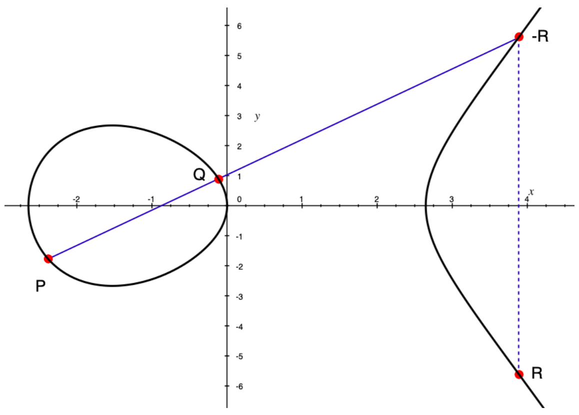


Figure 1: Point multiplication on the real solutions of the elliptic curve (on Weierstrass-form) $y^2 = x^3 - 7x$.

For our purposes, we are interested in $E(\mathbb{F}_{p^2})$, which is just a collection of distinct points (see fig. 2). Here our geometric intuition is no longer valid, but we may define the same point

[1] For a general field k of characteristic not 2 or 3, every elliptic curve is isomorphic to an elliptic curve on Weierstrass form $y^2 = x^3 + ax + b$.

multiplication algebraically: Given two distinct points $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ we define $P_3 := P_1 * P_2$ by

$$\left(\frac{(x_2 y_1 - x_1 y_2)^2}{x_1 x_2 (x_2 - x_1)^2}, \frac{(2x_1 + x_2 + a)(y_2 - y_1)}{x_2 - x_1} - \frac{(y_2 - y_1)^2}{(x_2 - x_1)^3} - y_1 \right).$$

An important subgroup is the group generated by one point P, denoted $E[P] := \langle P \rangle$.

An *isogeny* is a map $\phi : E \to E'$ that preserves the point at infinity, i.e. $\phi(\mathcal{O}) = \mathcal{O}$, and it induces a map between the corresponding group structure. An isogeny from a curve to itself is called an *endomorphism*, where the *multiplication-by-n* map

$$[n] : \begin{cases} E & \longrightarrow & E \\ P & \longmapsto & [n]P \end{cases},$$

where $[n]P$ denotes P added to itself n-times, is an important example. We denote the set of all endomorphism on E by $\mathrm{End}(E)$. Composing isogenies gives a new isogeny, and the existence of an *dual* isogeny $\hat{\phi}$ for every isogeny ϕ, equips this set with a group structure. For a supersingular elliptic curves it is always isomorphic to the $\mathbb{Q}$-algebra

$$\mathbb{Q} \oplus i\,\mathbb{Q} \oplus j\,\mathbb{Q} \oplus ij\,\mathbb{Q},$$

where $i^2 = -1, j^2 = -p$ and $ij = -ji$, although the isomorphism is often highly non-canonical.

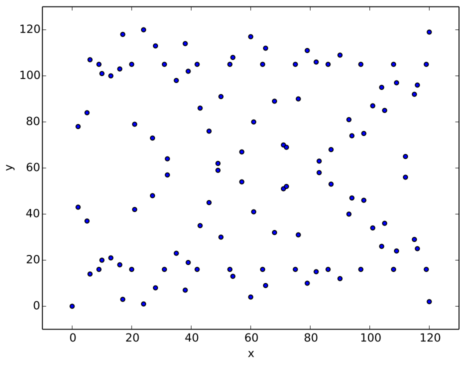


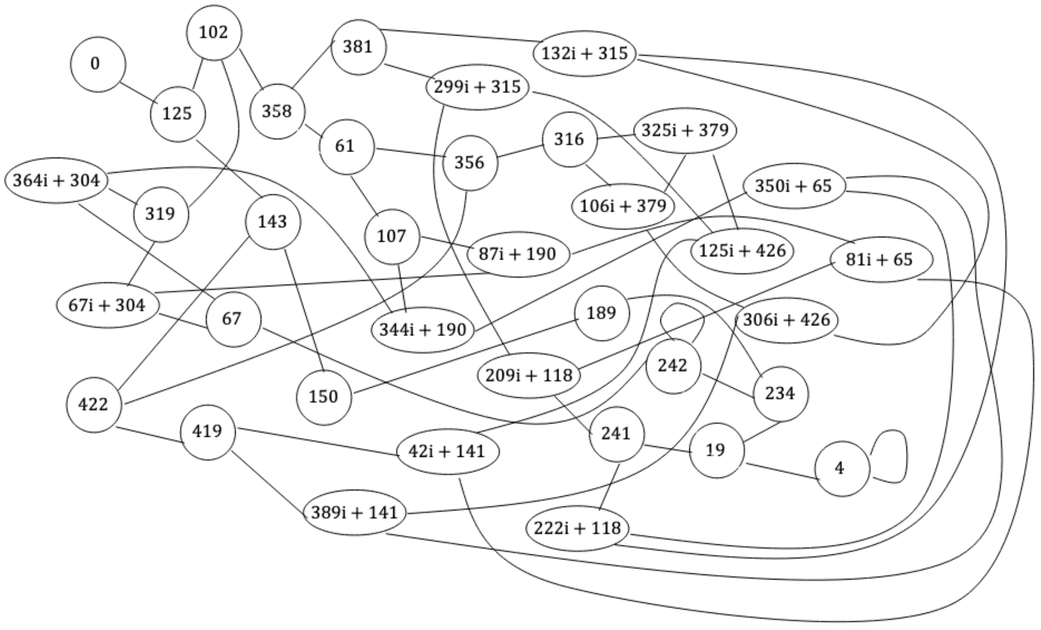
Figure 2: The elliptic curve $E_6(\mathbb{F}_{11}^2)$.

Montgomery curves are often the preferred choice of cryptographers since it enables x-only arithmetic. That is given a endomorphism ϕ we can recover the image of the y-coordinate from the x-coordinate by $c \cdot \phi'(x)$ for a fixed constant c. This gives the *doubling* map on Montgomery curves the simple form

$$[2] : \begin{cases} E_a & \longrightarrow & E_a \\ x & \longmapsto & \frac{(x^2-1)^2}{4x(x^2+ax+1)} \end{cases},$$

and the *tripling* map the form

$$[3] : \begin{cases} E_a & \longrightarrow & E_a \\ x & \longmapsto & \frac{x(x^4-6x^2-4ax^3-3)}{(3x^4+4ax^3+6x^2-1)^2} \end{cases}.$$



©Costello, *Supersingular isogeny key exchange*

Figure 3: The supersingular Isogeny graph $G(431, 2)$.

An important fact is that for every subgroup $G \subset E(\mathbb{F}_{p^2})$ there exists a unique curve E' and an (separable) isogeny $\phi : E \to E'$ with kernel G. Explicit formulas are provided by Vélu [Vél71].

The kernel of the multiplication-by-l map is isomorphic to $\mathbb{Z}/l \oplus \mathbb{Z}/l$ when $p \nmid l$, and an isogeny given as a subgroup of $\mathbb{Z}/l \oplus \mathbb{Z}/l$ is called an *l-isogeny*, for which there are $l + 1$ different maps. We call two elliptic curves *l-isogenous* if there is an l-isogeny between them. Moreover the image curve of a supersingular elliptic curve by an l-isogeny remains supersingular. Hence we can consider the *supersingular isogeny* graph $G(p, l)$ whose nodes are supersingular j-invariants over $\mathbb{F}_{p^2}$ and the edges are l-isogenies (see section 2). This graph is connected and l regular, and for sufficiently large p it is an *Ramanujan graph* ([Piz90]). This means roughly that a relatively small number of steps (logarithmic in the number of nodes) is needed for a random walk to converge to a random distribution, which is important for cryptographic applications. The graph is directed, but since there for every isogeny exists a dual isogeny going in the opposite direction, we often draw the graph undirected.

