Supersingular Isogeny Key Encapsulation

https://sike.org/

October 1, 2020

Principal submitter: David Jao

E-mail address: `djao@uwaterloo.ca`

Telephone: +1-519-888-4567 x32493

Affiliation: University of Waterloo & evolutionQ, Inc.

Postal address:

Department of Combinatorics & Optimization
University of Waterloo
200 University Ave. W
Waterloo, Ontario N2L 3G1
Canada

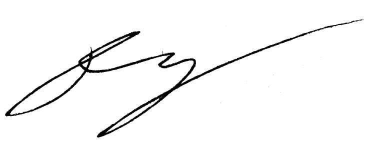
Auxiliary submitters:

- Reza Azarderakhsh, Florida Atlantic University & PQSecure Technologies, LLC, USA
- Matthew Campagna, Amazon Inc., USA
- Craig Costello, Microsoft Research, USA
- Luca De Feo, IBM Research Zürich, Switzerland
- Basil Hess, InfoSec Global, Switzerland
- Aaron Hutchinson, University of Waterloo, Canada
- Amir Jalali, LinkedIn Corporation, USA
- Koray Karabina, National Research Council, Canada
- Brian Koziel, Texas Instruments, USA
- Brian LaMacchia, Microsoft Research, USA
- Patrick Longa, Microsoft Research, USA
- Michael Naehrig, Microsoft Research, USA
- Geovandro Pereira, University of Waterloo & evolutionQ, Inc., Canada
- Joost Renes, NXP Semiconductors, The Netherlands
- Vladimir Soukharev, InfoSec Global, Canada
- David Urbanik, University of Toronto, Canada

Inventors/developers: Craig Costello, Luca De Feo, David Jao, Patrick Longa, Michael Naehrig and Joost Renes.

Owner: Same as submitters.

Signature:



Contents

Chapter 1

The SIKE protocol specification

This document presents a detailed description of the Supersingular Isogeny Key Encapsulation (SIKE) protocol. This protocol is based on a key-exchange construction, commonly referred to as Supersingular Isogeny Diffie-Hellman (SIDH), which was introduced by Jao and De Feo in 2011 [22], and subsequently improved in various ways by numerous authors [7, 8, 11, 30]. This specification gives an overview of the mathematical foundations necessary for SIKE, as well as a complete description of all the algorithms and data type conversions used in implementing SIKE, and a brief discussion of the security of the protocol.

For a summary of the notation used in this document, see Appendix G.

1.1 Mathematical Foundations

Use of the supersingular isogeny key encapsulation (SIKE) protocol described in this document involves arithmetic operations of elliptic curves over finite fields. This section provides the mathematical concepts and data type conversions used in the description of the SIKE protocol.

1.1.1 Finite Fields

A finite field consists of a finite set of elements closed under the operations of addition and multiplication defined over the set. There is an additive identity element (0) and a multiplicative identity element (1). Every element has a unique additive inverse, and every non-zero element has a unique multiplicative inverse.

For a positive integer q, there exists a finite field of q elements if and only if q is a power of a prime p. Further, there is a unique representative, up to isomorphism, of every finite field of q elements. We denote the finite field of q elements by $\mathbb{F}_q$. If $\mathbb{F}_q$ is a finite field with $q = p^t$ for prime p, we define the characteristic $\mathrm{char}(\mathbb{F}_q)$ of $\mathbb{F}_q$ to be p.

The finite fields used in supersingular isogeny cryptography are quadratic extension fields of a prime field $\mathbb{F}_p$, with $p = 2^{e_2}3^{e_3} - 1$, where e_2 and e_3 are fixed public parameters, and where the extension field is formed as $\mathbb{F}_{p^2} = \mathbb{F}_p(i)$ with $i^2 + 1 = 0$.

When abstraction is useful we will refer to $\ell, m \in \{2, 3\}$, such that $\ell \neq m$.

1.1.2 The Finite Field $\mathbb{F}_p$

The elements of $\mathbb{F}_p$ are represented by the integers:

$$\{0, 1, \ldots, p-1\}$$

with the field operations defined as follows:

- Addition: If $a, b \in \mathbb{F}_p$, then $a + b = r$ in $\mathbb{F}_p$, where $r \in [0, p-1]$ is the remainder of $a + b$ divided by p, also known as addition modulo p.
- Multiplication: If $a, b \in \mathbb{F}_p$, then $ab = s$ in $\mathbb{F}_p$, where $s \in [0, p-1]$ is the remainder of ab divided by p, also known as multiplication modulo p.
- Additive Inverse: If $a \in \mathbb{F}_p$, the unique solution in $[0, p-1]$ to the equation $a + x \equiv 0 \pmod{p}$ is the additive inverse $(-a)$.
- Multiplicative Inversion: If $a \in \mathbb{F}_p$, $a \neq 0$, the unique solution in $[0, p-1]$ to the equation $ax \equiv 1 \pmod{p}$ is the multiplicative inverse a^{-1}.

We make the convention that $a - b = a + (-b)$, and $a/b = a \cdot b^{-1}$ in the field $\mathbb{F}_p$.

1.1.3 The Finite Field $\mathbb{F}_{p^2}$

The elements of $\mathbb{F}_{p^2}$ are represented by $s = s_0 + s_1 \cdot i$, where $s_0, s_1 \in \mathbb{F}_p$, with the field operations defined as follows:

- Addition: If $a, b \in \mathbb{F}_{p^2}$, then $(a_0 + a_1 \cdot i) + (b_0 + b_1 \cdot i) = (a_0 + b_0) + (a_1 + b_1) \cdot i$ in $\mathbb{F}_{p^2}$, where the additions $(a_i + b_i)$ take place in $\mathbb{F}_p$.
- Multiplication: If $a, b \in \mathbb{F}_{p^2}$, then $(a_0 + a_1 \cdot i)(b_0 + b_1 \cdot i) = (a_0 b_0 - a_1 b_1) + (a_0 b_1 + a_1 b_0) \cdot i$ in $\mathbb{F}_{p^2}$, where the addition, additive inverse and multiplication operations take place in $\mathbb{F}_p$.
- Additive Inverse: If $a \in \mathbb{F}_{p^2}$, then $(-a_0) + (-a_1) \cdot i \in \mathbb{F}_{p^2}$ is the additive inverse $(-a)$, where the values $(-a_i)$ are computed in the field $\mathbb{F}_p$.
- Multiplicative Inversion: If $a \in \mathbb{F}_p$, $a \neq 0$, then $(a_0(a_0^2 + a_1^2)^{-1} + ((-a_1)(a_0^2 + a_1^2)^{-1}) \cdot i) \in \mathbb{F}_{p^2}$ is the multiplicative inverse a^{-1}, where the operations take place in $\mathbb{F}_p$.
- Square root: If there exists an $r = \alpha + \beta \cdot i \in \mathbb{F}_{p^2}$ with $\alpha, \beta \in \mathbb{F}_p$ such that $r^2 = s$, then we define $\sqrt{s} = r$ if either $\alpha \neq 0$ is an even integer or $\alpha = 0$ and β is an even integer, otherwise $\sqrt{s} = -r$.

1.1.4 Montgomery curves

A Montgomery curve is a special form of an elliptic curve. Let $A, B \in \mathbb{F}_q$ be field elements satisfying $B(A^2 - 4) \neq 0$ in $\mathbb{F}_q$ (where $\mathrm{char}(\mathbb{F}_q) \neq 2$). A Montgomery curve $E_{A,B}$ defined over $\mathbb{F}_q$, denoted $E_{A,B}/\mathbb{F}_q$, is defined to be the set of points $P = (x, y)$ of solutions in $\mathbb{F}_q$ to the equation

$$By^2 = x^3 + Ax^2 + x,$$

together with an extra point O, called the point at infinity. For convenience, we may refer to the curve as:

- $E_{A,B}$ when the underlying field $\mathbb{F}_q$ is either fixed by context, or unspecified,
- $E(\mathbb{F}_q)$ when the curve parameters are either fixed by context, or unspecified,
- E when both the field and the curve parameters A, B are either fixed by context, or unspecified.
- E_A when the underlying field $\mathbb{F}_q$ is fixed by context, or unspecified, and when B (which specifies the quadratic twist) is presumed to either be $B = 1$ or irrelevant.

At times it will be convenient to refer to the x-coordinate of a point P. We will use the notation x_P to refer to the x-coordinate of P, and analogously y_P to refer to the y-coordinate.

The set of points of E together with the point at infinity form a finite abelian group under a point addition rule. The order of an elliptic curve E over a finite field $\mathbb{F}_q$, denoted $\#E(\mathbb{F}_q)$, is the number of points in E including O.

Oftentimes, Montgomery curves are indicated by $M_{A,B}$, but we will use the notation $E_{A,B}$ instead.

1.1.5 Point addition

Given two points $P = (x_P, y_P)$ and $Q = (x_Q, y_Q)$ such that $P \neq \pm Q$ on a Montgomery curve $E_{A,B}$ over a finite field $\mathbb{F}_q$, we can compute $R = P + Q$ as

$$x_R = B\lambda^2 - (x_P + x_Q) - A$$

and

$$y_R = \lambda(x_P - x_R) - y_P,$$

where $R = (x_R, y_R)$ and $\lambda = (y_P - y_Q)/(x_P - x_Q)$.

We can add a point to itself multiple times, say k times, as follows: $P + P + \ldots + P = [k]P$.

The order $\mathrm{ord}(P)$ of a point P is the smallest positive integer n such that $[n]P = O$ (the point at infinity).

1.1.6 Point doubling

Let $P = (x_P, y_P) \in E_{A,B}$ be a point whose order does not divide 2. Then $[2]P = (x_{[2]P}, y_{[2]P}) \in E_{A,B}$ can be computed as

$$(x_{[2]P}\,,\ y_{[2]P}) = \left(\frac{(x_P^2 - 1)^2}{4x_P(x_P^2 + Ax_P + 1)} \quad , \quad y_P \cdot \frac{(x_P^2 - 1)(x_P^4 + 2Ax_P^3 + 6x_P^2 + 2Ax_P + 1)}{8x_P^2(x_P^2 + Ax_P + 1)^2} \right).$$

Observe that $x_{[2]P}$ only depends on x_P and A. The optimized, inversion-free algorithm that takes advantage of this is given in Algorithm 3 of Appendix A.

1.1.7 Point tripling

Let $P = (x_P, y_P) \in E_{A,B}$ be a point whose order does not divide 3. Then $[3]P = (x_{[3]P}, y_{[3]P}) \in E_{A,B}$ can be computed as

$$x_{[3]P} = \frac{(x_P^4 - 4Ax_P - 6x_P^2 - 3)^2 x_P}{(4Ax_P^3 + 3x_P^4 + 6x_P^2 - 1)^2} \quad ,$$

and

$$y_{[3]P} = y_P \cdot \frac{(x_P^4 - 4Ax_P - 6x_P^2 - 3)\left(x_P^8 + 4Ax_P^7 + 28x_P^6 + 28Ax_P^5 + (16A^2 + 6)x_P^4 + 28Ax_P^3 + 28x_P^2 + 4Ax_P + 1\right)}{(4Ax_P^3 + 3x_P^4 + 6x_P^2 - 1)^3}.$$

Again we see that $x_{[3]P}$ only depends on x_P and A. The algorithm that takes advantage of this is given in Algorithm 6 of Appendix A.

1.1.8 Additional properties of elliptic curves

For any group G, and a set of elements $\{P_1, P_2, \ldots, P_t\} \subseteq G$ we can define the subgroup $\langle P_1, P_2, \ldots P_t \rangle$ generated by this set to be the smallest subgroup of G containing the elements $P_1, P_2, \ldots, P_t$. For an abelian group G, we say a set of elements $\{P_1, P_2, \ldots P_t\} \subseteq G$ form a basis of G if every element P of G admits a unique expression of the form

$$P = [k_1]P_1 + [k_2]P_2 + \cdots [k_t]P_t$$

where $0 \leq k_i < \mathrm{ord}(P_i)$ for all i. Analogously, we say a set $\{P_1, P_2, \ldots, P_t\} \subseteq H$ forms a basis of a subgroup $H \subseteq G$ when all elements of the subgroup H admit a unique expression as above. The Weil pairing [34] can assist in determining whether or not a set forms a basis, since for $n = \mathrm{ord}(P) = \mathrm{ord}(Q)$, the order-$n$ Weil pairing e_n has the property that $\mathrm{ord}(e_n(P, Q)) = n$ if and only if $\langle P \rangle \cap \langle Q \rangle = \{\mathcal{O}\}$.

For a positive integer m, we define the set $E[m]$ of m-torsion elements of an elliptic curve $E(\mathbb{F}_q)$ to be the set of points in $E(\bar{\mathbb{F}}_q)$ such that $[m]P = \mathcal{O}$.

An elliptic curve $E(\mathbb{F}_q)$ over a field of characteristic p is called supersingular if $p \mid (q + 1 - \#E(\mathbb{F}_q))$, and ordinary otherwise.

The j-invariant of the elliptic curve $E_{A,B}$ is computed as

$$j(E_{A,B}) = \frac{256(A^2 - 3)^3}{A^2 - 4}.$$

The j-invariant of an elliptic curve over a field $\mathbb{F}_q$ is unique up to isomorphism of the elliptic curve. The SIKE protocol defines a shared secret as a j-invariant of an elliptic curve.

1.1.9 Isogenies

Let E_1 and E_2 be elliptic curves over a finite field $\mathbb{F}_q$. An isogeny $\phi\colon E_1 \to E_2$ is a non-constant rational map defined over $\mathbb{F}_q$ which is also a group homomorphism from $E_1(\mathbb{F}_q)$ to $E_2(\mathbb{F}_q)$. If such a map exists we say E_1 is isogenous to E_2, and two curves E_1 and E_2 over $\mathbb{F}_q$ are isogenous if and only if $\#E_1(\mathbb{F}_q) = \#E_2(\mathbb{F}_q)$.

An isogeny ϕ can be expressed in terms of two rational maps f and g over $\mathbb{F}_q$ such that $\phi((x, y)) = (f(x), y \cdot g(x))$. We can write $f(x) = p(x)/q(x)$ with polynomials $p(x)$ and $q(x)$ over $\mathbb{F}_q$ that do not have a common factor, and similarly for $g(x)$. We define the degree $\deg(\phi)$ of the isogeny to be $\max\{\deg(p(x)), \deg(q(x))\}$, where $p(x)$ and $q(x)$ are as above. It is often convenient to do isogeny calculations using only the $f(x)$ component of the isogeny.

Given an isogeny $\phi\colon E_1 \to E_2$ we define the kernel of ϕ as follows:

$$\ker(\phi) = \{P \in E_1 : \phi(P) = \mathcal{O}\}.$$

For any finite subgroup H of $E(\mathbb{F}_q)$, there is a unique isogeny (up to isomorphism) $\phi\colon E \to E'$ such that $\ker(\phi) = H$ and $\deg(\phi) = |H|$, where $|H|$ denotes the cardinality of H. In this case, we denote by E/H the curve E'. Given a subgroup $H \subseteq E(\mathbb{F}_q)$, Vélu's formula [51] can be used to find the isogeny ϕ and isogenous curve E/H. Vélu's formula is computationally impractical for arbitrary subgroups. SIKE uses isogenies over subgroups that are powers of 2, 3 and 4.

2-isogenies Let $(x_2, y_2) \in E_{A,B}$ be a point of order 2 with $x_2 \neq \pm 0$ and let $\phi_2\colon E_{A,B} \to E_{A',B'}$ be the unique (up to isomorphism) 2-isogeny with kernel $\langle (x_2, y_2) \rangle$. Then $E_{A',B'}$ can be computed as

$$(A', B') = \left(2 \cdot (1 - 2x_2^2)\ ,\ Bx_2\right)$$

Observe that A' only depends on x_2. The inversion-free algorithm that takes advantage of this is given in Algorithm 11 of Appendix A .

If $P = (x_P, y_P)$ is any point on $E_{A,B}$ that is not in $\ker(\phi_2)$, then $\phi_2\colon (x_P, y_P) \mapsto (x_{\phi_2(P)}, y_{\phi_2(P)})$, and this can be computed as

$$x_{\phi_2(P)} = \frac{x_P^2 x_2 - x_P}{x_P - x_2} \quad ,$$

and

$$y_{\phi_2(P)} = y_P \cdot \frac{x_P^2 x_2 - 2x_P x_2^2 + x_2}{(x_P - x_2)^2}.$$

Observe that $x_{\phi_2(P)}$ only depends on x_P and x_2. The inversion-free algorithm that takes advantage of this is given in Algorithm 12 of Appendix A.

4-isogenies Let $(x_4, y_4) \in E_{A,B}$ be a point of order 4 with $x_4 \neq \pm 1$ and let $\phi_4 \colon E_{A,B} \to E_{A',B'}$ be the unique (up to isomorphism) 4-isogeny with kernel $\langle (x_4, y_4) \rangle$. Then $E_{A',B'}$ can be computed as

$$(A', B') = \left(4x_4^4 - 2 \ , \ -x_4(x_4^2 + 1) \cdot B/2 \right)$$

Observe that A' only depends on x_4. The inversion-free algorithm that takes advantage of this is given in Algorithm 13 of Appendix A .

If $P = (x_P, y_P)$ is any point on $E_{A,B}$ that is not in $\ker(\phi_4)$, then $\phi_4 \colon (x_P, y_P) \mapsto (x_{\phi_4(P)}, y_{\phi_4(P)})$, and this can be computed as

$$x_{\phi_4(P)} = \frac{-(x_P x_4^2 + x_P - 2x_4) x_P (x_P x_4 - 1)^2}{(x_P - x_4)^2 (2x_P x_4 - x_4^2 - 1)} \quad ,$$

and

$$y_{\phi_4(P)} = y_P \cdot \frac{-2x_4^2(x_P x_4 - 1)(x_P^4(x_4^2 + 1) - 4x_P^3(x_4^3 + x_4) + 2x_P^2(x_4^4 + 5x_4^2) - 4x_P(x_4^3 + x_4) + x_4^2 + 1)}{(x_P - x_4)^3 (2x_P x_4 - x_4^2 - 1)^2}.$$

Observe that $x_{\phi_4(P)}$ only depends on x_P and x_4. The inversion-free algorithm that takes advantage of this is given in Algorithm 14 of Appendix A.

3-isogenies Let $(x_3, y_3) \in E_{A,B}$ be a point of order 3 and let $\phi_3 \colon E_{A,B} \to E_{A',B'}$ be the unique (up to isomorphism) 3-isogeny with kernel $\langle (x_3, y_3) \rangle$. Then $E_{A',B'}$ can be computed as

$$(A', B') = \left((Ax_3 - 6x_3^2 + 6)x_3 \ , \ Bx_3^2 \right)$$

The new coefficient A' only depends on A and x_3. The inversion-free algorithm that takes advantage of this is given in Algorithm 15 of Appendix A.

If $P = (x_P, y_P)$ is any point on $E_{A,B}$ that is not in $\ker(\phi_3)$, then $\phi_3 \colon (x_P, y_P) \mapsto (x_{\phi_3(P)}, y_{\phi_3(P)})$, and this can be computed as

$$(x_{\phi_3(P)}, y_{\phi_3(P)}) = \left(\frac{x_P(x_P x_3 - 1)^2}{(x_P - x_3)^2} \quad , \quad y_P \cdot \frac{(x_P x_3 - 1)(x_P^2 x_3 - 3x_P x_3^2 + x_P + x_3)}{(x_P - x_3)^3} \right).$$

Observe that $x_{\phi_3(P)}$ only depends on x_P and x_3. The inversion-free algorithm that takes advantage of this is given in Algorithm 16 of Appendix A.

The SIKE protocol defines secret keys from two separate key spaces, $\mathcal{K}_2$ and $\mathcal{K}_3$ (cf. §1.3.9). A secret key sk defines a subgroup H of $E(\mathbb{F}_q)$, which in turn defines an isogeny $\phi_{\text{sk}} \colon E \to E/H$. The public key is determined by the isogeny ϕ_{sk} and points $P, Q \in E(\mathbb{F}_q)$ (which are fixed globally as public parameters and do not depend on sk). More specifically, the public key corresponding to sk is determined by $\{E/H, \phi_{\text{sk}}(P), \phi_{\text{sk}}(Q)\}$. The points P and Q are chosen so that $\{P, Q\}$ forms a basis for $E[\ell^{e_\ell}]$. In our implementations, for efficiency reasons we represent a public key as a triplet of field elements, namely the three x-coordinates $\{x_{\phi_{\text{sk}}(P)}, x_{\phi_{\text{sk}}(Q)}, x_{\phi_{\text{sk}}(P-Q)}\}$ of three points under the isogeny. It is possible to convert between representations using the methods given in [8]. For example, the Montgomery curve coefficient A of E/H can be recovered by the three x-coordinates of a public key $\{x_{\phi_{\text{sk}}(P)}, x_{\phi_{\text{sk}}(Q)}, x_{\phi_{\text{sk}}(P-Q)}\}$ using the equation

$$A = \frac{(1 - x_{\phi_{\text{sk}}(P)}x_{\phi_{\text{sk}}(Q)} - x_{\phi_{\text{sk}}(P)}x_{\phi_{\text{sk}}(P-Q)} - x_{\phi_{\text{sk}}(Q)}x_{\phi_{\text{sk}}(P-Q)})^2}{4x_{\phi_{\text{sk}}(P)}x_{\phi_{\text{sk}}(Q)}x_{\phi_{\text{sk}}(P-Q)}} - x_{\phi_{\text{sk}}(P)} - x_{\phi_{\text{sk}}(Q)} - x_{\phi_{\text{sk}}(P-Q)}.$$

Similarly, the points $\phi_{\text{sk}}(P)$ and $\phi_{\text{sk}}(Q)$ can be recovered (up to simultaneous sign) from $x_{\phi_{\text{sk}}(P)}$ and $x_{\phi_{\text{sk}}(Q)}$ using the formula

$$y_{\phi_{\text{sk}}(P)} = \sqrt{x^3_{\phi_{\text{sk}}(P)} + Ax^2_{\phi_{\text{sk}}(P)} + x_{\phi_{\text{sk}}(P)}}$$

and

$$y_{\phi_{\text{sk}}(Q)} = \sqrt{x^3_{\phi_{\text{sk}}(Q)} + Ax^2_{\phi_{\text{sk}}(Q)} + x_{\phi_{\text{sk}}(Q)}},$$

and if

$$x_{\phi_{\text{sk}}(P-Q)} + x_{\phi_{\text{sk}}(Q)} + x_{\phi_{\text{sk}}(P)} + A \quad \neq \quad \left(\frac{y_{\phi_{\text{sk}}(Q)} - y_{\phi_{\text{sk}}(P)}}{x_{\phi_{\text{sk}}(Q)} - x_{\phi_{\text{sk}}(P)}}\right)^2 \quad ,$$

then set $y_{\phi_{\text{sk}}(Q)} = -y_{\phi_{\text{sk}}(Q)}$.

1.2 Data types and conversions

The SIKE protocol specified in this document involves operations using several data types. This section lists the different data types and describes how to convert one data type to another.

1.2.1 Curve-from-public-key computation - `cfpk`

An elliptic curve from a public key should be computed as described in this section. Informally, three field elements are interpreted as x-coordinates to three points P, Q, and $P - Q$, from which a curve E' is computed and returned.

Input: Three field elements (x_P, x_Q, x_R) of $\mathbb{F}_{p^2}$.

Output: A elliptic curve E' over $\mathbb{F}_{p^2}$ or FAIL.

Action: Convert (x_P, x_Q, x_R) to an elliptic curve as follows:

1. For $i \in [P, Q, R]$ verify $x_i \neq 0$ or return FAIL.
2. Compute $A = \frac{(1-x_Px_Q-x_Px_R-x_Qx_R)^2}{4x_Px_Qx_R} - x_P - x_Q - x_R$ in $\mathbb{F}_{p^2}$.
3. Set $E' = E_A$.
4. Output E'.

1.2.2 Octet-string-to-integer conversion - `ostoi`

Octet strings should be converted to integers as described in this section. This routine takes as input an octet string M of length mlen and interprets the octet string in base 2^8 of an integer.

Input: An octet string M of length mlen.

Output: An integer a.

Action: Convert M to an integer a as follows:

1. Parse $M = M_0 M_1 \ldots M_{\mathrm{mlen}-1}$ into mlen-many octets.
2. Interpret each octet M_i as an integer in $[0, 255]$.
3. Compute $a = \sum_{i=0}^{\mathrm{mlen}-1} M_i 2^{8i}$.
4. Output a.

1.2.3 Octet-string-to-field-p-element conversion - `ostofp`

Octet strings should be converted to elements of $\mathbb{F}_p$ as described in this section. This routine takes as input an octet string M of length $N_p = \lceil(\log_2 p)/8\rceil$ and converts it to an integer, verifying that the integer is in the range $[0, p-1]$.

Input: An octet string M of length N_p.

Output: A field element $a \in \mathbb{F}_p$ or FAIL.

Action: Convert the octet string M to field element as follows:

1. Convert M to an integer a (cf. §1.2.2) using M and N_p as inputs.
2. If $a \notin [0, p-1]$ output FAIL, otherwise output a.

1.2.4 Octet-string-to-field-p^2-element conversion - `ostofp2`

Octet strings should be converted to elements of $\mathbb{F}_{p^2}$ as described in this section. This routine takes as input an octet string M of length $2N_p$, where $N_p = \lceil(\log_2 p)/8\rceil$ and converts it to two integers, verifying each is in the range $[0, p-1]$, and interprets the results as an element of $\mathbb{F}_{p^2}$.

Input: An octet string M of length $2N_p$.

Output: A field element $a \in \mathbb{F}_{p^2}$ or FAIL.

Action: Convert the octet string M to field element as follows:

1. Parse $M = M_0 M_1$ where each M_i is of length N_p.
2. For $i \in [0, 1]$ convert M_i to a field element a_i (cf. §1.2.3) or output FAIL.
3. Form $a = a_0 + a_1 \cdot i$, and return a.

1.2.5 Octet-string-to-public-key conversion - `ostopk`

Octet strings should be converted to public keys as described in this section. This routine takes as input and octet string M of length $6N_p$, where $N_p = \lceil(\log_2 p)/8\rceil$ and converts it to three field elements of $\mathbb{F}_{p^2}$, interpreted as x-coordinates of three points P, Q, and R.

Input: An octet string M of length $6N_p$.

Output: A public key (x_P, x_Q, x_R) or FAIL.

Action: Convert the octet string M to a public key as follows:

1. Parse $M = M_1M_2M_3$, where each M_i is an octet string of length $2N_p$.
2. For $i \in [1, 2, 3]$ convert M_i to a field element x_i (cf. §1.2.4) or return FAIL.
3. Output $\mathrm{pk}_\ell = (x_1, x_2, x_3)$.

1.2.6 Integer-to-octet-string conversion - `itoos`

Integers should be converted to octet strings as described in this section. This routine takes as input an integer a and an octet length mlen is provided as input. The routine will represent a in base 2^8 and convert that to an octet string. A restriction is that $2^{8\cdot\mathrm{mlen}} > a$.

Input: A non-negative integer a together with a desired length mlen of the octet string, such that $2^{8\cdot\mathrm{mlen}} > a$.

Output: An octet string M of length mlen octets.

Actions: Convert a into an mlen-length octet string as follows:

1. Convert $a = a_{\mathrm{mlen}-1}2^{8(\mathrm{mlen}-1)} + a_{\mathrm{mlen}-2}2^{8(\mathrm{mlen}-2)} + \cdots + a_1 2^8 + a_0$ represented in base 2^8.
2. For $0 \le i < \mathrm{mlen}$, set $M_i = a_i$.
3. Form $M = M_0M_1 \ldots M_{\mathrm{mlen}-1}$.
4. Output M.

1.2.7 Field-p-to-octet-string conversion - `fptoos`

Field elements of $\mathbb{F}_p$ should be converted to octet strings as described in this section. Informally the idea is that an element of $\mathbb{F}_p$ is an integer in $[0, p-1]$ and is converted to a fixed length octet string.

Input: An element $a \in \mathbb{F}_p$.

Output: An octet string M of length $N_p = \lceil(\log_2 p)/8\rceil$.

Actions: Compute the octet string as follows:

1. Since a is an integer in the interval $[0, p-1]$, convert a to an octet string M (cf. §1.2.6), with inputs a and N_p.
2. Output M.

1.2.8 Field-p^2-to-octet-string conversion - `fp2toos`

Field elements $\mathbb{F}_{p^2}$ should be converted to octet strings as described in this section. Informally the idea is that the elements of $\mathbb{F}_{p^2}$ consists of two field elements of $\mathbb{F}_p$, each of these are converted to an octet string and the result is concatenated.

Input: An element $a \in \mathbb{F}_{p^2}$.

Output: An octet string M of length $2 \cdot N_p$ where $N_p = \lceil (\log_2 p)/8 \rceil$.

Actions: Compute the octet string as follows:

1. Since $a \in \mathbb{F}_{p^2}$, we can represent it as $a = a_0 + a_1 \cdot i$ where $a_i \in \mathbb{F}_p$.
2. Convert a_i into an octet string M_i of the length N_p (cf. §1.2.7).
3. Form $M = M_0 M_1$.
4. Output M.

1.2.9 Public-key-to-octet-string conversion - `pktoos`

Public keys (x_P, x_Q, x_R) should be converted to octet strings as described in this section. This routine converts each x-coordinate as an octet string encoding of a field elements and concatenates them to form the output octet string.

In portions of the spec we will refer to a public key pk in octet string format without explicitly referencing the public-key-to-octet-string conversion.

Input: A public key (x_P, x_Q, x_R) over a finite field $\mathbb{F}_{p^2}$

Output: An octet string M of length $6 \cdot N_p$ where $N_p = \lceil (\log_2 p)/8 \rceil$

Actions: Compute the octet string as follows:

1. Convert x_P, x_Q, x_R into the octet strings M_1, M_2, M_3 respectively, each of length $2N_p$ (cf. §1.2.6).
2. Form $M = M_1 M_2 M_3$.
3. Output M.

1.2.10 Compressed-public-key-to-octet-string conversion - `cpktoos`

Compressed public keys $(bit, bitEll_1, bitEll_2, t_1, t_2, t_3, A, s, r) \in \mathbb{Z}_2^3 \times (\mathbb{Z}_{\ell^e})^3 \times \mathbb{F}_{p^2} \times \mathbb{Z}_{2^8}^2$ should be converted to octet strings as described in this section. This routine converts each component as an octet string encoding and concatenates them to form the output octet string.

In portions of the spec we will refer to a public key pk_comp in octet string format without explicitly referencing the compressed-public-key-to-octet-string conversion.

Input: A compressed public key $(bit, bitEll_1, bitEll_2, t_1, t_2, t_3, A, r, s)$ consisting of 3 bits, 3 elements in $\mathbb{Z}_{\ell^e}$, one element of the finite field and 2 bytes.

Output: An octet string M of length $3 \cdot N_z + 2 \cdot N_p + 3$ where $N_z = \lceil(\lceil \log_2 \ell^e \rceil)/8)\rceil$ and $N_p = \lceil(\log_2 p)/8\rceil$.

Actions: Compute the octet string as follows:

1. Convert t_1, t_2, t_3 into the octet strings M_1, M_2, M_3 respectively, each of length N_z (cf. §1.2.8).
2. Convert A into an octet string M_4 of the length $2 \cdot N_p$ (cf. §1.2.7).
3. Let $M_5 = r$ if $bit = 1$ and $M_5 = r \vee 0x80$ if $bit = 0$, i.e., info about bit is encoded in the most significant bit of r.
4. Set $M_6 = s$, an octet.
5. Set $M_7 = bitEll_1 \vee (2 \cdot bitEll_2)$, an octet.
6. Form $M = M_1M_2M_3M_4M_5M_6M_7$.
7. Output M.

1.3 Detailed protocol specification

This section specifies the supersingular isogeny key encapsulation (SIKE) protocol. Some options have been omitted from this specification for the purpose of simplicity. In particular, the full specification below does not employ point compression. Users seeking the compression of public keys described in [2, 7, 35, 36] should refer to the implementation provided at https://github.com/Microsoft/PQCrypto-SIDH.

The set of public parameters for SIKE is defined in §1.3.1. The two necessary isogeny computation algorithms are defined in §1.3.5. The IND-CPA PKE scheme is defined in §1.3.10. The subsequent IND-CCA KEM is defined in §1.3.11. The security proofs of both the PKE and the KEM are in §4.3.

1.3.1 Public parameters

The public parameters in SIKE are:

- Two positive integers e_2 and e_3 that define a finite field $\mathbb{F}_{p^2}$ where $p = 2^{e_2}3^{e_3} - 1$,
- A starting supersingular elliptic curve $E_0/\mathbb{F}_{p^2}$,
- A set of three x-coordinates corresponding to points in $E_0[2^{e_2}]$, and
- A set of three x-coordinates corresponding to points in $E_0[3^{e_3}]$.

1.3.2 Starting curve

The public starting curve is the supersingular elliptic curve

$$E_0/\mathbb{F}_{p^2}\colon\, y^2 = x^3 + 6x^2 + x,$$

with $\#E_0(\mathbb{F}_{p^2}) = (2^{e_2}3^{e_3})^2$ and j-invariant equal to $j(E_0) = 287496$. This is the special instance of the Montgomery curve $By^2 = x^3 + Ax^2 + x$, where $A = 6$ and $B = 1$. Note that this has been updated since the initial proposal, for reasons that are further explained in [9, §5]. The original curve had $j = 1728$, for which E_0 above is the only non-isomorphic 2-isogenous curve, meaning an attacker would have known for certain the first step taken away from this starting point in the 2-isogeny graph, regardless of the secret. There also exist only two (as opposed to four generally) isomorphism classes that are 3-isogenous to $j = 1728$, so that distinct kernels can lead to isomorphic isogenies. Starting on E_0 avoids both of these problems. Moreover, the combination of the basis points on E_0 (defined in §1.3.3) and the computation of the secret kernel subgroups (defined in §1.3.6) ensures that the first 2-isogeny taken from E_0 is not in the direction of the curve with $j = 1728$, but rather to one of the two other 2-isogenous curves.

