Supersingular Isogeny Key Encapsulation
November 30, 2017

Principal submitter: David Jao

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

Telephone: +1-519-888-4567 x32493

Postal address:
David Jao
Combinatorics and Optimization
University of Waterloo
200 University Ave. W
Waterloo, Ontario N2L 3G1
Canada

Auxiliary submitters:

- Reza Azarderakhsh, Florida Atlantic University, USA
- Matthew Campagna, Amazon Inc., USA
- Craig Costello, Microsoft Research, USA
- Luca De Feo, UVSQ and Inria, Université de Paris-Saclay, France
- Basil Hess, InfoSec Global, Canada
- Amir Jalali, Florida Atlantic University, USA
- Brian Koziel, Texas Instruments, USA
- Brian LaMacchia, Microsoft Research, USA
- Patrick Longa, Microsoft Research, USA
- Michael Naehrig, Microsoft Research, USA
- Joost Renes, Radboud University, The Netherlands
- Vladimir Soukharev, InfoSec Global, Canada
- David Urbanik, University of Waterloo, Canada

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

Owner: Same as submitters.

Signature: [illegible]

Contents

1 The SIKE protocol specification **1**
1.1 Mathematical Foundations 1
1.2 Data types and conversions 6
1.3 Detailed protocol specification 10
1.4 Symmetric primitives 17
1.5 Parameter sets 17

2 Detailed performance analysis **25**
2.1 Reference implementation 26
2.2 Optimized and x64 assembly implementations 27
2.3 64-bit ARM assembly implementation 29
2.4 VHDL hardware implementation 30

3 Known Answer Test values **33**

4 Expected security strength **34**
4.1 Security 34
4.2 Other attacks 35
4.3 Security proofs 36

5 Analysis with respect to known attacks **38**
5.1 Security levels 38
5.2 Detailed analysis 39
5.3 Side-channel attacks 40

6 Advantages and Limitations **42**

A Explicit algorithms for `isogen`$_\ell$ and `isoex`$_\ell$: Optimized implementation **48**

B Explicit algorithms for `isogen`$_\ell$ and `isoex`$_\ell$: Reference implementation **58**

C Computing optimized strategies for fast isogeny computation **67**

C.1 Strategies for `SIKEp503` 68

C.2 Strategies for `SIKEp751` 68

C.3 Strategies for `SIKEp964` 69

D Notation **71**

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 [19], and subsequently improved in various ways by numerous authors [6, 7, 9, 25]. 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 D.

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 $r \in \mathbb{F}_{p^2}$ such that $r^2 = s$, then we define $\sqrt{s}$ as $t \in \{-r, r\}$ where $t = \alpha + \beta \cdot i$ with $\alpha, \beta \in \mathbb{F}_p$, such that $\alpha_0 = 0$, where $\alpha = \sum_{j=0}^{\lceil \log_2 p \rceil} \alpha_j 2^j$ in binary representation (i.e. $\alpha_j \in \{0, 1\}$ for all j).

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 $\text{char}(\mathbb{F}_q) \neq 2$). 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.

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 $\text{ord}(P)$ 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 [27] 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 [39] 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 4 and 3.

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

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

Observe that A' only depends on x_4. 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 $\ker(\phi_4)$, 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 12 of Appendix A.

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

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

The new coefficient A' only depends on A and x_3. 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 $\ker(\phi_3)$, then $\phi_3\colon (x_P, y_P) \mapsto (x_{\phi_3(P)}, y_{\phi_3(P)})$, and this can be computed as

$$(x_{\phi_3(P)}, y_{\phi_3(P)}) = \left(\frac{x_P(x_Px_3 - 1)^2}{(x_P - x_3)^2}, \quad y_P \cdot \frac{(x_Px_3 - 1)(x_P^2x_3 - 3x_Px_3^2 + x_P + x_3)}{(x_P - x_3)^3} \right).$$

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 14 of Appendix A.

The SIKE protocol defines secret keys from two separate key spaces, $\mathcal{K}_2$ and $\mathcal{K}_3$ (cf. §1.3.8). A secret key sk defines a subgroup H of $E(\mathbb{F}_q)$, which in turn defines an isogeny $\phi_{\mathrm{sk}}\colon E \to E/H$. 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 $\{E/H, \phi_{\mathrm{sk}}(P), \phi_{\mathrm{sk}}(Q)\}$. 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 $\{x_{\phi_{\mathrm{sk}}(P)}, x_{\phi_{\mathrm{sk}}(Q)}, x_{\phi_{\mathrm{sk}}(P-Q)}\}$ of three points under the isogeny. It is possible to convert between representations using the methods given in [7]. For example, the Montgomery curve coefficient A of E/H can be recovered by the three x-coordinates of a public key $\{x_{\phi_{\mathrm{sk}}(P)}, x_{\phi_{\mathrm{sk}}(Q)}, x_{\phi_{\mathrm{sk}}(P-Q)}\}$ using the equation

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

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

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

and

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

and if

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

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

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 $A = \frac{(1 - x_P x_Q - x_P x_R - x_Q x_R)^2}{4 x_P x_Q x_R} - x_P - x_Q - x_R$ 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 $\mathbb{F}_q$, 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 $M = M_1 M_2 M_3$, 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 $a = a_{\text{mlen}-1}2^{8(\text{mlen}-1)} + a_{\text{mlen}-2}2^{8(\text{mlen}-2)} + \cdots + a_1 2^8 + a_0$ represented in base 2^8.
2. For $0 \le i < \text{mlen}$, set $M_i = a_i$.
3. Form $M = M_0 M_1 \ldots M_{\text{mlen}-1}$.
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 an octet strings M_1, M_2, M_3 respectively, each of length $2N_p$ (cf. §1.2.8).
2. Form $M = M_1 M_2 M_3$.
3. 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 specification below does not employ point compression. Users seeking the compression of public keys described in [1, 6] 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 §1.3.4. The IND-CPA PKE scheme is defined in §1.3.9. The subsequent IND-CCA KEM is defined in §1.3.10. 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

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

with $\#E_0(\mathbb{F}_{p^2}) = (2^{e_2}3^{e_3})^2$ and j-invariant equal to $j(E_0) = 1728$. This is the special instance of the Montgomery curve $By^2 = x^3 + Ax^2 + x$, where $A = 0$ and $B = 1$.

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

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

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

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

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

The points P_2, Q_2, P_3, Q_3 are determined according to the following procedure:

- Q_2 is first specified by setting $x_0 = 1 \in \mathbb{F}_p$ and, if necessary, incrementing $x_0 \leftarrow x_0 + 1$ until both $x_0^3 + x_0$ is a square in $\mathbb{F}_p$, and the point $(x_0, y_0) \in E_0(\mathbb{F}_p)$ with $y_0 = \sqrt{x_0^3 + x_0}$ is such that $(x_{Q_2}, y_{Q_2}) = [3^{e_3}](x_0, y_0)$ has exact order 2^{e_2}. The point Q_2 is then defined as $Q_2 = (x_{Q_2}, y_{Q_2})$.
- P_2 is then specified by setting $x_0 = i + 1$ and, if necessary, incrementing $x_0 \leftarrow x_0 + 1$ until $x_0^3 + x_0$ is a square in $\mathbb{F}_{p^2}$, the point $(x_0, y_0) \in E_0(\mathbb{F}_{p^2})$ with $y_0 = \sqrt{x_0^3 + x_0}$ is such that $(x_{P_2}, y_{P_2}) = [3^{e_3}](x_0, y_0)$ has exact order 2^{e_2}, and the order-2^{e_2} Weil pairing of Q_2 and (x_{P_2}, y_{P_2}) has full order. The point P_2 is then defined as $P_2 = (x_{P_2}, y_{P_2})$.
- Q_3 is specified by setting $x_0 = 1 \in \mathbb{F}_p$ and, if necessary, incrementing $x_0 \leftarrow x_0+1$ until both $x_0^3+x_0$ is a square in $\mathbb{F}_p$, and the point $(x_0, y_0) \in E_0(\mathbb{F}_p)$ with $y_0 = \sqrt{x_0^3 + x_0}$ is such that $(x_{Q_3}, y_{Q_3}) = [2^{e_2}](x_0, y_0)$ has exact order 3^{e_3}. The point Q_3 is then defined as $Q_3 = (x_{Q_3}, y_{Q_3})$.
- P_3 is then specified by setting $x_0 = i + 1$ and, if necessary, incrementing $x_0 \leftarrow x_0 + 1$ until $x_0^3 + x_0$ is a square in $\mathbb{F}_{p^2}$, the point $(x_0, y_0) \in E_0(\mathbb{F}_{p^2})$ with $y_0 = \sqrt{x_0^3 + x_0}$ is such that $(x_{P_3}, y_{P_3}) = [2^{e_2}](x_0, y_0)$ has exact order 3^{e_3}, and the order-3^{e_3} Weil pairing of Q_3 and (x_{P_3}, y_{P_3}) has full order. The point P_3 is then defined as $P_3 = (x_{P_3}, y_{P_3})$.

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 [7]), 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 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 [7, 9], 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 [39], 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 $\phi_i\colon (x, —) \mapsto (f_i(x), —)$.

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.

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

A supersingular isogeny key pair consists of a secret key sk_ℓ, 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

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

   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 $x_S \leftarrow f_i(x_S)$;
    (d) Set $(x_1, x_2, x_3) \leftarrow (f_i(x_1), f_i(x_2), f_i(x_3))$;
4. Output $\mathrm{pk}_\ell = (x_1, x_2, x_3)$.
```

1.3.6 Establishing shared keys: `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 $\mathrm{pk}_m$ 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 $\ell$-isogeny
            $\phi_i : E'_i \to E'$
            $(x, —) \longmapsto (f_i(x), —)$
        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).
```

1.3.7 Optimized `isogen`$_\ell$ and `isoex`$_\ell$

The algorithms `isogen`$_\ell$ and `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 [9], 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[1]. If we draw such trees so that all nodes lie within a triangular region of a hexagonal lattice, with all

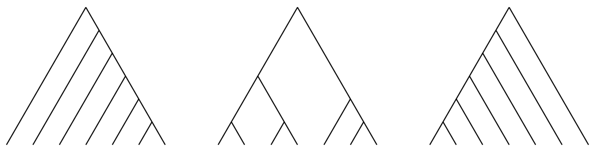
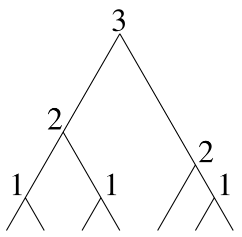


Figure 1.1: Three computational strategies of size $e_\ell - 1 = 6$. The simple approach used in Sections 1.3.5 and 1.3.6 corresponds to the leftmost strategy.

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 [9, §4] for a more formal definition.

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.



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

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

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:

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

$$\begin{aligned}\phi\colon E &\to E'\\ (x, —) &\longmapsto (f(x), —)\end{aligned}$$

   such that $\ker\phi = \langle S\rangle$;
   (b) Return $(E', f(x_1), f(x_2), \dots)$;
2. Let $n = s_1$;

[1]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.

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 $\mathtt{isoex}_\ell$ to obtain the same speedup. Remark that the simple algorithms of Sections 1.3.5 and 1.3.6 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:

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

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 $(s_1, \dots, s_{t-1})$ 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 $L = (s_2, \dots, s_{t-n})$ and $R = (s_{t-n+1}, \dots, s_{t-1})$;
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.

1.3.8 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 §1.3.9 and §1.3.10).

Let $\mathtt{N}_{\mathrm{sk}_2} = \lceil e_2/8 \rceil$. Secret keys sk_2 correspond to integers in the range $\{0, 1, \dots, 2^{e_2} - 1\}$, 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 $\{0, 1, \dots, 2^s - 1\}$, encoded as an octet string of length $\mathtt{N}_{\mathrm{sk}_3}$ using `itoos` (cf. §1.2.6). The corresponding keyspace is denoted $\mathcal{K}_3$.

1.3.9 Public-key encryption

Algorithm 1 defines a public-key encryption scheme $\mathtt{PKE} = (\mathtt{Gen}, \mathtt{Enc}, \mathtt{Dec})$ [9, §3.3]. The two keyspaces $\mathcal{K}_2$ and $\mathcal{K}_3$ are defined in 1.3.8. 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 $\mathtt{Enc}$ 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).

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

```
function Gen
    Input: ()
    Output: $(\mathrm{pk}_3, \mathrm{sk}_3)$
1   $\mathrm{sk}_3 \leftarrow_R \mathcal{K}_3$
2   $\mathrm{pk}_3 \leftarrow \mathtt{isogen}_3(\mathrm{sk}_3)$
3   return $(\mathrm{pk}_3, \mathrm{sk}_3)$

function Enc
    Input: $\mathrm{pk}_3, m \in \mathcal{M}$
    Output: $(c_0, c_1)$
4   $\mathrm{sk}_2 \leftarrow_R \mathcal{K}_2$
5   $c_0 \leftarrow \mathtt{isogen}_2(\mathrm{sk}_2)$
6   $j \leftarrow \mathtt{isoex}_2(\mathrm{pk}_3, \mathrm{sk}_2)$
7   $h \leftarrow F(j)$
8   $c_1 \leftarrow h \oplus m$
9   return $(c_0, c_1)$

function Dec
    Input: $\mathrm{sk}_3, (c_0, c_1)$
    Output: $m$
10  $j \leftarrow \mathtt{isoex}_3(c_0, \mathrm{sk}_3)$
11  $h \leftarrow F(j)$
12  $m \leftarrow h \oplus c_1$
13  return $m$
```

1.3.10 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 [18] to the PKE defined in §1.3.9. We slightly modify this transformation by including pk_3 in the input to G (as in [3]), 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 1.3.8. 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.