3 The SIDH-protocol

SIDH (Supersingular Isogeny Diffie-Hellman) was introduced in 2011 by Jao and De Feo in [JD11]. It is one of two cryptosystems that is based on walks in supersingular isogeny graphs [2]. The basic idea is to have Alice and Bob take random paths in two distinct isogeny graphs with the same vertex set.

We begin by choosing a prime p of the form $p := f \cdot 2^{e_A} 3^{e_B} - 1$, with $f, e_A, e_B \in \mathbb{N}$, and fix an elliptic curve E (on Montgomery form). We often have (for reasons we will see later) $2^{e_A} \approx 3^{e_B}$ and choose the parameter f to make p prime. Since such primes are abundant we can often have f equal to 1.

The protocol begin with Alice choosing points

$$\langle P_2, Q_2 \rangle \in E[2^{e_A}] \cong \mathbb{Z}_{2^{e_A}} \oplus \mathbb{Z}_{2^{e_A}},$$

on E, and a secret point S_2, typically (in SIKE) by

$$S_2 := P_2 + [k_A]Q_2, \qquad k_A \in [[0, 1, \cdots 2^{e_A})),$$

where $[[0, 1, \cdots 2^{e_A}))$ denotes the set of integers in the interval $[0, 1, \cdots 2^{e_A})$.

Meanwhile, Bob chooses points

$$\langle P_3, Q_3 \rangle = E[3^{e_B}] \cong \mathbb{Z}_{3^{e_B}} \oplus \mathbb{Z}_{3^{e_B}},$$

and a secret point

$$S_3 := P_3 + [b_2]Q_3, \qquad b_1, b_2 \in [[0, 1, \cdots 3^{e_B})).$$

Alice then finds an isogeny $\phi_A : E \to E_A$, with $E_A := E/\langle S_2 \rangle$ in the 2-isogeny graph $G(p^2, 2)$, and publishes her public key

$$PK_A := \big(E_A, P_3', Q_3'\big) = \big(\phi_A(E), \phi_A(P_3), \phi_A(Q_3)\big),$$

while Bob computes $\phi_B : E \to E_B$ in the 3-isogeny [3] $G(p^2, 3)$ and publishes his public key

$$PK_B := \big(E_B, P_2', Q_2'\big) = \big(\phi_B(E), \phi_B(P_2), \phi_B(Q_2)\big).$$

By using Bobs public key, Alice calculates $S_2' = P_2' + [k_a]Q_2'$ and $\phi_A' : E_B \to E_{AB}$ where $E_{AB} = E_B/\langle S_2' \rangle$, and finds the shared secret $j_{AB} = j(E_{AB})$.

On the other hand, Bob uses Alice's public key and computes $S_3' = P_B' + [k_B]Q_3'$ and $\phi_B' : E_A \to E_{BA}$ with $E_{BA} = E_A/\langle S_2' \rangle$, and finds the secret $j_{BA} = j(E_{BA})$, which agrees with Alice's secret j_{AB}.

The image points $\phi_B(P_2), \phi_B(Q_2), \phi_A(P_3), \phi_A(Q_3)$ are needed to circumvent the important fact (which does not make SIDH vulnerable to quantum attacks) that composing isogenies $\phi_A : E \to E_A$ and $\phi_B : E \to E_B$ does not make sense. Hence we need a way for Alice to compute a isogeny starting from Bob's curve, and visa versa. Moving through each other base points solves this problem, which makes it possible to arrive at the same $j-$invariant.

[2] The other one first appeared in [Cou06], and was later popularized in [RS06] and [Sto10]. That being said, the protocol is very slow, and more importantly, [CJS14] found an quantum algorithm that breaks the protocol in subexponential time.

[3] We choose the primes 2 and 3 for simplicity and efficiency. In general a larger prime l yields a more complicated l-isogeny graph.

Initial parameters:	
	Prime number $p := f \cdot 2^{e_A} 3^{e_B} - 1$
	Elliptic curve E_a, $a \in \mathbb{F}_{p^2}$
	Points $P_2, Q_2 \in E[2^{e_A}]$ and $P_3, Q_3 \in E[3^{e_B}]$
Public keys:	
	$PK_A := \left(E_A, P_3', Q_3'\right)$
	$PK_B := \left(E_B, P_2', Q_2'\right)$
Secret information:	
	Alice's secret key $k_A \in [[0, \cdots 2^{e_A}))$
	Bob's secret key $k_B \in [[0, \cdots 3^{e_B}))$
	Alice's secret isogeny $\phi_A : E \to E_A$
	Bob's secret isogeny $\phi_B : E \to E_B$
	Shared secret $j(E_{AB}) = j(E_{BA})$

Table 1: Information in the SIDH-protocol.

4 SIKE

SIKE (Supersingular Isogeny Key Encapsulation) is a implementation of SIDH currently under consideration by the US National Institute of Standards and Technology (NIST) for becoming (one of) the standardized post-quantum cryptosystems [Jao+19]. It is currently a round 3 "alternative candidate".

The submission consists of four proposals,

$$\texttt{SIKEpXXX} \text{ for } \texttt{XXX} \in \{434, 503, 610, 751\},$$

where `XXX` corresponds to the bit-length of the given prime. These are chosen so as to meet, respectfully, NIST's level 1, 2, 4, 5 security requirements.

The public parameters are

- Two positive integers e_A and e_B such that $p = 2^{e_A} 3^{e_B} - 1$ is a large prime and $2^{e_A} \cong 3^{e_B}$. The smallest instance of SIKE, `SIKEp434`, uses the prime $p = 2^{216} 3^{137} - 1$. We will refer to this prime as $p434$.
- The starting curve
$$E_6 : y^2 = x^3 + 6x^2 + x.$$
- Two points $P_2, Q_2 \in E[2^{e_A}]$ and two points $P_3, Q_3 \in E[3^{e_B}]$.

We make $2^{e_A} \cong 3^{e_B}$ in order for it to be equally hard to attack either Alice's or Bob's private key. Moreover, for very unbalanced parameters ($3^{e_B} \gg 2^{e_A}$) there are attacks on SIKE running in polynomial time (see section 5.1.3).

We usually fix the representation

$$\mathbb{F}_{p^2} \cong \mathbb{F}_p(i) \cong \mathbb{F}_p[x]/(x^2 + 1)$$

and define the points P_2, Q_2, P_3 and Q_3 as sums where $P_2 = P_{A,1} + i \cdot P_{A,2}$ and $Q_2 = Q_{A,1} + i \cdot Q_{A,2}$ (and similar for P_3 and Q_3). For efficiency reasons we also provide $R_2 = P_2 - Q_2$ and $R_3 = P_3 - Q_3$.

	Public paramteters: $a, p, e_A, e_B, P_2, Q_2, P_3, Q_3$		
	Alice	Public Domain	Bob
1	Choose $k_A \in [[0, 2^{e_A}))$		Choose $k_B \in [[0, 3^{e_B}))$
2	Calculate the group $\langle P_2 + [k_A]Q_2 \rangle$		Calculate the group $\langle P_3 + [k_B]Q_3 \rangle$
3	Find $E_A := {}^{E}/_{\langle P_2+[k_A]Q_2 \rangle}$, and $\phi_A : E \to E_A$		Find $E_B := {}^{E}/_{\langle P_3+[k_B]Q_3 \rangle}$, and $\phi_B : E \to E_B$
4	Compute $\phi_A(P_3), \phi_A(Q_3) \in E_A$		Compute $\phi_B(P_2)\phi_B(Q_2) \in E_B$
5	Send PK_A $\dashrightarrow$ Receive ● $\dashleftarrow$	$\dashrightarrow$ $\dashleftarrow$	Receive $\dashrightarrow$ ● Send $\dashleftarrow$ PK_B
6	Compute $E_{BA} := {}^{E_B}/_{\langle \phi_B(P_2)+[k_A]\phi_B(Q_2) \rangle}$		Compute $E_{AB} := {}^{E_A}/_{\langle \phi_A(P_3)+[k_B]\phi_A(Q_3) \rangle}$
7	Compute shared key $j(E_{BA})$		Compute shared key $j(E_{AB})$

Table 2: Summary of the SIDH-protocol.