1.3.3 Public generator points

The three x-coordinates in the public parameters corresponding to points in $E_0[2^{e_2}]$ are specified as follows. We first specify two points

$$P_2 \in E_0(\mathbb{F}_{p^2}) \quad \text{and} \quad Q_2 \in E_0(\mathbb{F}_{p^2})$$

such that both points have exact order 2^{e_2}, and $\{P_2, Q_2\}$ forms a basis for $E_0(\mathbb{F}_{p^2})[2^{e_2}]$, i.e., the order-$2^{e_2}$ Weil pairing $e_{2^{e_2}}(P_2, Q_2) \in \mathbb{F}_{p^2}^{\times}$ has full order, or equivalently, $e_2([2^{e_2-1}]P_2, [2^{e_2-1}]Q_2) \in \mathbb{F}_{p^2}^{\times}$ is not equal to 1. Similarly, we specify two points

$$P_3 \in E_0(\mathbb{F}_p) \quad \text{and} \quad Q_3 \in E_0(\mathbb{F}_{p^2}) \setminus E_0(\mathbb{F}_p)$$

such that both points have exact order 3^{e_3}, and $\{P_3, Q_3\}$ forms a basis for $E_0(\mathbb{F}_{p^2})[3^{e_3}]$.

Let $f := x^3 + 6x^2 + x$ and recall $\mathbb{F}_{p^2} = \mathbb{F}_p(i)$ with $i^2 + 1 = 0$. The points P_2, Q_2, P_3, Q_3 are determined according to the following procedure:

- $P_2 = [3^{e_3}]\left(i + c,\, \sqrt{f(i + c)}\right)$, where c is the smallest nonnegative integer such that $P_2 \in E_0(\mathbb{F}_{p^2})$ and $[2^{e_2-1}]P_2 = (-3 \pm 2\sqrt{2}, 0)$.
- $Q_2 = [3^{e_3}]\left(i + c,\, \sqrt{f(i + c)}\right)$, where c is the smallest nonnegative integer such that $Q_2 \in E_0(\mathbb{F}_{p^2})$ and $[2^{e_2-1}]Q_2 = (0, 0)$.
- $P_3 = [2^{e_2-1}]\left(c,\, \sqrt{f(c)}\right)$, where c is the smallest nonnegative integer such that $f(c)$ is square in $\mathbb{F}_p$ and P_3 has order 3^{e_3}.
- $Q_3 = [2^{e_2-1}]\left(c,\, \sqrt{f(c)}\right)$, where c is the smallest nonnegative integer such that $f(c)$ is non-square in $\mathbb{F}_p$ and Q_3 has order 3^{e_3}.

The points P_2, Q_2, P_3, Q_3 could serve as public parameters for SIKE, but instead, for efficiency reasons (as described in [8]), we encode the points P_2 and Q_2 using the three x-coordinates x_{P_2}, x_{Q_2} and x_{R_2}, where $R_2 = P_2 - Q_2$. Similarly, we encode P_3, Q_3 using the three x-coordinates x_{P_3}, x_{Q_3} and x_{R_3}, where $R_3 = P_3 - Q_3$.

1.3.4 Public generator points (compressed SIKE)

The methodology for building generators for the compressed variant of SIKE is described below. It was introduced in [35] and is motived by key compression optimizations. Specify two points

$$P_3 \in E_0(\mathbb{F}_p) \quad \text{and} \quad Q_3 \in E_0(\mathbb{F}_{p^2}) \setminus E_0(\mathbb{F}_p)$$

such that both points have exact order 3^{e_3}, and $\{P_3, Q_3\}$ forms a basis for $E_0(\mathbb{F}_{p^2})[3^{e_3}]$.

Let $f := x^3 + 6x^2 + x$ and recall $\mathbb{F}_{p^2} = \mathbb{F}_p(i)$ with $i^2 + 1 = 0$. The points P_2, Q_2, P_3, Q_3 are determined according to the following procedure:

- $P_2 = \left[\frac{1}{2}\right] P_2'$ [1] such that $P_2' = [3^{e_3}]\left(c, \sqrt{f(c)}\right)$ is a point of order 2^{e_2-1}, where c is the smallest nonnegative integer such that $f(c)$ is a non-square in $\mathbb{F}_p$.
- $Q_2 = \left[\frac{1}{2}\right] Q_2'$ such that $Q_2' = [3^{e_3}]\left(c, \sqrt{f(c)}\right)$ is a point of order 2^{e_2-1} and linearly independent with P_2', where c is the smallest nonnegative integer such that $f(c)$ is a square in $\mathbb{F}_p$.
- $P_3 = [2^{e_2}]\left(c, \sqrt{f(c)}\right)$, where c is the smallest nonnegative integer such that $f(c)$ is square in $\mathbb{F}_p$ and P_3 has order 3^{e_3}.
- $Q_3 = (-x(P_3), i \cdot y(P_3))$ is the distortion map of P_3.

The points P_2, Q_2, P_3, Q_3 could serve as public parameters for compressed SIKE, but instead, for efficiency reasons (as described in [8]), we encode the points P_2 and Q_2 using the three x-coordinates x_{P_2}, x_{Q_2} and x_{R_2}, where $R_2 = P_2 - Q_2$. Similarly, we encode P_3, Q_3 using the three x-coordinates x_{P_3}, x_{Q_3} and x_{R_3}, where $R_3 = P_3 - Q_3$.

1.3.5 Isogeny computations

In this section we fix $\ell, m \in \{2, 3\}$ such that $\ell \neq m$. The two fundamental isogeny algorithms described are $\texttt{isogen}_\ell$ and $\texttt{isoex}_\ell$. On input of the public parameters and a secret key, $\texttt{isogen}_\ell$ outputs the public key corresponding to the input secret key. On input of a secret key and a public key, $\texttt{isoex}_\ell$ outputs the corresponding shared key. These two algorithms will be used as building blocks for the PKE and KEM schemes defined in the subsequent sections.

Both algorithms compute an ℓ^{e_ℓ}-degree isogeny via the composition of e_ℓ individual ℓ-degree isogenies; these ℓ-degree isogenies are evaluated on at least one point lying on the domain curve. Following [8, 11], rather than evaluating the image of an isogeny on a point $R = (x_R, y_R)$, it is more efficient to evaluate its image under the x-only projection $(x_R, y_R) \mapsto x_R$. Since the coordinate maps for an isogeny $\psi \colon E \to E'$, $R \mapsto \psi(R)$ can always be written such that $x_{\psi(R)} = f(x_R)$ for some function f [51], the $\texttt{isogen}_\ell$ and $\texttt{isoex}_\ell$ algorithms will assume the i-th ℓ-degree isogeny ϕ_i adheres to this framework by writing $\phi_i \colon (x, —) \mapsto (f_i(x), —)$.

[1] The notation $\left[\frac{1}{2}\right]$ refers to the point halving map on Montgomery curves associated with some (arbitrary) square root convention.

Note that the definition of public parameters and public keys allows for the possibility of a generic implementation that reverts back to full isogeny computations which compute both the x- and y-coordinates of image points in either the Montgomery or short Weierstrass frameworks. In particular, the starting curve E_0 defined in §1.3.2 is a special instance of a Montgomery curve and a short Weierstrass curve, and the public generator points in §1.3.3 uniquely define the y-coordinates of P_2, Q_2, P_3 and Q_3.

1.3.6 Computing public keys: $\texttt{isogen}_\ell$

A supersingular isogeny key pair consists of a secret key sk_ℓ, which is an integer, and a set of three x-coordinates $\mathrm{pk}_\ell = (x_P, x_Q, x_R)$.

Public parameters. A prime $p = 2^{e_2}3^{e_3} - 1$, the starting curve $E_0/\mathbb{F}_{p^2}$, and public generators $\{x_{P_2}, x_{Q_2}, x_{R_2}\}$ and $\{x_{P_3}, x_{Q_3}, x_{R_3}\}$.

Input. A secret key sk_ℓ.

Output. A public key pk_ℓ.

Actions. Compute a public key pk_ℓ, as follows:

1. Set $x_S \leftarrow x_{P_\ell + [\mathrm{sk}_\ell]Q_\ell}$;
2. Set $(x_1, x_2, x_3) \leftarrow (x_{P_m}, x_{Q_m}, x_{R_m})$;
3. For i from 0 to $e_\ell - 1$ do
   (a) Compute the x portion for an ℓ-isogeny
   $$\begin{aligned} \phi_i : E_i &\to E' \\ (x, \text{—}) &\longmapsto (f_i(x), \text{—}) \end{aligned}$$
   such that $\ker \phi_i = \langle [\ell^{e_\ell - i - 1}]S \rangle$, where S is a point on E_i with x-coordinate x_S;
   (b) Set $E_{i+1} \leftarrow E'$;
   (c) Set $x_S \leftarrow f_i(x_S)$;
   (d) Set $(x_1, x_2, x_3) \leftarrow (f_i(x_1), f_i(x_2), f_i(x_3))$;
4. Output $\mathrm{pk}_\ell = (x_1, x_2, x_3)$.

1.3.7 Establishing shared keys: $\texttt{isoex}_\ell$

Public parameters. A prime $p = 2^{e_2}3^{e_3} - 1$.

Input. A public key $\mathrm{pk}_m = (x_{P_m}, x_{Q_m}, x_{R_m})$ and a secret key sk_ℓ.

Output. A shared secret j, an octet string of length $2N_p$.

Actions. Compute a shared secret j, as follows:

1. Compute E'_0 from pk_m using `cfpk` (cf. §1.2.1);
2. Set $x_S \leftarrow x_{P_m+[\mathrm{sk}_\ell]Q_m}$;
3. For i from 0 to $e_\ell - 1$ do
   (a) Compute the x portion for an ℓ-isogeny

$$\begin{aligned}\phi_i : E'_i &\to E'\\ (x, —) &\longmapsto (f_i(x), —)\end{aligned}$$

   such that $\ker\phi_i = \langle[\ell^{e_\ell-i-1}]S\rangle$, where S is a point on E'_i with x-coordinate x_S;
   (b) Set $E'_{i+1} \leftarrow E'$;
   (c) Set $x_S \leftarrow f_i(x_S)$;
4. Encode $j(E'_{e_\ell})$ into j using `fp2toos` (cf. §1.2.8).

1.3.8 Optimized $\texttt{isogen}_\ell$ and $\texttt{isoex}_\ell$

The algorithms $\texttt{isogen}_\ell$ and $\texttt{isoex}_\ell$ described above, though polynomial-time, are relatively inefficient in practice. In both cases, the most expensive part is the computation of the point $[\ell^{e_\ell-i-1}]S$ in step 4.a of each. Indeed, one such computation requires (at most) e_ℓ multiplications by the scalar ℓ, and is repeated e_ℓ times, for a total of $O(e_\ell^2)$ *elementary operations*.

In optimized implementations, following [11], it is recommended to replace the for loops by a recursive decomposition of the isogeny computation into *elementary operations*, requiring only $O(e_\ell \log e_\ell)$ multiplications by the scalar ℓ, and a similar amount of evaluations of ℓ-isogenies.

We call such a decomposition a *computational strategy*, and we describe it by a full binary tree on $e_\ell - 1$ nodes[2]. If we draw such trees so that all nodes lie within a triangular region of a hexagonal lattice, with all leaves on one border, then the path length of the tree is proportional to the computational effort required by the strategy. See Figure 1.1 for an example, and [11, §4] for a more formal definition.

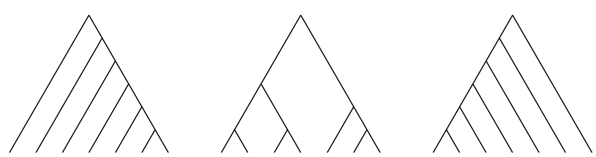
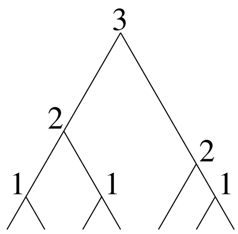


Figure 1.1: Three computational strategies of size $e_\ell - 1 = 6$. The simple approach used in Sections 1.3.6 and 1.3.7 corresponds to the leftmost strategy.

In practice, we represent any full binary tree on $e_\ell - 1$ nodes in the following way: associate to any internal node the number of leaves to its right, then walk the tree in depth-first left-first order and output the labels as they are encountered. See Figure 1.2 for an example.

Given any full binary tree represented this way, the computation in step 3 of $\texttt{isogen}_\ell$ can be replaced by the following recursive procedure:

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



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

Figure 1.2: Linear representation of a strategy on 6 nodes.

Input. A starting curve E, the x-coordinate x_S of a point S on E, a list of x-coordinates $(x_1, x_2, \dots)$ on E. A strategy $(s_1, \dots, s_{t-1})$ of size $t - 1$.

Output. The image curve $E' = E/\langle S \rangle$ of the isogeny $\psi : E \to E/\langle S \rangle$ with kernel $\langle S \rangle$, the list of image coordinates $(\psi(x_1), \psi(x_2), \dots)$ on E'.

Actions.

1. If $t = 1$ (i.e., the strategy is empty) then
   (a) Compute an ℓ-isogeny
   $$\begin{aligned} \phi \colon E &\to E' \\ (x, \text{—}) &\longmapsto (f(x), \text{—}) \end{aligned}$$
   such that $\ker \phi = \langle S \rangle$;
   (b) Return $(E', f(x_1), f(x_2), \dots)$;
2. Let $n = s_1$;
3. Let $L = (s_2, \dots, s_{t-n})$ and $R = (s_{t-n+1}, \dots, s_{t-1})$;
4. Set $x_T \leftarrow x_{[\ell^n]S}$;
5. Set $(E, (x_U, x_1, x_2, \dots)) \leftarrow$ Recurse on $(E, x_T, (x_S, x_1, x_2, \dots))$ with strategy L;
6. Set $(E, (x_1, x_2, \dots)) \leftarrow$ Recurse on $(E, x_U, (x_1, x_2, \dots))$ with strategy R;
7. Return $(E, (x_1, x_2, \dots))$.

A similar algorithm, without the inputs $(x_1, x_2, \dots)$, can be replaced inside $\texttt{isoex}_\ell$ to obtain the same speedup. Remark that the simple algorithms of Sections 1.3.6 and 1.3.7 correspond to the strategy $(e_\ell - 1, \dots, 2, 1)$. A *derecursivized* version of this algorithm is given in Appendix A.

We stress that the computational strategy is a public parameter independent of the (secret) input: it can be chosen once for all, and can possibly be hardcoded in the implementation. Changing it has no impact whatsoever on the security of the protocols (other than it affects the possible set of side-channel attacks). An implementer needs only be concerned with whether or not a given linear representation $(s_1, \dots, s_{t-1})$ correctly defines a strategy, i.e. that it belongs to the language S_t defined by the following grammar:

$$\begin{aligned} S_1 &::= \epsilon, \\ S_{a+b} &::= b \,.\, S_a \,.\, S_b. \end{aligned}$$

This can be readily verified with the following recursive procedure, that throws an error whenever a strategy is invalid, and terminates otherwise.

Input. A strategy $(s_1, \ldots, s_{t-1})$ of size $t - 1$.

Actions.

1. If $t = 1$ (i.e., the strategy is empty) return.
2. Let $n \leftarrow s_1$;
3. If $n < 1$ or $n \geq t$ halt with error "Invalid strategy";
4. Let $L = (s_2, \ldots, s_{t-n})$ and $R = (s_{t-n+1}, \ldots, s_{t-1})$;
5. Recurse on L;
6. Recurse on R.

These checks can easily be integrated into the isogeny computation algorithm. An analogous check is performed in the *derecursivized* versions of Appendix A.

1.3.9 Secret keys

The PKE and KEM schemes require two secret keys, sk_2 and sk_3, which are used to compute 2^{e_2}-isogenies and 3^{e_3}-isogenies, respectively (see §1.3.10 and §1.3.11).

Let $\mathtt{N}_{\mathrm{sk}_2} = \lceil e_2/8 \rceil$. Secret keys sk_2 correspond to integers in the range $\{0, 1, \ldots, 2^{e_2} - 1\}$, encoded as an octet string of length $\mathtt{N}_{\mathrm{sk}_2}$ using `itoos` (cf. §1.2.6). The corresponding keyspace is denoted $\mathcal{K}_2$.

Let $s = \lfloor \log_2 3^{e_3} \rfloor$ and $\mathtt{N}_{\mathrm{sk}_3} = \lceil s/8 \rceil$. Secret keys sk_3 correspond to integers in the range $\{0, 1, \ldots, 2^s - 1\}$, encoded as an octet string of length $\mathtt{N}_{\mathrm{sk}_3}$ using `itoos` (cf. §1.2.6). The corresponding keyspace is denoted $\mathcal{K}_3$.

1.3.10 Public-key encryption

Algorithm 1 defines a public-key encryption scheme $\mathtt{PKE} = (\mathtt{Gen}, \mathtt{Enc}, \mathtt{Dec})$ [11, §3.3]. The two keyspaces $\mathcal{K}_2$ and $\mathcal{K}_3$ are defined in 1.3.9. The size of the message space $\mathcal{M} = \{0, 1\}^n$, as well as the function F that maps the shared secret j to bitstrings, are left unspecified; concrete choices corresponding to our implementations are specified in Section 1.4. Note that the function `Enc` generates randomness sk_2. In the case of the key encapsulation mechanism we want to pass this randomness as input, in which case we write $(c_0, c_1) \leftarrow \mathtt{Enc}(\mathrm{pk}_3, m; \mathrm{sk}_2)$ (see Line 7 of Algorithm 2).

1.3.11 Key encapsulation mechanism

Algorithm 2 defines a key encapsulation mechanism $\mathtt{KEM} = (\mathtt{KeyGen}, \mathtt{Encaps}, \mathtt{Decaps})$, by applying a transformation of Hofheinz, Hövelmanns and Kiltz [19] to the PKE defined in §1.3.10. We slightly modify this transformation by including pk_3 in the input to G (as in [4]), and by simplifying "re-encryption" (see the proof of Theorem 1). Again, The two keyspaces $\mathcal{K}_2$ and $\mathcal{K}_3$ are defined in 1.3.9. The size of $\mathcal{M} = \{0, 1\}^n$ as well as the functions G and H, are left unspecified; concrete choices corresponding to our implementations are specified in Section 1.4.

Algorithm 1: $\texttt{PKE} = (\texttt{Gen}, \texttt{Enc}, \texttt{Dec})$

function $\texttt{Gen}$
- **Input:** ()
- **Output:** $(\mathrm{pk}_3, \mathrm{sk}_3)$
- $\mathrm{sk}_3 \leftarrow_R \mathcal{K}_3$
- $\mathrm{pk}_3 \leftarrow \texttt{isogen}_3(\mathrm{sk}_3)$
- **return** $(\mathrm{pk}_3, \mathrm{sk}_3)$

function $\texttt{Enc}$
- **Input:** $\mathrm{pk}_3, m \in \mathcal{M}, r \in \mathcal{K}_2$
- **Output:** (c_0, c_1)
- $\mathrm{sk}_2 \leftarrow r$
- $c_0 \leftarrow \texttt{isogen}_2(\mathrm{sk}_2)$
- $j \leftarrow \texttt{isoex}_2(\mathrm{pk}_3, \mathrm{sk}_2)$
- $h \leftarrow F(j)$
- $c_1 \leftarrow h \oplus m$
- **return** (c_0, c_1)

function $\texttt{Dec}$
- **Input:** $\mathrm{sk}_3, (c_0, c_1)$
- **Output:** m
- $j \leftarrow \texttt{isoex}_3(c_0, \mathrm{sk}_3)$
- $h \leftarrow F(j)$
- $m \leftarrow h \oplus c_1$
- **return** m

Algorithm 2: $\texttt{KEM} = (\texttt{KeyGen}, \texttt{Encaps}, \texttt{Decaps})$

function $\texttt{KeyGen}$
- **Input:** ()
- **Output:** $(s, \mathrm{sk}_3, \mathrm{pk}_3)$
- $\mathrm{sk}_3 \leftarrow_R \mathcal{K}_3$
- $\mathrm{pk}_3 \leftarrow \texttt{isogen}_3(\mathrm{sk}_3)$
- $s \leftarrow_R \{0,1\}^n$
- **return** $(s, \mathrm{sk}_3, \mathrm{pk}_3)$

function $\texttt{Encaps}$
- **Input:** pk_3
- **Output:** (c, K)
- $m \leftarrow_R \{0,1\}^n$
- $r \leftarrow G(m \,\|\, \mathrm{pk}_3)$
- $(c_0, c_1) \leftarrow \texttt{Enc}(\mathrm{pk}_3, m; r)$
- $K \leftarrow H(m \,\|\, (c_0, c_1))$
- **return** $((c_0, c_1), K)$

function $\texttt{Decaps}$
- **Input:** $(s, \mathrm{sk}_3, \mathrm{pk}_3), (c_0, c_1)$
- **Output:** K
- $m' \leftarrow \texttt{Dec}(\mathrm{sk}_3, (c_0, c_1))$
- $r' \leftarrow G(m' \,\|\, \mathrm{pk}_3)$
- $c_0' \leftarrow \texttt{isogen}_2(r')$
- **if** $c_0' = c_0$ **then**
  - $K \leftarrow H(m' \,\|\, (c_0, c_1))$
- **else**
  - $K \leftarrow H(s \,\|\, (c_0, c_1))$
- **return** K

NIST's API for the KEM

We now define how the inputs and outputs in Algorithm 2 match the API used in the implementations. NIST specifies the following API for the KEM:

```
int crypto_kem_keypair(unsigned char *pk, unsigned char *sk);
int crypto_kem_enc(unsigned char *ct, unsigned char *ss, const unsigned char *pk);
int crypto_kem_dec(unsigned char *ss, const unsigned char *ct, const unsigned char *sk);
```

The public key `pk` is given by pk_3. The secret key `sk` consists of the concatenation of s, sk_3 and pk_3 [3]. The ciphertext `ct` consists of the concatenation of c_0 and c_1. Finally, the shared secret `ss` is given by K.

[3] Since NIST's decapsulation API does not include an input for the public key, it needs to be included as part of the secret key.

1.4 Symmetric primitives

The three hash functions F, G and H that are used in the key encapsulation mechanism `KEM` are all instantiated with the SHA-3 derived function `SHAKE256` as specified by NIST in [13].

Specifically, the function G hashes the random bit string $m \in \mathcal{M} = \{0,1\}^n$ concatenated with the public key pk_3. It is instantiated with `SHAKE256`, taking $m \parallel \mathrm{pk}_3$ as the input, requesting e_2 output bits. In the notation of [13], this means $G(m \parallel \mathrm{pk}_3) = \texttt{SHAKE256}(m \parallel \mathrm{pk}_3, e_2)$. The value n corresponds to $n \in \{128, 192, 256\}$.

The function F is used as a key derivation function on the j-invariant during public key encryption and is computed as $F(j) = \texttt{SHAKE256}(j, n)$ using the notation of [13], where the requested output consists of n bits. Again, the value n corresponds to $n \in \{128, 192, 256\}$.

The third function H is used to derive the k-bit shared key K from the random bit string m and the ciphertext c produced by `Enc`. It is computed as $\texttt{SHAKE256}(m \parallel c, k)$ with $m \parallel c$ as the input. The value k corresponds to the number of bits of classical security, i.e., $k \in \{128, 192, 256\}$.

1.5 Public key compression

Recall that uncompressed SIKE public keys are of the form $(x_P, x_Q, x_R) \in (\mathbb{F}_{p^2})^3$, which correspond to three points $P, Q, R \in E_A(\mathbb{F}_{p^2})$ of exact order ℓ^e, for $\ell \in \{2, 3\}$. Following [2] (and the further improvements described in [7, 52]), the idea of public key compression is to instead represent these points as elements of $\mathbb{Z}_{\ell^e} \times \mathbb{Z}_{\ell^e}$ with respect to a ℓ^e-torsion basis that is determinstically chosen as a function of E_A. Encoding elements of $\mathbb{Z}_{\ell^e} \times \mathbb{Z}_{\ell^e}$ requires roughly half as many bits as encoding elements of $\mathbb{F}_{p^2}$. In particular, the compressed keys require $3.5 \log p$ bits instead of the $6 \log p$ to represent the triple (x_P, x_Q, x_R). This translates to a $\approx 41\%$ saving in key sizes.

Remark 1. Note that ciphertext compression is also possible although compressed ciphertexts occupy about $4 \log p$ bits ($\approx 33\%$ savings) instead of the $3.5 \log p$ bits achieved in key compression due to a tradeoff introduced in [36] allowing for a much faster decapsulation.

In Appendix C we describe the compression and decompression algorithms. Note that, since all of the operations in both algorithms are performed on public data, side-channel countermeasures (e.g., constant-time routines) are irrelevant except for the last step of the decompression algorithm in which we compute the last kernel generator using `Ladder3pt` (Algorithm 8). We point out that several alternatives for subroutines in both compression and decompression are possible. For example, compression requires the solutions of 2-dimensional discrete logarithm problems in $E(\mathbb{F}_{p^2})[\ell^e]$, which can be solved directly in $E(\mathbb{F}_{p^2})[\ell^e]$ (cf. [46]), but for improved performance our optimized implementation instead transports the problems to multiple 1-dimensional discrete logarithms in $\mathbb{F}_{p^2}^{\times}$, by way of the Tate pairing [2, 7, 52].

For more detailed information on the compression procedures followed in our implementation see [35] and [36].

Remark 2. It is worth mentioning that both SIKE public keys and ciphertexts are compressible. Due to the asymmetry in the original SIDH construction (binary and ternary torsions), compression techniques are faster in the binary torsion, and therefore torsions in Algorithms 2 are swapped for compression.

This implies that the most frequently used operation (Encapsulation) performs the fastest compression `compress`$_\ell$ for $\ell = 3$ which compressed points that are in the 2^{e_2}-torsion subgroup.[4]

1.6 Parameter sets

This section presents four different parameter sets, the concrete security of which is discussed in Chapter 5. The underlying prime fields are of the form $p = 2^{e_2}3^{e_3} - 1$ where $2^{e_2} \approx 3^{e_3}$.

The eight sets of parameters are `SIKEp434`, `SIKEp434_compressed`, `SIKEp503`, `SIKEp503_compressed` `SIKEp610`, `SIKEp610_compressed`, `SIKEp751` and `SIKEp751_compressed`, named so because of the bitlength of the prime field characteristic. In each case the parameters are, in order: the prime p and the values e_2 and e_3; the values $x_{Q_2,0}$ and $x_{Q_2,1}$ such that $x_{Q_2} = x_{Q_2,0} + x_{Q_2,1} \cdot i$; the values $x_{P_2,0}$ and $x_{P_2,1}$ such that $x_{P_2} = x_{P_2,0} + x_{P_2,1} \cdot i$; the values $x_{R_2,0}$ and $x_{R_2,1}$ such that $x_{R_2} = x_{R_2,0} + x_{R_2,1} \cdot i$; the values $x_{Q_3,0}$ and $x_{Q_3,1}$ such that $x_{Q_3} = x_{Q_3,0} + x_{Q_3,1} \cdot i$; the values $x_{P_3,0}$ and $x_{P_3,1}$ such that $x_{P_3} = x_{P_3,0} + x_{P_3,1} \cdot i$; the values $x_{R_3,0}$ and $x_{R_3,1}$ such that $x_{R_3} = x_{R_3,0} + x_{R_3,1} \cdot i$.

1.6.1 `SIKEp434`

```
   p = 0002341F 27177344 6CFC5FD6 81C52056 7BC65C78 3158AEA3 FDC1767A
       E2FFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
  e2 = 000000D8
  e3 = 00000089
xQ20 = 0000C746 1738340E FCF09CE3 88F666EB 38F7F3AF D42DC0B6 64D9F461
       F31AA2ED C6B4AB71 BD42F4D7 C058E13F 64B237EF 7DDD2ABC 0DEB0C6C
xQ21 = 000025DE 37157F50 D75D320D D0682AB4 A67E4715 86FBC2D3 1AA32E69
       57FA2B26 14C4CD40 A1E27283 EAAF4272 AE517847 197432E2 D61C85F5
yQ20 = 0001D407 B70B01E4 AEE172ED F491F4EF 32144F03 F5E054CE F9FDE5A3
       5EFA3642 A1181790 5ED0D4F1 93F31124 264924A5 F64EFE14 B6EC97E5
yQ21 = 0000E7DE C8C32F50 A4E735A8 39DCDB89 FE0763A1 84C525F7 B7D0EBC0
       E84E9D83 E9AC53A5 72A25D19 E1464B50 9D97272A E761657B 4765B3D6
xP20 = 00003CCF C5E1F050 030363E6 920A0F7A 4C6C71E6 3DE63A0E 6475AF62
       1995705F 7C84500C B2BB61E9 50E19EAB 8661D25C 4A50ED27 9646CB48
xP21 = 0001AD1C 1CAE7840 EDDA6D8A 924520F6 0E573D3B 9DFAC6D1 89941CB2
       2326D284 A8816CC4 249410FE 80D68047 D823C97D 705246F8 69E3EA50
yP20 = 0001AB06 6B849495 82E3F666 88452B92 55E72A01 7C45B148 D719D9A6
       3CDB7BE6 F48C812E 33B68161 D5AB3A0A 36906F04 A6A6957E 6F4FB2E0
yP21 = 0000FD87 F67EA576 CE97FF65 BF9F4F76 88C4C752 DCE9F8BD 2B36AD66
```

[4] Note that the points pk_ℓ are in the complementary torsion other than ℓ.