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);
```

Algorithm 2: KEM = (KeyGen, Encaps, Decaps)

```
function KeyGen
    Input: ()
    Output: $(s, \mathrm{sk}_3, \mathrm{pk}_3)$
1   $\mathrm{sk}_3 \leftarrow_R \mathcal{K}_3$
2   $\mathrm{pk}_3 \leftarrow \mathtt{isogen}_3(\mathrm{sk}_3)$
3   $s \leftarrow_R \{0,1\}^n$
4   return $(s, \mathrm{sk}_3, \mathrm{pk}_3)$

function Encaps
    Input: $\mathrm{pk}_3$
    Output: $(c, K)$
5   $m \leftarrow_R \{0,1\}^n$
6   $r \leftarrow G(m \,\|\, \mathrm{pk}_3)$
7   $(c_0, c_1) \leftarrow \mathtt{Enc}(\mathrm{pk}_3, m; r)$
8   $K \leftarrow H(m \,\|\, (c_0, c_1))$
9   return $((c_0, c_1), K)$

function Decaps
    Input: $(s, \mathrm{sk}_3, \mathrm{pk}_3), (c_0, c_1)$
    Output: $K$
10  $m' \leftarrow \mathtt{Dec}(\mathrm{sk}_3, (c_0, c_1))$
11  $r' \leftarrow G(m' \,\|\, \mathrm{pk}_3)$
12  $c_0' \leftarrow \mathtt{isogen}_2(r')$
13  if $c_0' = c_0$ then
14      $K \leftarrow H(m' \,\|\, (c_0, c_1))$
15  else
16      $K \leftarrow H(s \,\|\, (c_0, c_1))$
17  return $K$
```

The public key pk is given by pk_3. The secret key sk consists of the concatenation of s, sk_3 and pk_3 [2]. The ciphertext ct consists of the concatenation of c_0 and c_1. Finally, the shared secret ss is given by K.

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 cSHAKE256 as specified by NIST in [21].

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 cSHAKE256, taking $m \,\|\, \mathrm{pk}_3$ as the input, requesting e_2 output bits. In the notation of [21], this means $G(m \,\|\, \mathrm{pk}_3) = \mathtt{cSHAKE256}(m \,\|\, \mathrm{pk}_3, e_2, \texttt{""}, 0)$, where the function-name bit string is left empty, using the value 0 for the customization bit string. The value n corresponds to $n \in \{192, 256, 320\}$.

The function F is used as a key derivation function on the j-invariant during public key encryption and is computed as $F(j) = \mathtt{cSHAKE256}(j, n, \texttt{""}, 2)$ using the notation of [21], where the requested output consists of n bits, the function-name bit string is left empty and the value 2 is used for the customization bit string. Again, the value n corresponds to $n \in \{192, 256, 320\}$.

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 $\mathtt{cSHAKE256}(m \,\|\, c, k, \texttt{""}, 1)$ with $m \,\|\, c$ as the input and the value 1 for the customization bit string. The value k corresponds to the number of bits of classical security, i.e., $k \in \{128, 192, 256\}$.

1.5 Parameter sets

[2]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.

This section presents parameter sets for three different levels of security. The concrete security of these parameter sets is discussed in §5. For $w \in \{8, 12, 16\}$, the underlying prime fields with $p = 2^{e_2}3^{e_3} - 1$ were determined such that $\lceil \log_2 p \rceil \le 64w$, i.e., such that the prime does not exceed $64w$ bits. For each w, these primes were chosen with the aim of maximizing the overall security while achieving a close balance on both sides of the protocol, i.e., maximizing the value of $\min\{e_2, \log_2 3^{e_3}\}$ (see §5) such that $2^{e_2} \approx 3^{e_3}$.

The three sets of parameters are `SIKEp503`, `SIKEp751` and `SIKEp964`, 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 $x_{R_3} = x_{R_3,0} + x_{R_3,1} \cdot i$.

1.5.1 SIKEp503

```
   p  =  00000004 066F5418 11E1E604 5C6BDDA7 7A4D01B9 BF6C87B7 E7DAF130
         85BDA221 1E7A0ABF 809FFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
         FFFFFFFF FFFFFFFF FFFFFFFF
  e2  =  000000FA
  e3  =  0000009F
xQ20  =  00097453 912E12F3 DAF32EEF FD618BD9 3D3BBBF3 99137BD3 9858CADE
         FAE382E4 2D6E60A6 2FD62417 AD61A14B 60DB2612 5273EC98 0981325D
         86E55C45 E3BB46B1
xQ21  =  00000000
yQ20  =  0009B666 40A4CC79 F82B68D7 26092338 12DF76E8 B0422EF3 527A1F2A
         9915EFF1 6E094004 0DF4A15A 84A5ACF0 24FC2ED8 A50102A7 31E8D20D
         033B4803 5B63DD62
yQ21  =  00000000
xP20  =  001F6D52 A7563BB9 356B98A1 16A0CA97 75DBB738 2EB29E24 E45299D8
         939959EA EEB47FF3 113F6088 2D12103E 4B8B8CD2 B97DA146 57AE8C12
         8BE82209 D2DDFCA9
xP21  =  002D44C3 FAD24E4C BDDC8A2D 9DE336A9 2A9912EE 6D09E2DD 5C33AB26
         D60A268A C91F38E1 AF4C2D5B FA2B87DD 55C8CA60 19C6B0C0 8ED92B5A
         EB6C65A8 E06E53E9
yP20  =  003C9F7C 397283C0 871F78D9 F74ECC0A 8F89579C CBEF8FE6 0D07338A
         F0A0322E 3F0C66CA 826AA5BF 85EB5366 6C272C8E AEC9B808 B3B78E64
         22330617 AC23D6F2
yP21  =  0038222A E95DA234 ABD1B90F D897C2E2 E7995B2C 0006DC92 CC079B7C
         60C94DCA E9961CC7 A4BAEAC9 D294F6D5 760D4D65 4821193A E92AD42A
```

```
              C0047ADE 55C343FC
xR20    =     00173775 ECBEC79C 78FD1ED5 FE36075A ACE1F53F 8FFB97D2 A7E80DFC
              2875E77E C72D1D4A 99E13353 EC9D147B ADD96126 948A72B3 0BDD7CEB
              AD7B54F8 DDB5CD06
xR21    =     0002EAA2 24DDDA14 9BBBB908 9D2B2C47 1D068ECA 203465CE 97DBC1C8
              ED0EBB0F F90E4FBE 7E266BBA 99CBAE05 1797B4D3 5D28E36C 1B1CB994
              AEEED1CB 59FE5015
xQ30    =     001E7D6E BCEEC9CF C47779AF FD696A88 A971CDF3 EC61E009 DF55CAF4
              B6E01903 B2CD1A12 089C2ECE 106BDF74 5894C14D 7E39B699 7F70023E
              0A23B4B3 787EF08F
xQ31    =     00000000
yQ30    =     002EC0AA EF9FBBDD 75FBDA11 DA19725F 79E842FB C355071F D631C1CD
              F90E08E6 01929FAE C5DAEB0D 96BBB4AD 50FC7C8A D47064F0 5C06DC5D
              4AAE61CC CEFF1F26
yQ31    =     00000000
xP30    =     0021B709 8B640A01 D88708B7 29837E87 0CFF9DF6 D4DF86D8 6A7409F4
              1156CB5F 7B851482 2730940C 9B51E0D9 821B0A67 DD7ED98B 9793685F
              A2E22D6D 89D66A4E
xP31    =     002F37F5 75BEBBC3 3851F75B 7AB5D89F C3F07E4D F3CC5234 9804B8D1
              7A17000A 42FC6C57 34B9FCFD E669730F 3E8569CE B53821D3 E8012F7F
              391F5736 4F402909
yP30    =     0000078F 8A30AB36 B301BDF6 72D9E351 8AF741F8 227CC95A 9F351B99
              623A826D E3F8D90D D6ED42FF 298E394E 77B7AEFE E6010CDF 34A7DE9F
              9E239B10 3E7B3EEE
yP31    =     0037F3C6 00488EBB 6B11462C 4CAFC41C D5DC611A 9B0C804E 3BF50D6D
              8F75C4E7 A136E29E 00D80EB8 653CA830 F2AED61D 04F9F3A8 317F7916
              E016F273 3B828AC0
xR30    =     000D4818 D120A24A BF48DB51 D129E6B1 F24F4BBB 2C16FACC 0C8C0632
              3EEEC2FA 5B5E887E 17226417 B1907310 BFE6784F DEBBAC8C 2A9ABBE7
              53F52259 A7B7D70E
xR31    =     0019E75F 0F03312D 22CBBF15 3747525D 89E5155B ABB8BF0C 130CB567
              CA532F69 AAF57EA7 682B9957 021D9041 4433ABBE EDC233E9 08218578
              1C16724C 8C356777
```

1.5.2 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  =  00003E82 027A38E9 429C8D36 FF46BCC9 3FA23F89 F6BE06D2 B1317AD9
         04386217 83FDB7A4 AD3E83E8 6CAE096D 5DB822C9 8E561E00 8FA0E3F3
         B9AC2F40 C56D6FA4 A58A2044 9AF1F133 5661D14A B7347693 63264608
         6CE3ACD5 4B0346F5 CCE233E9
xQ21  =  00000000
yQ20  =  00003BBF 8DCD4E7E B6236F5F 598D56EB 5E15915A 755883B7 C331B043
         DA010E6A 163A7421 DFA8378D 1E911F50 BF3F721A 8ED5950D 80325A8D
         0F147EF3 BD0CFEC5 236C7FAC 9E69F7FD 5A99EBEC 3B5B8B00 0F8EEA73
         70893430 12E0D620 BFB341D
yQ21  =  00000000
xP20  =  00005492 1C31F0DC 9531CB89 0FC5EC66 DF2E7F0D 55761363 C6E375DA
         69B0682C ABE5C0FF FCBE6E1A D46563F0 42FA06B9 F207FCF3 CDD26736
         52828FF5 0C3F7B75 5C0BE072 950D16CA 747C1467 75C0267A 401FFC73
         8B03A49E 9A36B395 72AFB363
xP21  =  00002884 9BC0D81E 01993137 A5B63D6E 633C4E97 AB4FF118 CCF63DFE
         623092AC 86B6D4A9 B751797C BA1A1775 00E9EB5A F7852B7D F02C3348
         44D652EF C4729178 A1DBAD8C A47BB7E7 57C6D43B 799811A6 3BEBE649
         C18101F0 3AD752CD CD73BF66
yP20  =  00001961 19D87272 DC3AA722 3476C8C3 269D48CA EFAE692F 68DCF2D6
         E1BEB5B9 7525D502 6C157C7C 740B41AD E80A8CF2 E1E0B37E 5F5FD4ED
         88235BF7 404BE391 89C137E2 1C035EF6 339D7FAC BA38E72D 69043710
         E76266A5 FC14EFB9 5E5FBC7C
yP21  =  0000D3AC 09A67D59 CC8D78B0 FA6681AE 78BDF0C8 F558E386 6005E435
         5B0B1993 18D9CDD6 7C0A7DB2 34F9EA1E C4C5F1E5 9168B7DB D14281F0
         9E8DF904 A3D574CA D526DC5A 3667490A DE1A4C13 B09F7B11 5C4E488F
         D4DD5F76 70B58973 22AD41D
xR20  =  000022A0 B5A35A2B 0C56135A 7CEC5CFB 97964A7C 6226FE90 9F374362
         A8ECA3AB 14A1B7B0 C87AC875 DCE5888D 83B623BF 0011A4AC 138F62EF
         6B2D2D84 F636548A 9F920F23 8336E5A3 6E45E405 5940E3C9 4385B8FC
```

```
              53743964 32EEF2AE 178CEFDD
xR21    =     00000F9C 4AFCDA80 9C3358B0 96B250C6 9B20310F DF2EF631 711AA4EF
              EC49A4E7 6483F320 B793F2EB C63365EE D14AA3F6 EA33FEB5 6796F011
              BA6C6DFB 4D0A00AA C4D27866 46D914AD 026CBB4A 592EC74B 5485372E
              51382D44 528DD491 B83D9547
xQ30    =     00002F1D 80EF06EF 960A01AB 8FF409A2 F8D5BCE8 59ED725D E145FE2D
              525160E0 A3AD8E17 B9F9238C D5E69CF2 6DF23742 9BD37786 59023B9E
              CB610E30 288A7770 D3785AAA A4D646C5 76AECB94 B919AEED D9E1DF56
              6C1D26D3 76ED2325 DCC93103
xQ31    =     00000000
yQ30    =     00000127 A46D082A 1ACAF351 F09AB55A 15445287 ED1CC55D C3589212
              3951D4B6 E302C512 9C049EEB 399A6EDB 2EEB2F9B 0A94F06C DFB3EADE
              76EBA0C8 419745E9 7D12754F 00E898A3 15B52912 2CFE3CA6 BBC6BAF5
              F6BA40BB 91479226 A0687894
yQ31    =     00000000
xP30    =     000005FD 1A3C4DD0 F6309741 96FED351 9152BC70 98B9E2B1 21ECA46B
              D10A5CC9 F4BCC6C6 89B8E4C0 63B37980 75FCEE6E DAA9EB10 8B3CD004
              95CF04DD 8CE4A08F BE685A12 7D40E45F 4CF45098 A578DEB4 43686993
              94C43BFC 9BC5E000 52F78E8D
xP31    =     00002B88 A03360B3 38954773 2C9140C0 5DEA6516 881FE108 211BE887
              CC43FCB8 0C06A1D8 6FF5457D 3BB7DB93 6394EC33 821AA393 33A60AF8
              4B537974 CFA0BA82 87D699D2 BF79BA55 9026C64A 6ED61050 1D2357C1
              0B9A6C8F 83742492 2275ACBF
yP30    =     000053B5 5053E3F0 4FC315EF B1B7B2C4 AFCB4FEF 12CE744A F3B243C6
              E6B1417E 94A78D49 80DDE181 89646492 3E01AACC 3DA040A0 747CA675
              54A35268 4DA207C4 9022D930 732DF6BD 0BF37E1F 5C169176 69A70F88
              059C1C73 9A79D7CF A0C529D9
yP31    =     0000044E 44196909 252ECD7B 91643238 15294F02 AED22C4E 4EB43D2C
              E2BC5F29 EB575D45 CA8B6B4C 4242E369 AE3A1EFC 844E9D1C 57B0AE33
              74BC2CED AD16B0C6 99158332 E2D9AB3F 0025C034 8C5F70FD C4DD7C48
              65E64B8B 843F03D8 07447D5E
xR30    =     0000077B 3BB69009 428A327D 43CA6016 9715F547 454F88CD 017B32DF
              58A7252C 2B3C3D00 D52CCD31 33D54041 D8BCAEA2 91F20572 02328712
              CD395575 CD7CCD3C E70C0A1E BF633BA9 46559458 878F41F9 FDD1727E
              2C31125B 2FE5B713 06704829
xR31    =     00006D91 393A57DB F47FD6DC F841F17E CD719CAE 1D33C683 2A75B0F1
              68855BCC 38D2A479 2DFF9BC8 6DEACA10 B1AA808D 539B167D 73BBA321
```