Hence the actual protocol contains the parameter set

$$p, E_6, A, P_{A,1}, P_{A,2}, Q_{A,1}, Q_{A,2}, R_{A,1}, R_{A,2},$$
$$B, P_{B,1}, P_{B,2}, Q_{B,1}, Q_{B,2}, R_{B,1}, R_{B,2}.$$

The protocol also include parameters sets

$$\texttt{SIKEpXXXc} \text{ for } \texttt{XXX} \in \{434, 503, 610, 751\},$$

which employs public key compression to trade-off 12.5% larger ciphertexts with 1.6-1.7 times faster decapsulation.

SIKE's initial proposal used the curve $E_0 : y^2 = x^3 + x$ [Jao+19][NIST 1. round proposal], but they later changed it to $E_6 : y^2 = x^3 + 6x^2 + x$ [Jao+19][§1.3.2] (with j-invariant 287496). The reason for this was that the initial curve had j-invariant 1728, which made it having only one non-isomorphic 2-isogeny curve. This meant that an attacker would know the first step taken in $G(p^2, 2)$. Moreover, there are only two non-isomorphic 3-isogenous curves with j-invariant 1728. Although, the NIST submission does not mention the choice of curve, it seems likely that the initial curve was chosen since it is the *smallest* (of the possible values of a) elliptic curve over $\mathbb{F}_{p^2}$. It seems that the choice of the second curve, E_6, is due to being the smallest supersingular curve with j-invariant not 1728, see fig. 4. In general, [dQue+21] argues that for certain curves E_a one can generate backdoors which enables polynomial-time attacks. Moreover, such curves are difficult to distinguish from random curves, hence we should not trust curves coming from unknown sources.

As we will see below (section 5.1.1), SIDH is vulnerable to the use of static keys, that is where Alice (or Bob) uses a long term secret isogeny. For this reason instances of SIDH must either insist on all parties not reusing keys or maintain a way for Alice to reuse a long-term secret. SIKE employs the last strategy: Instead of Bob choosing his secret k_B at random, he lets $k_B = H(PK_A, m)$, where m is a chosen integer and H is a cryptographic hash function. He then

a	$j(E_a)$	Supersingular
0	1728	Yes
1	2048/3	No
2	-	-
3	55296/3	No
4	140608/3	No
5	2725888/21	No
6	287496	Yes
7	24918016/45	No
8	14526784/5	No
9	121485312/77	No
10	7301384/3	

Figure 4: j-invariants of the first curves over $\mathbb{F}_{p434^2}$. Here supersingular means supersingular for $p434, p503, p610$, and $p751$.

XOR the value m, and instead of sending the public key, he sends Alice $(PK_B, H(j(E_{BA})) \oplus m)$ where $H(j(E_{AB}))$ is a hash of the shared secret $j(E_{AB})$ and $\oplus$ denotes the XOR-operation. Alice can then hash her shared secret with the same hash and use this to recover m. She uses this to calculate Bob's secret k_B which enables her to ensure that PK_B is as it should be.

The advantage of SIKE compared to other NIST proposals is in its very short key size. However, this comes with the cost of being the slowest of all candidates; Using data from the *ECRYPT Benchmarking of Cryptographic Systems* we see that an implementation of `SIKEp503` uses 161613410 cycles (on average), in comparison to another NIST candidate McEliece ([Ber17]) which only uses 875025 cycles (a factor of 100 times less). Moreover, SIKE allows for no decryption errors, no complicated statistical distributions of error vectors, and no "reconciliation" in the protocol. We (as we will see in section 5.2) also emphasize that all known generic attacks on SIKE are exponential, even on quantum computers.

5 Attacking SIKE

Most of todays cryptosystems are breakable in subexponential time on quantum computers. For the widely used Elliptic Curve Diffie-Hellman (ECDH)-protocol this is largely due to the commutativity of the group structure of its points which enables an attack based on commutative ring theory [CJS14]. SIDH avoids such attack because of its non-commutative nature (see section 3).

All known attacks on SIKE focuses on first recovering one party's private key to obtain the shared secret. Since SIKE uses balances parameters we will always assume to be attacking Alice's private key. Specifically, they all try to solve the following problem:

The SuperSingular Isogeny problem.
[SSI] Given elliptic curves E and E_A, find an isogeny $\phi : E \to E_A$.

An answer to this problem gives explicit descriptions of $\text{End}(E)$ and $\text{End}(E_A)$ which, by [Gal+16a, Theorem 4.1], enables us to recover the isogeny of smallest degree between the curves. Since Alice's isogeny is much smaller than the diameter of the graph ($216 \ll log(p^2)$), this is very likely to be Alice's secret isogeny.

The SSI-problem has so far resisted all attacks, i.e. all known attacks requires exponential (in the bit length of the keys) time or memory, even on quantum computers. Actually, as [Cos21] points out, classical attacks currently beat attacks by quantum computers. However, SIKE has only been around for about a decade, which is relatively new with respect to cryptographic standards. Moreover, limited research has been done on quantum-algorithms, which is a general concern for all post-quantum cryptosystems.

An attacker not only has information of the two curves; she also has knowledge of the points $\phi_A(P_2), \phi_A(Q_2), \phi_B(P_3)$ and $\phi_B(Q_3)$. This she might use to her advantage, and instead solve the following problem:

The SuperSingular Isogeny problem 2.
[SSI-T] Let ϕ_A be a secret isogeny between E and E_A. Given knowledge of the points $\phi_A(P_3)$ and $\phi_A(Q_3)$ compute ϕ_A.

However, we know of no such attacks that are more efficient than those attacks approaching the first problem. Hence it is assumed that knowledge of the auxiliary points does not leak information to an attacker.

5.1 Specialized attacks

Specialized attacks are attacks that operates under certain assumptions, and in this section we will be presenting such attacks. An effective implementation of SIDH must therefore circumvent such conditions. In presenting the specialized attacks, we therefore also present their security relevance, and ways in which they can be avoided. SIKE employs all such security measures, hence no specialized attacks are known to break SIKE.

5.1.1 Static Keys

One of few successful attacks on SIDH is [Gal+16b] from 2016 where it was shown that the protocol is vulnerable to the use of static keys. Indeed, they show that a malicious party can recover the entire private secret by preforming as many interactions as the bit-length of the secret.

Lets say that Alice performs many interactions (ex. a server), and due to the inconvenience of computing points on elliptic curves and corresponding subgroups, she decides to reuse her public key. On the other hand, lets assume that Eve wants to learn Alice's secret. Then instead of following the protocol, Eve adds a point T_2 of order 2 (i.e. $[2]T_2 = \mathcal{O}$) to her second image point and sends

$$PK'_E = (E_B, P'_2, Q'_2 + T_2).$$

She then takes Alice's static key and her secret to arrive at what would be the shared secret, while Alice follows the protocol and calculates

$$S_2^! := P'_2 + [k_A](Q'_2 + T_2) = P'_2 + [k_A]Q'_2 + [k_A]T_2.$$

Since T_2 is a point of order 2, the extra addend will only vanish if k_A is even. If her secret is odd, the protocol fails, and she lands at a wrong shared secret $E_B/\langle S_2^! \rangle$. Either way Eve learns a bit of Alice's secret by analyzing whether the protocol fails or not.

Because of this, an implementation of SIDH must either insist on all parties using each key exactly once, or implement a way of securing that both parties arrive at the same common secret. SIKE overcomes this obstacle by insisting on Bob sending not only his public key PK_B, but also a hash of the common secret $H(j)$ (see section 4). Alice can then check if her common secret matches Bobs hash before using the common secret for encryption, which could otherwise leak information of her secret key.

5.1.2 Initial and terminal curve over $\mathbb{F}_p$

Exploiting the $\mathbb{F}_p$ subgraph was first studied in by Delfs-Gelbraith in [DG13] which led to a classic attack on SIKE. This attack applies when both the initial and terminal curve happen to be defined over $\mathbb{F}_p$ (contrary to the ordinary case of being defined over $\mathbb{F}_{p^2}$). Moreover, Biasse, Jao and Sankar improved this in [BJS14] to an efficient quantum attack which solves the SSI-problem in sub-exponential time.