E04249AA F8337C01 E6E4E1A8 44267BA1 A1887B43 3729E1DD 90C7DD2F

xR20 = 0000F37A B34BA0CE AD94F43C DC50DE06 AD19C67C E4928346 E829CB92
580DA84D 7C36506A 2516696B BE3AEB52 3AD7172A 6D239513 C5FD2516

xR21 = 000196CA 2ED06A65 7E90A735 43F3902C 208F4108 95B49CF8 4CD89BE9
ED6E4EE7 E8DF90B0 5F3FDB8B DFE489D1 B3558E98 7013F980 6036C5AC

xQ30 = 00012E84 D7652558 E694BF84 C1FBDAAF 99B83B42 66C32EC6 5B10457B
CAF94C63 EB063681 E8B1E739 8C0B241C 19B9665F DB9E1406 DA3D3846

xQ31 = 00000000

yQ30 = 00000000

yQ31 = 0000EBAA A6C73127 1673BEEC E467FD5E D9CC29AB 564BDED7 BDEAA86D
D1E0FDDF 399EDCC9 B49C829E F53C7D7A 35C3A074 5D73C424 FB4A5FD2

xP30 = 00008664 865EA7D8 16F03B31 E223C26D 406A2C6C D0C3D667 466056AA
E85895EC 37368BFC 009DFAFC B3D97E63 9F65E9E4 5F46573B 0637B7A9

xP31 = 00000000

yP30 = 00006AE5 15593E73 97609197 8DFBD70B DA0DD6BC AEEBFDD4 FB1E748D
DD9ED3FD CF679726 C67A3B2C C12B3980 5B32B612 E058A428 0764443B

yP31 = 00000000

xR30 = 0001CD28 597256D4 FFE7E002 E8787075 2A8F8A64 A1CC78B5 A2122074
783F51B4 FDE90E89 C48ED91A 8F4A0CCB ACBFA7F5 1A89CE51 8A52B76C

xR31 = 00014707 3290D78D D0CC8420 B1188187 D1A49DBF A24F26AA D46B2D9B
B547DBB6 F63A760E CB0C2B20 BE52FB77 BD2776C3 D14BCBC4 04736AE4

1.6.2 SIKEp434_compressed

p = 0002341F 27177344 6CFC5FD6 81C52056 7BC65C78 3158AEA3 FDC1767A
E2FFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF

e2 = 000000D8

e3 = 00000089

xQ20 = 0001F23C F1B907ED EBDB6BC4 AB18C1E6 2CA20EA3 437D5D58 EC34981F
17D2CFAE C244241D 8494AF8A C8C1A359 A0BA4167 6F2E0B4D 0605C092

xQ21 = 00009866 7831535E 5E02562D B217BD86 F84B9830 AAD60641 454AE862
50218091 3E1798D6 13D59676 0AE93F0C 143FD643 8E99D2A5 01379355

xP20 = 00002F75 7834CA38 E9A149EE 9AA469F5 F7044F21 E0FFC221 2177C82D
7C3D4520 68F15591 30263CE3 3E3BED84 1E332D93 7FAB2BC0 89E57BDA

xP21	=	0000123E E2393221 33D147DE EB520B4A 0AAC93CC D6E3505F 6B8B5E30 795A01C0 1EE1185C BF10F5A9 420E1AA3 EAF6013F F0232202 A614D2AE
xR20	=	000098A2 F720085C 56A0BEE7 3EC958DA 5EE05C6C 4E99E0C5 00C7C2ED FF10A433 56FF83F1 D082B5CF 6D37F0BF 78576711 78FA05E0 9636F391
xR21	=	0001E1E1 AD8568A6 9C98E305 6BCCF9D3 0B173824 682AF046 2E0DA766 6626E399 E428E873 7CBE44E6 A8C04533 2E6935C2 69B709F3 1DD34330
xQ30	=	0000F71F B8FC6F96 4822D12D 07448181 29564040 96D7A1B9 F6A36CD0 DAC0BA3A 0D05FEF8 DE7C9349 A293AA50 9F917803 B3312818 E432A58C
xQ31	=	000006C9 405C49DC ED859BBC 2A1EA424 00CF4E07 FE7AEAA8 1828E904 4AD1CD53 2441650F D103E1F1 81527785 73E647B6 E8F1EA18 55F28681
xP30	=	0000F71F B8FC6F96 4822D12D 07448181 29564040 96D7A1B9 F6A36CD0 DAC0BA3A 0D05FEF8 DE7C9349 A293AA50 9F917803 B3312818 E432A58C
xP31	=	00022D55 E6BB2967 7F76C41A 57A67C32 7AF70E70 32DDC3FB E5988D76 982E32AC DBBE9AF0 2EFC1E0E 7EAD887A 8C19B849 170E15E7 AA0D797E
xR30	=	0001606C 450113A8 35A7A52F 6A6A70C8 6D9225DD B1D8A858 748BF544 890B1610 299BFF5F 715D5643 8D0C69FA 211284A8 149C3EDF B42C18FD
xR31	=	00000000

1.6.3 SIKEp503

p	=	004066F5 41811E1E 6045C6BD DA77A4D0 1B9BF6C8 7B7E7DAF 13085BDA 2211E7A0 ABFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
e2	=	000000FA
e3	=	0000009F
xQ20	=	00325CF6 A8E2C618 3A8B9932 198039A7 F965BA85 87B67925 D08D809D BF9A69DE 1B621F7F 134FA2DA B82FF5A2 615F92CC 71419FFF AAF86A29 0D604AB1 67616461
xQ21	=	003E7B04 94C8E60A 8B72308A E09ED348 45B34EA0 911E356B 77A11872 CF7FEEFF 745D98D0 624097BC 1AD7CD2A DF7FFC2C 1AA5BA3C 6684B964 FA555A07 15E57DB1
yQ20	=	003A3465 4000BD4C B2612017 BD5A1965 A9F89FE1 1C55D517 DF91B89B 94F4F9C5 8B9A9DD0 56915573 FEDC09CC D4997E82 378759E0 0A2DE225 CE04589D 201FD754

yQ21	=	0019DEF0 E8930E51 23A22E34 6B1FFBD3 5EB01451 647D8708 A4835473 B2539BD2 6806ED10 5A29F2D3 F7EAA262 426A9653 38782C5D 20FBF478 E4D1C8DB FC5B8294
xP20	=	0002ED31 A03825FA 14BC1D92 C503C061 D843223E 611A92D7 C5FBEC0F 2C915EE7 EEE73374 DF6A1161 EA00CDCB 786155E2 1FD38220 C3772CE6 70BC6827 4B851678
xP21	=	001EE4E4 E9448FBB AB4B5BAE F280A99B 7BF86A1C E05D55BD 603C3BA9 D7C08FD8 DE7968B4 9A78851F FBC6D0A1 7CB2FA1B 57F3BABE F87720DD 9A489B55 81F915D2
yP20	=	00244D5F 814B6253 688138E3 17F24975 E596B09B B15C6418 E5295AAF 73BA7F96 EFED145D FAE1B21A 8B7B121F EFA1B6E8 B52F0047 8218589E 604B9735 9B8A6E0F
yP21	=	00181CCC 9F0CBE13 90CC1414 9E8DE88E E79992DA 32230DED B25F04FA DE07F242 A9057366 060CB599 27DB6DC8 B20E6B15 747156E3 C5300545 E9674487 AB393CA7
xR20	=	003D24CF 1F347F1D A54C1696 442E6AFC 192CEE5E 320905E0 EAB3C9D3 FB595CA2 6C154F39 427A0416 A9F36337 354CF1E6 E5AEDD73 DF80C710 026D4955 0AC8CE9F
xR21	=	0006869E A28E4CEE 05DCEE8B 08ACD597 75D03DAA 0DC8B094 C85156C2 12C23C72 CB2AB2D2 D90D4637 5AA6D66E 58E44F8F 219431D3 006FDED7 993F5164 9C029498
xQ30	=	0039014A 74763076 675D24CF 3FA28318 DAC75BCB 04E54ADD C6494693 F72EBB7D A7DC6A3B BCD188DA D5BECE9D 6BB4ABDD 05DB38C5 FBE52D98 5DCAF744 22C24D53
xQ31	=	00000000
yQ30	=	00000000
yQ31	=	00255120 12C90A68 69C4B29B 9A757A03 006BC7DF 0BF7A252 6A071393 9FA48018 AE3E249B D63699BE B3B8DEA2 15B7AE1B 5A30FE37 1B64C5F1 B0BF051A 11D68E04
xP30	=	0032D03F D1E99ED0 CB05C070 7AF74617 CBEA5AC6 B75905B4 B54B1B0C 2D736978 40155E7B 1005EFB0 2B5D0279 7A8B66A5 D258C76A 3C9EF745 CECE11E9 A178BADF
xP31	=	00000000
yP30	=	002D810F 828E3DC0 24D1BBBC 7D6FA6E3 02CC5D45 8571763B 7CCD0E4D BC9FA116 3F0C1F8F 4AE32A57 F89DF8D2 586D2A06 E9FA3044 2B94A725 266358C4 5236ADF3
yP31	=	00000000

xR30 = 0000C146 5FD048FF B8BF2158 ED57F0CF FF0C4D5A 4397C754 2D722567
700FDBB8 B2825CAB 4B725764 F5F52829 4B7F95C1 7D560E25 660AD3D0
7AB011D9 5B2CB522

xR31 = 00288165 466888BE 1E78DB33 9034E2B8 C7BDF048 3BFA7AB9 43DFA05B
2D171231 7916690F 5E713740 E7C7D483 8296E673 57DC34E3 460A95C3
30D51697 21981758

1.6.4 SIKEp503_compressed

p = 004066F5 41811E1E 6045C6BD DA77A4D0 1B9BF6C8 7B7E7DAF 13085BDA
2211E7A0 ABFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
FFFFFFFF FFFFFFFF

e2 = 000000FA

e3 = 0000009F

xQ20 = 00127E2A 93ACC146 DBAFF173 6C05A6AB C1D7A204 7ACAC67D 8376664F
752BF0C1 65BC8359 1A133452 94684CA7 905472FB 7B42CF74 279A103C
1160A907 9897F7FE

xQ21 = 0001D4E4 48A987A3 87135E53 AE756D50 BFE33804 1156E468 DF8496FA
C2FEE315 9D54E45A 77375EE6 1CE7A424 DEDF5E8C D76BE308 1EA3A0F8
2ADF1DC7 A27FED74

xP20 = 001B3B05 B5B16AB7 51539381 BC40C380 BBCA4C51 F83BE1A1 33C120FF
512AF82F D1B48DC3 01A68CBF 26CB9CB9 16827692 2641449A 235EF7DE
9EBE4BF6 195AE5BC

xP21 = 0008EB09 019A3B1E 83624F58 A80A2F3B 0DCC51CC AD5E184B 9AE75909
CF081D49 05B1DA2C C0FCAA55 34E63CF4 376017F9 1B04770D B0747BDE
00D86781 2667C39D

xR20 = 001ED5A3 856AF031 A9F37C2B 74301AB4 8806000B 840B5ED2 CE216DDA
4BA4E182 FA9C323A 470100DE 379C0092 3A7A6773 F2CFDDE9 FB645FC3
989DBC05 3D8E2C7D

xR21 = 000948FC DC0EFC46 B38B8F7E D2CB5680 AD864615 302C29D9 18E2EFD2
D317DB13 0AC1BADC ACC460E7 B9888640 B304FF9E 0D0FA616 BE343270
AF1D8BE1 829461EB

xQ30 = 003BAA0E 33084960 851B2BB6 8CF94538 C43CFA2B 2B296274 CD915284
4B843FCE F12839E7 E2AD41A9 0134A0DE C1D94B53 0FB5A80F DDAE6368

		C29E5CF8 43265AB3
xQ31	=	00167A0D 5F1DAC47 FAF2C217 2166CA3D 0DE573D2 C7ACB8E9 E9A0E5D1 1EB55A27 CCD7A8A8 34DA8A03 2D113D04 812366A8 7796C1DE 31FFA49D 17573D34 8A5C4FE0
xP30	=	003BAA0E 33084960 851B2BB6 8CF94538 C43CFA2B 2B296274 CD915284 4B843FCE F12839E7 E2AD41A9 0134A0DE C1D94B53 0FB5A80F DDAE6368 C29E5CF8 43265AB3
xP31	=	0029ECE7 E26371D6 655304A6 B910DA93 0DB682F5 B3D1C4C5 29677609 035C8D78 DF285757 CB2575FC D2EEC2FB 7EDC9957 88693E21 CE005B62 E8A8C2CB 75A3B01F
xR30	=	00264CBC 7895FF5B EED9E748 68E7C16A 90C73FFF 80F0E8A0 E806BC31 5C24BA51 AE9A3420 A8C0FF49 1C72EC37 911AD5D0 91DBB1FC 9FCF2360 AD730EDE D5B4F2E1
xR31	=	00000000

1.6.5 SIKEp610

p	=	00000002 7BF6A768 819010C2 51E7D88C B255B2FA 10C4252A 9AE7BF45 048FF9AB B1784DE8 AA5AB02E 6E01FFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
e2	=	00000131
e3	=	000000C0
xQ20	=	25DA39EC 90CDFB9B C0F772CD A52CB8B5 A9F478D7 AF8DBBA0 AEB3E524 32822DD8 8C38F4E3 AEC0746E 56149F1F E89707C7 7F8BA413 45686297 24F4A8E3 4B06BFE5 C5E66E08 67EC38B2 83798B8A
xQ21	=	00000002 250E1959 256AE502 428338CB 47153995 51AEC78D 8935B2DC 73FCDCFB DB1A0118 A2D3EF03 489BA6F6 37B1C7FE E7E5F313 40A1A537 B76B5B73 6B4CDD28 4918918E 8C986FC0 2741FB8C 98F0A0ED
yQ20	=	A4FD5539 025C0611 E4B1CEC3 C36F0D75 90C035D3 A25AD930 22849CCE B3F67E4B 1DBE9884 04290DD8 B87B8D5E 69ED3B0C 5CDBCA24 8DC9D174 CF762012 CFE2D725 CFD92057 F2DBF8B0 4C7B12CC
yQ21	=	00000002 01C807BD 738624E2 2B87554A 2E053A46 A9573BA8 63D4A9D3 09533E30 B27BF7DD 8137F5CE 0F79C263 D9D05054 1D69817A 839085A7 6395F879 315F6999 E3441FC8 FB3936DE E1BEF5B4 E0E25096

xP20	=	00000001 B368BC60 19B46CD8 02129209 B3E65B98 BC64A92B C4DB2F9F 3AC96B97 A1B9C124 DF549B52 8F18BEEC B1666D27 D4753043 5E842212 72F3A97F B80527D8 F8A359F8 F1598D36 5744CA30 70A5F26C
xP21	=	00000001 459685DC A7112D1F 6030DBC9 8F2C9CBB 41617B6A D913E652 3416CCBD 8ED9C784 1D97DF83 092B9B3F 2AF00D62 E08DAD8F A743CBCC CC1782BE 0186A343 2D3C97C3 7CA16873 BEDE01F0 637C1AA2
yP20	=	00000001 CD75CF51 2FFA9DF8 78EF4950 01A57ABC 07FC7CE9 BB488BB5 2DDCD727 2D8A4FD1 7DD258ED 3F844C86 2CF48803 B9AC2668 C7CB79C3 96128763 B578080C 30D14CA7 EB709F98 E3E682A3 91FB35A7
yP21	=	00000002 001062A6 289E4082 CED88402 9207A1AC DEC525D7 BC165A6C FF8BB469 A8588950 A416DBB9 24D2D673 E3D6C32D 232F6E6A DA62B376 08F652C0 B8628827 B304BF13 65D82113 46207B24 EFF09458
xR20	=	00000001 B36A006D 05F9E370 D5078CCA 54A16845 B2BFF737 C8653687 07C0DBBE 9F5A62A9 B9C79ADF 11932A9F A4806210 E25C92DB 019CC146 706DFBC7 FA2638EC C4343C1E 390426FA A7F2F07F DA163FB5
xR21	=	00000001 83C9ABF2 297CA696 99357F58 FED92553 436BBEBA 2C3600D8 9522E700 9D19EA5D 6C18CFF9 93AA3AA3 3923ED93 592B0637 ED0B33AD F12388AE 912BC4AE 4749E2DF 3C329299 4DCF3774 7518A992
xQ30	=	00000001 4E647CB1 9B7EAAAC 640A9C26 B9C26DB7 DEDA8FC9 399F4F8C E620D2B2 200480F4 338755AE 16D0E090 F15EA188 2166836A 478C6E16 1C938E4E B8C2DD77 9B45FFDD 17DCDF15 8AF48DE1 26B3A047
xQ31	=	00000000
yQ30	=	00000000
yQ31	=	E674067F 5EA6DE85 545C0A99 E9E71E64 FABFDC28 1D1E540F EDA47A56 ED3ADCDD E1841083 FABC7954 B467C71A C6349B04 974A7F9B 688C5F73 5632FEB3 94146B0A 08088006 9D8DA332 4EDF795B
xP30	=	00000001 587822E6 47707ED4 313D3BE6 A811A694 FB201561 111838A0 816BFB5D EC625D23 772DE48A 26D78C04 EEB26CA4 A571C67C E4DC4C62 0282876B 2F2FC263 3CA548C3 AB0C45CC 991417A5 6F7FEFEB
xP31	=	00000000
yP30	=	14F29511 4B69D476 9AC06DD0 7F051AD1 114BCB7F A6B6EDE1 9F840969 AFD56FD1 F728907D 3320A030 9462A944 4D24FE75 4666DB24 70080951 B31C2AC5 9704ABC7 670C3C3A 992C3C16 29791F30
yP31	=	00000000
xR30	=	00000001 DB73BC2D E666D24E 59AF5E23 B79251BA 0D189629 EF87E56C 38778A44 8FACE312 D08EDFB8 76C3FD45 ECF3746D 96E2CADB BA08B1A2 06C47DDD 93137059 E34C90E2 E42E10F3 0F6E5F52 DED74222

xR31 = 00000001 B2C30180 DAF5D918 71555CE8 EFEC76A4 D521F877 B7543112
28C7180A 3E2318B4 E7A00341 FF99F34E 35BF7A10 53CA76FD 77C0AFAE
38E20918 62AB4F1D D4C8D9C8 3DE37ACB A6646EDB 4C238B48

1.6.6 SIKEp610_compressed

p = 00000002 7BF6A768 819010C2 51E7D88C B255B2FA 10C4252A 9AE7BF45
048FF9AB B1784DE8 AA5AB02E 6E01FFFF FFFFFFFF FFFFFFFF FFFFFFFF
FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF

e2 = 00000131

e3 = 000000C0

xQ20 = 00000001 AAAF04FF FA0E920A 9D57B705 A019428F CDC7519A D382A3C2
452C47CB 0958477D DA6D112F 803F0B0F 93C1189A 79DC856F 653263E1
11A35C21 D5A9FF83 3915F8A6 AEABA65B 1D4949E2 C555B554

xQ21 = 32B70A9E C1ED3130 313FEB14 1828F365 9AF973AC 42AC8C4E FA89AC25
0A5C978C 29DA9D07 1606EAFE BFDBD97A 53548DF9 B5B860C2 AAEF8403
CF021DCE C87EE0C5 12F5D9AC BEECB66C 11E5AFE7

xP20 = 00000001 81858998 C9AA9597 F380DC28 ABF6173F 83F84A8F D45B0205
B42BCEFF EB609F11 B084205E C9F17599 B7215675 2BE0D893 1686BFB9
A6582EE8 D23460DE DF110BC4 E0E92734 CEC7D75E C902001D

xP21 = 1D339C0C E6B50439 45696090 3A352B20 7BDBF7ED 2188D1F2 42D8F96B
E97DC4FD AB9FE77A 7CB98619 F53292B4 4EBE7E26 A0344E8E 3E184B43
578CE0DC A512D5D0 89EFE77D 25DCDF3B 81C74B30

xR20 = 00000001 3979F57A F9D5E767 65CD95CF 685BE36D 3F3EA6DB ADAAA9F1
6EE7CC74 1F593A2F BCD5D24B 09AECB20 256455E0 DA41CA94 2224B061
03AD1B10 267ABC59 0AB8D90E 8CDA3973 4AB33CFA BB3AC133

xR21 = 00000001 BB80FA13 4385E8FA 1ABB9587 609F59C2 BB895505 6BF18460
451D86CD EBF0B1AE 5199F138 493F5016 E5C22740 64045251 1A4E3A1F
3EDA6AA5 ED0D3AC3 7615E26A B80F6DF2 2B656DCB A22838C1

xQ30 = 56DAE1DE 4A196594 7D689C6D 0F08F041 34F8C8D1 1CD3F938 68E827DF
93C580BD F781ED32 A3892752 1B638CCC 4E946AEF B179EBB0 B0F185BD
80097AFE 4242C7BE B46AB1BA C8FB3AE9 11A231D2

xQ31 = 00000001 BA6391FD F6EE6623 55BE2450 7FFB171A 6FF4A5BF E00AE62E
E208F738 41805CC8 E0F0C287 914E52FD 5684082C AEEC0DB6 909B1397

EF04EEC9 06545386 898FD7F1 949493F0 8CF45895 79C09C95

xP30 = 56DAE1DE 4A196594 7D689C6D 0F08F041 34F8C8D1 1CD3F938 68E827DF
93C580BD F781ED32 A3892752 1B638CCC 4E946AEF B179EBB0 B0F185BD
80097AFE 4242C7BE B46AB1BA C8FB3AE9 11A231D2

xP31 = C193156A 8AA1AA9E FC29B43C 325A9BDF A0CF7F6A BADCD916 22870273
6FF7F11F C969EDA6 DCB3AD02 A97BF7D3 5113F249 6F64EC68 10FB1136
F9ABAC79 7670280E 6B6B6C0F 730BA76A 863F636A

xR30 = 1797211F 860C7226 9AF31332 523E005D A5D6E2FA 48227319 BEEC8955
A8E3ECA2 3C590735 A275D333 351D1BD7 C41D5B2A 52AC5B75 84D9A17D
8E0AA956 E1C47B32 0EC98B94 6DCF3F89 9802B26A

xR31 = 00000000

1.6.7 SIKEp751

p = 00006FE5 D541F71C 0E12909F 97BADC66 8562B504 5CB25748 084E9867
D6EBE876 DA959B1A 13F7CC76 E3EC9685 49F878A8 EEAFFFFF FFFFFFFF
FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
FFFFFFFF FFFFFFFF FFFFFFFF

e2 = 00000174

e3 = 000000EF

xQ20 = 00001723 D2BFA01A 78BF4E39 E3A333F8 A7E0B415 A17F208D 3419E759
1D59D8AB DB7EE6D2 B2DFCB21 AC29A40F 837983C0 F057FD04 1AD93237
704F1597 D87F074F 682961A3 8B5489D1 019924F8 A0EF5E4F 1B2E64A7
BA536E21 9F5090F7 6276290E

xQ21 = 00002569 D7EAFB6C 60B244EF 49E05B5E 23F73C4F 44169A7E 02405E90
CEB680CB 0756054A C0E3DCE9 5E295033 4262CC97 3235C2F8 7D89500B
CD465B07 8BD0DEBD F322A2F8 6AEDFDCF EE65C093 77EFBA0C 5384DD83
7BEDB710 209FBC8D DB8C35C7

yQ20 = 000035B8 2D1BD2BA 608B4279 4C4820C5 6A3D8BBA D28380B8 D85A1910
E2609A61 F7BC0BCA 8ED8EF88 3E7E98C7 44A0AC85 D2893738 521B62EB
23D1983D 2EDCF2AB 437108DC 048AA853 FF9BC791 224B121E 8FDF1EA5
F617E6ED 5898663D DED49154

yQ21 = 00000F22 306A6963 907F16AA 38F89C67 2A4054DB 5FD1D265 98A3140E
A204B100 94AE6409 3142AEB0 56942494 D216A74E D9F51FFC 9272D177

21510133 34EC570B 532DB0C0 83CF3986 7F63D191 029033F9 42E977B8
5F69EC73 8B4C26D3 B72E2821

xP20 = 00004514 F8CC94B1 40F24874 F8B87281 FA6004CA 5B3637C6 8AC0C0BD
B2983805 1F385FBB CC300BBB 24BFBBF6 710D7DC8 B29ACB81 E429BD1B
D5629AD0 ECAD7C90 622F6BB8 01D0337E E6BC78A7 F12FDCB0 9DECFAE8
BFD643C8 9C3BAC1D 87F8B6FA

xP21 = 0000158A BF500B59 14B3A96C ED5FDB37 D6DD925F 2D6E4F7F EA3CC16E
10857540 77737EA6 F8CC7493 8D971DA2 89DCF243 5BCAC189 7D262769
3F9BB167 DC01BE34 AC494C60 B8A0F65A 28D7A31E A0D54640 653A8099
CE5A84E4 F0168D81 8AF02041

yP20 = 00000BF6 E4E7A28E 9A6EF66A 2F1614AE 2A2B5A58 3C9F2DC6 C83F84E2
D9E6577F 9E22B991 D58FB2F8 9666DC1D 40A2C0A3 AB876CF8 DA8878F1
2325BF8B 0CF92E45 AE006270 41C891BC 96FFBB87 4FC587E4 342F7809
8258DF2E 10A5708A 70A0D5A8

yP21 = 00001502 FB44178D 1DF80A53 858519CB CF233FE3 87905BC8 F9E41387
03C6DB7C 82302FBF B7E97153 F6001FE9 102D2597 AC2B300A 1C669D1A
2803F8D0 5BA3B1F2 ACBF27BC 1A127B4A 553916D6 2004FD21 633C5AEA
AB748338 53B4C5C4 2EB71F7E

xR20 = 00006066 E07F3C0D 964E8BC9 63519FAC 8397DF47 7AEA9A06 7F3BE343
BC53C883 AF29CCF0 08E5A307 19A29357 A8C33EB3 600CD078 AF1C40ED
5792763A 4D213EBD E44CC623 195C387E 0201E723 1C529A15 AF5AB743
EE9E7C9C 37AF3051 167525BB

xR21 = 000050E3 0C2C0649 4249BC4A 144EB5F3 1212BD05 A2AF0CB3 064C322F
C3604FC5 F5FE3A08 FB3A02B0 5A48557E 15C99225 4FFC8910 B72B8E13
28B4893C DCFBFC00 3878881C E390D909 E39F83C5 006E0AE9 79587775
443483D1 3C65B107 FADA5165

xQ30 = 00005BF9 54478180 3CBD7E0E A8B96D93 4C5CBCA9 70F9CC32 7A0A7E4D
AD931EC2 9BAA8A85 4B8A9FDE 5409AF96 C5426FA3 75D99C68 E9AE7141
72D7F045 02D45307 FA4839F3 9A28338B BAFD54A4 61A53540 8367D513
2E6AA0D3 DA697336 0F8CD0F1

xQ31 = 00000000

yQ30 = 00000000

yQ31 = 00003351 F421FC15 8472AC2D D8B4DABB 5B599456 748A5BCC 4449398F
05ED1AD1 414B4EEB BB70FB91 383474B7 12EA4B5B F096092C DDD57C0A
090B0410 22064C3A 8DD3D890 E7B5AC34 A24CEF50 7955F027 CC4CECFD
B67739CE 89F31FDC 5FE43243

xP30 = 0000605D 4697A245 C394B980 24A55547 46DC12FF 56D0C6F1 5D2F4812

3B6D9C49 8EEE98E8 F7CD6E21 6E2F1FF7 CE0C969C CA29CAA2 FAA57174
EF985AC0 A5042600 18760E9F DF67467E 20C13982 FF5B49B8 BEAB05F6
023AF873 F827400E 453432FE

xP31 = 00000000

yP30 = 00005634 690BFC14 C45E2FAA 930D6258 9855E5BD D1435CFF BDF60962
8FD043B4 BF295BB3 5F7B6D37 836F2C59 A27BB61E D0FF57FF 8093FE6B
712133D2 6502F17C B0D46CDC 8CF9BA76 64EA2B6C 1672A8CA 2FF1CE31
3FEEEF41 99FC7F14 FE720617

yP31 = 00000000

xR30 = 000055E5 124A05D4 809585F6 7FE9EA1F 02A06CD4 11F38588 BB631BF7
89C3F98D 1C332584 3BB53D9B 011D8BD1 F682C0E4 D8A5E723 364364E4
0DAD1B7A 476716AC 7D1BA705 CCDD680B FD4FE473 9CC21A9A 59ED544B
82566BF6 33E89501 86A79FE3

xR31 = 00005AC5 7EAFD6CC 7569E8B5 3A148721 953262C5 B404C143 380ADCC1
84B6C21F 0CAFE095 B7E9C79C A88791F9 A72F1B2F 3121829B 2622515B
694A1687 5ED637F4 21B539E6 6F2FEF1C E8DCEFC8 AEA60805 5E9C4407
7266AB64 611BF851 BA06C821

1.6.8 SIKEp751_compressed

p = 00006FE5 D541F71C 0E12909F 97BADC66 8562B504 5CB25748 084E9867
D6EBE876 DA959B1A 13F7CC76 E3EC9685 49F878A8 EEAFFFFF FFFFFFFF
FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
FFFFFFFF FFFFFFFF FFFFFFFF

e2 = 00000174

e3 = 000000EF

xQ20 = 00002F42 9823758D A73CC081 3C8D2A15 5539422C C3626F96 0894A5F0
C80EB816 2757BB4F 9973B0AD 3BE80434 D2F2274B B2B67382 E963F75D
095A7F4B 2117D035 87A0253A BFECE7DA C3D05EF9 B771D1AE F9CA904C
EDBFE5DD 06BDB972 1B084F9F

xQ21 = 000001EF 0F6ACD92 06BD5717 62E80C1C A9A70620 469CB867 4DC960DA
A92B8DBF 4356747C 98CF6CF5 FC3868D6 6A81C877 0D1FF1E2 464E8D16
232E0F2A B97A4502 523E86E4 5695747D 7549DE8D CE950C87 1422F4AD
1C910B72 89D03AFD F60F82F6

xP20 = 00002DAA 34E43B70 4940549A 6BD34F33 6869E9E3 9594567D C5D1D114
73A0905E F38A10FC 52BCCE8C 6626DB3A 14473A2D ABC9A0AE 65635619
590E3739 AEC25727 8D27A184 2DCF41DA AB8C9DC1 785E4B51 BDC11212
A65DAAF6 413E0B4B 01424D5D

xP21 = 000007C5 AD0C2460 8F54E811 A18B6C2C 38FB00A8 F10FBC51 AB3C3B01
6A98367C B1CE89FC 12583110 9517F405 517F588F FCB221D8 683750F2
D59C8D8A 4A35D14B A6B89451 5BD7E856 5CD5F779 FAFFB24C 27C9E87F
FD650EF5 D6B1A59D 62367E02

xR20 = 00001F0C FC586169 3969F69E 0BB690DB 3F6F7028 CCB78EEE 62759E54
94CF24DD 007392D4 526946DB 5B856192 9B49996D D22B1AE8 D8D7F05F
9BEC60F6 750B56A0 77F5066E D3C9F67C 89DBE517 B99F7A17 5868AD6E
192470E6 18FBED50 F5EDDD87

xR21 = 000015FD 8097ACA6 9AB4F140 6445749E 9743F440 1A40E03B AD039AA3
E55803AC 88E8D7AE 9ACB6068 105AD134 E0327915 1A1BE245 82CEF065
03295CEB 3F278799 D02E3C8E E0EB62BE BFE133F8 E31F83C4 B4CD6093
7DD1D191 83A7AB9F 04FF9AF6

xQ30 = 000049E7 19883EE7 671DC6B2 E480C96C F1B38889 3E510957 A2BA76CB
65916AD7 6751440F E9A2ED54 2C82B09A 7E81BE06 97A1D7E9 D6EFEC66
19CAAC92 9BDE6EB6 3AF379DF 8F290BBD AE3E8ECE 23C125E9 BAB87AA5
F1248050 DA1AF4F4 0B74B755

xQ31 = 00003047 02CEB5C2 F0B4B99C CAD5DFB6 66DB825D 662A3A34 A40E79C6
9CEAB5FC 7374C1EF A6F1A465 F7C90CB2 45CD8A32 BC2B560F 1FE20740
7DD86656 7C6981A3 EE57971A FC29AABA C22EB799 F946B2E4 287C45C1
80DE6D62 FA4FF221 9900C63A

xP30 = 000049E7 19883EE7 671DC6B2 E480C96C F1B38889 3E510957 A2BA76CB
65916AD7 6751440F E9A2ED54 2C82B09A 7E81BE06 97A1D7E9 D6EFEC66
19CAAC92 9BDE6EB6 3AF379DF 8F290BBD AE3E8ECE 23C125E9 BAB87AA5
F1248050 DA1AF4F4 0B74B755

xP31 = 00003F9E D2734159 1D5DD702 CCE4FCB0 1E8732A6 F6881D13 64401EA1
3A01327A 6720D92A 6D062810 EC2389D3 042AEE76 3284A9F0 E01DF8BF
822799A9 83967E5C 11A868E5 03D65545 3DD14866 06B94D1B D783BA3E
7F21929D 05B00DDE 66FF39C5

xR30 = 0000193E 2242DBEF FD930260 70F635B4 73459058 55C22F21 2E26946E
D8C84DFB 38C592AD C076400A 788DEE67 D6462706 64D4BFF7 78C48446
0250AE93 05D89BF3 832239B1 38D6D099 AD26B99C 5C83CE75 C20CDB7F
3FA01029 EDE53E4A B3B4DFD6

xR31 = 00000000

Chapter 2

Detailed performance analysis

The submission package includes:

1. A generic reference implementation written exclusively in portable C with simple algorithms to compute isogeny and field operations, using GMP for multi-precision arithmetic,
2. An optimized implementation written exclusively in portable C that includes efficient algorithms to compute isogeny and field operations,
3. An additional, optimized implementation for x64 platforms that exploits x64 assembly,
4. An additional, optimized implementation for x64 platforms that exploits x64 assembly and public key compression (§1.5),
5. An additional, optimized implementation for ARM64 platforms that exploits ARMv8 assembly,
6. An additional, optimized implementation for ARM Cortex M4 platforms that exploits ARM thumb assembly,
7. An additional, speed-optimized VHDL model for FPGA and ASIC platforms that parallelizes various aspects of the isogeny computation and field operations, and
8. An additional, simple textbook implementation written exclusively in portable C, using elliptic curves in short Weierstrass form.

All implementations except implementations number 1 and 8 are protected against timing and cache attacks at the software level. Specifically, they avoid the use of secret address accesses and secret branches.

The version with public key compression (number 4) uses the same public parameters as the uncompressed version, but requires different KAT files, because the output formats are different. Therefore, we formally assign to this implementation a different collection of parameter sets, denoted by `SIKEpXXX_compressed` for `XXX` $\in \{434, 503, 610, 751\}$.

The Weierstrass implementation (number 8) supports the same prime sizes as the main implementation (namely, 434, 503, 610, and 751 bits). However, it is not directly compatible with any of the parameter sets, because its main purpose is to illustrate isogeny computations using textbook formulas over elliptic curves in short Weierstrass form, whereas the parameter sets are defined using Montgomery curves. Converting between curves in short Weierstrass form and the curves of Montgomery form used in the parameter sets would defeat the purpose of having a simple textbook implementation.

In this chapter we describe the main features of the implementations and analyze their performance.

2.1 Reference implementation

The reference implementation is written in portable C, and uses simple algorithms for isogeny and elliptic curve computations. Isogenies are computed using a dense tree traversal algorithm, and elliptic curve computations use affine coordinates and a double-and-add scalar multiplication algorithm. Specifically, this implementation makes use of Algorithms 25–45 listed in Appendix B. As in the optimized implementation (see §2.2), the reference implementation uses Montgomery elliptic curves in the form $By^2 = x^3 + Ax^2 + x$, but with full x- and y-coordinates. The implementation is generic and is built to a single library supporting all SIKE instantiations. Additionally, a small library supporting the NIST KEM API is built for each of the SIKE instantiations. The code base is split in several layers:

1. Multiprecision arithmetic using GMP.
2. Finite field arithmetic over $\mathbb{F}_p$ is implemented with a generic API, hiding the underlying GMP functions. The same API is used for any prime. The function headers are available in fp.h.
3. Quadratic extension field arithmetic over $\mathbb{F}_{p^2}$ is built on top of the $\mathbb{F}_p$ API. The function headers are available in fp2.h.
4. Montgomery elliptic curve arithmetic uses the $\mathbb{F}_{p^2}$ code and implements point addition, point doubling, point tripling, 2/3/4-isogeny generation and evaluation, scalar multiplication and j-invariant computation. For simplicity reasons, the scalar multiplication algorithm is not safe against side-channel attacks, but could be protected with well known countermeasures against side-channel attacks for ECC. The headers for Montgomery curve arithmetic and 2/3/4-isogeny generation are available in montgomery.h and isogeny.h, respectively.
5. The SIDH key agreement scheme is implemented with the key-generation algorithm (corresponding to $\texttt{isogen}_\ell$) and the shared secret algorithm (corresponding to $\texttt{isoex}_\ell$). The function headers are available in sidh.h.
6. The SIKE key encapsulation protocol is built on top of SIDH and implements PKE encryption, PKE decryption, KEM encapsulation and KEM decapsulation. The function headers are available in sike.h and api_generic.h.