68687FA3 F85AE93A 1ADDE5BD 1FD5B681 DCC6C344 54D44969 76C22D80
C95E42B1 2576FC0F B4074B9F

1.5.3 SIKEp964

p	=	00000008 6B5BFF76 43C64F7A 10028248 AD4FC4B1 50CBAA75 A2A1FA44 CBAB2451 35469BAB 093F2B8D AD5281E7 E56EF6AA 57A94749 ABB38EAB 467ACDE5 451CD4BF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
e2	=	000001E6
e3	=	0000012D
xQ20	=	00000001 CD5AEB4E 02DBE2CE B712A45E ED7720D3 EA94116F 1E45C834 FDFF3A86 7BBB267B F8F5F9B1 9C369F7A FE141B85 D591243E 7310B6D0 2E78DB88 8615254D F178C1F7 5F2BDAF7 03E83BB9 7DBCDC3D FDB60BA3 85EC8F42 D4AD1505 21ECA6EC 4D3086A7 783698A7 1544E10A 45EA605E 1B86A894 7F14FA2E 03845DAE
xQ21	=	00000000
yQ20	=	00000002 2E751F1F 60841CF4 E8D4D3BD 8D400F58 9761CA1F 71A9C1F3 83C0FE55 3E6492DB E1F78D5F 9A768920 B682786E 8125398F 765A481B 32913561 FD16B270 19D9C10C 4F9062AC 1513FEB2 FE942DD2 2AC53F6E C319C4D1 8A53A481 430F3DFA 22E57EDA 0D067C37 F91EA8F1 3E4B1C65 4E974856 781F8E0A 397ED362
yQ21	=	00000000
xP20	=	00000006 ED767E28 04975D81 80368FB9 A72CE64E 838A5497 4865BFF1 A86AEF07 D6171A8A 4DF351F1 D4C94AAF 82BD6EBD 396F3342 48282F50 73178AB5 7B906BEF 89A2A152 A10D04A5 B20A0FFF 96B0B48F 0599FC9B D2AD52E0 81BB7FEA 7B5E8BF4 C3B0AB13 0731F4C5 A974CFA5 AD678121 7A20F9EC D30691D9 D1941D03
xP21	=	00000003 FE63FBDC A589518A 3DA694EC C8B65934 6693C45B D8AC86B6 F0C778CF 290C9F42 9163FEF6 4AFAD182 ADE1B0C4 DFC8CF29 C35455C7 BA69C225 59F2E0D4 20AE05BB 0AE3ADC0 9A4A0AF2 8CE1A1C5 93171033 7AA68884 EFCCD60A C76FD3F1 7ED50205 305509E0 5955F60D 5008D788 B83F5FA4 57FD79EC DC7179D1
yP20	=	00000000 4E0A8662 85403BC0 408F9BCA 912025E3 17111896 1C865461

```
             2AE20CEE AF91A98A 4F278EAE BB704602 8AD90CD9 5B99BF6B 34233CD4
             B084B2CA 4598DF3A 6D4839CA 6EA493ED 420CF4C1 3A1F37F1 EC59620F
             08693649 C72380A8 479E3753 93D3F4A7 59DF65F1 F74B4C65 6B79DF2A
             5DA2959E FB006BDA D015D252
yP21    =    00000007 38846048 2319281B 78C6AC1F E1A91DB7 2A2C9AA3 4BEE1EBE
             A33EF043 AA1BFA0C 45894142 95E94C91 1E19E808 246B2A0A 98593958
             E1F70888 E00332DE AE7D7FDB CA53398E 59530E5D 2A292463 7533F46C
             4684373D DDB8D09B 2A75C307 3EA3C19E CF946FCB 2B428B6E 9CF93F22
             33DD257C 5CAD4041 3F78CD1D
xR20    =    00000008 44F024B8 D993B660 48C9DA7F 1724AE2E 4C6162F8 4804FE3F
             E290FBEB 5ABF7DF2 5C395121 77C8E4A4 7A35F8EC D037B699 E34F58EF
             675AE188 A1537838 A4DDBA69 FC7BC3FA CB7E3815 F3031244 AF1BCCE1
             95AF45B4 2A587EE8 7A00BF2D A1E972D8 D662F4DC 5EC1CDB1 03D9EED2
             215D4DD7 48004985 8925A63A
xR21    =    00000007 35F06B97 66B69FF9 17835EAD C539A00F FC186ED2 5947F701
             FDA7EDEA F517039F 9B0DA172 1FDAE978 4838C75B 46A452DC 902EC8DD
             1F462564 3B42C596 C2CF0404 5A7AA804 3F07C9A9 F82611D2 02F06834
             512A3803 EF64650E 5309163F 25CCC336 CC852764 9E340F59 CDDFB51B
             24D1C02E 8CB2653E 7A05B709
xQ30    =    00000003 81DBCAB1 EE7A4CA3 192CDA85 3F4E0F42 6522EB9D 3277421C
             29D73CC4 F70BEFE7 009767C4 AE451600 3B237223 422C0E75 2ACD9D8F
             CE07263D 2C1D1013 08C0B97E DB8D4A8C 53C2064B 05DF9A61 E8216CA1
             FFAC55F4 CA043972 52704945 C27136A0 56F6B5CF 8838B7F6 52BC16C1
             392B5597 36CAF63B F0058A53
xQ31    =    00000000
yQ30    =    00000001 AFBFA81B 55C7D789 B6E89BC7 A311F3CE E4B733B0 FA5B7D56
             D29A644B 596B7729 778E0773 F908D76E 0377B3CA 41C03D79 7DE4F0B7
             985EB512 7D2151EC 4B6C1136 1AEB4CEA A3F776E0 8E4AD01B 7BB46074
             D425C8A2 61E88B14 5C4153BF 67732E82 9986EB9D 29C88385 1EEEB87C
             C4FD96A0 84332542 6C108687
yQ31    =    00000000
xP30    =    00000004 FE46F0B0 09171C87 0FE840B7 FD0C3F14 93813CD1 25C2191C
             9FA4BDE4 0941A603 124F1B81 BBFBEBFD AC06F808 07562639 FC61A579
             62AE6E6B 7EE793CF 7B359746 FEB0DA11 0D704681 F83EF6B4 40DC5DED
             D2A42471 49D0A44C 452ED374 A394319A 8888A2A0 9CC4A0F3 5A07AA3D
             248CF780 3E77EAD2 A4BEA308
xP31    =    00000006 DD4FC176 8E1ADEE5 4DC4E41C FAA7B810 4644DD69 0616D374
```

```
        A9139013 FD847C2C D11BA6CA 4C4FC26A 63FE198B 666B7912 FBF889E9
        91CF4B90 3651F441 4CD4AA50 BC02CE2E C986A7BF C1A8D364 F93410AD
        E3B959FF 1F036F36 EF3AFF88 D28DB500 8276C340 2158ADB4 A44BAECE
        D2AF6503 093FD8B6 A58EC136
yP30  = 00000003 EA8CDCF4 BC1C1B9D 2D449022 387D4DDE F05CE98C B63E722B
        0EA14717 C5FFA82E E107832E 5FE58C28 90185D2C 90D1BD94 AC13C69F
        D483AC80 66B1F1A4 844F7655 884B2379 0088A6DA 915FD709 EFE79A88
        028108F4 D4DFBFAD BA65EFBB C5D621BA 31F12BE6 FB717D3B 1D8CD78D
        CD05B0D2 B7E87C10 3D1897C0
yP31  = 00000007 77607A21 85C04FBF CFA5EAF7 7F38F40F 42746739 748CA176
        BBE31739 4BDA28F3 D971DFB9 CCB67207 E201FFB0 0A9A3E6B 9B7E804F
        BB6EF61E CEBDB8AC 68831E10 E8A72613 A47F132B D9A2309E 404FCFFF
        7EA7BC87 AD448B8B 8798AB61 CA6F97DB 3B240887 9DEB8A9F 930C4EE4
        69486FA1 129E89B6 7C084CD9
xR30  = 00000007 DE290085 EBBDC801 A1D6292D 1F2E89FF 463669ED 2F5F6C02
        B8010A75 245C4D39 84002821 B8A243C7 56512A5F C1FC0867 A84583D7
        6B0404E7 E73CEB70 71E2AE3B F43BFB77 A87BC98F DF888E28 5CD4A3C9
        4E4D1795 009E41ED B8A3AD8F 81321138 E4A87B69 416AEFF0 94E541F4
        8681863B AD30FB2F 32EA019A
xR31  = 00000007 A2A2DFA4 FB567336 60ACCB80 308A4482 B1A46D3B 9BB20313
        F164CC80 9A3A6B4D 2FBC4357 4994354D C06D409F 9F647E82 F4F05D6C
        5A70E340 DF4B9555 9787F82E AD7F7295 590FDCD9 D54B8001 094DC809
        29EF4C5A BB8E388A 53AA0BD3 88D890B5 980F1FD1 9404025B 582C640D
        DFDA1BDF 46D37046 4A812732
```

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 ARM64 platforms that exploits ARMv8 assembly,

5. An additional, speed-optimized VHDL model for FPGA and ASIC platforms that parallelizes various aspects of the isogeny computation and field operations, and

6. An additional, simple textbook implementation written exclusively in portable C, using elliptic curves in short Weierstrass form.

All implementations except implementations number 1 and 6 are protected against timing and cache attacks at the software level. Specifically, they avoid the use of secret address accesses and secret branches.

The generic reference implementation (number 1) supports all three parameter sets: `SIKEp503`, `SIKEp751` and `SIKEp964`. The optimized and assembly-optimized implementations (number 2, 3, 4) support the `SIKEp503` and `SIKEp751` parameter sets. The VHDL implementation (number 5) supports the `SIKEp751` parameter set. The Weierstrass implementation (number 6) is not 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 23–41 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, 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 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 `SIKEp503`, `SIKEp751` and `SIKEp964` 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 3–22 listed in Appendix A. The optimal tree traversal strategies used in Algorithms 17 and 18 are given in Appendix C 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 Montgomery reduction that has been specialized for primes of the form $2^{e_2}3^{e_3} - 1$ [7].

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 3.8.0 with the command `clang -O3`. Results are similar, although slightly slower, when compiling with GNU GCC version 5.4.0.

Table 2.1 details the performance of the reference, optimized and x64-assembly implementations of SIKE. As we can see, the *constant-time* optimized implementation is roughly 15 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 `SIKEp503` and `SIKEp751` (i.e., 10.1 msec. and 30.5 msec., respectively, obtained by adding the times for encapsulation and decapsulation) is achieved with the use of hand-tuned x64 assembly.

Scheme	KeyGen	Encaps	Decaps	**total** (Encaps + Decaps)
Reference Implementation				
SIKEp503	1,561,680	2,207,324	2,663,521	4,870,845
SIKEp751	4,735,527	6,485,322	7,996,219	14,481,541
SIKEp964	10,563,749	14,995,526	17,957,283	32,952,809
Optimized Implementation				
SIKEp503	86,156	141,821	150,790	292,611
SIKEp751	288,886	467,776	502,972	970,748
Additional implementation using x64 assembly				
SIKEp503	10,134	16,619	17,696	34,315
SIKEp751	30,919	50,014	53,838	103,852

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.

Scheme	**secret key** sk	**public key** pk	**ciphertext** ct	**shared secret** ss
SIKEp503	(56+378) 434	378	402	16
SIKEp751	(80+564) 644	564	596	24
SIKEp964	(100+726) 826	726	766	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 §1.3.10).