The $\mathbb{F}_p$-subgraph is a distinguished subset of the full isogeny graph with approximately $\sqrt{p/12}$ elements, which is easily recognizable once we have found it. The attack exploits the fact that it makes it very simple to compute isogenies between the corresponding nodes. Specifically, the quantum attack reduces the time cost of connecting two $\mathbb{F}_p$-curves to

$$L_p(1/2, \sqrt{3/2}) := \exp\left(\left(\sqrt{3/2} + o(1)\right) \frac{\sqrt{\log p}}{\sqrt{(\log \log p)}}\right)$$

We mention that the initial curve E_6 is already defined over $\mathbb{F}_p$, hence the security of SIKE relies on the curves E_A and E_B not randomly being defined over $\mathbb{F}_p$. That being said the plausibility of that happening becomes exceedingly small as p gets very large. We include a general discussion of exploiting other subgraphs in section 5.2.3.

5.1.3 Exploiting the torsion points, Petit's attack

In SIDH we are not only given the curves of Alice and Bob; We also have the auxiliary points P_2 and Q_2 on E_A, and P_3 and Q_3 on E_B. A torsion point attack is an attack on the Supersingular Isogeny problem 2.

Petit found in [Pet17] a way of exploiting the auxiliary points if the parameters 2^{e_A} and 3^{e_B} are very unbalanced, i.e. if $3^{e_B} \gg 2^{e_A}$. He used this information to construct an endomorphism $\phi \in \mathrm{End}(E_A)$, which using the auxiliary points, enables him to evaluate in on the 3^{e_B}-torsion in polynomial time to reconstruct Alice's secret isogeny ϕ_A. It is for this reason that SIKE uses balances parameters $2^{e_A} \cong 3^{e_B}$.

On balanced parameters this attack performs much worse than brute force attacks and is therefore irrelevant for the SIKE-protocol.

5.1.4 Exploiting the torsion points, Quantum Hidden Shift Attack (QHSA)

The security of the SIDH to quantum attacks lies in the non-commutative nature of its endomorphism ring, which does not make it vulnerable to Kuperberg's subexponential algorithm (that breaks DH, ECDH, etc.).

It turns out that [dQue+21] found another way of exploiting the torsion points, named the *Quantum Hidden shift attack* (QHSA). Namely elements θ in the quotient group

$$G = (\mathrm{End}(E)/(e_A \cdot \mathrm{End}(E)))^*$$

act on all curves that are 2^{e_A}-isogenous to E, which can be shown equal to $\mathrm{GL}_2(\mathbb{Z}/2^{e_A}\mathbb{Z})$, i.e. 2×2 matrices with entries in the group $\mathbb{Z}/2^{e_A}\mathbb{Z}$.

In detail, the group acts as follows: If an elliptic curve $E/\langle P \rangle$ is 2^{e_A}-isogenous curve and $\theta \in G$ an endomorphism, then if the degree of θ is coprime to 2^{e_A}, the curve $E/\langle \theta(P) \rangle$ is also a 2^{e_A}-isogenous curve. Hence endomorphisms of G acts on the set of e_A-isogenous curves [KP21].

Knowing the action of ϕ_A on $E[3^{e_B}]$, we approach the SSI-problem by computing $E/\langle \theta(\ker \phi_A) \rangle$ for an endomorphism $\theta \in G$, thus permuting the curves isogenous to E_A. The point is that we then have a commutative diagram

$$\begin{tikzcd}
E \arrow[r, "\phi_A"] \arrow[d, "\theta"] & E_A = E/\langle \ker \phi_A \rangle \arrow[d] \\
E/\langle \ker \theta \rangle \arrow[r] & E/\langle \theta(\ker \phi_A) \rangle \simeq E_A/\langle \phi_A(\ker \theta) \rangle.
\end{tikzcd}$$

Now, instead of calculating $E/\langle\theta(\ker \phi_A)\rangle$ knowing $E/\langle \ker \theta \rangle$, we can instead go the other way around and compute $E_A/\langle\phi_A(\ker \theta)\rangle$ by using E_A. This is computable if the degree of θ divides 3^{e_B}, which it in general does not. However, we can efficiently compute (in subexponential time) a different representative $\phi' \in G$ for ϕ such that the degree of ϕ' divides 3^{e_B} when $3^{e_B} > p \cdot 2^{4*e_A}$. For a suitable subgroup of G this action becomes free and transitive, which provides a commutative free group structure on the curves that are 2^{e^a}-isogenous to E. This allows us to apply Kuperberg's algorithm to solve SSI-problem in polynomial time.

If the condition on the parameters is not satisfied, we can still compute try to compute the different representative ϕ' for ϕ, but the complexity of such an approach is much worse than Petit's attack. Hence because of SIKE's balanced parameters, this attack does not present an security issue, but it undoubtedly presents a new approach to the problem.

The same paper also describes ways of constructing certain "backdoor" curves which allows for a polynomial time attack when $e_B > e_A^2$, and similarly also certain "backdoor" primes. They also point out the difficulty of detecting such curves, hence we should not trust curves coming from unknown sources.

5.2 General (black box) attacks

General (black box) attacks are attacks assuming no extra information, i.e. assuming no extra knowledge apart from the initial parameters and what passes through public channels.

The most naive way to attack SIKE is do a brute-force attack, starting at E or E_A, and search for a path connecting them of length e_A. Since there $(l+1)l^{e_A-1}$ paths of length e_A, we expect this to take $O(l^{e_A})$ time and memory (an infeasible bound), hence (as we would expect) has no impact on the security of SIKE.

5.2.1 Tani's algorithm

Better then the brute force attack is to preform a meet-in-the-middle attack, searching for paths emitting from both E and E_A of length $e_A/2$, and look for collisions. Letting X (resp. Y) be all such paths originating from E (resp. E_A), and $Z = \mathbb{F}_{p^2}$, this corresponds to solving the Claw finding problem.

The Claw-Finding problem. Let X, Y, Z be finite set and assume that there are functions $f : X \to Z$ and $g : Y \to Z$. The (resp. Golden) Claw-finding problem is to find (resp. unique element) $(x, y) \in X \times Y$ such that $f(x) = g(y)$ (i.e. compute the fibre product $X \times_Z Y$).

Here $f : X \to Z$ and $g : Y \to Z$ is the evaluation of the j-invariant of the end curve in the path.

Tani found in [Tan07b] an quantum algorithm that solves the Claw fining problem on a quantum computer. This represents the best known attack on SIKE using quantum computers. It relies on a generalization of Grover's algorithm to preform a search by quantum walks on the so-called *Johnson graph*.

We define the *Johnson graph*, denoted $J(X, R)$, of a finite set X and a parameter $0 < R < |X|$ as the graph with nodes being subsets of X with R elements (for which there are $\binom{|X|}{R}$) and where two nodes $U, V \subset X$ are adjacent if $|U \cap V| = R - 1$.

Now given the situation of the Claw-finding problem, consider the following two Johnson graphs

$$\begin{aligned} &J(X_f, R_f), \text{ where } X_f := \{(x, f(x) : x \in X\} && \text{and } 0 < R_f < |X_f|, \\ &J(Y_g, R_g), \text{ where } Y_g := \{(y, g(y) : y \in Y\} && \text{and } 0 < R_g < |Y_g|. \end{aligned}$$

Then compute the product graph $J(X_f, R_f) \times J(Y_g, R_g)$, i.e. where nodes are pairs (U, V) such that $U \in J(X_f, R_f)$ and $V \in J(Y_g, R_g)$ and two nodes (U, V) and (U', V') are adjacent if U is adjacent to U' and V is adjacent to V'. The parameter R is chosen in order to fit the number of qubits and memory, and the best known implementation uses $R \cong (XY)^{1/3}$ [Tan07a].