7. The parameters for SIKEp434, SIKEp503, SIKEp610 and SIKEp751 are instantiated, all using the same generic implementation. The parameters are defined in sike_params.h. Each instantiation leads to a small library that support the NIST KEM API defined in api.h.

The reference implementation uses the same public-key format and encoding that is used in the optimized implementation. KATs are compatible with both the reference implementation and the optimized implementation.

2.2 Optimized and x64 assembly implementations

The optimized implementation, which is written in portable C only, uses efficient algorithms for isogeny and elliptic curve computations using projective coordinates on Montgomery curves, the Montgomery ladder, and efficient tree traversal strategies for fast isogeny computation. Specifically, this implementation makes use of Algorithms 3–24 listed in Appendix A. The optimal tree traversal strategies used in Algorithms 19 and 20 are given in Appendix D along with the algorithm used to compute them. Operations over $\mathbb{F}_{p^2}$ exploit efficient techniques such as Karatsuba and lazy reduction. Multiprecision multiplication is implemented using a fully rolled version of Comba, and modular reduction is implemented using a fully rolled version of Montgomery reduction. Hence, the field arithmetic implementation is generic and very compact. Conveniently, the optimized implementation reuses the same codebase for all the security levels.

The only difference between the optimized and the additional x64 implementation is that the latter exploits x64 assembly to implement the field arithmetic. Thus the field arithmetic in the x64 implementation is specialized per security level. All the rest of the code between the optimized and x64 implementations is shared, making the library compact and simple.

In the case of the additional x64 implementation, integer multiplication is implemented using one-level Karatsuba built on top of schoolbook multiplication. For our implementation, schoolbook offers a better performance than Comba thanks to the availability of MULX and ADX instructions in modern x64 processors. Modular reduction is implemented using an efficient version of radix-r Montgomery reduction and exploiting the MULX and ADX instructions (when available), as done in [14].

As previously stated, the optimized and additional x64 implementations follow standard practices to protect against timing and cache attacks at the software level and, hence, are expected to run in *constant time* on typical x64 Intel platforms.

2.2.1 Performance on x64 Intel

To evaluate the performance of the optimized and x64-assembly implementations, we ran our benchmarking suite on a machine powered by a 3.4GHz Intel Core i7-6700 (Skylake) processor, running Ubuntu 16.04.3 LTS. The reference implementation is linked against GMP 6.1.1. As is standard practice, TurboBoost was disabled during the tests. For compilation we used clang version 6.0.1 with the command `clang -O3`. Results are similar, although slightly slower, when compiling with GNU GCC version 7.2.0.

Scheme	KeyGen	Encaps	Decaps	**total** (Encaps + Decaps)
Reference Implementation				
SIKEp434	1,047,991	1,482,681	1,790,304	3,272,987
SIKEp503	1,567,725	2,237,865	2,752,500	4,990,364
SIKEp610	2,661,251	3,882,513	4,595,856	8,478,369
SIKEp751	4,743,861	6,534,356	8,016,158	14,550,514
Optimized Implementation				
SIKEp434	56,378	90,773	96,592	187,365
SIKEp503	85,744	140,781	149,972	290,753
SIKEp610	160,401	294,628	296,577	591,205
SIKEp751	288,827	468,175	502,983	971,158
Additional implementation using x64 assembly				
SIKEp434	5,927	9,681	10,343	20,024
SIKEp503	8,243	13,544	14,415	27,959
SIKEp610	14,890	27,254	27,445	54,699
SIKEp751	25,197	40,703	43,851	84,553
Compressed SIKE implementation using x64 assembly				
SIKEp434_compressed	10,158	15,120	11,077	26,197
SIKEp503_compressed	14,452	21,190	15,733	36,923
SIKEp610_compressed	26,360	37,470	29,216	66,686
SIKEp751_compressed	40,935	63,254	46,606	109,860

Table 2.1: Performance (in thousands of cycles) of SIKE on a 3.4GHz Intel Core i7-6700 (Skylake) processor. Cycle counts are rounded to the nearest 10^3 cycles.

Table 2.1 details the performance of the reference, optimized, x64-assembly, and compressed implementations of SIKE. As we can see, the *constant-time* optimized implementation is about 14-18 times faster than the *variable-time* reference implementation, thanks to the use of more efficient elliptic curve arithmetic and optimal strategies for isogeny computation. The use of assembly optimizations further improves performance greatly. Compilers still do a poor job of generating efficient code for multiprecision operations, especially multiprecision multiplication and reduction. Thus, our best performance for SIKEp434, SIKEp503, SIKEp610 and SIKEp751 (i.e., 5.9 msec., 8.2 msec., 16.1 msec. and 24.9 msec., respectively, obtained by adding the times for encapsulation and decapsulation) is achieved with the use of hand-tuned x64 assembly.

Scheme	secret key sk	public key pk	ciphertext ct	shared secret ss
SIKEp434	374	330	346	16
SIKEp503	434	378	402	24
SIKEp610	524	462	486	24
SIKEp751	644	564	596	32
SIKEp434_compressed	350	197	236	16
SIKEp503_compressed	407	225	280	24
SIKEp610_compressed	491	274	336	24
SIKEp751_compressed	602	335	410	32

Table 2.2: Size (in bytes) of inputs and outputs in SIKE.

Memory analysis

First, in Table 2.2 we summarize the sizes, in terms of bytes, of the different inputs and outputs required by the KEM. We point out that we also include the public key in the secret key sizes in order to comply with NIST's API guidelines. Specifically, since NIST's decapsulation API does not include an input for the public key, it needs to be included as part of the secret key (see §1.3.11).

Table 2.3 shows the peak (stack) memory usage per function of the reference, optimized and additional x64-assembly implementations. In addition, on the right-most column we display the size of the produced static libraries.

To determine the memory usage we first run `valgrind` (http://valgrind.org/) to get "memory use snapshots" during execution of the test program:

```
$ valgrind --tool=massif --stacks=yes --detailed-freq=1 ./sike/test_KEM
```

The command above produces a file of the form `massif.out.xxxxx`. Afterwards, we run `massif-cherrypick` (https://github.com/lnishan/massif-cherrypick), which is an extension that outputs memory usage per function:

```
$ ./massif-cherrypick massif.out.xxxxx kem_function
```

Looking at the results in Table 2.3, one can note that the use of stack memory is relatively low. This is one advantage of supersingular isogeny based schemes, which is partly due to the fact that these schemes exhibit the most compact keys among popular post-quantum cryptosystems.

It can also be seen that the static library sizes can grow relatively high (see option compiled for speed). However, it is possible to reduce the library sizes significantly, to around 60KB, at little performance cost: compiling the additional implementations for size more than halves the library sizes and reduces speed by less than 5%. It should be noted that the reference implementation is a single library for all SIKE instantiations, and that GMP attributes to its size because of static linking. The stack memory usage is relatively low due to GMP's internal memory management.

Scheme	KeyGen (stack)	Encaps (stack)	Decaps (stack)	static library speed (-O3)	static library size (-Os)
Reference Implementation					
SIKEp434	448	448	448	89,148	81,860
SIKEp503	512	512	512	89,148	81,860
SIKEp610	640	640	640	89,148	81,860
SIKEp751	768	768	768	89,148	81,860
Optimized Implementation					
SIKEp434	8,040	8,360	8,744	105,474	54,170
SIKEp503	8,072	8,456	8,904	120,202	58,714
SIKEp610	12,008	12,408	12,936	163,312	56,400
SIKEp751	13,912	14,040	14,696	164,810	60,162
Additional implementation using x64 assembly					
SIKEp434	8,136	8,456	8,840	108,208	56,672
SIKEp503	8,152	8,536	8,984	116,022	61,166
SIKEp610	13,536	12,512	12,112	135,470	68,494
SIKEp751	14,064	14,192	14,960	159,032	76,840
Compressed SIKE implementation using x64 assembly					
SIKEp434	16,920	15,640	17,000	593,138	458,418
SIKEp503	18,872	17,560	19,128	648,404	509,636
SIKEp610	23,824	22,048	24,144	866,796	698,148
SIKEp751	28,024	27,936	28,320	1,296,540	1,070,688

Table 2.3: Peak memory usage (stack memory, in bytes) and static library size (in bytes) of the various implementations of SIKE on a 3.4GHz Intel Core i7-6700 (Skylake) processor. Static libraries were obtained by compiling with clang and optimizing for speed (`-O3`) and for size (`-Os`).

2.3 Compressed SIKE implementation

Compared to round 2 specification we provide an improved implementation of compressed SIKE that incorporates recent new techniques introduced in [35, 36]. The implementation is available at `https://github.com/microsoft/PQCrypto-SIDH`.

In particular, [35] speeds up pairing computation (which is the major bottleneck in compression) and basis generation exploiting the use of dual isogenies. Small precomputed tables are used during encapsulation for the faster pairings. Additionally, [36] exploits a trade-off where 12.5% larger ciphertexts plus 1 field element larger secret keys are employed in exchange of a $\approx$ 1.6 to 1.7 times faster decapsulation. Moreover, [36] gets rid of the previous discrete log tables (of the order of megabytes) for the decapsulation operation since compression-related operations like pairing or discrete logs were completely avoided. This is highly welcome for memory-constrained IoT devices that are intended to execute decapsulation and decrypt traffic.

The compressed SIKE implementation uses the same codebase as the optimized x64 implementation, but additionally performs public key compression. Compression is performed both for the static public key and for the component of the ciphertext corresponding to the ephemeral public key.

In SIDH (§4.3.1), the public key pk_2 encodes 3^{e_3}-torsion points, and the public key pk_3 encodes 2^{e_2}-torsion points. Key compression is faster on 2^{e_2}-torsion points than on 3^{e_3}-torsion points. If we assume that one public key may be used to encrypt multiple ciphertexts over its lifespan, then we must designate pk_2 for static public keys and pk_3 for ephemeral public keys in order to achieve optimal performance in the KEM in the key compression setting. Accordingly, **the compressed SIKE implementation swaps the roles** of the subscripts $_2$ and $_3$ in Algorithm 2.

Tables 2.1 and 2.2 respectively indicate the performance penalty and key size improvements offered by compressed SIKE. The performance penalties range from 62% to 77% (vs 139% to 161% in previous round 2) for key generation, 38% to 56% (vs 66% to 90% in previous round 2) for encapsulation, and **7% to 9% (vs 59% to 68% in previous round 2) for decapsulation**, depending on the parameter size. The size of the public key is reduced by 41%, and the size of the ciphertext by 31%.

As seen in Table 2.3, compressed SIKE has modest memory requirements but incurs a large penalty in terms of static library size. The size increase arises because our implementation uses large tables of discrete logarithms in order to speed up compression. A time-space tradeoff is possible here — smaller tables can be used, in exchange for slower performance. For some applications, such as IoT, which are constrained in both time and space, further work is needed in order to find the optimal trade-off point.

2.4 64-bit ARM assembly implementation

The submission includes an additional implementation for 64-bit ARM processors. This implementation is identical to the additional x64 implementation with the exception of the field arithmetic, which is written with hand-optimized ARMv8 assembly.

To evaluate the performance of this implementation, we ran our benchmarking suite on a 1.992GHz 64-bit ARM Cortex-A72 processor, running Ubuntu 16.04.2 LTS. For compilation we used GNU GCC version 5.4.0 with the command `gcc -O3`.

Table 2.4 compares the performance of the additional ARMv8-assembly implementation of SIKE to the (portable) optimized implementation of SIKE. As we can see, the specialized implementation is roughly 5 to 6 times faster than the generic optimized implementation, thanks to the use of assembly routines. Our best performance for `SIKEp434`, `SIKEp503`, `SIKEp610` and `SIKEp751` on the targeted platform is 29.4 ms, 40.9 ms, 94.9 ms and 141.6 ms respectively, corresponding to the total time that it takes to compute the encapsulation and decapsulation operations.

For more information on the ARM assembly optimizations, see [20, 21, 40].

Scheme	KeyGen	Encaps	Decaps	total (Encaps + Decaps)
Optimized implementation (portable)				
SIKEp434	95,077	155,810	166,111	321,921
SIKEp503	147,731	243,716	258,976	502,692
SIKEp610	280,776	516,892	519,834	1,036,726
SIKEp751	509,655	827,192	887,972	1,715,164
SIKEp434_compressed	157,804	238,739	176,645	415,383
SIKEp503_compressed	243,148	365,538	273,825	639,363
SIKEp610_compressed	487,244	696,832	548,679	1,245,511
SIKEp751_compressed	801,979	1,267,991	939,763	2,207,754
Additional implementation using ARMv8 assembly				
SIKEp434	17,321	28,321	30,221	58,542
SIKEp503	24,046	39,501	42,121	81,621
SIKEp610	49,108	90,276	90,831	181,107
SIKEp751	84,005	135,956	146,133	282,089
SIKEp434_compressed	29,129	44,352	32,623	76,975
SIKEp503_compressed	40,434	60,702	45,198	105,900
SIKEp610_compressed	86,010	123,086	96,805	219,891
SIKEp751_compressed	136,036	211,055	155,682	366,737

Table 2.4: Performance (in thousands of cycles) of SIKE on a 1.992GHz 64-bit ARM Cortex-A72 processor. Results are measured in ns and scaled to cycles using the nominal processor frequency. Cycle counts are rounded to the nearest 10^3 cycles.

2.5 ARM Cortex-M4 assembly implementation

The submission includes an additional implementation for the ARM Cortex-M4 processor. This implementation is identical to the additional x64 implementation with the exception of the field arithmetic, which is written with hand-optimized ARMv7 assembly targeting the Cortex-M4 processor. These results are available in [42] as well as [41].

To evaluate the performance of this implementation, we ran our benchmarking suite on a 168 MHz STM32F407G-DISC1 board and a 96 MHz PowerShield X-NUCLEO-LM01A board, both running the pqm4 OS. For compilation we used GNU GCC version 5.4.0 with the command `gcc -O3`.

Table 2.5 compares the performance of the additional M4-assembly implementation of SIKE to the (portable) optimized implementation of SIKE on the 168 MHz STM Discovery board. As we can see, the specialized implementation is roughly 13 to 16 times faster than the generic optimized implementation, thanks to the use of assembly routines. Our best performance for `SIKEp434`, `SIKEp503`, `SIKEp610`, and `SIKEp751` on the Discovery board is 1.09 sec., 1.53 sec., 2.94 sec., and 4.58 sec., corresponding to the total time that it takes to compute the encapsulation and decapsulation operations.

Scheme	KeyGen	Encaps	Decaps	total (Encaps + Decaps)
Optimized implementation (portable)				
SIKEp434	718	1,175	1,254	2,429
SIKEp503	1,076	1,773	1,886	3,659
SIKEp610	2,011	3,701	3,722	7,423
SIKEp751	3,657	5,915	6,353	12,267
Additional implementation using ARMv7 Cortex-M4 assembly				
SIKEp434	54	89	95	184
SIKEp503	76	125	133	257
SIKEp610	134	246	248	493
SIKEp751	229	371	399	770

Table 2.5: Performance (in millions of cycles) of SIKE on a 168MHz 32-bit ARM Cortex-M4 processor on the STM32F407G-DISC1 board. Results are measured in ms and scaled to cycles using the nominal processor frequency. Cycle counts are rounded to the nearest 10^6 cycles.

Scheme	Timing [cc $\times 10^6$]			Average Power [mW]			Energy [mJ]		
	KeyGen	Encaps	Decaps	KeyGen	Encaps	Decaps	KeyGen	Encaps	Decaps
Optimized implementation (portable)									
SIKEp434	696	1,139	1,215	66.91	67.28	67.20	485.00	798.32	850.72
SIKEp503	1,046	1,724	1,833	66.56	66.72	66.70	724.96	1,198.00	1,273.00
SIKEp610	1,957	3,602	3,623	66.63	67.06	66.98	1,358.00	2,516.00	2,528.00
SIKEp751	3,555	5,766	6,193	65.75	66.46	66.25	2,435.00	3,992.00	4,273.00
Additional implementation using ARMv7 Cortex-M4 assembly									
SIKEp434	49	79	85	73.74	74.70	74.34	37.26	61.60	65.54
SIKEp503	67	111	118	75.48	76.16	76.10	53.03	87.89	93.55
SIKEp610	120	219	211	78.17	78.75	78.74	97.36	180.50	181.30
SIKEp751	205	312	357	80.66	81.16	81.21	172.07	280.53	301.58

Table 2.6: Power and energy consumption of SIKE on a 96 MHz 32-bit ARM Cortex-M4 processor on the PowerShield X-NUCLEO-LM01A board.

Table 2.6 compares the average power and energy consumption of the additional M4-assembly implementation of SIKE to the (portable) optimized implementation of SIKE on the 96 MHz PowerShield board. As this is a different Cortex-M4 board, the timing results are slightly different from the Discovery board. In general, SIKE takes slightly less cycles on the PowerShield board than the Discovery board. As we can see, the specialized implementation consumes roughly 10% to 23% more average power, but is 13 to 14 times more energy efficient. Our best energy results for `SIKEp434`, `SIKEp503`, `SIKEp610`, and `SIKEp751` on the PowerShield board are 127.14 mJ, 181.44 mJ, 361.80 mJ, and 582.11 mJ, corresponding to the total energy consumption to compute the encapsulation and decapsulation operations.

#Multipliers	Scheme	Cycle counts ($cc \times 10^3$)			
		KeyGen	Encaps	Decaps	**Total**
6	SIKEp434	528	926	980	1,906
6	SIKEp503	643	1,144	1,201	2,345
6	SIKEp610	895	1,807	1,781	3,588
8	SIKEp751	1,252	2,206	2,335	4,541

Table 2.7: Summary of cycle counts for SIKE accelerator architecture over SIKE Round 2 parameter sets. The number of multipliers is a design parameter.

2.6 VHDL hardware implementation

The optimized VHDL hardware implementation accelerates SIKE operations by using Algorithms 3–24 listed in Appendix A. Thus, this hardware implementation uses projective coordinates on Montgomery curves, an efficient double-point multiplication ladder, and an efficient tree traversal algorithm for isogeny computation. A separate tree traversal strategy was computed with Algorithm 60 in Appendix D using $p = 2$ and $q = 1$ which emphasizes isogeny evaluations. Notably, the hardware implementation focuses on exploiting additional amounts of parallelism through the use of high-radix Montgomery multiplication, simultaneous isogeny evaluation, and efficient scheduling of resources. This hardware implementation emphasizes speed over area and power consumption. The details of this implementation can be found in [27], which went through several iterations of optimizations in [30, 31, 32].

There are four different instantiations of the SIKE hardware accelerator, each targeting a different parameter set: `SIKEp434`, `SIKEp503`, `SIKEp610`, and `SIKEp751`. The isogeny accelerator architecture includes a controller, program ROM, finite field arithmetic unit, register file, Keccak block, and secret message buffer. After populating the register file with the public parameters, adding keys, and writing a command, the controller can perform each step of the key encapsulation mechanism or the individual isogeny computations ($\texttt{isogen}_2$, $\texttt{isogen}_3$, $\texttt{isoex}_2$, and $\texttt{isoex}_3$) for the chosen public parameter set.

2.6.1 Performance

The SIKE hardware accelerators can perform KEM functions for the SIKE Round 2 public parameters: `SIKEp434`, `SIKEp503`, `SIKEp610`, and `SIKEp751`. There is some configurability in the number of replicated dual-multipliers which affects the number of cycles per operation. Since the isogeny operations require the most time, this implementation parallelizes various finite field arithmetic and isogeny calculations. In Table 2.7, we specify the total number of cycles to perform the key encapsulation operations based on the number of multipliers. In the hardware package, we use 3 dual-multipliers, or 6 total multipliers for `SIKEp434`, `SIKEp503`, and `SIKEp610`, as well as 4 dual-multipliers, or 8 total multipliers for `SIKEp751`.

2.6.2 FPGA SIKE Accelerator

The VHDL SIKE accelerator cores were compiled for FPGA with Xilinx Vivado design suite version 2019.2 to a Xilinx Artix-7 xc7a200tffg1156-3 device and a Xilinx Kintex Ultrascale+ xcku13p-ffve900-

Scheme	Area					Freq	Time (msec)			
	# FFs	# LUTs	# Slices	# DSPs	# BRAMs	(MHz)	KeyGen	Encaps	Decaps	total (Encaps + Decaps)
SIKEp434	24,328	21,946	8,006	240	26.5	132.2	4.00	7.01	7.42	14.43
SIKEp503	27,759	24,610	9,186	264	33.5	129.9	4.95	8.81	9.25	18.06
SIKEp610	33,198	29,447	10,843	312	39.5	125.3	7.15	14.43	14.22	28.65
SIKEp751	49,982	40,792	15,794	512	43.5	127.0	9.86	17.37	18.39	35.76

Table 2.8: FPGA implementation results of SIKE accelerator on a Xilinx Artix-7 FPGA.

Scheme	Area					Freq	Time (msec)			
	# FFs	# LUTs	# CLBs	# DSPs	# BRAMs	(MHz)	KeyGen	Encaps	Decaps	total (Encaps + Decaps)
SIKEp434	23,881	21,657	5,237	240	26.5	299.4	1.76	3.09	3.28	6.37
SIKEp503	27,783	24,255	4,601	264	33.5	305.3	2.11	3.75	3.93	7.68
SIKEp610	33,193	28,758	5,404	312	39.5	300.1	2.98	6.02	5.94	11.96
SIKEp751	50,143	40,700	7,726	512	43.5	296.9	4.22	7.43	7.87	15.30

Table 2.9: FPGA implementation results of SIKE accelerator on a Xilinx Kintex UltraScale+ FPGA.

3-e device. All results were obtained after place-and-route. The area and timing results of our accelerator cores on an Artix-7 FPGA and a Kintex UltraScale+ FPGA are shown in Tables 2.8 and 2.9, respectively. These are constant-time results. On the Artix-7, our FPGA implementations can perform SIKE key encapsulation and decapsulation over SIKEp434, SIKEp503, SIKEp610, and SIKEp751 in 14.43 msec., 18.06 msec., 28.65 msec., and 35.76 msec., respectively. On the Kintex UltraScale+, our FPGA implementations can perform SIKE key encapsulation and decapsulation over SIKEp434, SIKEp503, SIKEp610, and SIKEp751 in 6.37 msec., 7.68 msec., 11.96 msec., and 15.30 msec., respectively.

Chapter 3

Known Answer Test values

The submission includes KAT values with tuples containing secret keys (sk), public keys (pk), ciphertexts (ct) and shared secrets (ss) corresponding to the proposed KEM schemes SIKEp434, SIKEp503, SIKEp610, SIKEp751, SIKEp434_compressed, SIKEp503_compressed, SIKEp610_compressed and SIKEp751_compressed. The KAT files can be found in the media folder of the submission: \KAT\PQCkemKAT_374.rsp, \KAT\PQCkemKAT_434.rsp, \KAT\PQCkemKAT_524.rsp and \KAT\PQCkemKAT_644.rsp, \KAT\PQCkemKAT_239.rsp, \KAT\PQCkemKAT_280.rsp, \KAT\PQCkemKAT_336.rsp and \KAT\PQCkemKAT_413.rsp for SIKEp434, SIKEp503, SIKEp610, SIKEp751, SIKEp434_compressed, SIKEp503_compressed, SIKEp610_compressed and SIKEp751_compressed, respectively.

In addition, we provide a test suite that can be used to verify the KAT values against any of the implementations. Instructions to compile and run the KAT test suite can be found in the README file in the top-level directory of the media folder (see Section 2, “Quick Instructions”).

Chapter 4

Expected security strength

4.1 Security

The security of SIKE informally relies on the *(supersingular) isogeny walk problem*: given two elliptic curves E, E' in the same isogeny class, find a path made of isogenies of small degree between E and E'.

The isogeny walk problem has been considered in the literature even before the introduction of isogeny-based cryptography. The best generic algorithm currently known is due to Galbraith [15]: it is a meet-in-the-middle strategy that, on average, requires a number of elementary steps proportional to the square root of the size of the isogeny class of E and E'. In the supersingular case, an improvement due to Delfs and Galbraith [12] has roughly the same computational complexity, but only uses a constant amount of memory.

Over $\mathbb{F}_{p^2}$, there is a unique isogeny class of supersingular elliptic curves (up to twist), and it has size roughly $p/12$. Thus, the algorithm of Delfs and Galbraith would find an isogeny between the starting curve E_0 and a public curve E' in $O(\sqrt{p})$ time.[1] Nevertheless, these generic algorithms do not improve upon exhaustive search. Indeed, if $p = 2^{e_2} \cdot 3^{e_3} - 1$, the key spaces $\mathcal{K}_2$ and $\mathcal{K}_3$ have sizes roughly 2^{e_2} and 3^{e_3}; thus, if these are chosen to balance out, then the size of the key spaces is roughly $\sqrt{p}$.

However, the idea of Galbraith's meet-in-the-middle approach can be easily adapted to attack SIKE in only $O(\sqrt[4]{p})$ operations. To find the secret isogeny of degree ℓ^{e_ℓ} between E_0 and E', an attacker builds a tree of all curves isogenous to E_0 via isogenies of degree $\ell^{e_\ell/2}$, and a similar tree of all curves isogenous to E' of degree $\ell^{e_\ell/2}$. Since we suppose that an isogeny of degree ℓ^{e_ℓ} exists between E_0 and E', and since the length of this walk is much shorter than the size of the graph, with high probability the two trees will have exactly one curve E'' in common, so the secret isogeny will be recovered by composing the paths $E_0 \to E''$ and $E'' \to E'$. This procedure only requires $O(\sqrt{\ell^{e_\ell}})$ elementary steps, or $O(\sqrt[4]{p})$, as announced.

Given two functions $f : A \to C$ and $g : B \to C$ with domain of equal size, finding a pair (a, b) such that $f(a) = g(b)$ is known as the *claw problem* in complexity theory. The claw problem can obviously be solved using $O(|A| + |B|)$ invocations of f and g on average, by building a hash table holding $f(a)$ for any $a \in A$

[1] The attentive reader will have noticed that knowing a generic path between E_0 and E' is not necessarily equivalent to knowing the secret path generated by `isogen`$_\ell$. However, a complete reduction of the security of SIKE to the isogeny walk problem is presented in [17].

and looking for hits for $g(b)$ where $b \in B$. However, one can do better with a quantum computer using Tani's claw-finding algorithm [47], which only uses $O(\sqrt[3]{|A||B|})$ invocations to quantum oracles for f and g. These complexities are optimal for a black-box claw attack [53]. For given supersingular curves E, E' we could, for example, let A resp. B be the set of points of order exactly $\ell^{e_\ell/2}$ on E resp. E', and C the set of supersingular j-invariants. The functions f and g compute $\ell^{e_\ell/2}$-isogenies which have kernels generated by their input points and return the j-invariant of the final curve. Classically this is exactly the $O(\sqrt{\ell^{e_\ell}})$ attack described above, and applying Tani's algorithm to `SIKE` gives an attack requiring $O(\sqrt[3]{\ell^{e_\ell}}) = O(\sqrt[6]{p})$ invocations of a quantum isogeny computation oracle.

While the generic algorithms described above (and their asymptotic complexities) were used for the security analysis in the initial `SIKE` proposal, a series of subsequent papers beginning with [1] have since argued that the parallel collision finding algorithm of van Oorschot and Wiener [50] is the best classical claw-finding attack on `SIKE`, and [23] even argues that the above query-optimal instantiation of Tani's algorithm is outperformed by the classical van Oorschot and Wiener algorithm. We further discuss the concrete security of `SIKE` in Chapter 5.

We stress that, while breaking `SIKE` keys can be reduced to claw finding, no reduction is known in the opposite direction, nor is it widely believed that such a reduction should exist. The security of `SIKE` is modeled after a much more specific problem named SIDH (see Problem 1). In particular the knowledge of the coordinates (x_1, x_2, x_3) output by $\texttt{isogen}_\ell$ apparently gives more information than what is available in the claw problem. Nevertheless, to this day no attack seems to be able to exploit this auxiliary knowledge against `SIKE`. For this reason, we assume that the security of the claw problem and SIDH are equivalent, and analyze security accordingly.

4.2 Other attacks

Other attacks applying to specific security models have appeared in the literature in recent years.

Galbraith, Petit, Shani and Ti [17] exhibit a very efficient polynomial-time attack against SIDH with static keys. Their technique is readily adapted to a chosen ciphertext attack against the scheme `PKE`. However, their attack does not apply to `KEM`, as we will prove in the next section that the scheme is CCA secure.

Many authors have considered the security of SIDH under various side-channel scenarios:

- Galbraith, Petit, Shani and Ti [17] show how a secret j-invariant can be recovered from some partial knowledge of it.
- Ti [49] explains how a random perturbation to the inputs of $\texttt{isogen}_\ell$ yields to a key recovery with very high probability in most protocols derived from SIDH. It is not clear, however, how the technique can be used against the public key format specified in 1.2.9.
- Gélin and Wesolowski [18] present a loop-abort fault attack that potentially leads to an efficient key recovery against the "simple" version of $\texttt{isogen}_\ell$ given in Algorithms 17 and 18. However their attack is efficiently countered by the additional checks in Algorithms 19 and 20.

A recent preprint by Petit [37] presents various polynomial-time attacks against generalizations of SIDH. None of the systems successfully attacked by Petit had previously appeared in the literature, and in particular the schemes presented in this document are not affected by the attack. It is not clear that Petit's attacks could possibly be extended to break real uses of SIDH and derived schemes. The technique employed by Petit, however, sheds some light on the separation between the isogeny walk problem and the possibly (though not yet shown to be) easier SIDH problem.

Even more recently, Petit and Lauter [38] showed that the isogeny walk problem used to construct the Charles-Goren-Lauter hash function [5] is equivalent to the problem of computing endomorphism rings of supersingular elliptic curves, which is possibly (but not yet shown to be) harder than the SIDH problem. However, it does not appear to be possible to extend the Charles-Goren-Lauter hash construction to yield key exchange.

4.3 Security proofs

The PKE scheme in §1.3.10 is a modified version of the classical hashed ElGamal scheme that replaces the group-based computational Diffie-Hellman problem by its analogue in the setting of supersingular isogenies (Problem 1 below). As such, the proofs of the IND-CPA PKE scheme and the subsequent IND-CCA KEM are standard; these are given in §4.3.2 and §4.3.3.

4.3.1 The SIDH problem

Problem 1 is the Supersingular Isogeny Diffie-Hellman (SIDH) problem [11, Problem 5.3].

Problem 1. *Let* $\mathrm{sk}_2 \in \mathcal{K}_2$ *and* $\mathrm{sk}_3 \in \mathcal{K}_3$. *Let* $\mathrm{pk}_2 = \mathtt{isogen}_2(\mathrm{sk}_2)$ *and* $\mathrm{pk}_3 = \mathtt{isogen}_3(\mathrm{sk}_3)$. *Given* $(E_0, \mathrm{pk}_2, \mathrm{pk}_3)$*, compute* $j = \mathtt{isoex}_2(\mathrm{pk}_3, \mathrm{sk}_2) = \mathtt{isoex}_3(\mathrm{pk}_2, \mathrm{sk}_3)$.

4.3.2 IND-CPA PKE

Define the IND-CPA security of a public-key encryption scheme in the standard way (e.g. see [3, 25]). Assume that F is a random oracle.

Proposition 1. *In the random oracle model,* `PKE` *is IND-CPA if SIDH is hard.*

Proof. The public-key encryption scheme is the classical hashed ElGamal scheme converted to the setting of supersingular isogeny graphs. More specifically, note that we can view ElGamal as a static-ephemeral Diffie–Hellman key exchange to obtain a shared secret, which is hashed and used a secret key for a symmetric algorithm (for example the one-time pad) to encrypt a message. The scheme `PKE` simply replaces the original group-based Diffie–Hellman exchange by an SIDH key exchange, but is otherwise identical to hashed ElGamal. As a result, its proof of security is completely analogous. For example, see [25, Thm 5], [16, Thm 20.4.11] or [24, Thm 11.21]. □

Remark 3. There exist alternative proofs of security in the standard model, reducing the security to a decisional variant of SIDH [11, Problem 5.4] instead of SIDH (see [11, Thm 6.2], based on [44, Thm 2] and [43, §3.4]).

4.3.3 IND-CCA KEM

Theorem 1 ([19]). *For any IND-CCA adversary B against* $\texttt{KEM}$*, issuing at most q_G (resp. q_H) queries to the random oracle G (resp. H), there exists an IND-CPA adversary A against* $\texttt{PKE}$ *with*

$$\mathrm{Adv}^{\texttt{IND-CCA}}_{\texttt{KEM}}(B) \leq \frac{2q_G + q_H + 1}{2^n} + 3 \cdot \mathrm{Adv}^{\texttt{IND-CPA}}_{\texttt{PKE}}(A).$$

Proof. This is the bound obtained by combining the results from Theorem 3.2 and Theorem 3.4 from [19], setting $\texttt{KEM} = U^{\not\perp}[T[\texttt{PKE}, G], H]$.