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 out-

Scheme	`KeyGen` (stack)	`Encaps` (stack)	`Decaps` (stack)	static library speed (`-O3`)	static library size (`-Os`)
Reference Implementation					
`SIKEp503`	512	762	1528	107,450	96,386
`SIKEp751`	2880	1332	2280	107,450	96,386
`SIKEp964`	3744	2262	2936	107,450	96,386
Optimized Implementation					
`SIKEp503`	8,040	8,632	9,464	122,612	60,020
`SIKEp751`	13,864	14,024	14,680	167,508	61,404
Additional implementation using x64 assembly					
`SIKEp503`	8,120	8,520	8,952	132,688	62,488
`SIKEp751`	14,032	14,176	14,944	188,720	67,080

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`).

puts 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 1%. 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.

2.3 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 machine powered by a 1.992GHz 64-bit ARM Cortex-A72 processor, running Ubuntu 16.04.2 LTS. For compilation we used

Scheme	`KeyGen`	`Encaps`	`Decaps`	**total** (`Encaps + Decaps`)
Optimized Implementation				
`SIKEp503`	150,000	247,078	262,795	509,873
`SIKEp751`	515,036	834,451	896,767	1,731,217
Additional implementation using ARMv8 assembly				
`SIKEp503`	31,252	51,629	54,826	106,454
`SIKEp751`	101,963	164,890	176,935	341,825

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

clang version 3.8.0 with the command `clang -O3`. Results are similar, although slightly slower, when compiling with GNU GCC version 5.4.0.

Table 2.4 details the performance of the optimized and additional ARMv8-assembly implementations of SIKE. As we can see, the specialized implementation is roughly 5 times faster than the optimized implementation, which is mainly due to the use of assembly. Our best performance for `SIKEp503` and `SIKEp751` on the targeted platform is given by 53.4 msec. and 171.6 msec., respectively, which correspond to the time that takes to compute the encapsulation and decapsulation operations.

We comment that similar results (after scaling) were achieved on a 1.7GHz 64-bit ARM Cortex-A57 processor, running Ubuntu 14.04.5 LTS.

2.4 VHDL hardware implementation

The optimized VHDL hardware implementation accelerates SIKE operations by using Algorithms 3–22 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 42 in Appendix C 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 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 public parameters listed in `SIKEp751`.

#Multipliers	Scheme	Cycle counts ($cc \times 10^3$)			
		KeyGen	Encaps	Decaps	**Total**
2	SIKEp751	3,920	6,563	6,992	13,555
4		2,464	4,214	4,488	8,702
6		1,941	3,459	3,633	7,092
8		**1,798**	**3,221**	**3,383**	**6,603**
10		1,698	3,112	3,240	6,352

Table 2.5: Summary of cycle counts for SIKE accelerator architecture over parameters listed in `SIKEp751`. The number of multipliers is a design parameter.

	Area					Freq	Time (msec)			
# Mults	# FFs	# LUTs	# Slices	# DSPs	# BRAMs	(MHz)	KeyGen	Encaps	Decaps	total (Encaps + Decaps)
8	51,914	44,822	16,756	376	56.5	198	9.08	16.27	17.08	33.35

Table 2.6: FPGA implementation results of SIKE accelerator over `SIKEp751` on a Xilinx Virtex-7 FPGA.

2.4.1 Performance

The SIKE hardware accelerator can perform KEM functions for the public parameters listed in `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 2.5, we specify the total number of cycles to perform the key encapsulation operations based on the number of multipliers. In the hardware package, we include the version with 4 dual-multipliers, or 8 total multipliers.

2.4.2 FPGA SIKE Accelerator

The VHDL SIKE accelerator core was compiled for FPGA with Xilinx Vivado design suite version 2015.4 to a Xilinx Virtex-7 xc7vx690tffg1157-3 board. All results were obtained after place-and-route. The area and timing results of our `SIKEp751` accelerator core on FPGA are shown in Table 2.6. For our design, we had the option of choosing how many dual multipliers to replicate. We focused on 4 replicated multipliers in our design to ensure the parallelism in isogeny-based computations could be taken advantage of. These are constant-time results. For the FPGA implementation over `SIKEp751`, encapsulation and decapsulation can be performed in 16.27 and 17.08 msec, respectively. This results in a total KEM time of 33.35 msec.

2.4.3 ASIC SIKE Accelerator

The SIKE accelerator core was synthesized using Synopsys Design Compiler. The TSMC 65-nm CMOS standard technology and CORE65LPSVT standard cell library were used for results. This implementation was optimized for performance.

The area was converted to Gate Equivalents (GE), where the size of a single NAND gate is considered 1 GE. For our particular technology library, the size of a synthesized NAND gate was 1.41 μm^2, so this

Area			**Frequency**	**Time (msec)**			
Tech (nm)	**#Mults**	**Area (kGE)**	**(MHz)**	`KeyGen`	`Encaps`	`Decaps`	**total** `(Encaps + Decaps)`
65	8	1,210	350	5.14	9.20	9.67	18.87

Table 2.7: Optimized hardware synthesis results for SIKE accelerator over `SIKEp751`. The area results do not include synthesized program ROM, register file, or strategy lookup table results.

was used as the conversion factor. For the ASIC implementation over `SIKEp751`, encapsulation and decapsulation can be performed in 9.20 and 9.67 msec, respectively. Thus, the total KEM time is 18.87 msec. The area and timing results of our design are shown in Table 2.7.

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 `SIKEp503`, `SIKEp751`, and `SIKEp964`. The KAT files can be found in the media folder of the submission: `\KAT\PQCkemKAT_426.rsp`, `\KAT\PQCkemKAT_636.rsp`, and `\KAT\PQCkemPAT_826.rsp` for `SIKEp503`, `SIKEp751` and `SIKEp964`, 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 [12]: 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 [10] 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

[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 $\texttt{isogen}_\ell$. However, a complete reduction of the security of SIKE to the isogeny walk problem is presented in [14].

$a \in A$ 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 [37], 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 [42]. 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, while 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.

We further discuss the implications of the claw-finding attacks on the security of `SIKE` in Section 5.2. 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 recently appeared in the literature.

Galbraith, Petit, Shani and Ti [14] 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 [14] show how a secret j-invariant can be recovered from some partial knowledge of it.
- Ti [38] 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 [15] 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 15 and 16. However their attack is efficiently countered by the additional checks in Algorithms 17 and 18.

A recent preprint by Petit [29] 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 [30] showed that the isogeny walk problem used to construct the Charles-Goren-Lauter hash function [4] 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 §1.3.9 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 [9, 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 [2, 22]). 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 [22, Thm 5], [13, Thm 20.4.11] or [20, Thm 11.21]. □

Remark 1. There exist alternative proofs of security in the standard model, reducing the security to a decisional variant of SIDH [9, Problem 5.4] instead of SIDH (see [9, Thm 6.2], based on [34, Thm 2] and [33, §3.4]).

4.3.3 IND-CCA KEM

Theorem 1 ([18]). *For any IND-CCA adversary B against* $\mathtt{KEM}$*, issuing at most q_G (resp. q_H) queries to the random oracle G (resp. H), there exists an IND-CPA adversary A against* $\mathtt{PKE}$ *with*

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

Proof. This is the bound obtained by combining the results from Theorem 3.2 and Theorem 3.4 from [18], setting $\mathtt{KEM} = U^{\not\perp}[T[\mathtt{PKE}, G], H]$.

Note that $\mathtt{Decaps}$ slightly deviates from the definition in [18]. Instead of full "re-encryption" $(c_0', c_1') \leftarrow \mathtt{Enc}(\mathrm{pk}_3, m'; G(m' \parallel \mathrm{pk}_3))$, we only re-compute c_0'. However, full computation would yield

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

while $c_1 = m' \oplus F(\mathtt{isoex}_3(c_0, \mathrm{sk}_3))$. 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` $\in \{503, 751, 964\}$, requires resources comparable to those required for key search on a k-bit (ideal) block cipher $\mathcal{B}$, where $k \in \{128, 192, 256\}$. We give a precise statement of this design goal in §5.1, and a more detailed justification in §5.2.

5.1 Security levels

If a k-bit key block cipher $\mathcal{B}$ is viewed as a black box, the optimal way (classically) to retrieve the key is to try all 2^k different options. On average, this takes 2^{k-1} attempts. However, if we have $\mathcal{B}$ as a (black box) quantum circuit, we can apply Grover's algorithm [17]. This requires only (approximately) $\sqrt{2^k}$ applications of $\mathcal{B}$. This motivates the following definition.

Definition 1. *A block cipher $\mathcal{B}$ with a k-bit key is called classically (resp. quantum) secure if on average we need 2^{k-1} (resp. $\sqrt{2^k}$) invocations of $\mathcal{B}$ to retrieve the key.*

Similarly, let $p = 2^{e_2} \cdot 3^{e_3} - 1$, and $\ell^{e_\ell} = \min(2^{e_2}, 3^{e_3})$. Then the classical (resp. quantum) complexity of breaking the isogeny problem is assumed to be $O(\sqrt{\ell^{e_\ell}})$ (resp. $O(\sqrt[3]{\ell^{e_\ell}})$). The complexity of these algorithms is measured in the number of queries to an oracle $\mathcal{I}$, computing isogenies of degree $\ell^{e_\ell/2}$ (see §4.1). Assuming the cost of $\mathcal{I}$ to be larger than the cost of $\mathcal{B}$ (see §5.2), we obtain the following security statement.

Theorem 2. *The scheme* `SIKEpXXX`*, where* `XXX` $\in \{503, 751, 964\}$*, is classically (resp. quantum) k-bit secure if breaking the respective SIDH problem requires at least 2^{k-1} (resp. $\sqrt{2^k}$) invocations of $\mathcal{I}$.*

Proof. Follows from Proposition 1 and Theorem 1. □

The security levels are summarized in Table 5.1.

	k	2^{k-1}	$\min(\sqrt{2^{e_2}}, \sqrt{3^{e_3}})$	$\sqrt{2^k}$	$\min(\sqrt[3]{2^{e_2}}, \sqrt[3]{3^{e_3}})$
`SIKEp503`	128	2^{127}	$1.00 \cdot 2^{125}$	2^{64}	$1.26 \cdot 2^{83}$
`SIKEp751`	192	2^{191}	$1.00 \cdot 2^{186}$	2^{96}	$1.00 \cdot 2^{124}$
`SIKEp964`	256	2^{255}	$1.45 \cdot 2^{238}$	2^{128}	$1.02 \cdot 2^{159}$

Table 5.1: Classical and quantum security estimates

Remark 2. For a k-bit security level the prime p is chosen to be the largest prime of the form $p = 2^{2i} \cdot 3^j - 1$ such that $p < 2^{4k}$ and $2^{2i} \approx 3^j$. Due to the sparseness of primes of this shape this leads to security levels (slightly) below k, most notably for `SIKEp964`. However, choosing p to have fewer than $4k$ bits allows us to represent $\mathbb{F}_p$-elements using $4k$ bits. As a result, we use a register less (e.g. a 32-bit or 64-bit register) than we would need for $p \geq 2^{4k}$, and gain a significant efficiency improvement. Moreover, this seemingly lower security is (easily) made up for by the higher cost of constructing a circuit implementing isogeny computations when compared to implementing a block cipher (see Remark 3).

5.2 Detailed analysis

We observe that even though the definitions in the previous section may be theoretically sound, it remains unclear how they would reflect the size and run-time of actual cryptanalytic circuits. Therefore we elaborate on estimates of the gate count and depth of a circuit solving the above problems.

Classical circuits. Let $G^C_{\mathcal{B}}$ (resp. $G^C_{\mathcal{I}}$) be the number of classical gates necessary to construct a classical circuit implementing $\mathcal{B}$ (resp. $\mathcal{I}$). Let $D^C_{\mathcal{B}}$ and $D^C_{\mathcal{I}}$ be the respective depths of the circuits. If $\mathcal{B}$ has a k-bit key and is classically secure, the above definitions imply that a circuit retrieving the key contains at least $2^{k-1} \cdot G^C_{\mathcal{B}}$ gates. Since the invocations of $\mathcal{B}$ are completely independent, we can run anywhere between 1 and 2^{k-1} $\mathcal{B}$-circuits in parallel. That means we can retrieve the key with depth anywhere between $2^{k-1} \cdot D^C_{\mathcal{B}}$ and $D^C_{\mathcal{B}}$. Similarly, if we have a k-bit secure instantiation of `SIKE`, breaking the scheme requires a circuit of at least $2^{k-1} \cdot G^C_{\mathcal{I}}$ gates. Moreover, queries to $\mathcal{I}$ are all independent in the classic claw-finding algorithm. Assuming that $G^C_{\mathcal{B}} \leq G^C_{\mathcal{I}}$, $D^C_{\mathcal{B}} \leq D^C_{\mathcal{I}}$ (see Remark 3) and $\min(\sqrt{2^{e_2}}, \sqrt{3^{e_3}}) \approx 2^{k-1}$, this shows that the computational cost of classically breaking `SIKE` reflects the computational cost of classically retrieving the key of $\mathcal{B}$.