Such a meet-in-the-middle attack can be implemented on a quantum computer running in $O(\sqrt[4]{p})$ time. However, a significant drawback is the requirement of exponential storage; Indeed, when running the algorithm above, it is overwhelmingly unlikely that we hit the same endpoint from both sides simultaneously. We therefore need to store the endpoints of each path emanating from E and E_A, which we would expect to require $\sqrt(p)/2$ storage on average [JS19]. For the lowest case of SIKE, namely SIKEp434, this we would require more than 2^{108}-bits, which is more than the collective storage resources on the planet.

It is possible to generalize this attack by assuming knowledge some of the nodes $E_1, \dots, E_n$ that appears in computing Alice's secret isogeny, and applying Tani's algorithm to pairwise neighboring nodes. In the worst case scenario, all the known nodes line up in one end of Alice's path, and the search reduces to a search of length $e_A - n$ starting from E_n. In the best case scenario, $n = e - A$, in which case we are done. Acquiring such information may be performed by a side-channel attack (see section 5.2.4).

5.2.2 The vOW-Collision Finding Algorithm

One way to overcome the memory problem of Tani's algorithm (section 5.2.1)is to fix an upper bound on the possible memory, say 2^{80} (a possible but yet infeasible bound). The van Oorschot-Wiener's (vOW) algorithm [OW13] does exactly this in order to solve the SSI-problem. It is considered the best known general attack on SIKE [JS19].

The idea is that since we cannot store all the $2 \cdot 2^{e_A/2}$-isogenous curves of E or E_A simultaneously, we do a pseudorandom walk on the supersingular isogeny graph S, where we only store certain distinguished curves in memory. The procedure of distinguishing curves may be purely arbitrary (it should be efficiently computable, ex. checking a hash for 30 leading zeroes), and we should only distinguish one out of $2^{80}/2^{109} \approx 1/2^{30}$ curves to assure us that the memory never gets full. We achieve a pseudorandom walk in S by defining a deterministic, but pseudorandom, function $f : S \to S$ that inputs a $j-$invariant of a point and outputs a cryptographic hash. We use one bit of the hash to determine if we are to compute an isogeny from E or E_A, and then use the rest of the hash to produce an isogeny/subgroup from E or E_A respectfully, which in turn provides us with a new j-invariant.

We begin the procedure by choosing a random starting point $x_0 \in S$, and repeatedly apply f until we reach an distinguished element x_n after n iterations. We check the memory for other walks that end in x_n, and if there are none we store the triple (x_0, x_n, n), and pick a new random starting point. If there is already a path (x_0', x_n, n') in memory that ends in x_n, we have a *collision*, and a possible solution to our problem. Assuming that $n < n'$, we check whether this is indeed the correct solution by applying f n-times to check if $f^n(x_0) = f^n(x_0')$, in which case we are done. Otherwise we rewrite the old triple with the new triple and continue.

Since f is deterministic, we can recover the isogeny to x_n from the starting point x_0 without storing all the intermediate points x_i. However, f being pseudorandom introduces new problems: Although we know that there is a e_A-isogeny from E to E_A, this algorithm is not guarantied to find a solution after having filled the memory with distinguished points, and we have no way of knowing that this will happen with such an f in advance. In that case we must abandon all previous computations and start all over with a new function f. This makes both implementations and runtime analysis of this algorithm non-trivial. However, [OW13, §4.2] (and verified in [JS19]) has computed that holding w elements in memory from a set of size S running on m processors working in parallel, the runtime T of the algorithm is

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

where t is the cycle time of the processor. In SIDH we have $|S| = \lfloor p/12 \rfloor$, hence this algorithm becomes worse then the generic meet-in-the-middle attack above. However it remains the best known implementable algorithm for solving the isogeny problem underlying SIDH.

5.2.3 Finding subgraphs

When one (or both curves) are not defined over $\mathbb{F}_p$ we could try to localize the $\mathbb{F}_p$-subgraph, and by connecting them to the $\mathbb{F}_p$-subgraph apply an attack as in section 5.1.2. Nevertheless, there is no quantum-algorithm for finding the $\mathbb{F}_p$-subgraph in the first place, so such implementation must use a brute force attack which has a time cost of $\tilde{O}(\sqrt{p})$. Hence such an attack is not cheaper than breaking SIDH directly through the general meet-in-the-middle attack.

Like exploiting the $\mathbb{F}_p$ subgraph, we could try to find another subgraph of the supersingular isogeny graphs to exploit. However, we know of no such attacks in general. Nonetheless, always having knowledge of the supersingular isogeny graph and the initial curve, we could try to do a pre-computational attack on the graph. This we could use to narrow down the search once we know the public keys by remove loops and other unsuited paths, and in general try to connect initial and terminal curves with paths of specific lengths. The problem with such an attack is the sheer size of the graph, which would make such an approach requiring enormous storage. Therefore we conclude that this does not seem like a fruitful attempt.

5.2.4 Side-Channel Attacks

There have been several investigations into side-channel attacks on SIDH with various success [[GGK21][Tas+21], [KAJ17]]. Side-Channel attacks are attacks where one assumes some access to the machine carrying out the protocol, and employs methods like timing attacks, power analysis, electromagnetic analysis, etc., to obtain information of the secret.

In the case of SIDH, a side-channel attacks can focus on one of the two phases that take place during the secret isogeny computation:

- *Scalar Multiplication:* Computing the secret kernel point $S_2 = P_2 + [k_A]Q_2$ given knowledge of k_A, P_2 and Q_2.
- *Isogeny computation:* Given the kernel point S_2 and the public starting curve E, compute the secret isogeny $\phi_A : E \to E_A$.

The first point is also relevant for ECDH, hence we have over two decades of research into protecting against such attacks. However, as is the case with traditional elliptic curve cryptography, power analysis and fault injection attacks seems hard to defend against.

Attacking the second point can provide the attacker with knowledge of specific walks in the isogeny graph, thus reducing the problem of computing the secret isogeny on a smaller subgraph, weakening the security of SIKE. This is a point of concern, but such attacks applies to most, if not all, of the post-quantum cryptosystems candidates (at least to some extent). On the other hand, SIKE already has the benefit of good side-channel analysis of elliptic curve arithmetic from the much in use cryptosystem ECDH.

6 Generalizing SIDH

We briefly mentions that there has been attempts at generalizing the SIDH-protocol to curves of higher genus (hyperelliptic curves) and varieties of higher dimensions (abelian varieties) in [CS20]. Such generalizations allow for even shorter key sizes, but suffers under less efficient implementations. Compared to other NIST candidates, SIKE's efficiency is already concerning, hence we have not looked further at the security of such generalizations.

References

[Ber17] Bernstein, D. *Classic McEliece: Conservative Code-Based Cryptography.* 2017. (Visited on 07/22/2021).

[BJS14] Biasse, J.-F., Jao, D., and Sankar, A. "A Quantum Algorithm for Computing Isogenies between Supersingular Elliptic Curves". In: *Progress in Cryptology – INDOCRYPT 2014.* Ed. by Meier, W. and Mukhopadhyay, D. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2014, pp. 428–442.

[CJS14] Childs, A. M., Jao, D., and Soukharev, V. "Constructing Elliptic Curve Isogenies in Quantum Subexponential Time". In: *Journal of Mathematical Cryptology* vol. 8, no. 1 (Jan. 1, 2014), pp. 1–29. arXiv: `1012.4019`.

[Cos21] Costello, C. *The Case for SIKE: A Decade of the Supersingular Isogeny Problem.* 543. 2021.

[Cou06] Couveignes, J.-M. "Hard Homogeneous Spaces." In: *IACR Cryptology ePrint Archive* vol. 2006 (Jan. 1, 2006), p. 291.

[CS20] Costello, C. and Smith, B. "The Supersingular Isogeny Problem in Genus 2 and Beyond". In: *Post-Quantum Cryptography.* Ed. by Ding, J. and Tillich, J.-P. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 151–168.