Note that $\texttt{Decaps}$ slightly deviates from the definition in [19]. Instead of full "re-encryption" $(c_0', c_1') \leftarrow \texttt{Enc}(\mathrm{pk}_3, m'; G(m' \parallel \mathrm{pk}_3))$, we only re-compute c_0'. However, full computation would yield

$$c_1' = m' \oplus F(\texttt{isoex}_2(\mathrm{pk}_3, G(m' \parallel \mathrm{pk}_3))) = m' \oplus F(\texttt{isoex}_3(c_0', \mathrm{sk}_3)),$$

while $c_1 = m' \oplus F(\texttt{isoex}_3(c_0, \mathrm{sk}_3))$. Hence it is clear that $c_0' = c_0$ implies $c_1' = c_1$, making the computation of c_1' redundant. $\square$

Chapter 5

Analysis with respect to known attacks

In choosing concrete parameter sizes, our goal is to ensure that the computational cost of breaking SIKEpXXX, where XXX $\in \{434, 610, 751\}$, requires respective resources comparable to those required for key search on a k-bit (ideal) block cipher $\mathcal{B}$, where $k \in \{128, 192, 256\}$. In addition, our goal is to ensure that the computational cost of breaking SIKEp503 requires resources comparable to those required for collision search on a 256-bit (ideal) hash function.

We discuss the complexity of the best known classical attacks in §5.1 and the complexity of the best known quantum attacks in §5.2. Side-channel attacks are discussed in §5.3.

5.1 Classical security

Following the submission of SIKE to the NIST call in November of 2017, a series of papers have emerged that have scrutinized the application of generic meet-in-the-middle attacks described in §4.1. The work of Adj, Cervantes-Vázquez, Chi-Domínguez, Menezes and Rodríguez-Henríquez [1] was the first paper to argue that the parallel collision-finding algorithm of van Oorschot and Wiener (vOW) [50] is actually the attack that should be used to evaluate the security of SIKE. The reason is that the $O(p^{1/4})$ memory that is required to mount the generic meet-in-the-middle attack — that which runs in $O(p^{1/4})$ time — is far beyond feasible for SIKE parameters in the ranges of interest. Since the best known generic attack against ideal block ciphers (e.g., AES) use only a moderate amount of memory, in deriving SIKE parameters for which the computational resources are comparable to AES instantiations, the most appropriate model is to fix an upper bound on the classical memory available, and to evaluate the runtime of the best known attacks subject to this limit.

Under the assumption that the memory available permits the storage of 2^{80} *units*, Adj et al. [1] conclude that SIKEp434 and SIKEp610 meet the respective security requirements of NIST's categories 2 and 4. A subsequent paper by Jaques and Schanck [23] — which is largely geared towards the analysis of quantum algorithms, but also considers vOW — further endorses the classical complexity claims of Adj et al with respect to these two curves and the NIST requirements they satisfy. And, in addition to a further endorsement of these two curves, a recent paper by Costello, Longa, Naehrig, Renes and Virdia [9] argues that SIKEp751, which was initially proposed to meet level 3, actually meets NIST's category 5 requirements.

We refer to these three papers (and the original vOW paper) for the in-depth analyses, but we summarize their application to three SIKE parameterizations in Table 5.1, noting that they use slightly different memory assumptions and/or cost metrics in order to estimate the complexity of vOW against SIKE parameters. Adj et al. assume that the memory permits the storage of 2^{80} units, and present their results in "total time", where the unit of time is actually the time complexity of a degree $\ell^{e/2}$-isogeny; thus, although their *times* fall slightly below NIST's required *gate counts*, the corresponding conversion to gate counts would see these parameters comfortably exceed NIST's requirements. The classical analysis of Jaques and Schanck uses the PRAM model and estimates the number of classical gates under the assumption that the memory is 2^{96} bits. Their model does incorporate the cost of the isogeny computations, but is still rather conservative. Finally, the vOW analysis of Costello et al. estimates the total number of x64 instructions required to mount the vOW attack, and argues that this is also a conservative lower bound on the true classical gate count. In particular, for the `SIKEp751` parameterization, they conclude that the true gate count corresponding to their estimated 2^{262} x64 instructions would exceed NIST's 2^{272} gate count requirement.

	Target level	Classical gate requirement [48]	Classical security estimates		
			Total time [1] memory 2^{80} units	Gates [23, Fig. 4(d)] memory 2^{96} bits	x64 instructions [9] memory 2^{80} units
`SIKEp434`	1	143	128	142	143
`SIKEp503`	2	146	152	169*	169*
`SIKEp610`	3	207	189	209	210
`SIKEp751`	5	272	-	263*	262

Table 5.1: Classical security estimates of the three SIKE parameterizations according to Adj et al. [1], Jaques and Schanck [23], and Costello et al. [9]. Gate requirements and classical security estimates are all expressed as their base-2 logarithms. The values marked with (*) are not found in the actual papers. In the case of [9], we obtained the numbers for `SIKEp503` using their scripts, where (for the half-sized isogenies used in vOW) the optimal strategy for the 2-torsion resulted in 362 doublings and 189 4-isogenies, and the optimal strategy for the 3-torsion yielded 229 triplings and 275 3-isogenies. In the former scenario, a vOW isogeny required over 2^{22} x64 instructions, and in the latter, over 2^{23} x64 instructions. In the case of [23], the RAM operations for `SIKEp503` and `SIKEp751` were taken from the width-restricted table in §5.2.

5.2 Quantum security

The initial `SIKE` proposal used the asymptotic complexity of Tani's quantum claw-finding algorithm [47] together with crude lower bounds for the number of quantum gates used in an $\mathbb{F}_p$-multiplication and the number of such multiplications in a typical isogeny computation to provide conservative resource estimates for the cost of quantum cryptanalysis of `SIKE`. The recent paper by Jaques and Schanck [23] conducts a much more detailed analysis of the best known quantum algorithms to solve the computational supersingular isogeny problem. Jaques and Schanck propose a model of quantum computation that allows a direct comparison between quantum and classical algorithms. They treat qubit registers as memory peripherals for classical control processors, which run quantum circuits through RAM operations on qubit memory peripherals. This allows them to use the number of RAM operations as the algorithm's cost function, derived either from the quantum gate count in a quantum circuit or the product of its depth and width. The crucial difference to previous cost estimates lies in considering the complexity of implementing and querying quantum memory.

Jaques and Schanck consider both Tani's algorithm as well as a direct application of Grover's algorithm to the claw-finding problem, but also include the purely classical vOW algorithm. Their analysis provides various trade-offs between time, memory and RAM operations, which lead to the preference of different algorithm parameterizations depending on the given attack constraints. They conclude that in a scenario with limited memory, quantum algorithms promise to be more efficient, but that the classical vOW algorithm outperforms quantum algorithms for attackers with limited time. Therefore, in some scenarios, security against classical attacks is the limiting factor for selecting parameters. In particular, it is argued that the classical hardware required to run the query-optimal parameterization of Tani's algorithm can be used to break `SIKE` faster by running the classical vOW algorithm on that same hardware.

Figure 4 in [23] provides concrete cost estimates for solving the computational supersingular isogeny problem in different scenarios for the parameters `SIKEp434` and `SIKEp610`. The relevant constraint for matching the NIST security categories is imposing a depth restriction on quantum circuits between 2^{64} and 2^{96} (corresponding to the MAXDEPTH parameter in the NIST call for proposals [48]). Allowing depth 2^{96}, Jaques and Schanck conclude that no known quantum algorithm can break `SIKE` in their model of computation with less than 2^{143} classical gates and 2^{207} classical gates for `SIKEp434` and `SIKEp610`, respectively. Therefore, these two parameter sets are suitable for NIST categories 1 and 3. Running the scripts accompanying [23] to produce the same tables for `SIKEp503` and `SIKEp751` suggest that no quantum algorithm can break those with less than 2^{146} and 2^{272} classical gates, respectively, which confirms that these parameter sets are suitable for NIST categories 2 and 5.

Tables 5.2, 5.3, 5.4 and 5.5 were obtained with the methodology from [23], including all `SIKE` parameter sets[1]. They show the base-2 logarithm of the classical gate count costs (G) and the corresponding depth (D) and width (W) for a specific parameterisation of a given algorithm. The algorithms considered are a direct application of Grover search, Tani's claw-finding algorithm and the classical van Oorschot-Wiener collision search algorithm (vOW). Table 5.2 shows results when the depth is restricted to either 2^{64} or 2^{96}. This corresponds to the NIST model for quantum computation. In this case, the classical vOW algorithm does not show the optimal gate count, but instead minimizes the memory (width) with the given depth restriction. Table 5.3 instead restricts the width of the algorithm to either 2^{64} or 2^{96}. Tables 5.4 and 5.5 show the gate cost optimal and the depth-width cost optimal settings. It should be noted that these parameterizations either violate a reasonable depth or width constraint.

5.3 Side-channel attacks

Side-channel analysis targets various physical phenomena that are emitted by a cryptographic implementation to reveal critical internal information of the device. Power consumption information, timing information, and electromagnetic radiation are all emitted externally as cryptographic computations are performed. Simple power analysis (SPA) analyzes a single power signature of a device, while differential

[1] Sam Jaques kindly produced these tables for us with the scripts used to generate the tables in [23].

	SIKEp434			SIKEp503			SIKEp610			SIKEp751		
Algorithm	G	D	W	G	D	W	G	D	W	G	D	W
Grover	190	64	127	226	64	162	280	64	216	352	64	288
Tani	175	63	126	210	64	161	264	64	216	336	63	288
vOW	145	64	91	162	63	109	189	63	136	225	63	173
Grover	158	96	63	194	96	98	248	96	152	320	96	224
Tani	143	95	62	178	96	97	232	96	152	304	95	224
vOW	155	95	70	173	95	88	200	95	115	236	96	151

Table 5.2: Cost estimates for algorithms to solve the computational supersingular isogeny problem on SIKE parameter sets with depth constraints. The first three lines restrict to maximal depth close to 2^{64}, the last three to 2^{96}.

	SIKEp434			SIKEp503			SIKEp610			SIKEp751		
Algorithm	G	D	W	G	D	W	G	D	W	G	D	W
Grover	159	95	64	177	113	64	204	140	64	240	176	64
Tani	144	94	64	162	112	65	188	140	64	224	175	64
vOW	158	104	64	185	131	64	225	172	64	279	226	64
Grover	175	79	96	193	97	96	220	124	96	256	160	96
Tani	160	78	96	178	96	97	204	124	96	240	159	96
vOW	142	56	96	169	83	96	209	124	96	263	178	96

Table 5.3: Cost estimates for algorithms to solve the computational supersingular isogeny problem on SIKE parameter sets with width constraints. The first three lines restrict to maximal width close to 2^{64}, the last three to 2^{96}.

	SIKEp434			SIKEp503			SIKEp610			SIKEp751		
Algorithm	G	D	W	G	D	W	G	D	W	G	D	W
Grover	132	122	10	150	140	10	177	167	10	213	202	11
Tani	124	114	25	142	132	25	169	159	25	205	194	27
vOW	132	14	128	150	15	145	177	14	173	213	16	208

Table 5.4: Cost estimates for algorithms to solve the computational supersingular isogeny problem on SIKE parameter sets optimizing G-cost.

	`SIKEp434`			`SIKEp503`			`SIKEp610`			`SIKEp751`		
Algorithm	G	D	W	G	D	W	G	D	W	G	D	W
Grover	132	122	10	150	140	10	177	167	10	213	202	11
Tani	131	122	10	149	139	10	177	166	10	213	202	11
VW	132	14	128	150	15	145	177	14	173	213	16	208

Table 5.5: Cost estimates for algorithms to solve the computational supersingular isogeny problem on `SIKE` parameter sets optimizing DW-cost.

power analysis (DPA) statistically analyzes many power runs of a device. Timing analysis targets timing information of various portions of the computation. Electromagnetic radiation can be seen as an extension of power analysis attacks by analyzing electromagnetic emissions instead of power.

In general, isogeny-based cryptography comes down to two computations: generation of a secret kernel and computing a large-degree isogeny over that kernel. In schemes like SIKE, the secret kernel is found by computing a double-point multiplication over a torsion basis. Thus, there are 2 general approaches an attacker can exploit to attack the security of the cryptosystem via side-channel analysis:

1. Reveal parts of the hidden kernel point,

2. Reveal secret isogeny walks during the isogeny computation.

Regarding the first approach, a double-point multiplication over a torsion basis is used to compute the hidden kernel. This computation shares many similarities with traditional elliptic curve cryptography. Accordingly, existing techniques for elliptic curve cryptography side-channel attacks can be applied to reveal information about this ladder and what kind of hidden kernel point was generated. Further, invalid parameters may be injected by providing an invalid torsion basis or invalid curve, thus limiting the possible number of valid kernel points of full isogeny order.

For the second approach, the hidden kernel point is used to perform various walks of small degree on an isogeny graph. If an attacker can identify specific walks used during this computation, then the attacker has a subset of the isogeny computation between two distant isomorphism classes and the security of SIKE is weakened. As this part of the computation has no analogue in traditional ECC, this category of side-channels attacks is being actively investigated by the research community.

In targeting these parts of the SIKE cryptosystem, an attacker no doubt has access to a wide range of power, timing, fault, and various other side-channels. Constant-time implementations using a constant set of operations has been shown to be a good countermeasure against SPA and timing attacks. Higher level differential power analysis attacks and fault injection attacks are much harder to defend against. Papers and publications describing side-channel attacks against SIKE and countermeasures include [18, 28, 29, 49]. We remark that most, if not all, post-quantum cryptosystems are vulnerable to side-channel attacks to some extent, and research in this area is extremely active.

Chapter 6

Advantages and Limitations

Despite their relatively short lifespans as foundations for cryptographic key exchange, problems relating to the computation of isogenies between elliptic curves defined over finite fields have been studied since at least as far back as the mid 1990's [26]. Although there exists a subexponential quantum algorithm [6] that can solve the analogue of SIDH that uses ordinary curves (this scheme was first suggested by Couveignes in 1997 [10] and later published by Rostovtsev and Stolbunov in 2006 [39, 45]), the best classical attacks against this protocol remain exponential. Moreover, given that problems for which there exist subexponential classical algorithms (e.g., RSA) are widely used and considered secure in the classical sense, even the existence of a quantum subexponential attack against the ordinary analogue of SIDH does not necessarily preclude its consideration in the quantum setting. Nevertheless, the supersingular case is currently preferred because it is more efficient, and because the best known classical and quantum algorithms for solving well-formed instances of the SIDH problem (see §4.3.1) are exponential. Computational number theorists therefore have reasonable evidence that the underlying problems are hard. Furthermore, if the best algorithms for SIDH remain the claw-finding algorithms, then we have known lower bounds on the respective classical and quantum complexities in the asymptotic case (cf. §4.1). Moreover, two recent works [1, 9] have both shown that implementations of the vOW algorithm in the context of the computational supersingular isogeny problem essentially perform in exact accordance with the theoretical predictions made by van Oorschot and Wiener [50]. Coupled with the recent quantum cryptanalysis performed by Jaques and Schanck [23], these works provide confidence in the concrete security of the parameterizations in the present proposal.

We note that the number of isogeny classes that can be used at any given security level are plentiful; even when restricting to the case of 2^{e_2}- and 3^{e_3}-isogenies, there are many primes of the form $p = f \cdot 2^{e_2} 3^{e_3} - 1$ (where f is a small cofactor [11]) with $2^{e_2} \approx 3^{e_3}$ that can be used for secure SIKE instantiations. Fixing $f = 1$ still yields many choices at any given security level, and the `SIKEp434`, `SIKEp503`, `SIKEp610`, and `SIKEp751` parameters were selected from these candidates according to the criteria discussed in §1.6.

Following decades of intense research on traditional elliptic curve cryptography, one advantage of isogeny-based schemes is that there already exists a wide-reaching global expertise in the secure implementation of curve-based cryptography. History has shown that the most serious reported real-world attacks against public-key cryptography have not been a result of algorithms that break the underlying mathematical problems, but rather a result of attacks that exploit poor implementations (e.g., side-channel attacks). Isogeny-based cryptography essentially inherits all of its operations from elliptic curve cryptography, so any implementer that is experienced with producing secure code for real-world ECC should find little or no trouble developing secure code for the scheme in this proposal.

Compared to other primitives that are conjectured to offer reasonable quantum security, the main practical advantage of SIKE is its relatively small key sizes. The uncompressed public key (resp. ciphertext) sizes corresponding to `SIKEp434` and `SIKEp503` are 330 and 378 (resp. 378 and 402) bytes, which is comparable to the 384-byte (3072-bit) modulus that is conjectured to offer 128 bits of classical security. Likewise, `SIKEp610` public keys (resp. ciphertexts) are 462 (resp. 486) bytes, and the largest of our parameter sets, `SIKEp751`, has 564-byte uncompressed public keys and 596-byte ciphertexts. One main update made to this version of the proposal is the inclusion of a protocol specification that offers further public key and ciphertext compression; this reduces all of the above numbers to roughly 60% of their former size, for performance overheads ranging from 139% to 161% during public key generation, 66% to 90% during encapsulation, and 59% to 68% during decapsulation (Tables 2.1 and 2.2), and a significant cost in static library size (Table 2.3).

The ease of partnering supersingular isogeny-based public-key cryptography with strong classical elliptic curve cryptography (ECC) is discussed in [8, §8]. In particular, a sound SIKE software library contains all of the ingredients necessary to securely implement elliptic curve Diffie-Hellman in a hybrid key exchange scheme, with a minimal amount of additional coding effort required. As in the case of high-performance ECC implementations, a large portion of the code is dedicated to tailored arithmetic in the underlying finite field. Strong, well-chosen Montgomery curves (like those recently chosen for adoption in TLS [33]) can be defined over any large enough prime field, and (beyond the field arithmetic) are essentially implemented in the same way. Even when defined over the 434 and 503-bit prime fields corresponding to `SIKEp434` and `SIKEp503`, this technique gives rise to respective SIKE+ECDH hybrids that offer around 216 and 251 bits of ECDLP security, the latter being comparable to the NISTp521 curve. The corresponding (uncompressed) SIKE+ECDH public keys inflate by a factor of no more than 1.17x relative to SIKE alone, and the benchmarks reported in [8, Table 3] show that the performance slowdown is even less than this factor.

Relative to other post-quantum candidates, the main practical limitation of SIKE currently lies in its performance. Although the benchmarks in §2.2 show that, especially for the `SIKEp434` and `SIKEp503` parameters, SIKE is already practical enough for many applications, it is still at least an order of magnitude slower than some popular lattice- and code-based alternatives. Nevertheless, high-performance supersingular isogeny-based public-key cryptography is arguably much less developed than its counterparts, and a similar trade-off (small keys versus larger latencies) was seen in the early days of classical elliptic curve cryptography; this was before the decades of research and performance optimizations brought ECC to the high-performance alternative it is today. In addition, for many applications, such as protocols with fixed-size packets, bandwidth is a more precious commodity than computational cycles, and SIKE represents a good fit for such situations.

Bibliography

[1] Gora Adj, Daniel Cervantes-Vázquez, Jesús-Javier Chi-Domínguez, Alfred Menezes, and Francisco Rodríguez-Henríquez. On the cost of computing isogenies between supersingular elliptic curves. In Carlos Cid and Michael J. Jacobson Jr., editors, *Selected Areas in Cryptography - SAC 2018 - 25th International Conference, Calgary, AB, Canada, August 15-17, 2018, Revised Selected Papers*, volume 11349 of *Lecture Notes in Computer Science*, pages 322–343. Springer, 2018. 46, 49, 50, 54

[2] Reza Azarderakhsh, David Jao, Kassem Kalach, Brian Koziel, and Christopher Leonardi. Key compression for isogeny-based cryptosystems. In Keita Emura, Goichiro Hanaoka, and Rui Zhang, editors, *Proceedings of the 3rd ACM International Workshop on ASIA Public-Key Cryptography, AsiaPKC@AsiaCCS, Xi'an, China, May 30 – June 03, 2016*, pages 1–10. ACM, 2016. 11, 19

[3] Mihir Bellare, Anand Desai, David Pointcheval, and Phillip Rogaway. Relations among notions of security for public-key encryption schemes. In *Advances in Cryptology — CRYPTO '98, 18th Annual International Cryptology Conference, Santa Barbara, California, USA, August 23-27, 1998, Proceedings*, pages 26–45, 1998. 47

[4] Joppe W. Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, and Damien Stehlé. CRYSTALS – Kyber: a CCA-secure module-lattice-based KEM. Cryptology ePrint Archive, Report 2017/634, 2017. http://eprint.iacr.org/2017/634. 17

[5] Denis Charles, Kristin Lauter, and Eyal Goren. Cryptographic hash functions from expander graphs. *J. Cryptology*, 22(1):93–113, 2009. 47

[6] Andrew M. Childs, David Jao, and Vladimir Soukharev. Constructing elliptic curve isogenies in quantum subexponential time. *J. Mathematical Cryptology*, 8(1):1–29, 2014. 54

[7] Craig Costello, David Jao, Patrick Longa, Michael Naehrig, Joost Renes, and David Urbanik. Efficient compression of SIDH public keys. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology — EUROCRYPT 2017 — 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 – May 4, 2017, Proceedings, Part I*, volume 10210 of *Lecture Notes in Computer Science*, pages 679–706, 2017. 1, 11, 19

[8] Craig Costello, Patrick Longa, and Michael Naehrig. Efficient algorithms for supersingular isogeny Diffie-Hellman. In Matt Robshaw and Jonathan Katz, editors, *Advances in Cryptology — CRYPTO 2016 — 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part I*, volume 9814 of *Lecture Notes in Computer Science*, pages 572–601. Springer, 2016. 1, 7, 12, 13, 55

[9] Craig Costello, Patrick Longa, Michael Naehrig, Joost Renes, and Fernando Virdia. Improved classical cryptanalysis of the computational supersingular isogeny problem. Cryptology ePrint Archive, Report 2019/298, 2019. https://eprint.iacr.org/2019/298. 12, 49, 50, 54

[10] Jean-Marc Couveignes. Hard homogeneous spaces. Cryptology ePrint Archive, Report 2006/291, 2006. http://eprint.iacr.org/2006/291. 54

[11] Luca De Feo, David Jao, and Jérôme Plût. Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. *J. Mathematical Cryptology*, 8(3):209–247, 2014. 1, 13, 15, 17, 47, 54

[12] Christina Delfs and Steven D. Galbraith. Computing isogenies between supersingular elliptic curves over $\mathbb{F}_p$. *Designs, Codes and Cryptography*, 78(2):425–440, Feb 2016. 45

[13] Morris J. Dworkin. SHA-3 standard: Permutation-based hash and extendable-output functions. *Federal Inf. Process. Stds. (NIST FIPS) âĂŞ 202*, 2015. Available at https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf. 19

[14] Armando Faz-Hernández, Julio López, Eduardo Ochoa Ochoa-Jiménez, and Francisco Rodríguez-Henríquez. A faster software implementation of the supersingular isogeny Diffie-Hellman key exchange protocol. *IEEE Transactions on Computers*, preprint(99):1–1, 2017. 35, 61

[15] Steven D. Galbraith. Constructing isogenies between elliptic curves over finite fields. *LMS Journal of Computation and Mathematics*, 2:118–138, 1999. 45

[16] Steven D. Galbraith. *Mathematics of Public Key Cryptography*. Cambridge University Press, 2012. 47

[17] Steven D. Galbraith, Christophe Petit, Barak Shani, and Yan Bo Ti. On the security of supersingular isogeny cryptosystems. In Jung-Hee Cheon and Tsuyoshi Takagi, editors, *Advances in Cryptology — ASIACRYPT 2016 — 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part I*, volume 10031 of *Lecture Notes in Computer Science*, pages 63–91, 2016. 45, 46

[18] Alexandre Gélin and Benjamin Wesolowski. Loop-abort faults on supersingular isogeny cryptosystems. In Tanja Lange and Tsuyoshi Takagi, editors, *Post-Quantum Cryptography : 8th International Workshop, PQCrypto 2017, Utrecht, The Netherlands, June 26-28, 2017, Proceedings*, pages 93–106, Cham, 2017. Springer International Publishing. 46, 53

[19] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. A Modular Analysis of the Fujisaki-Okamoto Transformation. Cryptology ePrint Archive, Report 2017/604, 2017. 17, 48

[20] Amir Jalali, Reza Azarderakhsh, Mehran Mozaffari Kermani, Matthew Campagna, and David Jao. ARMv8 SIKE: Optimized supersingular isogeny key encapsulation on ARMv8 processors. *IEEE Trans. on Circuits and Systems*, 66-I(11):4209–4218, 2019. 39

[21] Amir Jalali, Reza Azarderakhsh, Mehran Mozaffari Kermani, and David Jao. Supersingular isogeny Diffie-Hellman key exchange on 64-bit ARM. *IEEE Trans. Dependable Secur. Comput.*, 16(5):902–912, 2019. 39

[22] David Jao and Luca De Feo. Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. In Bo-Yin Yang, editor, *Post-Quantum Cryptography — 4th International Workshop, PQCrypto 2011, Taipei, Taiwan, November 29 – December 2, 2011. Proceedings*, volume 7071 of *Lecture Notes in Computer Science*, pages 19–34. Springer, 2011. 1

[23] Samuel Jaques and John M. Schanck. Quantum cryptanalysis in the RAM model: Claw-finding attacks on SIKE. *IACR Cryptology ePrint Archive*, 2019:103, 2019. 46, 49, 50, 51, 54

[24] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography, Second Edition*. CRC Press, 2014. 47

[25] Eike Kiltz and John Malone-Lee. A General Construction of IND-CCA2 Secure Public Key Encryption. In *Cryptography and Coding, 9th IMA International Conference, Cirencester, UK, December 16-18, 2003, Proceedings*, pages 152–166, 2003. 47

[26] David R. Kohel. *Endomorphism rings of elliptic curves over finite fields*. PhD thesis, University of California, Berkeley, 1996. 54

[27] Brian Koziel, A-Bon Ackie, Rami El Khatib, Reza Azarderakhsh, and Mehran Mozaffari-Kermani. SIKE'd Up: Fast and Secure Hardware Architectures for Supersingular Isogeny Key Encapsulation. Cryptology ePrint Archive, Report 2019/711, 2019. https://eprint.iacr.org/2019/711. 42

[28] Brian Koziel, Reza Azarderakhsh, and David Jao. Side-channel attacks on quantum-resistant Supersingular Isogeny Diffie-Hellman. In Carlisle Adams and Jan Camenisch, editors, *Selected Areas in Cryptography: 24th International Conference, SAC 2017, Ottawa, ON, Canada, August 16-18, 2017, Revised Selected Papers*, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg. To appear. 53

[29] Brian Koziel, Reza Azarderakhsh, and David Jao. An exposure model for supersingular isogeny Diffie-Hellman key exchange. In Nigel P. Smart, editor, *Topics in Cryptology — CT-RSA 2018*, pages 452–469, Cham, 2018. Springer International Publishing. 53

[30] Brian Koziel, Reza Azarderakhsh, Mehran Mozaffari Kermani, and David Jao. Post-quantum cryptography on FPGA based on isogenies on elliptic curves. *IEEE Trans. on Circuits and Systems*, 64-I(1):86–99, 2017. 1, 42

[31] Brian Koziel, Reza Azarderakhsh, and Mehran Mozaffari-Kermani. Fast Hardware Architectures for Supersingular Isogeny Diffie-Hellman Key Exchange on FPGA. In *Progress in Cryptology – INDOCRYPT 2016: 17th International Conference on Cryptology in India*, pages 191–206, 2016. 42

[32] Brian Koziel, Reza Azarderakhsh, and Mehran Mozaffari-Kermani. A High-Performance and Scalable Hardware Architecture for Isogeny-Based Cryptography. *IEEE Transactions on Computers*, 67(11):1594–1609, Nov 2018. 42

[33] Adam Langley, Mike Hamburg, and Sean Turner. Elliptic curves for security. Internet Research Task Force (IRTF) RFC, 2016. https://tools.ietf.org/html/rfc7748. 55

[34] Victor S. Miller. The Weil pairing, and its efficient calculation. *Journal of Cryptology*, 17(4):235–261, 2004. 4

[35] Michael Naehrig and Joost Renes. Dual isogenies and their application to public-key compression for isogeny-based cryptography. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 243–272. Springer, 2019. 11, 13, 19, 38, 85, 86, 91, 92

[36] Geovandro Pereira, Javad Doliskani, and David Jao. x-only point addition formula and faster torsion basis generation in compressed SIKE. Cryptology ePrint Archive, Report 2020/431, 2020. http://eprint.iacr.org/2020/431. 11, 19, 38, 85, 86, 92, 93

[37] Christophe Petit. Faster algorithms for isogeny problems using torsion point images. Cryptology ePrint Archive, Report 2017/571, 2017. http://eprint.iacr.org/2017/571. 47

[38] Christophe Petit and Kristin Lauter. Hard and easy problems for supersingular isogeny graphs. Cryptology ePrint Archive, Report 2017/962, 2017. http://eprint.iacr.org/2017/962. 47

[39] Alexander Rostovtsev and Anton Stolbunov. Public-key cryptosystem based on isogenies. Cryptology ePrint Archive, Report 2006/145, 2006. http://eprint.iacr.org/. 54

[40] H. Seo, P. Sanal, A. Jalali, and R. Azarderakhsh. Optimized implementation of SIKE Round 2 on 64-bit ARM Cortex-A processors. *IEEE Transactions on Circuits and Systems I: Regular Papers*, pages 1–13, 2020. 39

[41] Hwajeong Seo, Mila Anastasova, Amir Jalali, and Reza Azarderakhsh. Supersingular isogeny key encapsulation (SIKE) round 2 on ARM Cortex-M4. Cryptology ePrint Archive, Report 2020/410, 2020. https://eprint.iacr.org/2020/410. 40

[42] Hwajeong Seo, Amir Jalali, and Reza Azarderakhsh. SIKE round 2 speed record on ARM Cortex-M4. In Yi Mu, Robert H. Deng, and Xinyi Huang, editors, *Cryptology and Network Security - 18th International Conference, CANS 2019, Fuzhou, China, October 25-27, 2019, Proceedings*, volume 11829 of *Lecture Notes in Computer Science*, pages 39–60. Springer, 2019. 40

[43] Victor Shoup. Sequences of Games: A Tool for Taming Complexity in Security Proofs. *IACR Cryptology ePrint Archive*, 2004:332, 2004. 47

[44] Anton Stolbunov. Reductionist Security Arguments for Public-Key Cryptographic Schemes Based on Group Action. NISK-2009 conference, 2009. 47

[45] Anton Stolbunov. Constructing public-key cryptographic schemes based on class group action on a set of isogenous elliptic curves. *Adv. Math. Commun.*, 4(2):215–235, 2010. 54

[46] Andrew Sutherland. Structure computation and discrete logarithms in finite abelian p-groups. *Mathematics of Computation*, 80(273):477–500, 2011. 19

[47] Seiichiro Tani. Claw finding algorithms using quantum walk. *Theor. Comput. Sci.*, 410(50):5285–5297, 2009. 46, 50

[48] The National Institute of Standards and Technology (NIST). Submission requirements and evaluation criteria for the post-quantum cryptography standardization process, December, 2016. 50, 51

[49] Yan Bo Ti. Fault attack on supersingular isogeny cryptosystems. In Tanja Lange and Tsuyoshi Takagi, editors, *Post-Quantum Cryptography : 8th International Workshop, PQCrypto 2017, Utrecht, The Netherlands, June 26-28, 2017, Proceedings*, pages 107–122, Cham, 2017. Springer International Publishing. 46, 53

[50] Paul C. van Oorschot and Michael J. Wiener. Parallel collision search with cryptanalytic applications. *J. Cryptology*, 12(1):1–28, 1999. 46, 49, 54

[51] Jacques Vélu. Isogénies entre courbes elliptiques. *CR Acad. Sci. Paris Sér. AB*, 273:A238–A241, 1971. 5, 13

[52] Gustavo Zanon, Marcos A. Simplício Jr., Geovandro C. C. F. Pereira, Javad Doliskani, and Paulo S. L. M. Barreto. Faster isogeny-based compressed key agreement. In Tanja Lange and Rainer Steinwandt, editors, *Post-Quantum Cryptography - 9th International Conference, PQCrypto 2018, Fort Lauderdale, FL, USA, April 9-11, 2018, Proceedings*, volume 10786 of *Lecture Notes in Computer Science*, pages 248–268. Springer, 2018. 19, 83

[53] Shengyu Zhang. Promised and Distributed Quantum Search Computing and Combinatorics. In *Proceedings of the Eleventh Annual International Conference on Computing and Combinatorics*, volume 3595 of *Lecture Notes in Computer Science*, pages 430–439, Berlin, Heidelberg, 2005. Springer Berlin / Heidelberg. 46

Appendix A

Explicit algorithms for $\texttt{isogen}_\ell$ and $\texttt{isoex}_\ell$: Optimized implementation

This section contains explicit formulas for computing the isogenies described in §1.3.6 and §1.3.7. Assuming access to all of the field operations in $\mathbb{F}_{p^2}$, Algorithms 3–24 can compute $\texttt{isogen}_\ell$ and $\texttt{isoex}_\ell$ for $\ell \in \{2, 3\}$ in their entirety for the three sets of parameters `SIKEp434`, `SIKEp503`, and `SIKEp751`. In the case of `SIKEp610`, the exponent of 2 is odd, meaning the algorithm needs to start or finish with a single 2-isogeny; for simplicity, we have presented the algorithms using 4-isogenies only, but refer to the codebase(s) in the case of `SIKEp610` for the scenario where a 2-isogeny is needed.

The notation $(X_P : Z_P)$ with $Z_P \neq 0$ is used for the projective tuple in $\mathbb{P}^1(\mathbb{F}_{p^2})$ representing the Montgomery x-coordinate $x_P = X_P/Z_P$; lower case letters are used for normalized coordinates, upper cases for projective coordinates.