Quantum circuits. Let $G^Q_{\mathcal{B}}$ (resp. $G^Q_{\mathcal{I}}$) be the number of quantum gates needed to construct a quantum circuit implementing $\mathcal{B}$ (resp. $\mathcal{I}$). Let $D^Q_{\mathcal{B}}$ and $D^Q_{\mathcal{I}}$ be the respective depths of the circuits. If $\mathcal{B}$ is quantum secure, a quantum circuit retrieving the key similarly needs at least $\sqrt{2^k} \cdot G^Q_{\mathcal{B}}$ gates (by Grover's algorithm). Estimates for these costs using T-gates and Clifford gates are made by Grassl et al. [16, Table 5] for `AES`-$\{128, 192, 256\}$. They assume all invocations of `AES` to be done in serial[1]. By breaking up the search space into smaller parts, it is obvious that one can reduce the depth of such a circuit. However, a result by

[1]There are in fact parallel applications of `AES`, but these are only done to ensure uniqueness of the solution.

Zalka [40] shows that reducing the depth by a factor M, increases the number of gates by at least the same factor. In particular, if we restrict the depth to maximum value MAXDEPTH, we need at least

$$(\#\text{gates} \times \text{depth}) \ / \, \text{MAXDEPTH} \tag{5.1}$$

gates.

On the other hand, to break a k-bit quantum secure instantiation of `SIKE`, we require a quantum circuit containing at least $\sqrt{2^k} \cdot G_{\mathcal{I}}^{Q}$ gates. Naïvely, we run all queries in serial and obtain a circuit of depth $\sqrt{2^k} \cdot D_{\mathcal{I}}^{Q}$. In this case, unfortunately, there is no result analog to Zalka's about the consequences of reducing the depth. However, we conjecture that reducing the depth by a factor M also increases the gate count by a factor M. The idea is that the quantum claw-finding algorithm by Tani [37] relies on a quantum algorithm by Szegedy [36], which generalizes Grover's algorithm. Therefore we expect the behavior with respect to parallelization to be comparable, but emphasize that this fact remains to be proven. As a result, we expect Equation (5.1) to hold. Therefore, assuming $G_{\mathcal{B}}^{Q} \leq G_{\mathcal{I}}^{Q}$, $D_{\mathcal{B}}^{Q} \leq D_{\mathcal{I}}^{Q}$ (see Remark 3) and $\sqrt[3]{\ell^{e_\ell}} \approx \sqrt{2^k}$, the computational cost of breaking `SIKE` with a quantum circuit reflects the computational cost of retrieving the key of $\mathcal{B}$.

Moreover, this analysis ignores the number of qubits required. The algorithm of Tani [37] requires $O(\sqrt[3]{\ell^{e_\ell}})$ qubits whereas, e.g., the implementation by Grassl et al. [16] only uses a few thousand.

Remark 3. For $\mathcal{B} =$ `AES`, estimates for $G_{\mathcal{B}}^{C}$ are 2^{16}, 2^{16} and 2^{17} for 128, 192 and 256-bit security levels, respectively (see [28]). On the other hand, isogeny computations of $\mathcal{I}$ rely on $\mathbb{F}_{p^2}$-element operations, which in turn are composed of $\mathbb{F}_p$-multiplications. If we assume an n-bit multiplication to have a lower bound of $n \log_2 n$ gates, where $n = \log_2 p$, then we need as few as 15 $\mathbb{F}_p$-multiplications to obtain $G_{\mathcal{B}}^{C} \leq G_{\mathcal{I}}^{C}$ at any of our security levels. In reality, an isogeny computation requires many more $\mathbb{F}_p$-multiplications while $\mathbb{F}_p$-multiplications do not achieve the lower bound of $n \log_2 n$, so this analysis remains very conservative. Moreover, many of these multiplications must be done in serial, so that also $D_{\mathcal{B}}^{C} \leq D_{\mathcal{I}}^{C}$. The same results are expected to hold for quantum circuits.

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 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 [15, 24, 38]. 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 [23]. Although there exists a subexponential quantum algorithm [5] that can solve the analogue of SIDH that uses ordinary curves (this scheme was first suggested by Couveignes in 1997 [8] and later published by Rostovtsev and Stolbunov in 2006 [31, 35]), 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-style algorithms, then the complexities used to generate the sets of parameters in §1.5 are already known to be optimal (cf. §4.1), and this gives us some confidence in the long-term quantum-security of the proposed SIKE instantiations, analogous to the (classical) confidence in discrete logarithm security that one derives from generic lower bounds [32].

Notwithstanding the current limitations in the knowledge of the future of quantum computation, the well-understood (classical and quantum) complexities of the claw algorithms make for relatively straightforward scaling of SIKE parameters at different levels of security. In addition, 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 [9]) 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 `SIKEp503`, `SIKEp751` and `SIKEp964` parameters were selected from these candidates according to the criteria discussed in §1.5.

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 keys corresponding to `SIKEp503` are 378 bytes, which is comparable to the 384-byte RSA public keys corresponding to a 3072-bit modulus, those which are conjectured to offer 128 bits of classical security. Likewise, `SIKEp751` public keys are 564 bytes, and the largest of our parameter sets, `SIKEp964`, has 723-byte uncompressed public keys. Note that the technique of public-key compression [1, 6], which we have omitted from this specification, affords users the option of compressing SIKE public keys to around 60% of their original size. Under current cryptanalytic knowledge, SIKE with key compression provides smaller public keys than any other post-quantum public-key cryptosystem at comparable security levels. The performance cost of key compression in the published literature is approximately double the computation time of regular key exchange [6]. A very recent preprint [41], not yet published, improves key compression performance by an additional factor of two to five.

The ease of partnering supersingular isogeny-based public-key cryptography with strong classical elliptic curve cryptography (ECC) is discussed in [7, §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 [26]) 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 503-bit prime field corresponding to `SIKEp503`, this technique gives rise to a SIKE+ECDH hybrid that offers around the same classical ECDLP security as the strongest NISTp521 curve. The corresponding uncompressed public keys inflate by a factor of no more than 1.17x relative to SIKE alone, and the benchmarks reported in [7, 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 `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] 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.

[2] 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.

[3] 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.

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

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

[6] 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.

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

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

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

[14] 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.

[15] 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.

[16] Markus Grassl, Brandon Langenberg, Martin Roetteler, and Rainer Steinwandt. Applying Grover's algorithm to AES: quantum resource estimates. In *Post-Quantum Cryptography — 7th International Workshop, PQCrypto 2016, Fukuoka, Japan, February 24-26, 2016, Proceedings*, pages 29–43, 2016.

[17] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996*, pages 212–219, 1996.

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

[19] 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.

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

[21] John M. Kelsey, Shu-Jen H. Chang, and Ray Perlner. SHA-3 derived functions: cSHAKE, KMAC, TupleHash and ParallelHash, 2016. Available at `http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-185.pdf`.

[22] 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.

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

[24] 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.

[25] 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.

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

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

[28] National Institute of Standards and Technology. Post-quantum cryptography standardization — call for proposals (draft), 2016. https://www.nist.gov/pqcrypto/.

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

[30] 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.

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

[32] Victor Shoup. Lower bounds for discrete logarithms and related problems. In Walter Fumy, editor, *Advances in Cryptology — EUROCRYPT '97: International Conference on the Theory and Application of Cryptographic Techniques Konstanz, Germany, May 11–15, 1997 Proceedings*, pages 256–266, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

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

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

[35] 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.

[36] Mario Szegedy. Quantum speed-up of markov chain based algorithms. In *45th Symposium on Foundations of Computer Science (FOCS 2004), 17-19 October 2004, Rome, Italy, Proceedings*, pages 32–41, 2004.

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

[38] 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.

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

[40] Christof Zalka. Grover's quantum searching algorithm is optimal. *Rev. A*, 1999.

[41] Gustavo H. M. Zanon, Marcos A. Simplicio Jr., Geovandro C. C. F. Pereira, Javad Doliskani, and Paulo S. L. M. Barreto. Faster isogeny-based compressed key agreement. Cryptology ePrint Archive, Report 2017/1143, 2017. https://eprint.iacr.org/2017/1143.

[42] 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.

Appendix A

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

This section contains explicit formulas for computing the isogenies described in §1.3.5 and §1.3.6. Assuming access to all of the field operations in $\mathbb{F}_{p^2}$, Algorithms 3–22 can compute `isogen`$_\ell$ and `isoex`$_\ell$ for $\ell \in \{2, 3\}$ in their entirety.

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 $E_a/\mathbb{F}_{p^2} : y^2 = x^3 + ax^2 + x$ 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 [11].

Algorithms 17 and 18 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: $(X_{[2]P} : Z_{[2]P})$
1  $t_0 \leftarrow X_P - Z_P$
2  $t_1 \leftarrow X_P + Z_P$
3  $t_0 \leftarrow t_0^2$
4  $t_1 \leftarrow t_1^2$
5  $Z_{[2]P} \leftarrow C_{24} \cdot t_0$
6  $X_{[2]P} \leftarrow Z_{[2]P} \cdot t_1$
7  $t_1 \leftarrow t_1 - t_0$
8  $t_0 \leftarrow A_{24}^+ \cdot t_1$
9  $Z_{[2]P} \leftarrow Z_{[2]P} + t_0$
10 $Z_{[2]P} \leftarrow Z_{[2]P} \cdot t_1$
11 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: $(X_{[2^e]P} : Z_{[2^e]P})$
1 $(X' : Z') \leftarrow (X_P : Z_P)$
2 for $i = 1$ to $e$ do
3     $(X' : Z') \leftarrow$ xDBL$\big((X' : Z'), (A^+_{24} : C_{24})\big)$        // Alg. 3
4 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: $(X_{[2]P} : Z_{[2]P})$, $(X_{P+Q} : Z_{P+Q})$
1  $t_0 \leftarrow X_P + Z_P$
2  $t_1 \leftarrow X_P - Z_P$
3  $X_{[2]P} \leftarrow t_0^2$
4  $t_2 \leftarrow X_Q - Z_Q$
5  $X_{P+Q} \leftarrow X_Q + Z_Q$
6  $t_0 \leftarrow t_0 \cdot t_2$
7  $Z_{[2]P} \leftarrow t_1^2$
8  $t_1 \leftarrow t_1 \cdot X_{P+Q}$
9  $t_2 \leftarrow X_{[2]P} - Z_{[2]P}$
10 $X_{[2]P} \leftarrow X_{[2]P} \cdot Z_{[2]P}$
11 $X_{P+Q} \leftarrow a^+_{24} \cdot t_2$
12 $Z_{P+Q} \leftarrow t_0 - t_1$
13 $Z_{[2]P} \leftarrow X_{P+Q} + Z_{[2]P}$
14 $X_{P+Q} \leftarrow t_0 + t_1$
15 $Z_{[2]P} \leftarrow Z_{[2]P} \cdot t_2$
16 $Z_{P+Q} \leftarrow Z_{P+Q}^2$
17 $X_{P+Q} \leftarrow X_{P+Q}^2$
18 $Z_{P+Q} \leftarrow X_{Q-P} \cdot Z_{P+Q}$
19 $X_{P+Q} \leftarrow Z_{Q-P} \cdot X_{P+Q}$
20 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: $(X_{[3]P} : Z_{[3]P})$
1  $t_0 \leftarrow X_P - Z_P$
2  $t_2 \leftarrow t_0^2$
3  $t_1 \leftarrow X_P + Z_P$
4  $t_3 \leftarrow t_1^2$
5  $t_4 \leftarrow t_1 + t_0$
6  $t_0 \leftarrow t_1 - t_0$
7  $t_1 \leftarrow t_4^2$
8  $t_1 \leftarrow t_1 - t_3$
9  $t_1 \leftarrow t_1 - t_2$
10 $t_5 \leftarrow t_3 \cdot A^+_{24}$
11 $t_3 \leftarrow t_5 \cdot t_3$
12 $t_6 \leftarrow t_2 \cdot A^-_{24}$
13 $t_2 \leftarrow t_2 \cdot t_6$
14 $t_3 \leftarrow t_2 - t_3$
15 $t_2 \leftarrow t_5 - t_6$
16 $t_1 \leftarrow t_2 \cdot t_1$
17 $t_2 \leftarrow t_3 + t_1$
18 $t_2 \leftarrow t_2^2$
19 $X_{[3]P} \leftarrow t_2 \cdot t_4$
20 $t_1 \leftarrow t_3 - t_1$
21 $t_1 \leftarrow t_1^2$
22 $Z_{[3]P} \leftarrow t_1 \cdot t_0$
23 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: $(X_{[3^e]P} : Z_{[3^e]P})$
1 $(X' : Z') \leftarrow (X_P : Z_P)$
2 for $i = 1$ to $e$ do
3     $(X' : Z') \leftarrow$ xTPL$\big((X' : Z'), (A_{24}^+ : A_{24}^-)\big)$        // Alg. 6
4 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 $(a_{24}^+ : 1)$
    Output: $(X_{Q+[m]P} : Z_{Q+[m]P})$
1 $((X_0 : Z_0), (X_1 : Z_1), (X_2 : Z_2)) \leftarrow ((x_Q : 1), (x_P : 1), (x_{Q-P} : 1))$
2 for $i = 0$ to $\ell - 1$ do
3     if $m_i = 1$ then
4         $((X_0 : Z_0), (X_1 : Z_1)) \leftarrow$ xDBLADD$((X_0 : Z_0), (X_1 : Z_1), (X_2 : Z_2), (a_{24}^+ : 1))$        // Alg. 5
5     else
6         $((X_0 : Z_0), (X_2 : Z_2)) \leftarrow$ xDBLADD$((X_0 : Z_0), (X_2 : Z_2), (X_1 : Z_1), (a_{24}^+ : 1))$        // Alg. 5
7 return $(X_1 : Z_1)$
```

Algorithm 9: Montgomery j-invariant computation