[DG13] Delfs, C. and Galbraith, S. D. "Computing isogenies between supersingular elliptic curves over Fp". In: (2013). arXiv: `1310.7789 [math.NT]`.

[dQue+21] De Quehen, V., Kutas, P., Leonardi, C., Martindale, C., Panny, L., Petit, C., and Stange, K. E. *Improved Torsion Point Attacks on SIDH Variants.* Mar. 2, 2021. (Visited on 07/07/2021).

[Gal+16a] Galbraith, S. D., Petit, C., Shani, B., and Ti, Y. B. *On the Security of Supersingular Isogeny Cryptosystems.* 859. 2016.

[Gal+16b] Galbraith, S. D., Petit, C., Shani, B., and Ti, Y. B. "On the Security of Supersingular Isogeny Cryptosystems". In: (2016).

[GGK21] Genêt, A., Guertechin, N. L. de, and Kaluđerović, N. *Full Key Recovery Side-Channel Attack against Ephemeral SIKE on the Cortex-M4.* 858. 2021.

[Jao+19] Jao, D. et al. "Supersingular Isogeny Key Encapsulation (NIST Round 2)". In: (Apr. 2019).

[JD11] Jao, D. and De Feo, L. “Towards Quantum-Resistant Cryptosystems from Supersingular Elliptic Curve Isogenies”. eng. In: *Post-Quantum Cryptography.* Vol. 7071. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 19–34.

[JS19] Jaques, S. and Schanck, J. “Quantum Cryptanalysis in the RAM Model: Claw-Finding Attacks on SIKE”. In: Aug. 2019, pp. 32–61.

[KAJ17] Koziel, B., Azarderakhsh, R., and Jao, D. “Side-Channel Attacks on Quantum-Resistant Supersingular Isogeny Diffie-Hellman”. In: *SAC.* 2017.

[KP21] Kutas, P. and Petit, C. *Torsion Point Attacks on "SIDH-like" Cryptosystems.* 2021. URL: `https://csrc.nist.gov/CSRC/media/Events/third-pqc-standardization-conference/documents/accepted-papers/kutas-torsion-point-pqc2021.pdf`.

[OW13] Oorschot, P. V. and Wiener, M. “Parallel Collision Search with Cryptanalytic Applications”. In: *Journal of Cryptology* (2013).

[Pet17] Petit, C. “Faster Algorithms for Isogeny Problems Using Torsion Point Images”. In: *Advances in Cryptology – ASIACRYPT 2017.* Ed. by Takagi, T. and Peyrin, T. Vol. 10625. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2017, pp. 330–353.

[Piz90] Pizer, A. K. “Ramanujan Graphs and Hecke Operators”. In: *Bulletin (New Series) of the American Mathematical Society* vol. 23, no. 1 (July 1990), pp. 127–137.

[RS06] Rostovtsev, A. and Stolbunov, A. “Public-Key Cryptosystem Based on Isogenies”. In: *IACR Cryptol. ePrint Arch.* (2006).

[Sil09] Silverman, J. H. *The Arithmetic of Elliptic Curves.* 2nd ed. Graduate Texts in Mathematics. New York: Springer-Verlag, 2009.

[Sto10] Stolbunov, A. “Constructing Public-Key Cryptographic Schemes Based on Class Group Action on a Set of Isogenous Elliptic Curves”. In: *Advances in Mathematics of Communications - ADV MATH COMMUN* vol. 4 (May 1, 2010), pp. 215–235.

[Tan07a] Tani, S. “An Improved Claw Finding Algorithm Using Quantum Walk”. In: *Mathematical Foundations of Computer Science 2007.* Ed. by Kučera, L. and Kučera, A. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2007, pp. 536–547.

[Tan07b] Tani, S. “Claw Finding Algorithms Using Quantum Walk”. In: *Theoretical Computer Science - TCS* vol. 410 (Aug. 20, 2007).

[Tas+21] Tasso, É., Feo, L. D., Mrabet, N. E., and Pontié, a. S. *Resistance of Isogeny-Based Cryptographic Implementations to a Fault Attack.* 850. 2021.

[Vél71] Vélu, J. “Isogénies entre courbes elliptiques”. In: *CR Acad. Sci. Paris, Séries A* vol. 273 (1971), pp. 305–347.