Several variants of the Montgomery curve constants are used below for enhanced performance. Write E_a for the curve $E_a/\mathbb{F}_{p^2}\colon y^2 = x^3 + ax^2 + x$ and use $(A : C)$ to denote the equivalence $(A : C) \sim (a : 1)$ in $\mathbb{P}^1(\mathbb{F}_{p^2})$. Furthermore, define $(A^+_{24} : C_{24}) \sim (A + 2C : 4C)$, $(A^+_{24} : A^-_{24}) \sim (A + 2C : A - 2C)$, and $(a^+_{24} : 1) \sim (A + 2C : 4C)$.

Algorithm 8, which computes the *three point ladder*, uses the recent and improved algorithm from [14].

Algorithms 19 and 20 use a *deque* (double ended queue) data structure with three defined operations: `push` adds an item on *top* of the deque, `pop` removes an item from the *top* of the deque, and `pull` removes an item from the *bottom* of the deque.

Algorithm 3: Coordinate doubling

function `xDBL`
- **Input:** $(X_P : Z_P)$ and $(A^+_{24} : C_{24})$
- **Output:** $(X_{[2]P} : Z_{[2]P})$

$t_0 \leftarrow X_P - Z_P$
$t_1 \leftarrow X_P + Z_P$
$t_0 \leftarrow t_0^2$
$t_1 \leftarrow t_1^2$
$Z_{[2]P} \leftarrow C_{24} \cdot t_0$
$X_{[2]P} \leftarrow Z_{[2]P} \cdot t_1$
$t_1 \leftarrow t_1 - t_0$
$t_0 \leftarrow A^+_{24} \cdot t_1$
$Z_{[2]P} \leftarrow Z_{[2]P} + t_0$
$Z_{[2]P} \leftarrow Z_{[2]P} \cdot t_1$
return $(X_{[2]P} : Z_{[2]P})$

Algorithm 4: Repeated coordinate doubling

function xDBLe

  Input: $(X_P : Z_P)$, $(A_{24}^{+} : C_{24})$, and $e \in \mathbb{Z}$

  Output: $(X_{[2^e]P} : Z_{[2^e]P})$

$(X' : Z') \leftarrow (X_P : Z_P)$

for $i = 1$ **to** e **do**

  $(X' : Z') \leftarrow \mathtt{xDBL}\big((X' : Z'), (A_{24}^{+} : C_{24})\big)$ `// Alg. 3`

return $(X' : Z')$

Algorithm 5: Combined coordinate doubling and differential addition

function xDBLADD

  Input: $(X_P : Z_P)$, $(X_Q : Z_Q)$, $(X_{Q-P} : Z_{Q-P})$, and $(a_{24}^{+} : 1) \sim (A + 2C : 4C)$

  Output: $(X_{[2]P} : Z_{[2]P})$, $(X_{P+Q} : Z_{P+Q})$

$t_0 \leftarrow X_P + Z_P$

$t_1 \leftarrow X_P - Z_P$

$X_{[2]P} \leftarrow t_0^2$

$t_2 \leftarrow X_Q - Z_Q$

$X_{P+Q} \leftarrow X_Q + Z_Q$

$t_0 \leftarrow t_0 \cdot t_2$

$Z_{[2]P} \leftarrow t_1^2$

$t_1 \leftarrow t_1 \cdot X_{P+Q}$

$t_2 \leftarrow X_{[2]P} - Z_{[2]P}$

$X_{[2]P} \leftarrow X_{[2]P} \cdot Z_{[2]P}$

$X_{P+Q} \leftarrow a_{24}^{+} \cdot t_2$

$Z_{P+Q} \leftarrow t_0 - t_1$

$Z_{[2]P} \leftarrow X_{P+Q} + Z_{[2]P}$

$X_{P+Q} \leftarrow t_0 + t_1$

$Z_{[2]P} \leftarrow Z_{[2]P} \cdot t_2$

$Z_{P+Q} \leftarrow Z_{P+Q}^2$

$X_{P+Q} \leftarrow X_{P+Q}^2$

$Z_{P+Q} \leftarrow X_{Q-P} \cdot Z_{P+Q}$

$X_{P+Q} \leftarrow Z_{Q-P} \cdot X_{P+Q}$

return $\{(X_{[2]P} : Z_{[2]P}),$
  $(X_{P+Q} : Z_{P+Q})\}$

Algorithm 6: Coordinate tripling

function xTPL

  Input: $(X_P : Z_P)$ and $(A_{24}^{+} : A_{24}^{-})$

  Output: $(X_{[3]P} : Z_{[3]P})$

$t_0 \leftarrow X_P - Z_P$

$t_2 \leftarrow t_0^2$

$t_1 \leftarrow X_P + Z_P$

$t_3 \leftarrow t_1^2$

$t_4 \leftarrow t_1 + t_0$

$t_0 \leftarrow t_1 - t_0$

$t_1 \leftarrow t_4^2$

$t_1 \leftarrow t_1 - t_3$

$t_1 \leftarrow t_1 - t_2$

$t_5 \leftarrow t_3 \cdot A_{24}^{+}$

$t_3 \leftarrow t_5 \cdot t_3$

$t_6 \leftarrow t_2 \cdot A_{24}^{-}$

$t_2 \leftarrow t_2 \cdot t_6$

$t_3 \leftarrow t_2 - t_3$

$t_2 \leftarrow t_5 - t_6$

$t_1 \leftarrow t_2 \cdot t_1$

$t_2 \leftarrow t_3 + t_1$

$t_2 \leftarrow t_2^2$

$X_{[3]P} \leftarrow t_2 \cdot t_4$

$t_1 \leftarrow t_3 - t_1$

$t_1 \leftarrow t_1^2$

$Z_{[3]P} \leftarrow t_1 \cdot t_0$

return $(X_{[3]P} : Z_{[3]P})$

Algorithm 7: Repeated coordinate tripling

function `xTPLe`

Input: $(X_P : Z_P)$, $(A_{24}^+ : A_{24}^-)$, and $e \in \mathbb{Z}^+$

Output: $(X_{[3^e]P} : Z_{[3^e]P})$

$(X' : Z') \leftarrow (X_P : Z_P)$

for $i = 1$ **to** e **do**

  $(X' : Z') \leftarrow \mathtt{xTPL}\big((X' : Z'), (A_{24}^+ : A_{24}^-)\big)$ `// Alg. 6`

return $(X' : Z')$

Algorithm 8: Three point ladder

function `Ladder3pt`

Input: $m = (m_{\ell-1}, \ldots, m_0)_2 \in \mathbb{Z}$, (x_P, x_Q, x_{Q-P}), and $(A : 1)$

Output: $(X_{P+[m]Q} : Z_{P+[m]Q})$

$((X_0 : Z_0), (X_1 : Z_1), (X_2 : Z_2)) \leftarrow ((x_Q : 1), (x_P : 1), (x_{Q-P} : 1))$

$a_{24}^+ \leftarrow (A + 2)/4$

for $i = 0$ **to** $\ell - 1$ **do**

  if $m_i = 1$ **then**

    $((X_0 : Z_0), (X_1 : Z_1)) \leftarrow \mathtt{xDBLADD}((X_0 : Z_0), (X_1 : Z_1), (X_2 : Z_2), (a_{24}^+ : 1))$ `// Alg. 5`

  else

    $((X_0 : Z_0), (X_2 : Z_2)) \leftarrow \mathtt{xDBLADD}((X_0 : Z_0), (X_2 : Z_2), (X_1 : Z_1), (a_{24}^+ : 1))$ `// Alg. 5`

return $(X_1 : Z_1)$

Algorithm 9: Montgomery j-invariant computation

function `jInvariant`

Input: $(A : C)$

Output: j-invariant $j(E_{A/C}) \in \mathbb{F}_{p^2}$

$j \leftarrow A^2$

$t_1 \leftarrow C^2$

$t_0 \leftarrow t_1 + t_1$

$t_0 \leftarrow j - t_0$

$t_0 \leftarrow t_0 - t_1$

$j \leftarrow t_0 - t_1$

$t_1 \leftarrow t_1^2$

$j \leftarrow j \cdot t_1$

$t_0 \leftarrow t_0 + t_0$

$t_0 \leftarrow t_0 + t_0$

$t_1 \leftarrow t_0^2$

$t_0 \leftarrow t_0 \cdot t_1$

$t_0 \leftarrow t_0 + t_0$

$t_0 \leftarrow t_0 + t_0$

$j \leftarrow 1/j$

$j \leftarrow t_0 \cdot j$

return j

Algorithm 10: Recovering the Montgomery curve coefficient

function `get_A`

Input: x_P, x_Q and x_{Q-P} corresponding to points on $E_A : y^2 = x^3 + Ax^2 + x$

Output: $A \in \mathbb{F}_{p^2}$

$t_1 \leftarrow x_P + x_Q$

$t_0 \leftarrow x_P \cdot x_Q$

$A \leftarrow x_{Q-P} \cdot t_1$

$A \leftarrow A + t_0$

$t_0 \leftarrow t_0 \cdot x_{Q-P}$

$A \leftarrow A - 1$

$t_0 \leftarrow t_0 + t_0$

$t_1 \leftarrow t_1 + x_{Q-P}$

$t_0 \leftarrow t_0 + t_0$

$A \leftarrow A^2$

$t_0 \leftarrow 1/t_0$

$A \leftarrow A \cdot t_0$

$A \leftarrow A - t_1$

return A

Algorithm 11: Computing the 2-isogenous curve

function `2_iso_curve`

Input: $(X_{P_2} : Z_{P_2})$, where P_2 has exact order 2 on $E_{A/C}$

Output: $(A_{24}^+ : C_{24}) \sim (A' + 2C' : 4C')$ corresponding to $E_{A'/C'} = E_{A/C}/\langle P_2 \rangle$

$A_{24}^+ \leftarrow X_{P_2}^2$,

$C_{24} \leftarrow Z_{P_2}^2$,

$A_{24}^+ \leftarrow C_{24}^+ - A_{24}$,

return A_{24}^+, C_{24}

Algorithm 12: Evaluating a 2-isogeny at a point

function `2_iso_eval`

Input: $(X_{P_2} : Z_{P_2})$, where P_2 has exact order 2 on $E_{A/C}$, and $(X_Q : Z_Q)$ where $Q \in E_{A/C}$

Output: $(X_{Q'} : Z_{Q'})$ corresponding to $Q' \in E_{A'/C'}$, where $E_{A'/C'}$ is the curve 2-isogenous to $E_{A/C}$ output from `2_iso_curve`

$t_0 \leftarrow X_{P_2} + Z_{P_2}$,

$t_1 \leftarrow X_{P_2} - Z_{P_2}$,

$t_2 \leftarrow X_Q + Z_Q$,

$t_3 \leftarrow X_Q - Z_Q$,

$t_0 \leftarrow t_0 \cdot t_3$,

$t_1 \leftarrow t_1 \cdot t_2$,

$t_2 \leftarrow t_0 + t_1$,

$t_3 \leftarrow t_0 - t_1$,

$X_{Q'} \leftarrow X_Q \cdot t_2$,

$Z_{Q'} \leftarrow Z_Q \cdot t_3$,

return $(X_{Q'} : Z_{Q'})$

Algorithm 13: Computing the 4-isogenous curve

function `4_iso_curve`

Input: $(X_{P_4} : Z_{P_4})$, where P_4 has exact order 4 on $E_{A/C}$

Output: $(A_{24}^+ : C_{24}) \sim (A' + 2C' : 4C')$ corresponding to $E_{A'/C'} = E_{A/C}/\langle P_4 \rangle$, and constants $(K_1, K_2, K_3) \in (\mathbb{F}_{p^2})^3$

$K_2 \leftarrow X_{P_4} - Z_{P_4}$,

$K_3 \leftarrow X_{P_4} + Z_{P_4}$,

$K_1 \leftarrow Z_{P_4}^2$,

$K_1 \leftarrow K_1 + K_1$,

$C_{24} \leftarrow K_1^2$,

$K_1 \leftarrow K_1 + K_1$,

$A_{24}^+ \leftarrow X_{P_4}^2$,

$A_{24}^+ \leftarrow A_{24}^+ + A_{24}^+$,

$A_{24}^+ \leftarrow (A_{24}^+)^2$,

return A_{24}^+, C_{24}, (K_1, K_2, K_3)

Algorithm 14: Evaluating a 4-isogeny at a point

function `4_iso_eval`

Input: Constants $(K_1, K_2, K_3) \in (\mathbb{F}_{p^2})^3$ from `4_iso_curve`, and $(X_Q : Z_Q)$ where $Q \in E_{A/C}$

Output: $(X_{Q'} : Z_{Q'})$ corresponding to $Q' \in E_{A'/C'}$, where $E_{A'/C'}$ is the curve 4-isogenous to $E_{A/C}$ output from `4_iso_curve`

$t_0 \leftarrow X_Q + Z_Q$,
$t_1 \leftarrow X_Q - Z_Q$,
$X_Q \leftarrow t_0 \cdot K_2$,
$Z_Q \leftarrow t_1 \cdot K_3$,
$t_0 \leftarrow t_0 \cdot t_1$,
$t_0 \leftarrow t_0 \cdot K_1$,
$t_1 \leftarrow X_Q + Z_Q$,
$Z_Q \leftarrow X_Q - Z_Q$,
$t_1 \leftarrow t_1^2$,
$Z_Q \leftarrow Z_Q^2$,
$X_Q \leftarrow t_0 + t_1$,
$t_0 \leftarrow Z_Q - t_0$,
$X_{Q'} \leftarrow X_Q \cdot t_1$,
$Z_{Q'} \leftarrow Z_Q \cdot t_0$,
return $(X_{Q'} : Z_{Q'})$

Algorithm 15: Computing the 3-isogenous curve

function `3_iso_curve`

Input: $(X_{P_3} : Z_{P_3})$, where P_3 has exact order 3 on $E_{A/C}$

Output: Curve constant $(A_{24}^+ : A_{24}^-) \sim (A' + 2C' : A' - 2C')$ corresponding to $E_{A'/C'} = E_{A/C}/\langle P_3 \rangle$, and constants $(K_1, K_2) \in (\mathbb{F}_{p^2})^2$

$K_1 \leftarrow X_{P_3} - Z_{P_3}$,
$t_0 \leftarrow K_1^2$,
$K_2 \leftarrow X_{P_3} + Z_{P_3}$,
$t_1 \leftarrow K_2^2$,
$t_2 \leftarrow t_0 + t_1$,
$t_3 \leftarrow K_1 + K_2$,
$t_3 \leftarrow t_3^2$,
$t_3 \leftarrow t_3 - t_2$,
$t_2 \leftarrow t_1 + t_3$,
$t_3 \leftarrow t_3 + t_0$,
$t_4 \leftarrow t_3 + t_0$,
$t_4 \leftarrow t_4 + t_4$,
$t_4 \leftarrow t_1 + t_4$,
$A_{24}^- \leftarrow t_2 \cdot t_4$,
$t_4 \leftarrow t_1 + t_2$,
$t_4 \leftarrow t_4 + t_4$,
$t_4 \leftarrow t_0 + t_4$,
$A_{24}^+ \leftarrow t_3 \cdot t_4$,
return $(A_{24}^+ : A_{24}^-)$, $(K_1, K_2) \in (\mathbb{F}_{p^2})^2$

Algorithm 16: Evaluating a 3-isogeny at a point

function `3_iso_eval`

Input: Constants $(K_1, K_2) \in (\mathbb{F}_{p^2})^3$ output from `3_iso_curve` together with $(X_Q : Z_Q)$ corresponding to $Q \in E_{A/C}$

Output: $(X_{Q'} : Z_{Q'})$ corresponding to $Q' \in E_{A'/C'}$, where $E_{A'/C'}$ is 3-isogenous to $E_{A/C}$

$t_0 \leftarrow X_Q + Z_Q$,
$t_1 \leftarrow X_Q - Z_Q$,
$t_0 \leftarrow K_1 \cdot t_0$,
$t_1 \leftarrow K_2 \cdot t_1$,
$t_2 \leftarrow t_0 + t_1$,
$t_0 \leftarrow t_1 - t_0$,
$t_2 \leftarrow t_2^2$,
$t_0 \leftarrow t_0^2$,
$X_{Q'} \leftarrow X_Q \cdot t_2$,
$Z_{Q'} \leftarrow Z_Q \cdot t_0$.
return $(X_{Q'} : Z_{Q'})$

Algorithm 17: Computing and evaluating a 2^e-isogeny, simple version

function `2_e_iso`

Static parameters: Integer $\mathtt{e}_2$ from the public parameters

Input: Constants $(A_{24}^{+} : C_{24})$ corresponding to a curve $E_{A/C}$, $(X_S : Z_S)$ where S has exact order $2^{\mathtt{e}_2}$ on $E_{A/C}$

Optional input: $(X_1 : Z_1)$, $(X_2 : Z_2)$ and $(X_3 : Z_3)$ on $E_{A/C}$

Output: $(A_{24}^{+\prime} : C_{24}')$ corresponding to the curve $E_{A'/C'} = E/\langle S \rangle$

Optional output: $(X_1' : Z_1')$, $(X_2' : Z_2')$ and $(X_3' : Z_3')$ on $E_{A'/C'}$

for $e = \mathtt{e}_2 - 2$ **downto** 0 **by** -2 **do**

  $(X_T : Z_T) \leftarrow$ `xDBLe` $\big((X_S : Z_S), (A_{24}^{+} : C_{24}), e\big)$ `// Alg. 4`

  $\big((A_{24}^{+} : C_{24}), (K_1, K_2, K_3)\big) \leftarrow$ `4_iso_curve` $((X_T : Z_T))$ `// Alg. 13`

  if $e \neq 0$ **then**

    $(X_S : Z_S) \leftarrow$ `4_iso_eval` $((K_1, K_2, K_3), (X_S : Z_S))$ `// Alg. 14`

  for $(X_j : Z_j)$ **in** *optional input* **do**

    $(X_j : Z_j) \leftarrow$ `4_iso_eval` $\big((K_1, K_2, K_3), (X_j : Z_j)\big)$ `// Alg. 14`

return $(A_{24}^{+} : C_{24}), [(X_1 : Z_1), (X_2 : Z_2), (X_3 : Z_3)]$

Algorithm 18: Computing and evaluating a 3^e-isogeny, simple version

function `3_e_iso`

Static parameters: Integer $\mathtt{e}_3$ from the public parameters

Input: Constants $(A_{24}^+ : A_{24}^-)$ corresponding to a curve $E_{A/C}$, $(X_S : Z_S)$ where S has exact order $3^{\mathtt{e}_3}$ on $E_{A/C}$

Optional input: $(X_1 : Z_1)$, $(X_2 : Z_2)$ and $(X_3 : Z_3)$ on $E_{A/C}$

Output: $(A_{24}^{+}{}' : A_{24}^{-}{}')$ corresponding to the curve $E_{A'/C'} = E/\langle S \rangle$

Optional output: $(X_1' : Z_1')$, $(X_2' : Z_2')$ and $(X_3' : Z_3')$ on $E_{A'/C'}$

for $e = \mathtt{e}_3 - 1$ **downto** 0 **by** -1 **do**
    $(X_T : Z_T) \leftarrow$ `xTPLe` $\big((X_S : Z_S), (A_{24}^+ : A_{24}^-), e\big)$ `// Alg. 7`
    $\big((A_{24}^+ : A_{24}^-), (K_1, K_2)\big) \leftarrow$ `3_iso_curve` $((X_T : Z_T))$ `// Alg. 15`
    if $e \neq 0$ **then**
        $(X_S : Z_S) \leftarrow$ `3_iso_eval` $((K_1, K_2), (X_S : Z_S))$ `// Alg. 16`
    for $(X_j : Z_j)$ **in** *optional input* **do**
        $(X_j : Z_j) \leftarrow$ `3_iso_eval` $\big((K_1, K_2), (X_j : Z_j)\big)$ `// Alg. 16`

return $(A_{24}^+ : A_{24}^-), [(X_1 : Z_1), (X_2 : Z_2), (X_3 : Z_3)]$

Algorithm 19: Computing and evaluating a 2^e-isogeny, optimized version

function `2_e_iso`

Static parameters: Integer $\mathtt{e}_2$ from the public parameters, a *strategy* $(s_1, \ldots, s_{\mathtt{e}_2/2-1}) \in (\mathbb{N}^+)^{\mathtt{e}_2/2-1}$

Input: Constants $(A_{24}^+ : C_{24})$ corresponding to a curve $E_{A/C}$, $(X_S : Z_S)$ where S has exact order $2^{\mathtt{e}_2}$ on $E_{A/C}$

Optional input: $(X_1 : Z_1)$, $(X_2 : Z_2)$ and $(X_3 : Z_3)$ on $E_{A/C}$

Output: $(A_{24}^{+\prime} : C_{24}')$ corresponding to the curve $E_{A'/C'} = E/\langle S \rangle$

Optional output: $(X_1' : Z_1')$, $(X_2' : Z_2')$ and $(X_3' : Z_3')$ on $E_{A'/C'}$

Initialize empty deque $\mathbf{S}$
`push`$(\mathbf{S}, (\mathtt{e}_2/2, (X_S : Z_S)))$
$i \leftarrow 1$
while $\mathbf{S}$ *not empty* **do**
  $(h, (X : Z)) \leftarrow$ `pop`$(\mathbf{S})$
  if $h = 1$ **then**
    $\big((A_{24}^+ : C_{24}), (K_1, K_2, K_3)\big) \leftarrow$ `4_iso_curve` $((X : Z))$ `// Alg. 13`
    Initialize empty deque $\mathbf{S}'$
    while $\mathbf{S}$ *not empty* **do**
      $(h, (X : Z)) \leftarrow$ `pull`$(\mathbf{S})$
      $(X : Z) \leftarrow$ `4_iso_eval` $((K_1, K_2, K_3), (X : Z))$ `// Alg. 14`
      `push`$(\mathbf{S}', (h-1, (X : Z)))$
    $\mathbf{S} \leftarrow \mathbf{S}'$
    for $(X_j : Z_j)$ **in** *optional input* **do**
      $(X_j : Z_j) \leftarrow$ `4_iso_eval` $\big((K_1, K_2, K_3), (X_j : Z_j)\big)$ `// Alg. 14`
  else if $0 < s_i < h$ **then**
    `push`$(\mathbf{S}, (h, (X : Z)))$
    $(X : Z) \leftarrow$ `xDBLe` $\big((X : Z), (A_{24}^+ : C_{24}), 2 \cdot s_i\big)$ `// Alg. 4`
    `push`$(\mathbf{S}, (h - s_i, (X : Z)))$
    $i \leftarrow i + 1$
  else
    Error: Invalid strategy
return $(A_{24}^+ : C_{24}), [(X_1 : Z_1), (X_2 : Z_2), (X_3 : Z_3)]$

Algorithm 20: Computing and evaluating a 3^e-isogeny, optimized version

function `3_e_iso`

Static parameters: Integer $\mathtt{e}_3$ from the public parameters, a *strategy* $(s_1, \ldots, s_{\mathtt{e}_3-1}) \in (\mathbb{N}^+)^{\mathtt{e}_3-1}$

Input: Constants $(A_{24}^+ : A_{24}^-)$ corresponding to a curve $E_{A/C}$, $(X_S : Z_S)$ where S has exact order $3^{\mathtt{e}_3}$ on $E_{A/C}$

Optional input: $(X_1 : Z_1)$, $(X_2 : Z_2)$ and $(X_3 : Z_3)$ on $E_{A/C}$

Output: $(A_{24}^{+\prime} : A_{24}^{-\prime})$ corresponding to the curve $E_{A'/C'} = E/\langle S \rangle$

Optional output: $(X_1' : Z_1')$, $(X_2' : Z_2')$ and $(X_3' : Z_3')$ on $E_{A'/C'}$

Initialize empty deque $\mathbf{S}$
$\mathtt{push}(\mathbf{S}, (\mathtt{e}_3, (X_S : Z_S)))$
$i \leftarrow 1$
while $\mathbf{S}$ *not empty* **do**
  $(h, (X : Z)) \leftarrow \mathtt{pop}(\mathbf{S})$
  if $h = 1$ **then**
    $\big((A_{24}^+ : A_{24}^-), (K_1, K_2)\big) \leftarrow \mathtt{3\_iso\_curve}\,((X : Z))$ `// Alg. 15`
    Initialize empty deque $\mathbf{S}'$
    while $\mathbf{S}$ *not empty* **do**
      $(h, (X : Z)) \leftarrow \mathtt{pull}(\mathbf{S})$
      $(X : Z) \leftarrow \mathtt{3\_iso\_eval}\,((K_1, K_2), (X : Z))$ `// Alg. 16`
      $\mathtt{push}(\mathbf{S}', (h - 1, (X : Z)))$
    $\mathbf{S} \leftarrow \mathbf{S}'$
    for $(X_j : Z_j)$ **in** *optional input* **do**
      $(X_j : Z_j) \leftarrow \mathtt{3\_iso\_eval}\big((K_1, K_2), (X_j : Z_j)\big)$ `// Alg. 16`
  else if $0 < s_i < h$ **then**
    $\mathtt{push}(\mathbf{S}, (h, (X : Z)))$
    $(X : Z) \leftarrow \mathtt{xTPLe}\big((X : Z), (A_{24}^+ : A_{24}^-), s_i\big)$ `// Alg. 7`
    $\mathtt{push}(\mathbf{S}, (h - s_i, (X : Z)))$
    $i \leftarrow i + 1$
  else
    Error: invalid strategy
return $(A_{24}^+ : A_{24}^-), [(X_1 : Z_1), (X_2 : Z_2), (X_3 : Z_3)]$

Algorithm 21: Computing public keys in the 2-torsion

function `isogen`$_2$

Input: Secret key $\mathrm{sk}_2 \in \mathbb{Z}$ (see §1.2.6) and public parameters $\{\texttt{e2}, \texttt{e3}, \texttt{p}, \texttt{xP2}, \texttt{xQ2}, \texttt{xR2}, \texttt{xP3}, \texttt{xQ3}, \texttt{xR3}\}$ (see §1.6)

Output: Public key $pk_2 = (x_1, x_2, x_3)$ equivalent to the output of Step 4 of `isogen`$_\ell$ (see §1.3.6)

$\big((A : C), (A_{24}^+ : C_{24})\big) \leftarrow ((6 : 1), (8 : 4))$

$((X_1 : Z_1), (X_2 : Z_2), (X_3 : Z_3)) \leftarrow ((\texttt{xP3} : 1), (\texttt{xQ3} : 1), (\texttt{xR3} : 1))$

$(X_S : Z_S) \leftarrow \texttt{Ladder3pt}(\mathrm{sk}_2, (\texttt{xP2}, \texttt{xQ2}, \texttt{xR2}), (A : C))$ `// Alg. 8`

$\big((A_{24}^+ : C_{24}), (X_1 : Z_1), (X_2 : Z_2), (X_3 : Z_3)\big) \leftarrow$
$\texttt{2\_e\_iso}\big((A_{24}^+ : C_{24}), (X_S : Z_S), (X_1 : Z_1), (X_2 : Z_2), (X_3 : Z_3)\big)$ `// Alg. 17 or Alg. 19`

$((x_1 : 1), (x_2 : 1), (x_3 : 1)) \leftarrow ((X_1 : Z_1), (X_2 : Z_2), (X_3 : Z_3))$

return $pk_2 = (x_1, x_2, x_3)$ `// Encoded as in §1.2.9`

Algorithm 22: Computing public keys in the 3-torsion

function `isogen`$_3$

Input: Secret key $\mathrm{sk}_3 \in \mathbb{Z}$ (see §1.2.6) and public parameters $\{\texttt{e2}, \texttt{e3}, \texttt{p}, \texttt{xP2}, \texttt{xQ2}, \texttt{xR2}, \texttt{xP3}, \texttt{xQ3}, \texttt{xR3}\}$ (see §1.6)

Output: Public key $pk_3 = (x_1, x_2, x_3)$ equivalent to the output of Step 4 of `isogen`$_\ell$ (see §1.3.6)

$\big((A : C), (A_{24}^+ : A_{24}^-)\big) \leftarrow ((6 : 1), (8 : 4))$

$((X_1 : Z_1), (X_2 : Z_2), (X_3 : Z_3)) \leftarrow ((\texttt{xP2} : 1), (\texttt{xQ2} : 1), (\texttt{xR2} : 1))$

$(X_S : Z_S) \leftarrow \texttt{Ladder3pt}(\mathrm{sk}_3, (\texttt{xP3}, \texttt{xQ3}, \texttt{xR3}), (A : C))$ `// Alg. 8`

$\big((A_{24}^+ : A_{24}^-), (X_1 : Z_1), (X_2 : Z_2), (X_3 : Z_3)\big) \leftarrow$
$\texttt{3\_e\_iso}\big((A_{24}^+ : A_{24}^-), (X_S : Z_S), (X_1 : Z_1), (X_2 : Z_2), (X_3 : Z_3)\big)$ `// Alg. 18 or Alg. 20`

$((x_1 : 1), (x_2 : 1), (x_3 : 1)) \leftarrow ((X_1 : Z_1), (X_2 : Z_2), (X_3 : Z_3))$

return $pk_3 = (x_1, x_2, x_3)$ `// Encoded as in §1.2.9`

Algorithm 23: Establishing shared keys in the 2-torsion

function `isoex`$_2$

Input: Secret key $\mathrm{sk}_2 \in \mathbb{Z}$ (see §1.2.6), public key $pk_3 = (x_1, x_2, x_3) \in (\mathbb{F}_{p^2})^3$ (see §1.2.9), and parameter `e2` (see §1.6)

Output: A j-invariant j_2 equivalent to the output of Step 4 of `isogen`$_\ell$ (see §1.3.7)

$(A : C) \leftarrow (\mathtt{get\_A}(x_1, x_2, x_3) : 1)$ // Alg. 10

$(X_S : Z_S) \leftarrow \mathtt{Ladder3pt}(\mathrm{sk}_2, (x_1, x_2, x_3), (A : C))$ // Alg. 8

$(A_{24}^+ : C_{24}) \leftarrow (A + 2 : 4)$

$(A_{24}^+ : C_{24}) \leftarrow \mathtt{2\_e\_iso}\big((A_{24}^+ : C_{24}), (X_S : Z_S)\big)$ // Alg. 17 or Alg. 19

$(A : C) \leftarrow (4A_{24}^+ - 2C_{24} : C_{24})$

$j = \mathtt{j\_inv}((A : C))$ // Alg. 9

return j // Encoded as in §1.2.8

Algorithm 24: Establishing shared keys in the 3-torsion

function `isoex`$_3$

Input: Secret key $\mathrm{sk}_3 \in \mathbb{Z}$ (see §1.2.6), public key $pk_2 = (x_1, x_2, x_3) \in (\mathbb{F}_{p^2})^3$ (see §1.2.9), and parameter `e3` (see §1.6)

Output: A j-invariant j_3 equivalent to the output of Step 4 of `isogen`$_\ell$ (see §1.3.7)

$(A : C) \leftarrow (\mathtt{get\_A}(x_1, x_2, x_3) : 1)$ // Alg. 10

$(X_S : Z_S) \leftarrow \mathtt{Ladder3pt}(\mathrm{sk}_3, (x_1, x_2, x_3), (A : C))$ // Alg. 8

$(A_{24}^+ : A_{24}^-) \leftarrow (A + 2 : A - 2)$

$(A_{24}^+ : A_{24}^-) \leftarrow \mathtt{3\_e\_iso}\big((A_{24}^+ : A_{24}^-), (X_S : Z_S)\big)$ // Alg. 18 or Alg. 20

$(A : C) \leftarrow (2 \cdot (A_{24}^- + A_{24}^+) : A_{24}^+ - A_{24}^-)$

$j = \mathtt{j\_inv}((A : C))$ // Alg. 9

return j // Encoded as in §1.2.8

Appendix B

Explicit algorithms for `isogen`$_\ell$ and `isoex`$_\ell$: Reference implementation

This section contains explicit formulas for computing the isogenies described in §1.3.6 and §1.3.7 as used in the reference implementation. Assuming access to all of the field operations in $\mathbb{F}_{p^2}$, Algorithms 25–45 can compute `isogen`$_\ell$ and `isoex`$_\ell$ for $\ell \in \{2, 3\}$ in their entirety.

The notation (x_P, y_P) is used for the affine tuple in $\mathbb{P}^1(\mathbb{F}_{p^2})$ representing the Montgomery x/y-coordinate. For simplicity, the reference implementation operates only on normalized, affine coordinates.

Only a single variant of the Montgomery curve constants are used with the tuple (a, b). Write $E_{a,b}$ for the curve $E_{a,b}/\mathbb{F}_{p^2}: by^2 = x^3 + ax^2 + x$.