```
function jInvariant
    Input: $(A : C)$
    Output: $j$-invariant $j(E_{A/C}) \in \mathbb{F}_{p^2}$
1 $j \leftarrow A^2$
2 $t_1 \leftarrow C^2$
3 $t_0 \leftarrow t_1 + t_1$
4 $t_0 \leftarrow j - t_0$
5 $t_0 \leftarrow t_0 - t_1$
6 $j \leftarrow t_0 - t_1$
7 $t_1 \leftarrow t_1^2$
8 $j \leftarrow j \cdot t_1$
9 $t_0 \leftarrow t_0 + t_0$
10 $t_0 \leftarrow t_0 + t_0$
11 $t_1 \leftarrow t_0^2$
12 $t_0 \leftarrow t_0 \cdot t_1$
13 $t_0 \leftarrow t_0 + t_0$
14 $t_0 \leftarrow t_0 + t_0$
15 $j \leftarrow 1/j$
16 $j \leftarrow t_0 \cdot j$
17 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: $A \in \mathbb{F}_{p^2}$
1  $t_1 \leftarrow x_P + x_Q$
2  $t_0 \leftarrow x_P \cdot x_Q$
3  $A \leftarrow x_{Q-P} \cdot t_1$
4  $A \leftarrow A + t_0$
5  $t_0 \leftarrow t_0 \cdot x_{Q-P}$
6  $A \leftarrow A - 1$
7  $t_0 \leftarrow t_0 + t_0$
8  $t_1 \leftarrow t_1 + x_{Q-P}$
9  $t_0 \leftarrow t_0 + t_0$
10 $A \leftarrow A^2$
11 $t_0 \leftarrow 1/t_0$
12 $A \leftarrow A \cdot t_0$
13 $A \leftarrow A - t_1$
14 return $A$
```

Algorithm 11: 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
            $(K_1, K_2, K_3) \in (\mathbb{F}_{p^2})^3$
1  $K_2 \leftarrow X_{P_4} - Z_{P_4}$
2  $K_3 \leftarrow X_{P_4} + Z_{P_4}$
3  $K_1 \leftarrow Z_{P_4}^2$
4  $K_1 \leftarrow K_1 + K_1$
5  $C_{24} \leftarrow K_1^2$
6  $K_1 \leftarrow K_1 + K_1$
7  $A_{24}^+ \leftarrow X_{P_4}^2$
8  $A_{24}^+ \leftarrow A_{24}^+ + A_{24}^+$
9  $A_{24}^+ \leftarrow (A_{24}^+)^2$
10 return $A_{24}^+$, $C_{24}$,
       $(K_1, K_2, K_3)$
```

Algorithm 12: 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
1  $t_0 \leftarrow X_Q + Z_Q$ ,
2  $t_1 \leftarrow X_Q - Z_Q$ ,
3  $X_Q \leftarrow t_0 \cdot K_2$,
4  $Z_Q \leftarrow t_1 \cdot K_3$,
5  $t_0 \leftarrow t_0 \cdot t_1$,
6  $t_0 \leftarrow t_0 \cdot K_1$,
7  $t_1 \leftarrow X_Q + Z_Q$ ,
8  $Z_Q \leftarrow X_Q - Z_Q$ ,
9  $t_1 \leftarrow t_1^2$ ,
10 $Z_Q \leftarrow Z_Q^2$ ,
11 $X_Q \leftarrow t_0 + t_1$,
12 $t_0 \leftarrow Z_Q - t_0$ ,
13 $X_Q \leftarrow X_Q \cdot t_1$,
14 $Z_Q \leftarrow Z_Q \cdot t_0$,
15 return $(X_{Q'} : Z_{Q'})$
```

Algorithm 13: 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 $(K_1, K_2) \in (\mathbb{F}_{p^2})^2$
1  $K_1 \leftarrow X_{P_3} - Z_{P_3}$ ,
2  $t_0 \leftarrow K_1^2$ ,
3  $K_2 \leftarrow X_{P_3} + Z_{P_3}$ ,
4  $t_1 \leftarrow K_2^2$ ,
5  $t_2 \leftarrow t_0 + t_1$ ,
6  $t_3 \leftarrow K_1 + K_2$ ,
7  $t_3 \leftarrow t_3^2$ ,
8  $t_3 \leftarrow t_3 - t_2$ ,
9  $t_2 \leftarrow t_1 + t_3$ ,
10 $t_3 \leftarrow t_3 + t_0$ ,
11 $t_4 \leftarrow t_3 + t_0$ ,
12 $t_4 \leftarrow t_4 + t_4$ ,
13 $t_4 \leftarrow t_1 + t_4$ ,
14 $A^-_{24} \leftarrow t_2 \cdot t_4$ ,
15 $t_4 \leftarrow t_1 + t_2$ ,
16 $t_4 \leftarrow t_4 + t_4$ ,
17 $t_4 \leftarrow t_0 + t_4$ ,
18 $t_4 \leftarrow t_3 \cdot t_4$ ,
19 $t_0 \leftarrow t_4 - A^-_{24}$ ,
20 $A^+_{24} \leftarrow A^-_{24} + t_0$ ,
21 return $(A^+_{24} : A^-_{24})$,
       $(K_1, K_2) \in (\mathbb{F}_{p^2})^2$
```

Algorithm 14: 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 $E_{A/C}$
1  $t_0 \leftarrow X_Q + Z_Q$ ,
2  $t_1 \leftarrow X_Q - Z_Q$ ,
3  $t_2 \leftarrow K_1 \cdot t_0$ ,
4  $t_1 \leftarrow K_2 \cdot t_1$ ,
5  $t_2 \leftarrow t_0 + t_1$ ,
6  $t_0 \leftarrow t_1 - t_0$ ,
7  $t_2 \leftarrow t_2^2$ ,
8  $t_0 \leftarrow t_0^2$ ,
9  $X_{Q'} \leftarrow X_Q \cdot t_2$ ,
10 $Z_{Q'} \leftarrow Z_Q \cdot t_0$.
11 return $(X_{Q'} : Z_{Q'})$,
       $(K_1, K_2) \in (\mathbb{F}_{p^2})^2$
```

Algorithm 15: 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 $E_{A'/C'}$
1 for $e = \mathtt{e}_2 - 2$ downto $0$ by $-2$ do
2     $(X_T : Z_T) \leftarrow$ xDBLe $\big((X_S : Z_S), (A_{24}^+ : C_{24}), e\big)$        // Alg. 4
3     $\big((A_{24}^+ : C_{24}), (K_1, K_2, K_3)\big) \leftarrow$ 4_iso_curve $((X_T : Z_T))$        // Alg. 11
4     $(X_S : Z_S) \leftarrow$ 4_iso_eval $((K_1, K_2, K_3), (X_S : Z_S))$        // Alg. 12
5     for $(X_j : Z_j)$ in optional input do
6         $(X_j : Z_j) \leftarrow$ 4_iso_eval $\big((K_1, K_2, K_3), (X_j : Z_j)\big)$        // Alg. 12
7 return $(A_{24}^+ : C_{24}), [(X_1 : Z_1), (X_2 : Z_2), (X_3 : Z_3)]$
```

Algorithm 16: 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: $(A_{24}^{+\prime} : A_{24}^{-\prime})$ 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 $E_{A'/C'}$
1 for $e = \mathtt{e3} - 1$ downto $0$ by $-1$ do
2     $(X_T : Z_T) \leftarrow$ xTPLe $\big((X_S : Z_S), (A_{24}^+ : A_{24}^-), e\big)$        // Alg. 7
3     $\big((A_{24}^+ : A_{24}^-), (K_1, K_2)\big) \leftarrow$ 3_iso_curve $((X_T : Z_T))$        // Alg. 13
4     $(X_S : Z_S) \leftarrow$ 3_iso_eval $((K_1, K_2), (X_S : Z_S))$        // Alg. 14
5     for $(X_j : Z_j)$ in optional input do
6         $(X_j : Z_j) \leftarrow$ 3_iso_eval $\big((K_1, K_2), (X_j : Z_j)\big)$        // Alg. 14
7 return $(A_{24}^+ : A_{24}^-), [(X_1 : Z_1), (X_2 : Z_2), (X_3 : Z_3)]$
```

Algorithm 17: 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: $(A_{24}^{+}{}' : C_{24}')$ 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 S
push(S, $(\mathtt{e}_2/2, (X_S : Z_S)))($
$i \leftarrow 1$
while S *not empty* do
    $(h, (X : Z)) \leftarrow$ pop(S)
    if $h = 1$ then
        $(((A_{24}^+ : C_{24}), (K_1, K_2, K_3))($← 4_iso_curve$((X : Z))$    // Alg. 11
        Initialize empty deque S′
        while S *not empty* do
            $(h, (X : Z)) \leftarrow$ pull(S)
            $(X : Z) \leftarrow$ 4_iso_eval$(((K_1, K_2, K_3), (X : Z))$    // Alg. 12
            push(S′, $(h-1, (X : Z)))($
        S ← S′
        for $(X_j : Z_j)$ in *optional input* do
            $(X_j : Z_j) \leftarrow$ 4_iso_eval$(((K_1, K_2, K_3), (X_j : Z_j))($    // Alg. 12
    else if $0 < s_i < h$ then
        push(S, $(h, (X : Z)))($
        $(X : Z) \leftarrow$ xDBLe$(((X : Z), (A_{24}^+ : C_{24}), 2 \cdot s_i)($    // Alg. 4
        push(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 18: 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: $(A^+_{24}{}' : A^-_{24}{}')$ 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 $E_{A'/C'}$
1  Initialize empty deque $\mathbf{S}$
2  push($\mathbf{S}$, $(\mathtt{e}_3, (X_S : Z_S))$)
3  $i \leftarrow 1$
4  while $\mathbf{S}$ not empty do
5      $(h, (X : Z)) \leftarrow$ pop($\mathbf{S}$)
6      if $h = 1$ then
7          $((A^+_{24} : A^-_{24}), (K_1, K_2)) \leftarrow$ 3_iso_curve($(X : Z)$)          // Alg. 13
8          Initialize empty deque $\mathbf{S}'$
9          while $\mathbf{S}$ not empty do
10             $(h, (X : Z)) \leftarrow$ pull($\mathbf{S}$)
11             $(X : Z) \leftarrow$ 3_iso_eval($(K_1, K_2), (X : Z)$)          // Alg. 14
12             push($\mathbf{S}'$, $(h - 1, (X : Z))$)
13         $\mathbf{S} \leftarrow \mathbf{S}'$
14         for $(X_j : Z_j)$ in optional input do
15             $(X_j : Z_j) \leftarrow$ 3_iso_eval($(K_1, K_2), (X_j : Z_j)$)          // Alg. 14
16     else if $0 < s_i < h$ then
17         push($\mathbf{S}$, $(h, (X : Z))$)
18         $(X : Z) \leftarrow$ xTPLe($(X : Z), (A^+_{24} : A^-_{24}), s_i$)          // Alg. 7
19         push($\mathbf{S}$, $(h - s_i, (X : Z))$)
20         $i \leftarrow i + 1$
21     else
22         Error: invalid strategy
23 return $(A^+_{24} : A^-_{24})$, $[(X_1 : Z_1), (X_2 : Z_2), (X_3 : Z_3)]$
```

Algorithm 19: 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, xP2, xQ2, xR2, xP3, xQ3, xR3} (see §1.5)
    Output: Public key $pk_2 = (x_1, x_2, x_3)$ equivalent to the output of Step 4 of isogen_$\ell$ (see §1.3.5)
1 $((A: C), (A_{24}^{+}: C_{24})) \leftarrow ((0: 1), (1: 2))$
2 $((X_1: Z_1), (X_2: Z_2), (X_3: Z_3)) \leftarrow$ ((xP3: 1), (xQ3: 1), (xR3: 1))
3 $(X_S: Z_S) \leftarrow$ Ladder3pt($\mathrm{sk}_2$, (xP2, xQ2, xR2), $(A: C)$)                // Alg. 8
4 $((A_{24}^{+}: C_{24}), (X_1: Z_1), (X_2: Z_2), (X_3: Z_3)) \leftarrow$
    2_e_iso$((A_{24}^{+}: C_{24}), (X_S: Z_S), (X_1: Z_1), (X_2: Z_2), (X_3: Z_3))$    // Alg. 15 or Alg. 17
5 $((x_1: 1), (x_2: 1), (x_3: 1)) \leftarrow ((X_1: Z_1), (X_2: Z_2), (X_3: Z_3))$
6 return $pk_2 = (x_1, x_2, x_3)$                                      // Encoded as in §1.2.9
```

Algorithm 20: 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
           {e2, e3, p, xP2, xQ2, xR2, xP3, xQ3, xR3} (see §1.5)
    Output: Public key $pk_3 = (x_1, x_2, x_3)$ equivalent to the output of Step 4 of isogen_$\ell$ (see §1.3.5)
1 $((A: C), (A_{24}^{+}: A_{24}^{-})) \leftarrow ((0: 1), (2: -2))$
2 $((X_1: Z_1), (X_2: Z_2), (X_3: Z_3)) \leftarrow$ ((xP2: 1), (xQ2: 1), (xR2: 1))
3 $(X_S: Z_S) \leftarrow$ Ladder3pt($\mathrm{sk}_3$, (xP3, xQ3, xR3), $(A: C)$)                // Alg. 8
4 $((A_{24}^{+}: A_{24}^{-}), (X_1: Z_1), (X_2: Z_2), (X_3: Z_3)) \leftarrow$
    3_e_iso$((A_{24}^{+}: A_{24}^{-}), (X_S: Z_S), (X_1: Z_1), (X_2: Z_2), (X_3: Z_3))$    // Alg. 16 or Alg. 18
5 $((x_1: 1), (x_2: 1), (x_3: 1)) \leftarrow ((X_1: Z_1), (X_2: Z_2), (X_3: Z_3))$
6 return $pk_3 = (x_1, x_2, x_3)$                                      // Encoded as in §1.2.9
```