A Implementing SIDH

The following program contains an minimal working example of the SIDH-protocol implemented in Python for illustrative purposes. It has not been heavily optimized, hence struggles for large primes (like the case of SIKE). In general we do not recommend an actual implementation of SIKE in Python. See SIKE's NIST proposal for implementation purposes.

```
import numpy as np
import sys

##############################################
# This program implements a instance of the
# SIDH-protocol.
#
# WARNING: This program in not optimized, and
#          will not terminate for large
#          primes. It is only intended as an
#          illustrating example.
#          An effective implementation of
#          SIKE should be written in a more
#          optimized language than Python.
#          (See SIKE"s NIST-submission on
#          which this program is based upon)
##############################################

def modular_sqrt(a, p):
    """ Find square roots of numbers modulo p,
        that is \sqrt{2}{n} mod p.

        The code is due to Svetlin Nakov.
    """

    def legendre_symbol(a, p):
        """ Compute the Legendre symbol a|p using
            Eulers criterion. p is a prime, a is
            relatively prime to p (if p divides
            a, then a|p = 0)
            Returns 1 if a has a square root modulo
            p, -1 otherwise.
        """
        ls = pow(a, (p - 1) // 2, p)
        return -1 if ls == p - 1 else ls

    """ Find a quadratic residue (mod p) of "a". p
        must be an odd prime.
        Solve the congruence of the form:
            x^2 = a (mod p)
        And returns x. Note that p - x
        is also a root. 0 is returned is no square
        root exists for these a and p.
        The Tonelli-Shanks algorithm is used
        (except for some simple cases in which
        the solution is known from an identity).
        This algorithm runs in polynomial time
        (unless the generalized Riemann
        hypothesis is false).
    """
    # Simple cases
    if legendre_symbol(a, p) != 1:
        return 0
    elif a == 0:
        return 0
    elif p == 2:
        return p
    elif p % 4 == 3:
        return pow(a, (p + 1) // 4, p)
    # Partition p-1 to s * 2^e for an odd s (i.e.
    # reduce all the powers of 2 from p-1)
    s = p - 1
    e = 0
    while s % 2 == 0:
        s //= 2
        e += 1
    # Find some "n" with a legendre symbol n|p = -1.
    n = 2
    while legendre_symbol(n, p) != -1:
        n += 1
    x = pow(a, (s + 1) // 2, p)
    b = pow(a, s, p)
    g = pow(n, s, p)
    r = e
    while True:
        t = b
        m = 0
        for m in range(r):
            if t == 1:
                break
            t = pow(t, 2, p)
        if m == 0:
            return x
        gs = pow(g, 2 ** (r - m - 1), p)
        g = (gs * gs) % p
        x = (x * gs) % p
        b = (b * g) % p
        r = m

class SIDH:
    def __init__(self, e_2=4, e_3=3, a=6, b=1):
        ##############################################
        # Initial parameters:
        # Values a, b defining an elliptic curve
        # E_ab : b*y^2 = x^3 + a*x^2+bx.
        # Values e_2 and e_3 such that
        # p=2^e_2 * 3^e_3 -1
        # is prime, and 2^{e_2} \cong 3^{e_3}.
        ##############################################

        self.e_2 = e_2
        self.e_3 = e_3
        p = 2**self.e_2 * 3**self.e_3-1
        self.p = p
        self.q = p**2
        if a == 2:
            sys.exit(
                "Error: a=2 is not a valid parameter"
                    )
        else:
            self.a = a
            #The curve E_ab : by^2 = x^3 + ax^2 + x
        self.b=b

    def primality(self, p):
        """ Returns True if p is prime.
        Otherwise returns False.
        """
        for i in range(2, np.floor(np.sqrt(p))+1):
            if p%i == 0:
                return False
        return True

    def inv(self, n):
        n=n%self.q
        """ Finds the (multiplicative) inverse of
        n mod q. Returns the inverse or 0.
        """
        if n%self.q == 0:
            return 0
        for i in range(1, self.q):
            if (i*n)%self.q == 1:
                return i
        return 0

    def sq(self, n):
        """ Finds the square root of n mod q
        """
        return modular_sqrt(n, self.q)

    def FindPoints(self, n=0):
        """ Finds all points up to a limit n
        on the curve and return them in a list
        """
        if n == 0:
            n=self.q
        list = [(0, 0)]
        for i in range(1, n):
            for j in range(1, n):
                if (self.b*j**2-
                    (i**3+self.a
                    *i**2+i))%self.q == 0:
                    list.append((i%self.q, j%self.q))
        return list

    def xDBL(self, x, y, a=6, b=1, e=1):
        """ e-times doublings of the point
        P=(x,y) on E_a
        """
        if x == float("inf") or y== float("inf"):
            return (float("inf"), float("inf"))
        else:
            X = ((x**2-1)**2
                )*self.inv(4*x*(x**2+a*x+1))
            Y = y*(x**2-1)*(x**4+2*a*x**3+
                6*x**2+2*a*x+1)*self.inv(8*
                x**2*(x**2+a*x+1)**2)
            for i in range(1, e):
                X, Y = self.xDBL(X, Y, a, b)
            if X==float("inf") and Y == float("inf"):
                return X, Y
            print("bla")
            return X%self.q, Y%self.q

    def xADD(self, x_p, y_p, x_q, y_q, a=6, b=1):
        """ Adds the points (x_p, y_p) and (x_q, y_q)."""
        if x_p == float("inf") and y_p == float("inf"):
            return x_q, y_q
        elif x_q == float("inf") and y_q == float("inf"):
            return x_p, y_p
        elif x_p == x_q and y_p==y_q:
            return self.xDBL(x_p, y_p, a, b)
        elif (x_p+x_q)%self.q==0 and (y_p+y_q)%self.q==0:
            return (float("inf"), float("inf"))
        else:
            l = (y_p-y_q)*self.inv(x_p-x_q)
            X = b*l**2-(x_p+x_q)-a
            Y = l*(x_p - X)-y_p
            return X%self.q, Y%self.q

    def xTPL(self, x, y, a=6, b=1, e=1):
        """ Tripling a point e times, i.e. P ---> [3^e]P"""
        Double = self.xDBL(x, y, a, b, e=1)
        Triple = self.xADD(x, y, Double[0], Double[1], a, b)
        X = Triple[0]
        Y = Triple[1]
        for i in range(1, e):
            Triplee = self.xTPL(X, Y, a, b, 1)
            X = Triplee[0]
            Y = Triplee[1]
        if (X, Y) == (float("inf"), float("inf")):
            return (X, Y)
        return X%self.q, Y%self.q

    def double_and_add(self, binary, x, y, a=6, b=1):
        """ Double and add scalar multiplication"""
        x_0 = 0
        y_0 = 0
        for i in range(len(binary)-1, 0, -1):
            x_0, y_0 = self.xDBL(x_0, y_0, a, b)
            if binary[i]==1:
                x_0, y_0 = self.xADD(x_0, y_0, x, y, a, b)
        return x_0%self.q, y_0%self.q

    def j_inv(self, a=6):
        """ Computes the j-invariant of the curve E_a"""
        if a == 2:
            sys.exit("Error: a=2 is not a valid parameter")
        j = 256*(a-3)**3*self.inv(a**2-4)
        return j%self.q

    def curve_2_iso(self, x_2, y_2, a=6, b=1):
        """ Calculates the 2-isogenous curve when
        P_2 has exact order 2 on E_ab. Returns a- b-."""
        if a == 2:
            sys.exit("Error: a=2 is not a valid parameter")
        a_1 = 2*(1-2*x_2**2)
        b_1 = x_2 * b
        return a_1%self.q, b_1%self.q

    def eval_2_iso(self, x_p, y_p, x_2):
        """ Evaluates the 2-isogeny corresponding to
        P_2 of the point P on E_ab"""
        X = (x_p**2*x_2-x_p)*self.inv(x_p-x_2)
        Y = y_p * (x_p**2*x_2-2*x_p*x_2**2+x_2
                    )*self.inv((x_p-x_2)**2)
        return X%self.q, Y%self.q

    def curve_3_iso(self, x_3, y_3, a=6, b=1):
        """ Calculates the 2-isogenous curve when
        P_3 has exact order 3 on E_ab. Returns a- b-."""
        if a == 2:
            sys.exit("Error: a=2 is not a valid parameter")
        a_1 = (a*x_3-6*x_3**2+6)*x_3
        b_1 = b*x_3**2
        return a_1%self.q, b_1%self.q

    def eval_3_iso(self, x_p, y_p, x_3):
        """ Evaluates the 4-isogeny corresponding to
        P_3 of the point P on E_ab"""
        X = (x_p*(x_p*x_3-1)**2)*self.inv((x_p-x_3)**2)
        Y = y_p*((x_p*x_3-1)*(x_p**2*x_3-3*x_p*x_3**2
                +x_p+x_3))*self.inv((x_p-x_3)**3)
        return X%self.q, Y%self.q

    def curve_4_iso(self, x_4, y_4, b=1):
        """ Calculates the 2-isogenous curve when
        P_4 has exact order 2 on E_ab. Returns a- b-."""
        a_1 = 4*x_4**4-2
        b_1 = -x_4*(x_4**2+1)*b*self.inv(2)
        return a_1%self.q, b_1%self.q

    def eval_4_iso(self, x_p, y_p, x_4):
        """ Evaluates the 2-isogeny corresponding to
        P_4 of the point P"""
        X = (-(x_p*x_4**2+x_p-2*x_4)*x_p*(x_p*x_4-1)**2
            )*self.inv((x_p-x_4)**2*(2*x_p*x_4-x_4**2-1))
        Y = (-2*x_4**2*(x_p*x_4-1)*(x_4**4*(x_4**2+1)
        -4*x_p**3*(x_4**3+x_4)+2*x_p**2*(x_4**4+5*x_4**2)
        -4*x_p*(x_4**3+x_4)+x_4**2+1))*self.inv((
            x_p-x_4)**3*(2*x_p*x_4-x_4**2-1)**2)
        return X%self.q, Y%self.q

    def iso_2_e(self, x_s, y_s, a=6, b=1, list=[]):
        """ Comutes (a", b") corresponding to the curve
        E_a"b" = E/<S>"""
        e_2 = self.e_2
        if self.e_2%2 == 1:
            print("hello")
            x_t, y_t = self.xDBL(x_s, y_s, a, b, self.e_2-1)
            a, b = self.curve_2_iso(x_t, b)
            x_s, y_s = self.eval_2_iso(x_s, y_s, x_t)
            list2=[]
            for instance in list:
                x = instance[0]
                y = instance[1]
                x, y = self.eval_2_iso(x, y, x_t)
                list2.append((x,y))
            e_2 = self.e_2-1
            list=list2
        for e in range(e_2-2, 0 -2):
            x_t, y_t = self.xDBL(x_s, y_s, a, b, e)
            a, b = self.curve_4_iso(x_t, y_t, b)
            if e != 0:
                x_s, y_s = self.eval_4_iso(x_s, y_s, x_t)
            list3=[]
            for instance in list:
                print(instance[0], instance[1])
                x = instance[0]
                y = instance[1]
                x, y = self.eval_4_iso(x, y, x_t)
                list3.append((x, y))
            list=list3
        return a%self.q, b%self.q, list

    def iso_3_e(self, x_s, y_s, a=6, b=1, list=[]):
        """ Comutes (a", b") corresponding to the
        curve E_a"b" = E/<S>, where S has exact
        order 3^e_3 on E_ab.
        """
        for e in range(self.e_3-2, 0 -1):
            x_t, y_t = self.xTPL(x_s, y_s, a, b, e)
            a, b = self.curve_3_iso(x_t, y_t, b)
            if e != 0:
                x_s, y_s = self.eval_3_iso(x_s, y_s, x_t)
            for instance in list:
                instance[0], instance[1] = self.eval(
                    instance[0], instance[1], x_t)
        return a%self.q, b%self.q, list

    def get_xR(self, a, b, x_p, y_p, x_q, y_q):
        """ Recovering of the x-coordinate of R"""
        x_r, y_r = self.xADD(x_p, y_p, x_q, -y_q, a, b)
        if x_r == float("inf"):
            return x_r
        else:
            return x_r%self.q

    def get_A(self, x_p, x_q, x_q_p):
        """ Recovers the Montgomery curve coefficient
        from the points x_p, x_q, x_{q-p}
        """
        a = ((1-x_p*x_q-x_p*x_q_p-x_q*x_q_p)**2
        )*self.inv(4*x_p*x_q*x_q_p) -x_p - x_q - x_q_p
        return a%self.q

    def get_yP_yQ_A_B(self,x_p, x_q, x_r):
        """ Recovers the y-coordinates of P and Q,
        and the Montgomery curve coefficient a
        """
        a = self.get_A(x_p, x_q, x_r)
        b = 1
        t_1 = x_p**2
        t_2 = x_p * t_1
        t_1 = a*t_1+t_2+x_p
        y_p = self.sq(t_1)
        t_1 = x_q**2
        t_2 = x_q*t_1
        t_1 = a*t_1 + t_2+x_q
        y_q = self.sq(t_1)
        x_t, y_t = self.xADD(x_p, y_p, x_q, -y_q, a, b)
        if x_t != x_r:
            y_q = -y_q
        return y_p%self.q, y_q%self.q, a%self.q, b%self.q

    def isogen2(self, sk2, x_p2, y_p2, x_q2, y_q2,
                x_p3, y_p3, x_q3, y_q3, a=6, b=1):
        """Computing public keys in the 3-torsion"""
        x_s, y_s = self.double_and_add([int(digit) for
                   digit in bin(sk2)[2:]], x_q2, y_q2)
        x_s, y_s = self.xADD(x_p2, y_p2, x_s, y_s)
        a, b, list = self.iso_2_e(x_s, y_s,
        list=[(x_s, y_s), (x_p3, y_p3), (x_q3, y_q3)])
        x_p3 = list[1][0]
        y_p3 = list[1][1]
        x_q3 = list[2][0]
        y_q3 = list[2][1]
        x_r3 = self.get_xR(a, b, x_p3, y_p3,
                                 x_q3, y_q3)
        return x_p3%self.q, x_q3%self.q, x_r3

    def isogen3(self, sk3, x_p2, y_p2, x_q2, y_q2,
                x_p3, y_p3, x_q3, y_q3, a=6, b=1):
        """Computing public keys in the 3-torsion"""
        x_s, y_s = self.double_and_add([int(digit)
        for digit in bin(sk3)[2:]], x_q3, y_q3, a, b)
        x_s, y_s = self.xADD(x_p3, y_p3, x_s, y_s, a, b)
        a, b, list = self.iso_3_e(a, b,
        list=[(x_s, y_s), (x_p2, y_p2), (x_q2, y_q2)])
        x_p2 = list[1][0]
        y_p2 = list[1][1]
        x_q2 = list[2][0]
        y_q2 = list[2][1]
        x_r2 = self.get_xR(a, b, x_p2, y_p2,
                                 x_q2, y_q2)
        return x_p2%self.q, x_q2%self.q, x_r2

    def isoex2(self, sk2, x_p2, x_q2, x_r2):
        """Establishes the shared key in the 2-torsion"""
        y_p2, y_q2, a, b = self.get_yP_yQ_A_B(x_p2, x_q2, x_r2)
        x_s, y_s = self.double_and_add([int(digit)
        for digit in bin(sk2)[2:]], x_q2, y_q2, a, b)
        x_s, y_s = self.xADD(x_p2, y_p2, x_s, y_s, a, b)
        a, b, list = self.iso_2_e(a, b, x_s, y_s)
        j2 = self.j_inv(a)
        return j2%self.q

    def isoex3(self, sk3, x_p3, x_q3, x_r3):
        """Establishes the shared key in the 3-torsion"""
        y_p3, y_q3, a, b = self.get_yP_yQ_A_B(x_p3, x_q3, x_r3)
        x_s, y_s = self.double_and_add([int(digit) for
        digit in bin(sk3)[2:]], x_q3, y_q3, a, b)
        x_s, y_s = self.xADD(x_p3, y_p3, x_s, y_s, a, b)
        a, b, list = self.iso_2_e(a, b, x_s, y_s)
        j3 = self.j_inv(a)
        return j3%self.q

## Public parameters:
## \\Mini-implementation of SIDH
## See the paper of Costello
## entitled "Supersingular isogeny
## key exchange for beginners"

# e_2 = 2, e_3 = 1, a=6, b=1
# p = 2**e_2 * 3**e_3-1 = 11
# q = p**2 # 11**2=121
# P2 = (10, 20), Q2 = (76, 90)
# P3 = (60, 4), Q3 = (60, 117)

# S = SIDH(e_2=2, e_3=1)
# k_A = 3
# PK_A = (60, 60, 116)
# k_B = 4
# PK_B = (10, 76, 29)

## Computing the shared secret
# print(S.isoex2(3, 10, 76, 29))
# print(S.isoex3(4, 60, 60, 116))
```