Algorithm 25: Affine coordinate doubling

function `xDBL`
  Input: (x_P, y_P) and (a, b)
  Output: $(x_{[2]P}, y_{[2]P})$

if $P = \infty$ **then**
  return ∞

$t_0 \leftarrow {x_P}^2$
$t_1 \leftarrow t_0 + t_0$
$t_2 \leftarrow 1$
$t_0 \leftarrow t_0 + t_1$
$t_1 \leftarrow a \cdot x_P$
$t_1 \leftarrow t_1 + t_1$
$t_0 \leftarrow t_0 + t_1$
$t_0 \leftarrow t_0 + t_2$
$t_1 \leftarrow b \cdot y_P$
$t_1 \leftarrow t_1 + t_1$
$t_1 \leftarrow {t_1}^{-1}$
$t_0 \leftarrow t_0 \cdot t_1$
$t_1 \leftarrow {t_0}^2$
$t_2 \leftarrow b \cdot t_1$
$t_2 \leftarrow t_2 - a$
$t_2 \leftarrow t_2 - x_P$
$t_2 \leftarrow t_2 - x_P$
$t_1 \leftarrow t_0 \cdot t_1$
$t_1 \leftarrow b \cdot t_1$
$t_1 \leftarrow t_1 + y_P$
$y_{[2]P} \leftarrow x_P + x_P$
$y_{[2]P} \leftarrow y_{[2]P} + x_P$
$y_{[2]P} \leftarrow y_{[2]P} + a$
$y_{[2]P} \leftarrow y_{[2]P} \cdot t_0$
$y_{[2]P} \leftarrow y_{[2]P} - t_1$
$x_{[2]P} \leftarrow t_2$
return $(x_{[2]P}, y_{[2]P})$

Algorithm 26: Repeated affine coordinate doubling

function `xDBLe`

Input: (x_P, y_P), (a, b), and $e \in \mathbb{Z}$

Output: $(x_{[2^e]P}, y_{[2^e]P})$

$(x', y') \leftarrow (x_P, y_P)$

for $i = 1$ **to** e **do**

  $(x', y') \leftarrow$ `xDBL` $((x', y'), (a, b))$ // Alg. 25

return (x', y')

Algorithm 27: Affine coordinate addition

function `xADD`

Input: $P = (x_P, y_P)$, $Q = (x_Q, y_Q)$, and (a, b)

Output: (x_{P+Q}, y_{P+Q})

if $P = \infty$ **then**

  return (x_Q, y_Q)

if $Q = \infty$ **then**

  return (x_P, y_P)

if $P = Q$ **then**

  return `xDBL` $((x_P, y_P), (a, b))$

if $P = -Q$ **then**

  return ∞

$t_0 \leftarrow y_Q - y_P$

$t_1 \leftarrow x_Q - x_P$

$t_1 \leftarrow {t_1}^{-1}$

$t_0 \leftarrow t_0 \cdot t_1$

$t_1 \leftarrow {t_0}^2$

$t_2 \leftarrow x_P + x_P$

$t_2 \leftarrow t_2 + x_Q$

$t_2 \leftarrow t_2 + a$

$t_2 \leftarrow t_2 \cdot t_0$

$t_0 \leftarrow t_0 \cdot t_1$

$t_0 \leftarrow b \cdot t_0$

$t_0 \leftarrow t_0 + y_P$

$t_0 \leftarrow t_2 - t_0$

$t_1 \leftarrow b \cdot t_1$

$t_1 \leftarrow t_1 - a$

$t_1 \leftarrow t_1 - x_P$

$x_{[P+Q]} \leftarrow t_1 - x_Q$

$y_{[P+Q]} \leftarrow t_0$

return (x_{P+Q}, y_{P+Q})

Algorithm 28: Affine coordinate tripling

function `xTPL`

Input: (x_P, y_P) and (a, b)

Output: $(x_{[3]P}, y_{[3]P})$

$(x_{[2]P}, y_{[2]P}) \leftarrow$ `xDBL` $((x_P, y_P), (a, b))$ // Alg. 25

$(x_{[3]P}, y_{[3]P}) \leftarrow$ `xADD` $((x_P, y_P), (x_{[2]P}, y_{[2]P}), (a, b))$ // Alg. 27

return $(x_{[3]P}, y_{[3]P})$

Algorithm 29: Repeated affine coordinate tripling

function `xTPLe`
  Input: (x_P, y_P), (a, b), and $e \in \mathbb{Z}^+$
  Output: $(x_{[3^e]P}, y_{[3^e]P})$

$(x', y') \leftarrow (x_P, y_P)$
for $i = 1$ **to** e **do**
  $(x', y') \leftarrow$ `xTPL` $((x', y'), (a, b))$ `// Alg. 28`
return (x', y')

Algorithm 30: Double-and-add scalar multiplication

function `double_and_add`
  Input: $m = (m_{\ell-1}, \ldots, m_0)_2 \in \mathbb{Z}$, $P = (x, y)$, and (a, b)
  Output: $(x_{[m]P}, y_{[m]P})$

$(x_0, y_0) \leftarrow (0, 0)$
for $i = \ell - 1$ **to** 0 **by** -1 **do**
  $(x_0, y_0) \leftarrow$ `xDBL` $((x_0, y_0), (a, b))$ `// Alg. 25`
  if $m_i = 1$ **then**
    $(x_0, y_0) \leftarrow$ `xADD` $((x_0, y_0), (x, y), (a, b))$ `// Alg. 27`
return (x_0, y_0)

Algorithm 31: Montgomery j-invariant computation

function `j_inv`
  Input: a
  Output: j-invariant $j(E_{a,b}) \in \mathbb{F}_{p^2}$

$t_0 \leftarrow a^2$
$j \leftarrow 3$
$j \leftarrow t_0 - j$
$t_1 \leftarrow j^2$
$j \leftarrow j \cdot t_1$
$j \leftarrow j + j$
$j \leftarrow j + j$
$j \leftarrow j + j$
$j \leftarrow j + j$
$j \leftarrow j + j$
$j \leftarrow j + j$
$j \leftarrow j + j$
$j \leftarrow j + j$
$t_1 \leftarrow 4$
$t_0 \leftarrow t_0 - t_1$
$t_0 \leftarrow t_0^{-1}$
$j \leftarrow j \cdot t_0$
return j

Algorithm 32: Computing the 2-isogenous curve

function `curve_2_iso`

Input: x_{P_2} and b, where P_2 has exact order 2 on $E_{a,b}$

Output: (a', b') corresponding to $E_{a',b'} = E_{a,b}/\langle P_2 \rangle$

$t_1 \leftarrow {x_{P_2}}^2$

$t_1 \leftarrow t_1 + t_1$

$t_1 \leftarrow 1 - t_1$

$a' \leftarrow t_1 + t_1$

$b' \leftarrow x_{P_2} \cdot b$

return (a', b')

Algorithm 33: Evaluating a 2-isogeny at a point

function `eval_2_iso`

Input: (x_Q, y_Q) and x_{P_2}, where $P \in E_{a,b}$, and P_2 has exact order 2 on $E_{a,b}$

Output: $(x_{Q'}, y_{Q'})$ corresponding to $Q' \in E_{a',b'}$, where $E_{a',b'}$ is the curve 2-isogenous to $E_{a,b}$ output from `curve_2_iso`

$t_1 \leftarrow x_Q \cdot x_{P_2}$

$t_2 \leftarrow x_Q \cdot t_1$

$t_3 \leftarrow t_1 \cdot x_{P_2}$

$t_3 \leftarrow t_3 + t_3$

$t_3 \leftarrow t_2 - t_3$

$t_3 \leftarrow t_3 + x_{P_2}$

$t_3 \leftarrow y_Q \cdot t_3$

$t_2 \leftarrow t_2 - x_Q$

$t_1 \leftarrow x_Q - x_{P_2}$

$t_1 \leftarrow {t_1}^{-1}$

$x_{Q'} \leftarrow t_2 \cdot t_1$

$t_1 \leftarrow {t_1}^2$

$y_{Q'} \leftarrow t_3 \cdot t_1$

return $(x_{Q'}, y_{Q'})$

Algorithm 34: Computing the 4-isogenous curve

function `curve_4_iso`

Input: x_{P_4} and b, where P_4 has exact order 4 on $E_{a,b}$

Output: (a', b') corresponding to $E_{a',b'} = E_{a,b}/\langle P_4 \rangle$

$t_1 \leftarrow {x_{P_4}}^2$

$a' \leftarrow {t_1}^2$

$a' \leftarrow a' + a'$

$a' \leftarrow a' + a'$

$t_2 \leftarrow 2$

$a' \leftarrow a' - t_2$

$t_1 \leftarrow x_{P_4} \cdot t_1$

$t_1 \leftarrow t_1 + x_{P_4}$

$t_1 \leftarrow t_1 \cdot b$

$t_2 \leftarrow {t_2}^{-1}$

$t_2 \leftarrow -t_2$

$b' \leftarrow t_2 \cdot t_1$

return (a', b')

Algorithm 35: Evaluating a 4-isogeny at a point

function `eval_4_iso`

Input: (x_Q, y_Q) and x_{P_4}, where $P \in E_{a,b}$, and P_4 has exact order 4 on $E_{a,b}$

Output: $(x_{Q'}, y_{Q'})$ corresponding to $Q' \in E_{a',b'}$, where $E_{a',b'}$ is the curve 4-isogenous to $E_{a,b}$ output from `curve_4_iso`

$t_1 \leftarrow {x_Q}^2$
$t_2 \leftarrow {t_1}^2$
$t_3 \leftarrow {x_{P_4}}^2$
$t_4 \leftarrow t_2 \cdot t_3$
$t_2 \leftarrow t_2 + t_4$
$t_4 \leftarrow t_1 \cdot t_3$
$t_4 \leftarrow t_4 + t_4$
$t_5 \leftarrow t_4 + t_4$
$t_5 \leftarrow t_5 + t_5$
$t_4 \leftarrow t_4 + t_5$
$t_2 \leftarrow t_2 + t_4$
$t_4 \leftarrow {t_3}^2$
$t_5 \leftarrow t_1 \cdot t_4$
$t_5 \leftarrow t_5 + t_5$
$t_2 \leftarrow t_2 + t_5$
$t_1 \leftarrow t_1 \cdot x_Q$
$t_4 \leftarrow x_{P_4} \cdot t_3$
$t_5 \leftarrow t_1 \cdot t_4$
$t_5 \leftarrow t_5 + t_5$
$t_5 \leftarrow t_5 + t_5$
$t_2 \leftarrow t_2 - t_5$
$t_1 \leftarrow t_1 \cdot x_{P_4}$
$t_1 \leftarrow t_1 + t_1$
$t_1 \leftarrow t_1 + t_1$
$t_1 \leftarrow t_2 - t_1$
$t_2 \leftarrow x_Q \cdot t_4$
$t_2 \leftarrow t_2 + t_2$
$t_2 \leftarrow t_2 + t_2$
$t_1 \leftarrow t_1 - t_2$
$t_1 \leftarrow t_1 + t_3$
$t_1 \leftarrow t_1 + 1$
$t_2 \leftarrow x_Q \cdot x_{P_4}$
$t_4 \leftarrow t_2 - 1$
$t_2 \leftarrow t_2 + t_2$
$t_5 \leftarrow t_2 + t_2$
$t_1 \leftarrow t_1 - t_5$
$t_1 \leftarrow t_4 \cdot t_1$
$t_1 \leftarrow t_3 \cdot t_1$
$t_1 \leftarrow y_Q \cdot t_1$
$t_1 \leftarrow t_1 + t_1$
$y_{Q'} \leftarrow -t_1$
$t_2 \leftarrow t_2 - t_3$
$t_1 \leftarrow t_2 - 1$
$t_2 \leftarrow x_Q - x_{P_4}$
$t_1 \leftarrow t_2 \cdot t_1$
$t_5 \leftarrow {t_1}^2$
$t_5 \leftarrow t_5 \cdot t_2$
$t_5 \leftarrow {t_5}^{-1}$
$y_{Q'} \leftarrow y_{Q'} \cdot t_5$
$t_1 \leftarrow t_1 \cdot t_2$
$t_1 \leftarrow {t_1}^{-1}$
$t_4 \leftarrow {t_4}^2$
$t_1 \leftarrow t_1 \cdot t_4$
$t_1 \leftarrow x_Q \cdot t_1$
$t_2 \leftarrow x_Q \cdot t_3$
$t_2 \leftarrow t_2 + x_Q$
$t_3 \leftarrow x_{P_4} + x_{P_4}$
$t_2 \leftarrow t_2 - t_3$
$t_2 \leftarrow -t_2$
$x_{Q'} \leftarrow t_1 \cdot t_2$
return $(x_{Q'}, y_{Q'})$

Algorithm 36: Computing the 3-isogenous curve

function `curve_3_iso`

Input: x_{P_3} and (a, b), where P_3 has exact order 3 on $E_{a,b}$

Output: Curve constant (a', b') corresponding to $E_{a',b'} = E_{a,b}/\langle P_3 \rangle$

$t_1 \leftarrow {x_{P_3}}^2$
$b' \leftarrow b \cdot t_1$
$t_1 \leftarrow t_1 + t_1$
$t_2 \leftarrow t_1 + t_1$
$t_1 \leftarrow t_1 + t_2$
$t_2 \leftarrow 6$
$t_1 \leftarrow t_1 - t_2$
$t_2 \leftarrow a \cdot x_{P_3}$
$t_1 \leftarrow t_2 - t_1$
$a' \leftarrow t_1 \cdot x_{P_3}$
return (a', b')

Algorithm 37: Evaluating a 3-isogeny at a point

function `eval_3_iso`

Input: (x_Q, y_Q) and x_{P_3}, where $P \in E_{a,b}$, and P_3 has exact order 3 on $E_{a,b}$

Output: $(x_{Q'}, y_{Q'})$ corresponding to $Q' \in E_{a',b'}$, where $E_{a',b'}$ is the curve 3-isogenous to $E_{a,b}$ output from `curve_3_iso`

$t_1 \leftarrow {x_Q}^2$
$t_1 \leftarrow t_1 \cdot x_{P_3}$
$t_2 \leftarrow {x_{P_3}}^2$
$t_2 \leftarrow x_Q \cdot t_2$
$t_3 \leftarrow t_2 + t_2$
$t_2 \leftarrow t_2 + t_3$
$t_1 \leftarrow t_1 - t_2$
$t_1 \leftarrow t_1 + x_Q$
$t_1 \leftarrow t_1 + x_{P_3}$
$t_2 \leftarrow x_Q - x_{P_3}$
$t_2 \leftarrow {t_2}^{-1}$
$t_3 \leftarrow {t_2}^2$
$t_2 \leftarrow t_2 \cdot t_3$
$t_4 \leftarrow x_Q \cdot x_{P_3}$
$t_4 \leftarrow t_4 - 1$
$t_1 \leftarrow t_4 \cdot t_1$
$t_1 \leftarrow t_1 \cdot t_2$
$t_2 \leftarrow {t_4}^2$
$t_2 \leftarrow t_2 \cdot t_3$
$x_{Q'} \leftarrow x_Q \cdot t_2$
$y_{Q'} \leftarrow y_Q \cdot t_1$
return $(x_{Q'}, y_{Q'})$

Algorithm 38: Computing and evaluating a 2^e-isogeny, simple version

function `iso_2_e`

Static parameters: Integer $\mathtt{e}_2$ from the public parameters

Input: Constants (a, b) corresponding to a curve $E_{a,b}$, (x_S, y_S) where S has exact order $2^{\mathtt{e}_2}$ on $E_{a,b}$

Optional input: $\{(x_1, y_1), ..., (x_n, y_n)\}$ on $E_{a,b}$

Output: (a', b') corresponding to the curve $E_{a',b'} = E/\langle S \rangle$

Optional output: $\{(x'_1, y'_1), ..., (x'_n, y'_n)\}$ on $E_{a',b'}$

$(a', b') \leftarrow (a, b)$

$e'_2 \leftarrow \mathtt{e}_2$

if e'_2 *is odd* **then**

    $(x_T, y_T) \leftarrow \mathtt{xDBLe}\big((x_S, y_S), (a', b'), e'_2 - 1\big)$ // Alg. 26

    $(a', b') \leftarrow \mathtt{curve\_2\_iso}(x_T, b')$ // Alg. 32

    $(x_S, y_S) \leftarrow \mathtt{eval\_2\_iso}((x_S, y_S), x_T)$ // Alg. 33

    for (x_j, y_j) **in** *optional input* **do**

        $(x'_j, y'_j) \leftarrow \mathtt{eval\_2\_iso}\big((x_j, y_j), x_T\big)$ // Alg. 33

    $e'_2 \leftarrow e'_2 - 1$

for $e = e'_2 - 2$ **downto** 0 **by** -2 **do**

    $(x_T, y_T) \leftarrow \mathtt{xDBLe}((x_S, y_S), (a', b'), e)$ // Alg. 26

    $(a', b') \leftarrow \mathtt{curve\_4\_iso}(x_T, b')$ // Alg. 34

    if $e \neq 0$ **then**

        $(x_S, y_S) \leftarrow \mathtt{eval\_4\_iso}((x_S, y_S), x_T)$ // Alg. 35

    for (x_j, y_j) **in** *optional input* **do**

        $(x'_j, y'_j) \leftarrow \mathtt{eval\_4\_iso}\big((x_j, y_j), x_T\big)$ // Alg. 35

return $(a', b'), [(x'_1, y'_1), ..., (x'_n, y'_n)]$

Algorithm 39: Computing and evaluating a 3^e-isogeny, simple version

function `iso_3_e`

Static parameters: Integer $\mathtt{e}_3$ from the public parameters

Input: Constants (a, b) corresponding to a curve $E_{a,b}$, (x_S, y_S) where S has exact order $3^{\mathtt{e}_3}$ on $E_{a,b}$

Optional input: $\{(x_1, y_1), ..., (x_n, y_n)\}$ on $E_{a,b}$

Output: (a', b') corresponding to the curve $E_{a',b'} = E/\langle S\rangle$

Optional output: $\{(x'_1, y'_1), ..., (x'_n, y'_n)\}$ on $E_{a',b'}$

$(a', b') \leftarrow (a, b)$

for $e = e_3 - 1$ **downto** 0 **by** -1 **do**

  $(x_T, y_T) \leftarrow$ `xTPLe` $((x_S, y_S), (a', b'), e)$ `// Alg. 29`

  $(a', b') \leftarrow$ `curve_3_iso` (x_T, y_T) `// Alg. 36`

  if $e \neq 0$ **then**

    $(x_S, y_S) \leftarrow$ `eval_3_iso` $((x_S, y_S), x_T)$ `// Alg. 37`

  for (x_j, y_j) **in** *optional input* **do**

    $(x'_j, y'_j) \leftarrow$ `eval_3_iso` $\big((x_j, y_j), x_T\big)$ `// Alg. 37`

return $(a', b'), [(x'_1, y'_1), ..., (x'_n, y'_n)]$

Algorithm 40: Recovering the x-coordinate of R

function `get_xR`

Input: Parameters of $E_{a,b}$ with generator points: (a, b), $P = (x_P, y_P)$, $Q = (x_Q, y_Q)$

Output: x_R, such that $R = P - Q$

$(x_R, y_R) \leftarrow$ `xADD` $((x_P, y_P), (x_Q, -y_Q), (a, b))$ `// Alg. 27`

return x_R

Algorithm 41: Recovering the y-coordinates of P and Q, and the Montgomery curve coefficient a

function `get_yP_yQ_A_B`

Input: $pk = (x_P, x_Q, x_R)$ // Encoded as in §1.2.8

Output: (y_P, y_Q, a, b)

$a \leftarrow$ `get_A`(x_P, x_Q, x_R) // Alg. 10

$b \leftarrow 1$

$t_1 \leftarrow {x_P}^2$

$t_2 \leftarrow x_P \cdot t_1$

$t_1 \leftarrow a \cdot t_1$

$t_1 \leftarrow t_2 + t_1$

$t_1 \leftarrow t_1 + x_P$

$y_P \leftarrow \sqrt{t_1}$

$t_1 \leftarrow {x_Q}^2$

$t_2 \leftarrow x_Q \cdot t_1$

$t_1 \leftarrow a \cdot t_1$

$t_1 \leftarrow t_2 + t_1$

$t_1 \leftarrow t_1 + x_Q$

$y_Q \leftarrow \sqrt{t_1}$

$(x_T, y_T) \leftarrow$ `xADD`$((x_P, y_P), (x_Q, -y_Q), (a, b))$ // Alg. 27

if $x_T \neq x_R$ **then**

$\quad y_Q \leftarrow -y_Q$

return (y_P, y_Q, a, b)

Algorithm 42: Computing public keys in the 2-torsion

function `isogen`$_2$

Input: Secret key $\mathrm{sk}_2 \in \mathbb{Z}$ (see §1.2.6) and public parameters $\{e2, e3, p, (x_{P2}, y_{P2}), (x_{Q2}, y_{Q2}), (x_{P3}, y_{P3}), (x_{Q3}, y_{Q3})\}$ (see §1.6)

Output: Public key $pk_2 = (x'_{P3}, x'_{Q3}, x'_{R3})$ equivalent to the output of Step 4 of `isogen`$_\ell$ (see §1.3.6)

$(a, b) \leftarrow (6, 1)$

$(x_S, y_S) \leftarrow$ `double_and_add`$(\mathrm{sk}_2, (x_{Q2}, y_{Q2}), (a, b))$ // Alg. 30

$(x_S, y_S) \leftarrow$ `xADD`$((x_{P2}, y_{P2}), (x_S, y_S), (a, b))$ // Alg. 27

$\big((a', b'), (x'_{P3}, y'_{P3}), (x'_{Q3}, y'_{Q3})\big) \leftarrow$ `iso_2_e`$((a, b), (x_S, y_S), (x_{P3}, y_{P3}), (x_{Q3}, y_{Q3}))$ // Alg. 38

$x'_{R3} \leftarrow$ `get_xR`$\big((a', b'), (x'_{P3}, y'_{P3}), (x'_{Q3}, y'_{Q3})\big)$ // Alg. 40

return $pk_2 = (x'_{P3}, x'_{Q3}, x'_{R3})$ // Encoded as in §1.2.9

Algorithm 43: Computing public keys in the 3-torsion

function `isogen`$_3$

Input: Secret key $\text{sk}_3 \in \mathbb{Z}$ (see §1.2.6) and public parameters $\{e2, e3, p, (x_{P2}, y_{P2}), (x_{Q2}, y_{Q2}), (x_{P3}, y_{P3}), (x_{Q3}, y_{Q3})\}$ (see §1.6)

Output: Public key $pk_3 = (x'_{P2}, x'_{Q2}, x'_{R2})$ equivalent to the output of Step 4 of `isogen`$_\ell$ (see §1.3.6)

1 $(a, b) \leftarrow (6, 1)$

2 $(x_S, y_S) \leftarrow$ `double_and_add` $(\text{sk}_3, (x_{Q3}, y_{Q3}), (a, b))$ `// Alg. 30`

3 $(x_S, y_S) \leftarrow$ `xADD` $((x_{P3}, y_{P3}), (x_S, y_S), (a, b))$ `// Alg. 27`

4 $\big((a', b'), (x'_{P2}, y'_{P2}), (x'_{Q2}, y'_{Q2})\big) \leftarrow$ `iso_3_e` $((a, b), (x_S, y_S), (x_{P2}, y_{P2}), (x_{Q2}, y_{Q2}))$ `// Alg. 39`

5 $x'_{R2} \leftarrow$ `get_xR` $\big((a', b'), (x'_{P2}, y'_{P2}), (x'_{Q2}, y'_{Q2})\big)$ `// Alg. 40`

6 **return** $pk_3 = (x'_{P2}, x'_{Q2}, x'_{R2})$ `// Encoded as in §1.2.9`

Algorithm 44: Establishing shared keys in the 2-torsion

function `isoex`$_2$

Input: Secret key $\text{sk}_2 \in \mathbb{Z}$ (see §1.2.6), public key $pk_3 = (x'_{P2}, x'_{Q2}, x'_{R2}) \in (\mathbb{F}_{p^2})^3$ (see §1.2.9), and parameter `e2` (see §1.6)

Output: A j-invariant j_2 equivalent to the output of Step 4 of `isogen`$_\ell$ (see §1.3.7)

1 $(y'_{P2}, y'_{Q2}, a, b) \leftarrow$ `get_yP_yQ_A_B` $(x'_{P2}, x'_{Q2}, x'_{R2})$ `// Alg. 41`

2 $(x_S, y_S) \leftarrow$ `mult_double_add` $\big(\text{sk}_2, (x'_{Q2}, y'_{Q2}), (a, b)\big)$ `// Alg. 30`

3 $(x_S, y_S) \leftarrow$ `xADD` $\big((x'_{P2}, y'_{P2}), (x_S, y_S), (a, b)\big)$ `// Alg. 27`

4 $(a, b) \leftarrow$ `2_e_iso` $((a, b), (x_S, y_S))$ `// Alg. 38`

5 $j_2 =$ `j_inv` (a) `// Alg. 31`

6 **return** j_2 `// Encoded as in §1.2.8`

Algorithm 45: Establishing shared keys in the 3-torsion

function `isoex`$_3$

Input: Secret key $\mathrm{sk}_3 \in \mathbb{Z}$ (see §1.2.6), public key $pk_2 = (x'_{P3}, x'_{Q3}, x'_{R3}) \in (\mathbb{F}_{p^2})^3$ (see §1.2.9), and parameter `e3` (see §1.6)

Output: A j-invariant j_3 equivalent to the output of Step 4 of `isogen`$_\ell$ (see §1.3.7)

$(y'_{P3}, y'_{Q3}, a, b) \leftarrow$ `get_yP_yQ_A_B`$(x'_{P3}, x'_{Q3}, x'_{R3})$ `// Alg. 41`

$(x_S, y_S) \leftarrow$ `mult_double_add`$\big(\mathrm{sk}_3, (x'_{Q3}, y'_{Q3}), (a, b)\big)$ `// Alg. 30`

$(x_S, y_S) \leftarrow$ `xADD`$\big((x'_{P3}, y'_{P3}), (x_S, y_S), (a, b)\big)$ `// Alg. 27`

$(a, b) \leftarrow$ `3_e_iso`$((a, b), (x_S, y_S))$ `// Alg. 39`

$j_3 =$ `j_inv`$(a))$ `// Alg. 31`

return j_3 `// Encoded as in §1.2.8`

Appendix C

Explicit algorithms for compressed SIKE: Optimized implementation

The major algorithms underlying the current key compression techniques [52] are listed next. In this section we assume that the public parameters include the following data.

- The torsion basis generation algorithms 47 and 48 use elligator-like techniques to get points on the curve. This involves computing values of the form $v = 1/(1+u\cdot r^2) \in \mathbb{F}_{p^2}$. In order to avoid multiple inversions, a precomputed table T is employed in the optimized implementation. Experimentally, less than 20 elements are enough for storage.

- Optimal traversal paths $\{\text{path}_k = (s_1, \ldots, s_{\mathsf{e}_k/2-1}) \in (\mathbb{N}^+)^{\mathsf{e}_k/2-1}\}$ where $k \in \{2, 3\}$ for solving discrete logarithms via Pohlig-Hellman. Optimal paths for discrete logarithms can be generated by *compute_strategy* (Algorithm 60), the same that allows for computing smooth degree isogenies in complexity $O(e_\ell \log e_\ell)$. In this context the input parameters p, q will consist of the costs of powering an element of $\mathbb{F}_{p^2}$ to ℓ (or ℓ^w in the general case) and of multiplying two elements in $\mathbb{F}_{p^2}$, respectively. In the implementation a different encoding of the strategies shown in Appendix D is used.

- When computing a discrete logarithm of the form $\log_g(r)$ with $r = g^d$ and $d = (d_{k-1}\cdots d_1 d_0)_{\ell^w}$, the base $g \in \mathbb{F}_{p^2}$ is fixed and can be included in the public parameters. In particular, precomputed tables are employed to speed up computations. The compression algorithms uses tables $T_1[u][d] := g^{-\ell^{w\cdot u}\cdot d}$ and $T_2[0][d_i] := g^{-d_i}$ and $T_2[u][d_i] := g^{-d_i\cdot \ell^{e \bmod w+(u-1)w}}$ for $0 < u < \lceil e/w \rceil$ and 0 as described in [52] to speed up computations. The parameter w can be seen as a tradeoff between speed vs. storage. Larger w implies smaller trees to be traversed but larger discrete logarithm instances at the leaves.

- The size of the tables in discrete logarithm computations can be compressed by a factor of two by utilizing the signed digit representation of exponents and the fact that the cost of inverting a cyclotomic group element is negligible: if $h = a + bi$, then $h^{-1} = a - bi$. More precisely, for an element $h \in \mathbb{F}_{p^2}$, whose order divides ℓ^e, we first use the equaltiy $h^d = h^{d-\ell^e}$ for $d > \lfloor \ell^e/2 \rfloor$ to replace the interval for exponents from $[0, \ell^e)$ to $I_- \cup I_+ = [-\lfloor \ell^e/2 \rfloor, 0) \cup [0, \lfloor \ell^e/2 \rfloor]$. Then, in the implementation, the interval I_- with negative exponents can be discarded because, for $d \in I_-$, an element h^d can be recovered through an inversion $(h^{-d})^{-1}$ at a negligible cost. Note that $-d \in I_+$ and h^{-d} would already be stored in one of the tables T, T_1, or T_2.

Algorithm 46: x-only tripling k times on the Montgomery curve $E_A : y^2 = x^3 + Ax^2 + x$

function `xTPLe_fast`

  Input: $P = (x, z) \in E_A$, the coefficient $A_2 = A/2 \in \mathbb{F}_{p^2}$ and the number of triplings k.

  Output: $[3^k]P = (x', z')$

---

for $j = 1$ **to** k **do**

  $t_1 \leftarrow x^2$

  $t_2 \leftarrow z^2$

  $t_3 \leftarrow t_1 + t_2$

  $t_4 \leftarrow A_2 \cdot ((x + z)^2 - t_3) + t_3$

  $t_3 \leftarrow (t_1 - t_2)^2$

  $t_1 \leftarrow (t_1 \cdot t_4 - t_3)^2$

  $t_2 \leftarrow (t_2 \cdot t_4 - t_3)^2$

  $x' \leftarrow x \cdot t_2$

  $z' \leftarrow z \cdot t_1$

return (x', z')

Algorithm 47: Entangled basis generation for $E[2^{e_2}](\mathbb{F}_{p^2}) : y^2 = x^3 + Ax^2 + x$

function `get_2_torsion_entangled_basis_compression`

Input: $A = a + bi \in \mathbb{F}_{p^2}$ and the public parameters $u_0 \in \mathbb{F}_{p^2} : u = u_0^2 \in \mathbb{F}_{p^2} \backslash \mathbb{F}_p$; tables T_1, T_2 of pairs $(r \in \mathbb{F}_p, v = 1/(1 + ur^2) \in \mathbb{F}_{p^2})$ of QNR and QR.

Output: $\{S_1, S_2\}$ such that $\langle [3^{e_3}]S_1, [3^{e_3}]S_2 \rangle = E[2^{e_2}](\mathbb{F}_{p^2})$, a bit *bit* indicating the quadraticity of A and the table entry for r

$z \leftarrow a^2 + b^2$

$s \leftarrow z^{(p+1)/4}$

$T \leftarrow (s^2 \stackrel{?}{=} z)\ T_1 : T_2$ select proper table by testing quadraticity of A

repeat

  lookup next entry (r, v) from T

  $x \leftarrow -A \cdot v$

  $t \leftarrow x \cdot (x^2 + A \cdot x + 1)$

  test quadraticity of $t = c + di$

  $z \leftarrow c^2 + d^2$,

  $s \leftarrow z^{(p+1)/4}$

until $s^2 = z$

compute $y \leftarrow \sqrt{x^3 + A \cdot x^2 + x}$

$z \leftarrow (c + s)/2$

$\alpha \leftarrow z^{(p+1)/4}$

$\beta \leftarrow d \cdot (2\alpha)^{-1}$

$y \leftarrow (\alpha^2 \stackrel{?}{=} z)\ \alpha + \beta i : -\beta - \alpha i$

return $S_1 \leftarrow (x, y),\ S_2 \leftarrow (ur^2 x, u_0 r y)$, $bit \leftarrow (T \stackrel{?}{=} T_1)\ 1 : 0, r$

Algorithm 48: Generating a basis for $E[3^{e_3}](\mathbb{F}_{p^2}) : y^2 = x^3 + Ax^2 + x$

function `BuildOrdinaryE3nBasis`

Input: Montgomery curve coefficient A

Output: 3^{e_3}-torsion basis information

`// See details of basis generation and further optimizations in [35, 36]`

return 3^{e_3}*-torsion basis information*

Algorithm 49: Deterministically generating a basis for $E[3^{e_3}](\mathbb{F}_{p^2}) : y^2 = x^3 + Ax^2 + x$ from A elligator counters r_1, r_2 and bits b_1, b_2

function `BuildOrdinaryE3nBasis_decompression`

Input: Montgomery curve coefficient A, elligator counters $(r_1, r_2) \in \mathbb{Z}^2_{2^8}$ and bits b_1, b_2 indicating the correct elligator abscissa

Output: Canonical basis for the 3^{e_3}-torsion on E

```
// See details of basis recovery and further optimizations in [35, 36]
```

return *Canonical basis for the* 3^{e_3}*-torsion on* E

Algorithm 50: Compute $log_g(r)$ for a fixed $g \in \mathbb{F}_{p^2}$ using an optimal traversal strategy

function `Traverse_w_div_e_fullsigned`

Input: r: value of root vertex Δ_{jk}, i.e. $r := r_k^{\ell^{wj}}$; j, k: coordinates of root vertex Δ_{jk}; z: number of leaves in subtree rooted at Δ_{jk}, w: an integer dividing e_k (the exponent of the respective torsion) and the public parameters P: traversal path and T: lookup table.

Output: d: signed digits (radix ℓ^w) of $\log_g r_0$

Remark: initial call is Traverse_w_div_e_fullsigned($r_0, 0, 0, len(P) - 1, \ell, w, P, T, d$).

Remark: assume w divides the exponent of the respective torsion e.