Algorithm 21: 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 §1.5)
    Output: A $j$-invariant $j_2$ equivalent to the output of Step 4 of isogen_ℓ (see §1.3.6)
1 $(A : C) \leftarrow$ (get_A$(x_1, x_2, x_3)$ : 1)                                    // Alg. 10
2 $(X_S : Z_S) \leftarrow$ Ladder3pt$(\mathrm{sk}_2, (x_1, x_2, x_3), (A : C))$          // Alg. 8
3 $(A^+_{24} : C_{24}) \leftarrow (A + 2 : 4)$
4 $(A^+_{24} : C_{24}) \leftarrow$ 2_e_iso$\big((A^+_{24} : C_{24}), (X_S : Z_S)\big)$   // Alg. 15 or Alg. 17
5 $(A : C) \leftarrow (4A^+_{24} - 2C_{24} : C_{24})$
6 $j$ = j_inv$((A : C))$                                                                // Alg. 9
7 return $j$                                                        // Encoded as in §1.2.8
```

Algorithm 22: 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 §1.5)
    Output: A $j$-invariant $j_3$ equivalent to the output of Step 4 of isogen_ℓ (see §1.3.6)
1 $(A : C) \leftarrow$ (get_A$(x_1, x_2, x_3)$ : 1)                                    // Alg. 10
2 $(X_S : Z_S) \leftarrow$ Ladder3pt$(\mathrm{sk}_3, (x_1, x_2, x_3), (A : C))$          // Alg. 8
3 $(A^+_{24} : A^-_{24}) \leftarrow (A + 2 : A - 2)$
4 $(A^+_{24} : A^-_{24}) \leftarrow$ 3_e_iso$\big((A^+_{24} : A_{24}), (X_S : Z_S)\big)$   // Alg. 16 or Alg. 18
5 $(A : C) \leftarrow (2 \cdot (A^-_{24} + A^+_{24}) : A^+_{24} - A^-_{24})$
6 $j$ = j_inv$((A : C))$                                                                // Alg. 9
7 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 §1.3.5 and §1.3.6 as used in the reference implementation. Assuming access to all of the field operations in $\mathbb{F}_{p^2}$, Algorithms 23–41 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 23: Affine coordinate doubling

```
function xDBL
    Input: $(x_P, y_P)$ and $(a, b)$
    Output: $(x_{[2]P}, y_{[2]P})$
1 if $P = \infty$ then
2     return $\infty$
3 $t_0 \leftarrow {x_P}^2$
4 $t_1 \leftarrow t_0 + t_0$
5 $t_0 \leftarrow t_0 + t_1$
6 $t_1 \leftarrow a \cdot x_P$
7 $t_1 \leftarrow t_1 + t_1$
8 $t_0 \leftarrow t_0 + t_1$
9 $t_0 \leftarrow t_0 + t_2$
10 $t_1 \leftarrow b \cdot y_P$
11 $t_1 \leftarrow t_1 + t_1$
12 $t_1 \leftarrow {t_1}^{-1}$
13 $t_0 \leftarrow t_0 \cdot t_1$
14 $t_1 \leftarrow {t_0}^2$
15 $t_2 \leftarrow b \cdot t_1$
16 $t_2 \leftarrow t_2 - a$
17 $t_2 \leftarrow t_2 - x_P$
18 $t_2 \leftarrow t_2 - x_P$
19 $t_1 \leftarrow t_0 \cdot t_1$
20 $t_1 \leftarrow b \cdot t_1$
21 $t_1 \leftarrow t_1 + y_P$
22 $y_{[2]P} \leftarrow x_P + x_P$
23 $y_{[2]P} \leftarrow y_{[2]P} + x_P$
24 $y_{[2]P} \leftarrow y_{[2]P} + a$
25 $y_{[2]P} \leftarrow y_{[2]P} \cdot t_0$
26 $y_{[2]P} \leftarrow y_{[2]P} - t_1$
27 $x_{[2]P} \leftarrow t_2$
28 return $(x_{[2]P}, y_{[2]P})$
```

Algorithm 24: 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
    $(x', y') \leftarrow$ xDBLAffine $((x', y'), (a, b))$        // Alg. 23
return $(x', y')$
```

Algorithm 25: 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 $\infty$
$t_0 \leftarrow y_Q - y_P$
$t_1 \leftarrow x_Q - x_P$
$t_1 \leftarrow {t_1}^{-1}$
$t_0 \leftarrow t_0 \cdot t_1$
$t_1 \leftarrow {t_0}^2$
$t_2 \leftarrow x_P + x_P$
$t_2 \leftarrow t_2 + x_Q$
$t_2 \leftarrow t_2 + a$
$t_2 \leftarrow t_2 \cdot t_0$
$t_0 \leftarrow t_0 \cdot t_1$
$t_0 \leftarrow b \cdot t_0$
$t_0 \leftarrow t_0 + y_P$
$t_0 \leftarrow t_2 - t_0$
$t_1 \leftarrow b \cdot t_1$
$t_1 \leftarrow t_1 - a$
$t_1 \leftarrow t_1 - x_P$
$x_{[P+Q]} \leftarrow t_1 - x_Q$
$y_{[P+Q]} \leftarrow t_0$
return $(x_{P+Q}, y_{P+Q})$
```

Algorithm 26: 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. 23
$(x_{[3]P}, y_{[3]P}) \leftarrow$ xADD $((x_P, y_P), (x_{[2]P}, y_{[2]P}), (a, b))$        // Alg. 25
return $(x_{[3]P}, y_{[3]P})$
```

Algorithm 27: Repeated affine coordinate tripling

```
function xTPLe
    Input: $(x_P, y_P)$, $(a, b)$, and $e \in \mathbb{Z}^+$
    Output: $(x_{[3^e]P}, y_{[3^e]P})$
1 $(x', y') \leftarrow (x_P, y_P)$
2 for $i = 1$ to $e$ do
3     $(x', y') \leftarrow$ xTPL $((x', y'), (a, b))$                // Alg. 26
4 return $(x', y')$
```

Algorithm 28: 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: $(x_{[m]P}, y_{[m]P})$
1 $(x_0, y_0) \leftarrow (0, 0)$
2 for $i = \ell - 1$ to $0$ by $-1$ do
3     $(x_0, y_0) \leftarrow$ xDBL $((x_0, y_0), (a, b))$                // Alg. 23
4     if $m_i = 1$ then
5         $(x_0, y_0) \leftarrow$ xADD $((x_0, y_0), (x, y), (a, b))$    // Alg. 25
6 return $(x_0, y_0)$
```

Algorithm 29: Montgomery j-invariant computation

```
function j_inv
    Input: $a$
    Output: $j$-invariant $j(E_{a,b}) \in \mathbb{F}_{p^2}$
1  $t_0 \leftarrow a^2$
2  $j \leftarrow 3$
3  $j \leftarrow t_0 - j$
4  $t_1 \leftarrow j^2$
5  $j \leftarrow j \cdot t_1$
6  $j \leftarrow j + j$
7  $j \leftarrow j + j$
8  $j \leftarrow j + j$
9  $j \leftarrow j + j$
10 $j \leftarrow j + j$
11 $j \leftarrow j + j$
12 $j \leftarrow j + j$
13 $j \leftarrow j + j$
14 $t_1 \leftarrow 4$
15 $t_0 \leftarrow t_0 - t_1$
16 $t_0 \leftarrow t_0^{-1}$
17 $j \leftarrow j \cdot t_0$
18 return $j$
```

Algorithm 30: 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 31: 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
$t_1 \leftarrow {x_Q}^2$
$t_2 \leftarrow {t_1}^2$
$t_3 \leftarrow {x_{P_4}}^2$
$t_4 \leftarrow t_2 \cdot t_3$
$t_2 \leftarrow t_2 + t_4$
$t_4 \leftarrow t_1 \cdot t_3$
$t_4 \leftarrow t_4 + t_4$
$t_5 \leftarrow t_4 + t_4$
$t_5 \leftarrow t_5 + t_5$
$t_4 \leftarrow t_4 + t_5$
$t_2 \leftarrow t_2 + t_4$
$t_4 \leftarrow t_3 \cdot t_3$
$t_5 \leftarrow t_1 \cdot t_4$
$t_5 \leftarrow t_5 + t_5$
$t_2 \leftarrow t_2 + t_5$
$t_1 \leftarrow t_1 \cdot x_Q$
$t_4 \leftarrow x_{P_4} \cdot t_3$
$t_5 \leftarrow t_1 \cdot t_4$
$t_5 \leftarrow t_5 + t_5$
$t_5 \leftarrow t_5 + t_5$
$t_2 \leftarrow t_2 - t_5$
$t_1 \leftarrow t_1 \cdot x_{P_4}$
$t_1 \leftarrow t_1 + t_1$
$t_1 \leftarrow t_1 + t_1$
$t_1 \leftarrow t_2 - t_1$
$t_2 \leftarrow x_Q \cdot t_4$
$t_2 \leftarrow t_2 + t_2$
$t_2 \leftarrow t_2 + t_2$
$t_1 \leftarrow t_1 - t_2$
$t_1 \leftarrow t_1 + t_3$
$t_1 \leftarrow t_1 + 1$
$t_2 \leftarrow x_Q \cdot x_{P_4}$
$t_4 \leftarrow t_2 - 1$
$t_2 \leftarrow t_2 + t_2$
$t_5 \leftarrow t_2 + t_2$
$t_1 \leftarrow t_1 - t_5$
$t_1 \leftarrow t_4 \cdot t_1$
$t_1 \leftarrow t_3 \cdot t_1$
$t_1 \leftarrow y_Q \cdot t_1$
$t_1 \leftarrow t_1 + t_1$
$y'_Q \leftarrow -t_1$
$t_2 \leftarrow t_2 - t_3$
$t_1 \leftarrow t_2 - 1$
$t_2 \leftarrow x_Q - x_{P_4}$
$t_1 \leftarrow t_2 \cdot t_1$
$t_5 \leftarrow {t_1}^2$
$t_5 \leftarrow t_5 \cdot t_2$
$t_5 \leftarrow {t_5}^{-1}$
$y'_Q \leftarrow y'_Q \cdot t_5$
$t_1 \leftarrow t_1 \cdot t_2$
$t_1 \leftarrow {t_1}^{-1}$
$t_4 \leftarrow {t_4}^2$
$t_1 \leftarrow t_1 \cdot t_4$
$t_1 \leftarrow x_Q \cdot t_1$
$t_2 \leftarrow x_Q \cdot t_3$
$t_2 \leftarrow t_2 + x_Q$
$t_3 \leftarrow x_{P_4} + x_{P_4}$
$t_2 \leftarrow t_2 - t_3$
$t_2 \leftarrow -t_2$
$x'_Q \leftarrow t_1 \cdot t_2$
return $(x_{Q'}, y_{Q'})$
```

Algorithm 32: 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 $E_{a',b'} = E_{a,b}/\langle P_3 \rangle$
1  $t_1 \leftarrow {x_{P_3}}^2$
2  $b' \leftarrow b \cdot t_1$
3  $t_1 \leftarrow t_1 + t_1$
4  $t_2 \leftarrow t_1 + t_1$
5  $t_1 \leftarrow t_1 + t_2$
6  $t_2 \leftarrow 6$
7  $t_1 \leftarrow t_1 - t_2$
8  $t_2 \leftarrow a \cdot x_{P_3}$
9  $t_1 \leftarrow t_2 - t_1$
10 $a' \leftarrow t_1 \cdot x_{P_3}$
11 return $(a', b')$
```

Algorithm 33: 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
1  $t_1 \leftarrow {x_Q}^2$
2  $t_1 \leftarrow t_1 \cdot x_{P_3}$
3  $t_2 \leftarrow {x_{P_3}}^2$
4  $t_2 \leftarrow x_Q \cdot t_2$
5  $t_3 \leftarrow t_2 + t_2$
6  $t_2 \leftarrow t_2 + t_3$
7  $t_1 \leftarrow t_1 - t_2$
8  $t_1 \leftarrow t_1 + x_Q$
9  $t_1 \leftarrow t_1 + x_{P_3}$
10 $t_2 \leftarrow x_Q - x_{P_3}$
11 $t_2 \leftarrow {t_2}^{-1}$
12 $t_3 \leftarrow {t_2}^2$
13 $t_2 \leftarrow t_2 \cdot t_3$
14 $t_4 \leftarrow x_Q \cdot x_{P_3}$
15 $t_4 \leftarrow t_4 - 1$
16 $t_1 \leftarrow t_4 \cdot t_1$
17 $t_1 \leftarrow t_1 \cdot t_2$
18 $t_2 \leftarrow {t_4}^2$
19 $t_2 \leftarrow t_2 \cdot t_3$
20 $x'_Q \leftarrow x_Q \cdot t_2$
21 $y'_Q \leftarrow y_Q \cdot t_1$
22 return $(x_{Q'}, y_{Q'})$
```