B Finding Supersingular j-invariants

The following program finds all the j-invariants of $\mathbb{F}_{p^2}$, and the second program checks for supersingularity through a Monte Carlo method.

```
import numpy.polynomial.polynomial as poly
from SIDH import SIDH

###################################################
# Find all supersingular j-invariants in F_p^2.
#
# This program uses the fact that a curve E_a
# is supersingular iff it on Weierstrassform
# given by y^2=f(x) we have that the coefficient
# of x^{p-1} in f^{(p-1)/2}(x) is zero.
#
# The function returns a list of tuples (i, a),
# where i corresponds to the curve E_i and a
# correspons to its corresponding j-invariant.
#
# WARNING: This program in not optimized, and
#          will not terminate for large primes.
#          It is only intended as an illustrating
#          example.
###################################################

```

```
S = SIDH()

def Supersingular(p):
    list = []
    for i in range(0, p):
        a = S.j_inv(i, p)
        a_1 = ((3-a**2)*S.inv(3, p))%p
        a_0 = ((2*a**3 - 9*a)*S.inv(27, p))%p
        e = poly.polypow([1, a_1, a_0],
                        (p-1)*S.inv(2, p))
        if e[-(p-1)-1]%p==0:
            list.append(i, a)
    return list
```

```
from SIDH import SIDH
import random

def MonteCarlo(A, n=1000, e_a=4, e_b=3):
    ##########################################
    # Employs a Monte Carlo simulation
    # in order to determine if E_a is
    # Supersingular. According to [Sut16]
    # a curve is likely to be supersingu-
    # ar if this algorithm returns True
    # for large primes p.
    #
    # The function takes in an variable
    # A determining the curve E_A,
    # a number n that restrict the search
    # of points on E_A, and parameters
    # e_a and e_b determing the prime p.
    # It returns False if the curve is
    # ordinary, and True if it is likely
    # to be supersingular.
    ##########################################

    p = 2**e_a*3**e_b-1 #prime number
    S = SIDH(e_2=e_a, e_3=e_b, a=A)
    P = random.choice(S.FindPoints(n)[1:])
    #Choosing a random point in the list
    #(without zero)
    Q = (0,0)
    for i in range(p-1):
        Q = S.xADD(P[0], P[1], Q[0], Q[1], a=A)
        #Q = [p-1]P
    if Q == (0,0):
        return True #E_A[p-1] not 0!
    else:
        Q = S.xADD(P[0], P[1], Q[0], Q[1], a=A)
        Q = S.xADD(P[0], P[1], Q[0], Q[1], a=A)
        #Q = [P+1]P
        if Q == (0,0):
            return True #E_A[p+1] not 0!
        else:
            return False, P, Q
```