---

if $z > 1$ **then**
    $t \leftarrow P[z]$
    $r' \leftarrow r^{\ell^{w(z-t)}}$ `// go left` $w(z-t)$ `times`
    Traverse_w_div_e_fullsigned($r', j + (z - t), k, t, \ell, w, P, T, d$)
    $r' \leftarrow r$
    for $h \leftarrow k$ **to** $k + t - 1$ **do** `// go right` t `times`
        if $d_h > 0$ **then**
            $r' \leftarrow r' \cdot T[j + h][d_h]$
        else if $d_h < 0$ **then**
            $r' \leftarrow r' \cdot \overline{T[j + h][-d_h]}$
    Traverse_w_div_e_fullsigned($r', j, k + t, z - t, \ell, w, P, T, d$)
else `// leaf`
    for $t \leftarrow 0$ **to** $\lceil(\ell^w - 1)/2\rceil$ **do** `// recover` k`-th digit` d_k
        if $r = \overline{T[e/w - 1][t]}$ **then**
            $d_k \leftarrow t$
            break
        else if $r = T[e/w - 1][t]$ **then**
            $d_k \leftarrow -t$
            break

Algorithm 51: Compute $log_g(r)$ for a fixed $g \in \mathbb{F}_{p^2}$ using an optimal traversal strategy

function `Traverse_w_notdiv_e_fullsigned`

Input: r: value of root vertex $\Delta_{j,k}$, i.e. $r := r_k^{\ell^{e \bmod w + (j-1)w}}$; j, k: coordinates of root vertex $\Delta_{j,k}$; z: number of leaves in subtree rooted at $\Delta_{j,k}$, w: an integer not dividing e (the exponent of the respective torsion) and the public parameters P: traversal path, T_1, T_2: lookup tables.

Output: d: signed digits (radix ℓ^w) of $\log_g r_0$

Remark: initial call is Traverse_w_notdiv_e_fullsigned($r_0, 0, 0, len(P) - 1, \ell, w, P, T_1, T_2, d$).

Remark: assume w divides the exponent of the torsion e.

---

if $z > 1$ **then**

  $t \leftarrow P[z]$ // z leaves

  if $j > 0$ **then**

    $r' \leftarrow r^{\ell^{w(z-t)}}$ // go left $w(z-t)$ times

  else

    $r' \leftarrow r^{\ell^{e \bmod w + w(z-t-1)}}$ // go left $e \pmod w + w(z-t-1)$ times

  Traverse_w_notdiv_e_fullsigned($r', j + (z-t), k, t, \ell, w, P, T_1, T_2, d$)

  $r' \leftarrow r$

  for $h \leftarrow k$ **to** $k + t - 1$ **do** // go right t times

    if $j = 0$ **then** // left-most leaf

      if $d_h > 0$ **then**

        $r' \leftarrow r' \cdot T_1[j+h][d_h]$

      else if $d_h < 0$ **then**

        $r' \leftarrow r' \cdot \overline{T_1[j+h][d_h]}$

    else

      if $d_h > 0$ **then**

        $r' \leftarrow r' \cdot T_2[j+h][d_h]$

      else if $d_h < 0$ **then**

        $r' \leftarrow r' \cdot \overline{T_2[j+h][d_h]}$

  Traverse_w_notdiv_e_fullsigned($r', j, k + t, z - t, \ell, w, P, T_1, T_2, d$)

  // Continued on next page...

```
// Continued from previous page...
else                                                        // leaf
    if $j = 0$ and $k = \lceil e/w \rceil - 1$ then         // left-most leaf
        for $t \leftarrow 0$ to $\lceil \ell^{e \bmod w}/2 \rceil$ do   // recover the $k$-th digit $d_k$
            if $r = \overline{T_1[\lceil e/w \rceil - 1][t]}$ then
                $d_k \leftarrow t$
                break
            else if $r = T_1[\lceil e/w \rceil - 1][t]$ then
                $d_k \leftarrow -t$
                break
    else
        for $t \leftarrow 0$ to $\lceil \ell^{w}/2 \rceil$ do   // recover the $k$-th digit $d_k$
            if $r = \overline{T_2[\lceil e/w \rceil - 1][t]}$ then
                $d_k \leftarrow t$
                break
            else if $r = T_2[\lceil e/w \rceil - 1][t]$ then
                $d_k \leftarrow -t$
                break
```

Algorithm 52: Convert a point on a Montgomery curve $E : y^2 = x^3 + Ax^2 + x$ into the corresponding point on its short Weierstrass form $E_W : y^2 = x^3 + ax + b$.

function `PointMonty2Weier`

  Input: Point $(x, y, z) \in E$ and A

  Output: Affine point $(x', y') \in E_W$

if $z = 0$ **then**

  $x' \leftarrow 0; y' \leftarrow 1; z' \leftarrow 0$

else

  $x' \leftarrow x + A/3$

  $y' \leftarrow y$

  $z' \leftarrow 1$

return (x', y', z')

Algorithm 53: Convert a Montgomery curve $E : y^2 = x^3 + Ax^2 + x$ into the corresponding short Weierstrass form $E_W : y^2 = x^3 + ax + b$.

function `Monty2Weier`

  Input: Montgomery curve coefficient A

  Output: Weierstrass coefficients (a, b)

$a \leftarrow 1 - A^2/3$

$b \leftarrow (2 \cdot A^3 - 9 \cdot A)/27$

return (a, b)

Algorithm 54: Compute the discrete logarithm (optimal Pohlig-Hellman traversal strategy) $d = log_g(r)$ where $g = e_{\ell_k^{e_k}}(P_k, Q_k)^{\ell_{\bar{k}}^{e_{\bar{k}}}} \in \mathbb{F}_{p^2}$ and $\bar{k}$ is the complement of the torsion $k \in \{2, 3\}$.

function `solve_dlog`

Input: Element $r \in \mathbb{F}_{p^2}$, the corresponding torsion k and the following public parameters corresponding to torsion k: optimal Pohlig-Hellman traversal path $\text{path}_k \in \mathbb{Z}^{plen_k}$, tables $(T_1)_k, (T_2)_k$ of precomputed values in $\mathbb{F}_{p^2}$, and exponent w_k.

Output: The discrete logarithm $d \in \mathbb{Z}_{\ell_k^{e_k}}$

if $e_k \pmod{w} = 0$ **then**

  $d \leftarrow Traverse\_w\_div\_e\_fullsigned(r, 0, 0, plen_k - 1, \ell, w_k, \text{path}_k, (T_1)_k)$ `// Alg. 50`

else

  $d \leftarrow Traverse\_w\_notdiv\_e\_fullsigned(r, 0, 0, plen_k - 1, \ell, w_k, \text{path}_k, (T_1)_k, (T_2)_k)$ `// Alg. 51`

return $d \in \mathbb{Z}_{\ell_k^{e_k}}$

Algorithm 55: Compute 4 discrete logarithms (optimal Pohlig-Hellman strategy) on the multiplicative subgroup of order $\ell_k^{e_k}$

function *ph*

Input: Points $P = (x_{P_M}, y_{P_M}), Q = (x_{Q_M}, y_{Q_M}), S_1 = (x_{S_1}, y_{S_1}), S_2 = (x_{S_2}, y_{S_2})$ on $E : y^2 = x^3 + Ax^2 + x$, the coefficient A, and the corresponding torsion $k \in \{2, 3\}$

Output: The discrete logs $(c_0, c_1, d_0, d_1) \in \mathbb{Z}_{\ell_k^{e_k}}$ such that $P = [c_0]S_1 + [c_1]S_2$ and $Q = [d_0]S_1 + [d_1]S_2$

$n_1, n_2, n_3, n_4 \leftarrow Tate\_4\_pairings\_k\_torsion(P, Q, S_1, S_2, A)$ `// see Alg. 2 or 3 of [35]`

$d_0 \leftarrow solve\_dlog(n_1, k)$ `// Alg. 54`

$c_0 \leftarrow solve\_dlog(n_3, k)$ `// Alg. 54`

$d_1 \leftarrow solve\_dlog(n_2, k)$ `// Alg. 54`

$c_1 \leftarrow solve\_dlog(n_4, k)$ `// Alg. 54`

return $c_0, d_0, c_1, d_1 \in (\mathbb{Z}_{\ell_k^{e_k}})^4$

Algorithm 56: Computing compressed public keys in the 3^{e_3}-torsion

function `PublicKeyCompression_2`

Input: Private key $sk_2 \in \mathbb{Z}_{2^{e_2}}$ and public key $pk_2 = (x_1, x_2, x_3)$ and Montgomery coefficient A

Output: Compressed public key $pk\_comp_2 = (bit, b_1, b_2, t_1, t_2, t_3, A, r_1, r_2)$ according to compressed encoding

$((x_P, x_Q, x_R) : P, Q, R \in E_{A'=0}) \leftarrow IsogenyDual(sk_2, A)$ `// see [35] for dual iso. computation`

$x_U, y_U, x_V, y_V, r_1, r_2, b_1, b_2 \leftarrow BuildOrdinaryE3nBasis(A)$ `// Alg. 48`

$x'_U, y'_U, x'_V, y'_V \leftarrow MoveBasisToE_{A'=0}(sk_2, A, x_U, y_U, x_V, y_V)$ `// see [35] for dual iso. evaluation`

$y_P, y_Q, A' \leftarrow$ `get_yP_yQ_A_B`(x_P, x_Q, x_R) `// Alg. 41`

$c_0, d_0, c_1, d_1 \leftarrow ph(x_P, y_P, x_Q, y_Q, x'_U, y'_U, x'_V, y'_V, A', 3)$ `// Alg. 55`

if $d_1 \pmod{3^{e_3}} \neq 0$ **then**

- $bit \leftarrow 0$
- $t_1 \leftarrow -d_0 \cdot d_1^{-1}$
- $t_2 \leftarrow -c_1 \cdot d_1^{-1}$
- $t_3 \leftarrow c_0 \cdot d_1^{-1}$

else

- $bit \leftarrow 1$
- $t_1 \leftarrow -d_1 \cdot d_0^{-1}$
- $t_2 \leftarrow c_1 \cdot d_0^{-1}$
- $t_3 \leftarrow -c_0 \cdot d_0^{-1}$

return $(bit, b_1, b_2, t_1, t_2, t_3, A, r_1, r_2)$ `// Encoded as in §1.2.10`

Algorithm 57: Computing compressed ciphertexts in the 2^{e_2}-torsion

function `CiphertextCompression_3`

Input: uncompressed points in ciphertext

Output: compressed points for ciphertext

`// See reference [36] for details on ciphertext computation and encoding`

return *compressed points for ciphertext*

Algorithm 58: Compute a kernel generator for the last 2^{e_2}-isogeny

function `CiphertextDecompression_2`

Input: Secret key $\mathrm{sk}_2 \in \mathbb{Z}_{2^{e_2}}$ and compressed ciphertext

Output: A kernel generator of the last 2^{e_2}-isogeny

`// See [36] for more details in ciphertext decompression computation`

return *kernel generator of last* 2^{e_2} *isogeny*

Algorithm 59: Compute a kernel generator for the last 3^{e_3}-isogeny

function `PublicKeyDecompression_3`

Input: Secret key $sk_3 \in \mathbb{Z}_{3^{e_3}}$ and compressed public key $\{bit, b_1, b_2, (t_1, t_2, t_3) \in (\mathbb{Z}_{3^{e_3}})^3, A, r_1, r_2\}$

Output: A kernel generator $(x', z') \in E_A[3^{e_3}]$ of the last 3^{e_3}-isogeny

$(x_1, y_1)_{P_1}, (x_2, y_2)_{P_2} \leftarrow BuildOrdinaryE3nBasis\_decompression(A, bit, b_1, b_2, r_1, r_2)$ `// Alg. 49`

if $bit = 0$ **then**

  $scal \leftarrow (t_1 + sk_3 \cdot t_3)(1 + sk_3 \cdot t_2))^{-1}$

  $x, z \leftarrow$ `Ladder3pt`$(scal, (x_1, x_2, x(P_1 - P_2)), (A : 1)))$ `// Alg. 8`

else

  $scal \leftarrow (t_1 + sk_3 \cdot t_2)(1 + sk_3 \cdot t_3))^{-1}$

  $x, z \leftarrow$ `Ladder3pt`$(scal, (x_2, x_1, x(P_2 - P_1)), (A : 1)))$ `// Alg. 8`

$(x', z') \leftarrow xTPLe\_fast(x, z, A/2, e_3)$ `// Alg. 46`

return (x', z')

Appendix D

Computing optimized strategies for fast isogeny and discrete logarithm computations

Algorithms 19 and 20 need to be parameterized by a computational strategy as described in Section 1.3.8. Any valid strategy, i.e. any sequence $(s_1, \ldots, s_{n-1})$ corresponding to a full binary tree, can be used without affecting the security of the protocol.

For the sake of efficiency, we recommend using the parameters specified in this section. They were generated by the algorithm below. The inputs to the algorithm are the strategy size n, which is one less than the number of leaves in the tree, the cost for a scalar multiplication step p and the cost for an isogeny computation and evaluation step q. Specifically, we use n_4, the size of the strategy for computations using the 2-torsion group, p_4 the cost of two `xDBL` operations, q_4 the cost of computation and evaluation of a 4-isogeny, i.e. of the functions `4_iso_curve` and `4_iso_eval`. Similarly, n_3 is the size of the strategy for computations using the 3-torsion group, p_3 the cost of a `xTPL` operation, and q_3 the cost of computation and evaluation of a 3-isogeny, i.e. of the functions `3_iso_curve` and `3_iso_eval`. We denote the respective strategies by S_4 and S_3, respectively.

Algorithm 60: Computing optimized strategy

function `compute_strategy`

  Input: Strategy size n, parameters $p, q > 0$

  Output: Optimal strategy of size n

$S \leftarrow [1 \to \epsilon]$

$C \leftarrow [1 \to 0]$

for $i = 2$ **to** $n + 1$ **do**

  Set $b \leftarrow \text{argmin}_{0<b<i}(C[i-b] + C[b] + bp + (i-b)q)$

  Set $S[i] \leftarrow b \,.\, S[i-b] \,.\, S[b]$

  Set $C[i] \leftarrow C[i-b] + C[b] + bp + (i-b)q$

return $S[n+1]$

D.1 Strategies for `SIKEp434`

D.1.1 Isogeny in the 2-torsion

$$
\begin{aligned}
\mathtt{n}_4 &= \mathtt{107}\\
\mathtt{p}_4 &= \mathtt{5633}\\
\mathtt{q}_4 &= \mathtt{5461}\\
\mathtt{S}_4 &= (48, 28, 16, 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 13,\\
&\quad 7, 4, 2, 1, 1, 2, 1, 1, 3, 2, 1, 1, 1, 1, 5, 4, 2, 1, 1, 2, 1, 1, 2, 1, 1, 1, 21, 12, 7, 4, 2, 1, 1, 2, 1,\\
&\quad 1, 3, 2, 1, 1, 1, 1, 5, 3, 2, 1, 1, 1, 1, 2, 1, 1, 1, 9, 5, 3, 2, 1, 1, 1, 1, 2, 1, 1, 1, 4, 2, 1, 1, 1, 2,\\
&\quad 1, 1)
\end{aligned}
$$

D.1.2 Isogeny in the 3-torsion

$$
\begin{aligned}
\mathtt{n}_3 &= \mathtt{136}\\
\mathtt{p}_3 &= \mathtt{5322}\\
\mathtt{q}_3 &= \mathtt{5282}\\
\mathtt{S}_3 &= (66, 33, 17, 9, 5, 3, 2, 1, 1, 1, 1, 2, 1, 1, 1, 4, 2, 1, 1, 1, 2, 1, 1, 8, 4, 2, 1, 1, 1, 2, 1, 1, 4, 2, 1, 1,\\
&\quad 2, 1, 1, 16, 8, 4, 2, 1, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 32,\\
&\quad 16, 8, 4, 3, 1, 1, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 16, 8, 4,\\
&\quad 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1)
\end{aligned}
$$

D.1.3 DLP in the subgroup of order 2^{e_2}

$$
\begin{aligned}
\mathtt{e}_2 &= \mathtt{216}\\
\mathtt{w}_2 &= \mathtt{4}\\
\mathtt{n}_2 &= \mathtt{53}\\
\mathtt{p} &= \mathtt{8}\\
\mathtt{q} &= \mathtt{3}\\
\mathtt{S}_2 &= (19, 13, 8, 5, 3, 2, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 3, 2, 1, 1, 1, 1, 1, 4, 3, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 6, 4,\\
&\quad 3, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1)
\end{aligned}
$$

D.1.4 DLP in the subgroup of order 3^{e_3}

$$
\begin{aligned}
\mathtt{e}_3 &= 137\\
\mathtt{w}_3 &= 3\\
\mathtt{n}_3 &= 45\\
\mathtt{p} &= 9\\
\mathtt{q} &= 3\\
\mathtt{S}_3 &= (18, 9, 6, 4, 3, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 3, 2, 1, 1, 1, 1, 1, 1, 6, 4, 3, 2, 1, 1, 1, 1, 1,\\
&\quad 1, 1, 1, 2, 1, 1, 1, 1)
\end{aligned}
$$

D.2 Strategies for SIKEp503

D.2.1 Isogeny in the 2-torsion

$$
\begin{aligned}
\mathtt{n}_4 &= 124\\
\mathtt{p}_4 &= 7490\\
\mathtt{q}_4 &= 7278\\
\mathtt{S}_4 &= (61, 32, 16, 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1,\\
&\quad 16, 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 29, 16,\\
&\quad 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 13, 8, 4, 2,\\
&\quad 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 5, 4, 2, 1, 1, 2, 1, 1, 2, 1, 1, 1)
\end{aligned}
$$

D.2.2 Isogeny in the 3-torsion

$$
\begin{aligned}
\mathtt{n}_3 &= 158\\
\mathtt{p}_3 &= 7189\\
\mathtt{q}_3 &= 7051\\
\mathtt{S}_3 &= (71, 38, 21, 13, 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 5, 4, 2, 1, 1, 2, 1, 1, 2, 1, 1, 1,\\
&\quad 9, 5, 3, 2, 1, 1, 1, 1, 2, 1, 1, 1, 4, 2, 1, 1, 1, 2, 1, 1, 17, 9, 5, 3, 2, 1, 1, 1, 1, 2, 1, 1, 1,\\
&\quad 4, 2, 1, 1, 1, 2, 1, 1, 8, 4, 2, 1, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 33, 17, 9, 5, 3, 2, 1, 1,\\
&\quad 1, 1, 2, 1, 1, 1, 4, 2, 1, 1, 1, 2, 1, 1, 8, 4, 2, 1, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 16, 8, 4,\\
&\quad 2, 1, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1)
\end{aligned}
$$

D.2.3 DLP in the subgroup of order 2^{e_2}

$$
\begin{aligned}
\mathtt{e_2} &= \mathtt{250} \\
\mathtt{w_2} &= \mathtt{5} \\
\mathtt{n_2} &= \mathtt{49} \\
\mathtt{p} &= \mathtt{10} \\
\mathtt{q} &= \mathtt{3} \\
\mathtt{S_2} &= (15, 10, 7, 5, 4, 3, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 3, 2, 1, 1, 1, 1, \\
&\quad 1, 1, 1, 5, 3, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1)
\end{aligned}
$$

D.2.4 DLP in the subgroup of order 3^{e_3}

$$
\begin{aligned}
\mathtt{e_3} &= \mathtt{159} \\
\mathtt{w_3} &= \mathtt{3} \\
\mathtt{n_3} &= \mathtt{52} \\
\mathtt{p} &= \mathtt{9} \\
\mathtt{q} &= \mathtt{3} \\
\mathtt{S_3} &= (19, 13, 8, 4, 3, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 3, 2, 1, 1, 1, 1, 1, 4, 3, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, \\
&\quad 6, 4, 3, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1)
\end{aligned}
$$

D.3 Strategies for `SIKEp610`

D.3.1 Isogeny in the 2-torsion

$$
\begin{aligned}
\mathtt{n_4} &= \mathtt{151} \\
\mathtt{p_4} &= \mathtt{10370} \\
\mathtt{q_4} &= \mathtt{10096} \\
\mathtt{S_4} &= (67, 37, 21, 12, 7, 4, 2, 1, 1, 2, 1, 1, 3, 2, 1, 1, 1, 1, 5, 3, 2, 1, 1, 1, 1, 2, 1, 1, 1, 9, 5, 3, 2, 1, \\
&\quad 1, 1, 1, 2, 1, 1, 1, 4, 2, 1, 1, 1, 2, 1, 1, 16, 9, 5, 3, 2, 1, 1, 1, 1, 2, 1, 1, 1, 4, 2, 1, 1, 1, 2, 1, \\
&\quad 1, 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 33, 16, 8, 5, 2, 1, 1, 1, 2, 1, 1, 1, 4, 2, 1, 1, 2, 1, \\
&\quad 1, 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 16, 8, 4, 2, 1, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 8, \\
&\quad 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1)
\end{aligned}
$$

D.3.2 Isogeny in the 3-torsion

$$
\begin{aligned}
\mathtt{n}_3 &= \mathtt{191}\\
\mathtt{p}_3 &= \mathtt{10084}\\
\mathtt{q}_3 &= \mathtt{9794}\\
\mathtt{S}_3 &= (86, 48, 27, 15, 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 7, 4, 2, 1, 1, 2, 1, 1, 3, 2, 1, 1, 1, 1, 12,\\
&\quad 7, 4, 2, 1, 1, 2, 1, 1, 3, 2, 1, 1, 1, 1, 5, 3, 2, 1, 1, 1, 1, 2, 1, 1, 1, 21, 12, 7, 4, 2, 1, 1, 2, 1, 1,\\
&\quad 3, 2, 1, 1, 1, 1, 5, 3, 2, 1, 1, 1, 1, 2, 1, 1, 1, 9, 5, 3, 2, 1, 1, 1, 1, 2, 1, 1, 1, 4, 2, 1, 1, 1, 2, 1,\\
&\quad 1, 38, 21, 12, 7, 4, 2, 1, 1, 2, 1, 1, 3, 2, 1, 1, 1, 1, 5, 3, 2, 1, 1, 1, 1, 2, 1, 1, 1, 9, 5, 3, 2, 1,\\
&\quad 1, 1, 1, 2, 1, 1, 1, 4, 2, 1, 1, 1, 2, 1, 1, 17, 9, 5, 3, 2, 1, 1, 1, 1, 2, 1, 1, 1, 4, 2, 1, 1, 1, 2, 1, 1,\\
&\quad 8, 4, 2, 1, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1)
\end{aligned}
$$

D.3.3 DLP in the subgroup of order 2^{e_2}

$$
\begin{aligned}
\mathtt{e}_2 &= \mathtt{305}\\
\mathtt{w}_2 &= \mathtt{5}\\
\mathtt{n}_2 &= \mathtt{60}\\
\mathtt{p} &= \mathtt{10}\\
\mathtt{q} &= \mathtt{3}\\
\mathtt{S}_2 &= (19, 13, 9, 6, 4, 3, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 3, 2, 1, 1, 1, 1, 1, 1, 4, 3, 2, 1, 1, 1, 1,\\
&\quad 1, 1, 1, 1, 1, 6, 4, 3, 2, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1)
\end{aligned}
$$

D.3.4 DLP in the subgroup of order 3^{e_3}

$$
\begin{aligned}
\mathtt{e}_3 &= \mathtt{192}\\
\mathtt{w}_3 &= \mathtt{3}\\
\mathtt{n}_3 &= \mathtt{63}\\
\mathtt{p} &= \mathtt{9}\\
\mathtt{q} &= \mathtt{3}\\
\mathtt{S}_3 &= (23, 13, 9, 6, 4, 3, 2, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 3, 2, 1, 1, 1, 1, 1, 1, 4, 3, 2, 1, 1, 1, 1, 1, 1,\\
&\quad 1, 1, 1, 9, 5, 3, 2, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 3, 2, 1, 1, 1, 1, 1, 1)
\end{aligned}
$$

D.4 Strategies for SIKEp751

D.4.1 Isogeny in the 2-torsion

$$
\begin{aligned}
\mathtt{n}_4 &= 185\\
\mathtt{p}_4 &= 14166\\
\mathtt{q}_4 &= 13810\\
\mathtt{S}_4 &= (80, 48, 27, 15, 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 7, 4, 2, 1, 1, 2, 1, 1, 3, 2, 1, 1, 1, 1,\\
&\quad 12, 7, 4, 2, 1, 1, 2, 1, 1, 3, 2, 1, 1, 1, 1, 5, 3, 2, 1, 1, 1, 1, 2, 1, 1, 1, 21, 12, 7, 4, 2, 1, 1,\\
&\quad 2, 1, 1, 3, 2, 1, 1, 1, 1, 5, 3, 2, 1, 1, 1, 1, 2, 1, 1, 1, 9, 5, 3, 2, 1, 1, 1, 1, 2, 1, 1, 1, 4, 2, 1,\\
&\quad 1, 1, 2, 1, 1, 33, 20, 12, 7, 4, 2, 1, 1, 2, 1, 1, 3, 2, 1, 1, 1, 1, 5, 3, 2, 1, 1, 1, 1, 2, 1, 1, 1,\\
&\quad 8, 5, 3, 2, 1, 1, 1, 1, 2, 1, 1, 1, 4, 2, 1, 1, 2, 1, 1, 16, 8, 4, 2, 1, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1,\\
&\quad 1, 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1)
\end{aligned}
$$

D.4.2 Isogeny in the 3-torsion

$$
\begin{aligned}
\mathtt{n}_3 &= 238\\
\mathtt{p}_3 &= 13898\\
\mathtt{q}_3 &= 13409\\
\mathtt{S}_3 &= (112, 63, 32, 16, 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1,\\
&\quad 1, 16, 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 31, 16,\\
&\quad 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 15, 8, 4, 2, 1,\\
&\quad 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 7, 4, 2, 1, 1, 2, 1, 1, 3, 2, 1, 1, 1, 1, 49, 31, 16, 8, 4, 2, 1, 1, 2,\\
&\quad 1, 1, 4, 2, 1, 1, 2, 1, 1, 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 15, 8, 4, 2, 1, 1, 2, 1, 1, 4, 2,\\
&\quad 1, 1, 2, 1, 1, 7, 4, 2, 1, 1, 2, 1, 1, 3, 2, 1, 1, 1, 1, 21, 12, 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1,\\
&\quad 1, 5, 3, 2, 1, 1, 1, 1, 2, 1, 1, 1, 9, 5, 3, 2, 1, 1, 1, 1, 2, 1, 1, 1, 4, 2, 1, 1, 1, 2, 1, 1)
\end{aligned}
$$

D.4.3 DLP in the subgroup of order 2^{e_2}

$$
\begin{aligned}
\mathtt{e}_2 &= 372\\
\mathtt{w}_2 &= 4\\
\mathtt{n}_2 &= 92\\
\mathtt{p} &= 8
\end{aligned}
$$

$$
\begin{aligned}
\mathtt{q} &= 3\\
\mathtt{S}_2 &= (34, 19, 13, 9, 6, 4, 3, 2, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 3, 2, 1, 1, 1, 1, 1, 1, 4, 3, 2, 1, 1, 1, 1, 1,\\
&\quad 1, 1, 1, 1, 6, 4, 3, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 12, 8, 5, 3, 2, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 3, 2,\\
&\quad 1, 1, 1, 1, 1, 4, 3, 2, 1, 1, 1, 1, 1, 1, 1, 1)
\end{aligned}
$$

D.4.4 DLP in the subgroup of order 3^{e_3}

$$
\begin{aligned}
\mathtt{e}_3 &= 239\\
\mathtt{w}_3 &= 3\\
\mathtt{n}_3 &= 79\\
\mathtt{p} &= 9\\
\mathtt{q} &= 3\\
\mathtt{S}_3 &= (28, 19, 13, 7, 4, 3, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 3, 1, 1, 1, 1, 1, 4, 3, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 6, 4, 3, 2, 1,\\
&\quad 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 9, 6, 4, 3, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 3, 2, 1, 1, 1, 1, 1, 1)
\end{aligned}
$$

Appendix E

Changes made in the 2nd round

The main differences between the first round and second round SIKE submissions are as follows.

- Two new parameter sets have been added: `SIKEp434` (§1.6.1) and `SIKEp610` (§1.6.5).
- One parameter set (`SIKEp964`) has been removed.
- Security categories for parameter sets have been adjusted upward. Chapter 5 presents the rationale for this change.
- The starting curve has been changed from $A = 0$ to $A = 6$. §1.3.2 presents the rationale for this change.
- An additional implementation including public key compression has been added (§1.5, §2.3).

Appendix F

Changes made in the 3rd round

The main differences between the second round and third round SIKE submissions are as follows.

- Optimized ARMv8 implementations are now available for all parameter sets.
- Optimized Cortex M4 and VHDL implementations are now available for all uncompressed parameter sets.
- New (space and time) optimizations for compressed SIKE have been added; see Appendix C.
- New pre-computation tables for discrete logarithms have been added, reducing static library sizes for compressed SIKE by 80-90%; see Appendix D.
- Appendix C and Appendix D in the previous version have been swapped.

Appendix G

Notation

ℓ, m	Integers $\ell, m \in \{2, 3\}$ such that $\ell \neq m$
e_ℓ	The index of ℓ in the degree of the ℓ-power isogeny
sk_ℓ	The secret key corresponding to points in the ℓ^{e_ℓ}-torsion
pk_ℓ	The public key corresponding to points in the ℓ^{e_ℓ}-torsion
ϕ_ℓ	The secret ℓ^{e_ℓ}-isogeny corresponding to sk_ℓ
P_ℓ	A point of exact order ℓ^{e_ℓ} in $E_0(\mathbb{F}_{p^2}) \setminus E_0(\mathbb{F}_p)$, such that the order-$\ell^{e_\ell}$ Weil pairing, $e_{\ell^{e_\ell}}(P_\ell, Q_\ell)$, has exact order ℓ^{e_ℓ}
Q_ℓ	A point of exact order ℓ^{e_ℓ} in $E_0(\mathbb{F}_p)$
R_ℓ	The point defined as $R_\ell = Q_\ell - P_\ell$
$\texttt{isogen}_\ell$	The algorithm that computes public keys — see §1.3.6
$\texttt{isoex}_\ell$	The algorithm that establishes shared keys — see §1.3.7
E_a	The Montgomery curve defined by $E_a/\mathbb{F}_{p^2} : y^2 = x^3 + ax^2 + x$
E_a	The Montgomery curve defined by $E_a/\mathbb{F}_{p^2} : y^2 = x^3 + ax^2 + x$
p	The prime field characteristic defined as $p = 2^{e_2}3^{e_3} - 1$
x_P	The x-coordinate of the point P
y_P	The y-coordinate of the point P
$\mathcal{K}_2$	The keyspace corresponding to points in the 2^{e_2}-torsion
$\mathcal{K}_3$	The keyspace corresponding to points in the 3^{e_3}-torsion
$\texttt{N}_p$	The number of bytes used to represent elements in $\mathbb{F}_p$
$\texttt{N}_{\mathrm{sk}}$	The number of bytes used to represent secret keys
$\texttt{N}_{pk}$	The number of bytes used to represent public keys
$\mathbb{Z}$	The ring of integers
$\mathbb{F}_q$	The finite field with q elements
$\bar{\mathbb{F}}_q$	The algebraic closure of the finite field with q elements
$\mathbb{F}_p$	The prime field with p elements
$\mathbb{F}_{p^2}$	The quadratic extension field $\mathbb{F}_{p^2}$, constructed over the prime field $\mathbb{F}_p$ as $\mathbb{F}_{p^2} = \mathbb{F}_p(i)$ with $i^2 + 1 = 0$
$\mathbb{P}^n(K)$	The projective space of dimension n over the field K
Q_2	A point of exact order 2^{e_2} in $E_0(\mathbb{F}_p)$
P_2	A point of exact order 2^{e_2} in $E_0(\mathbb{F}_{p^2}) \setminus E_0(\mathbb{F}_p)$, such that the order-$2^{e_2}$ Weil pairing, $e_{2^{e_2}}(P_2, Q_2)$, has exact order 2^{e_2}

R_2	The point defined as $R_2 = Q_2 - P_2$
Q_3	A point of exact order 3^{e_3} in $E_0(\mathbb{F}_p)$
P_3	A point of exact order 3^{e_3} in $E_0(\mathbb{F}_{p^2}) \setminus E_0(\mathbb{F}_p)$, such that the order-$3^{e_3}$ Weil pairing, $e_{3^{e_3}}(P_3, Q_3)$, has exact order 3^{e_3}
R_3	The point defined as $R_3 = Q_3 - P_3$
SIKE	Supersingular isogeny key encapsulation
SIDH	Supersingular isogeny Diffie–Hellman
PKE	Public-key encryption
KEM	Key encapsulation mechanism
IND-CPA	Indistinguishability under chosen plaintext attack
IND-CCA	Indistinguishability under chosen ciphertext attack
RSA	Rivest–Shamir–Adleman (cryptosystem)
ECC	Elliptic curve cryptography
$\oplus$	The binary exclusive or (XOR) of equal-length bitstrings
$\mathcal{I}$	An oracle computing isogenies of degree $\ell^{e_\ell/2}$
$\mathcal{B}$	A block cipher
G^C	The number of gates of a classical circuit
G^Q	The number of gates of a quantum circuit
D^C	The depth of a classical circuit
D^Q	The depth of a quantum circuit
`AES`	Advanced Encryption Standard
`PKE`	An isogeny-based public-key encryption scheme
`KEM`	An isogeny-based key encapsulation mechanism
`Gen`	Key generation algorithm for `PKE`
`Enc`	Encryption algorithm for `PKE`
`Dec`	Decryption algorithm for `PKE`
`KeyGen`	Key generation algorithm for `KEM`
`Encaps`	Encapsulation algorithm for `KEM`
`Decaps`	Decapsulation algorithm for `KEM`
F	A random oracle
G	A random oracle
H	A random oracle
`SHAKE256`	A customizable extendable-output function standardized by NIST
c_0	First part of an encapsulation of `KEM`
c_1	Second part of an encapsulation of `KEM`