Algorithm 34: 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 $E_{a',b'}$
for $e = \mathtt{e}_2 - 2$ downto $0$ by $-2$ do
    $(x_T, y_T) \leftarrow$ xDBLe $((x_S, y_S), (a, b), e)$                    // Alg. 24
    $(a', b') \leftarrow$ curve_4_iso $(x_T, y_T)$                              // Alg. 30
    $(x_S, y_S) \leftarrow$ eval_4_iso $((x_S, y_S), x_T)$                      // Alg. 31
    for $(x_j, y_j)$ in optional input do
        $(x'_j, y'_j) \leftarrow$ eval_4_iso $\big(\big(x_j, y_j), x_T\big)\big($   // Alg. 31
return $(a', b'), [(x'_1, y'_1), ..., (x'_n, y'_n)]\big($
```

Algorithm 35: 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 $E_{a',b'}$
for $e = \mathtt{e3} - 1$ downto $0$ by $-1$ do
    $(x_T, y_T) \leftarrow$ xTPLe $((x_S, y_S), (a, b), e)$                    // Alg. 27
    $(a', b') \leftarrow$ curve_3_iso $(x_T, y_T)$                              // Alg. 32
    $(x_S, y_S) \leftarrow$ eval_3_iso $((x_S, y_S), x_T)$                      // Alg. 33
    for $(x_j, y_j)$ in optional input do
        $(x'_j, y'_j) \leftarrow$ eval_3_iso $\big(\big(x_j, y_j), x_T\big)\big($   // Alg. 33
return $(a', b'), [(x'_1, y'_1), ..., (x'_n, y'_n)]\big($
```

Algorithm 36: 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 $R = P - Q$
1 $(x_R, y_R) \leftarrow$ xADD $((x_P, y_P), (x_Q, -y_Q), (a, b))$                    // Alg. 25
2 return $x_R$
```

Algorithm 37: 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: $(y_P, y_Q, a, b)$
1 $a \leftarrow$ get_A$(x_P, x_Q, x_R)$                    // Alg. 1.2.9
2 $b \leftarrow 1$
3 $t_1 \leftarrow {x_P}^2$          6 $t_1 \leftarrow t_2 + t_1$          9 $t_1 \leftarrow {x_Q}^2$          12 $t_1 \leftarrow t_2 + t_1$
4 $t_2 \leftarrow x_P \cdot t_1$    7 $t_1 \leftarrow t_1 + x_P$          10 $t_2 \leftarrow x_Q \cdot t_1$   13 $t_1 \leftarrow t_1 + x_Q$
5 $t_1 \leftarrow a \cdot t_1$      8 $y_P \leftarrow \sqrt{t_1}$         11 $t_1 \leftarrow a \cdot t_1$     14 $y_Q \leftarrow \sqrt{t_1}$
15 $(x_T, y_T) \leftarrow$ xADD $((x_P, y_P), (x_Q, -y_Q), (a, b))$                    // Alg. 25
16 if $x_T \neq x_R$ then
17     $y_Q \leftarrow -y_Q$
18 return $(y_P, y_Q, a, b)$
```

Algorithm 38: Computing public keys in the 2-torsion

```
function isogen$_2$
    Input: Secret key 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 §1.5)
    Output: Public key $pk_2 = (x'_{P3}, x'_{Q3}, x'_{R3})$ equivalent to the output of Step 4 of isogen$_\ell$
            (see §1.3.5)
1 $(a, b) \leftarrow (0, 1)$
2 $(x_S, y_S) \leftarrow$ mult_double_add $(\text{sk}_2, (x_{Q2}, y_{Q2}), (a, b))$                    // Alg. 28
3 $(x_S, y_S) \leftarrow$ xADD $((x_{P2}, y_{P2}), (x_S, y_S), (a, b))$                    // Alg. 25
4 $((a', b'), (x'_{P3}, y'_{P3}), (x'_{Q3}, y'_{Q3})) \leftarrow$ 2_e_iso $((a, b), (x_S, y_S), (x_{P3}, y_{P3}), (x_{Q3}, y_{Q3}))$   // Alg. 34
5 $x'_{R3} \leftarrow$ get_xR $((a, b), (x'_{P3}, y'_{P3}), (x'_{Q3}, y'_{Q3}))$
6 return $pk_2 = (x'_{P3}, x'_{Q3}, x'_{R3})$                    // Encoded as in §1.2.9
```

Algorithm 39: 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
        $\{e2, e3, p, (x_{P2}, y_{P2}), (x_{Q2}, y_{Q2}), (x_{P3}, y_{P3}), (x_{Q3}, y_{Q3})\}$ (see §1.5)
    Output: Public key $pk_3 = (x'_{P2}, x'_{Q2}, x'_{R2})$ equivalent to the output of Step 4 of isogen_ℓ
        (see §1.3.5)
1 $(a, b) \leftarrow (0, 1)$
2 $(x_S, y_S) \leftarrow$ mult_double_add $(\mathrm{sk}_3, (x_{Q3}, y_{Q3}), (a, b))$        // Alg. 28
3 $(x_S, y_S) \leftarrow$ xADD $((x_{P3}, y_{P3}), (x_S, y_S), (a, b))$        // Alg. 25
4 $\big((a', b'), (x'_{P2}, y'_{P2}), (x'_{Q2}, y'_{Q2})\big) \leftarrow$ 3_e_iso $((a, b), (x_S, y_S), (x_{P2}, y_{P2}), (x_{Q2}, y_{Q2}))$        // Alg. 35
5 $x'_{R3} \leftarrow$ get_xR $\big((a, b), (x'_{P2}, y'_{P2}), (x'_{Q2}, y'_{Q2})\big)$
6 return $pk_3 = (x'_{P2}, x'_{Q2}, x'_{R2})$        // Encoded as in §1.2.9
```

Algorithm 40: 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 $pk_3 = (x'_{P2}, x'_{Q2}, x'_{R2}) \in (\mathbb{F}_{p^2})^3$ (see §1.2.9),
        and parameter e2 (see §1.5)
    Output: A $j$-invariant $j_2$ equivalent to the output of Step 4 of isogen_ℓ (see §1.3.6)
1 $(y'_{P2}, y'_{Q2}, a, b) \leftarrow$ get_yP_yQ_A_B$(x'_{P2}, x'_{Q2}, x'_{R2})$        // Alg. 37
2 $(x_S, y_S) \leftarrow$ mult_double_add $\big(\mathrm{sk}_2, (x'_{Q2}, y'_{Q2}), (a, b)\big)$        // Alg. 28
3 $(x_S, y_S) \leftarrow$ xADD $\big((x'_{P2}, y'_{P2}), (x_S, y_S), (a, b)\big)$        // Alg. 25
4 $(a, b) \leftarrow$ 2_e_iso $((a, b), (x_S, y_S))$        // Alg. 34
5 $j_2 =$ j_inv$((a, b))$        // Alg. 29
6 return $j_2$        // Encoded as in §1.2.8
```

Algorithm 41: 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 §1.5)
    Output: A $j$-invariant $j_3$ equivalent to the output of Step 4 of isogen_$\ell$ (see §1.3.6)
1 $(y'_{P3}, y'_{Q3}, a, b) \leftarrow$ get_yP_yQ_A_B$(x'_{P3}, x'_{Q3}, x'_{R3})$          // Alg. 37
2 $(x_S, y_S) \leftarrow$ mult_double_add$\big(\mathrm{sk}_3, (x'_{Q3}, y'_{Q3}), (a, b)\big)\Big($          // Alg. 28
3 $(x_S, y_S) \leftarrow$ xADD$\big((x'_{P3}, y'_{P3}), (x_S, y_S), (a, b)\big)\Big($          // Alg. 25
4 $(a, b) \leftarrow$ 3_e_iso$((a, b), (x_S, y_S))$          // Alg. 35
5 $j_3 =$ j_inv$((a, b))$          // Alg. 29
6 return $j_3$          // Encoded as in §1.2.8
```

Appendix C

Computing optimized strategies for fast isogeny computation

Algorithms 17 and 18 need to be parameterized by a computational strategy as described in Section 1.3.7. 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 42: Computing optimized strategy

```
function compute_strategy
    Input: Strategy size $n$, parameters $p, q > 0$
    Output: Optimal strategy of size $n$
1 $S \leftarrow [1 \rightarrow \epsilon]$
2 $C \leftarrow [1 \rightarrow 0]$
3 for $i = 2$ to $n + 1$ do
4     Set $b \leftarrow \operatorname{argmin}_{0<b<i}(C[i-b] + C[b] + bp + (i-b)q)$
5     Set $S[i] \leftarrow b \,.\, S[i-b] \,.\, S[b]$
6     Set $C[i] \leftarrow C[i-b] + C[b] + bp + (i-b)q$
7 return $S[n+1]$
```

C.1 Strategies for SIKEp503

C.1.1 2-torsion

$$
\begin{aligned}
n_4 &= 124 \\
p_4 &= 7290 \\
q_4 &= 7278 \\
S_4 &= (61, 32, 16, 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, \\
&\quad 16, 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 29, 16, \\
&\quad 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 13, 8, 4, 2, \\
&\quad 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 5, 4, 2, 1, 1, 2, 1, 1, 2, 1, 1, 1)
\end{aligned}
$$

C.1.2 3-torsion

$$
\begin{aligned}
n_3 &= 158 \\
p_3 &= 7189 \\
q_3 &= 7051 \\
S_3 &= (71, 38, 21, 13, 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 5, 4, 2, 1, 1, 2, 1, 1, 2, 1, 1, 1, \\
&\quad 9, 5, 3, 2, 1, 1, 1, 1, 2, 1, 1, 1, 4, 2, 1, 1, 1, 2, 1, 1, 17, 9, 5, 3, 2, 1, 1, 1, 1, 2, 1, 1, 1, \\
&\quad 4, 2, 1, 1, 1, 2, 1, 1, 8, 4, 2, 1, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 33, 17, 9, 5, 3, 2, 1, 1, \\
&\quad 1, 1, 2, 1, 1, 1, 4, 2, 1, 1, 1, 2, 1, 1, 8, 4, 2, 1, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 16, 8, 4, \\
&\quad 2, 1, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1)
\end{aligned}
$$

C.2 Strategies for SIKEp751

C.2.1 2-torsion

$$
\begin{aligned}
n_4 &= 185 \\
p_4 &= 14166 \\
q_4 &= 13810 \\
S_4 &= (80, 48, 27, 15, 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 7, 4, 2, 1, 1, 2, 1, 1, 3, 2, 1, 1, 1, 1, \\
&\quad 12, 7, 4, 2, 1, 1, 2, 1, 1, 3, 2, 1, 1, 1, 1, 5, 3, 2, 1, 1, 1, 1, 2, 1, 1, 1, 21, 12, 7, 4, 2, 1, 1,
\end{aligned}
$$

2, 1, 1, 3, 2, 1, 1, 1, 1, 5, 3, 2, 1, 1, 1, 1, 2, 1, 1, 1, 9, 5, 3, 2, 1, 1, 1, 1, 2, 1, 1, 1, 4, 2, 1,
1, 1, 2, 1, 1, 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,
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,
1, 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1)

C.2.2 3-torsion

n_3 = 238

p_3 = 13898

q_3 = 13409

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,
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,
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,
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,
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,
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,
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)

C.3 Strategies for SIKEp964

C.3.1 2-torsion

n_4 = 242

p_4 = 19900

q_4 = 19420

S_4 = (116, 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, 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, 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, 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, 57, 27, 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, 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, 27, 14, 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2,
1, 1, 6, 4, 2, 1, 1, 2, 1, 1, 2, 2, 1, 1, 1, 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)

C.3.2 3-torsion

$$
\begin{aligned}
\mathtt{n}_3 &= \mathtt{300}\\
\mathtt{p}_3 &= \mathtt{19660}\\
\mathtt{q}_3 &= \mathtt{18870}\\
\mathtt{S}_3 &= (137, 73, 39, 22, 14, 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 6, 4, 2, 1, 1, 2, 1, 1, 2, 2, 1, 1,\\
&\quad 1, 9, 5, 4, 2, 1, 1, 2, 1, 1, 2, 1, 1, 1, 4, 2, 1, 1, 1, 2, 1, 1, 17, 9, 5, 4, 2, 1, 1, 2, 1, 1, 2, 1, 1,\\
&\quad 1, 4, 2, 1, 1, 1, 2, 1, 1, 8, 4, 2, 1, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 33, 17, 9, 6, 4, 2, 1, 1, 2, 1,\\
&\quad 1, 2, 2, 1, 1, 1, 4, 2, 1, 1, 1, 2, 1, 1, 8, 4, 2, 1, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 16, 8, 4, 2, 1,\\
&\quad 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 65, 33, 17, 9, 5, 4, 2,\\
&\quad 1, 1, 2, 1, 1, 2, 1, 1, 1, 4, 2, 1, 1, 1, 2, 1, 1, 8, 4, 2, 1, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 16, 8,\\
&\quad 4, 2, 1, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 32, 16, 8, 4,\\
&\quad 2, 1, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 16, 8, 4, 2, 1, 1,\\
&\quad 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1)
\end{aligned}
$$

Appendix D

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 §1.3.5
$\texttt{isoex}_\ell$	The algorithm that establishes shared keys – see §1.3.6
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, $e_{2^{e_3}}(P_3, Q_3)$, 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
SIDH	Supersingular Isogeny Diffie–Hellman
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
`cSHAKE256`	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`