Supersingular Isogeny Key Encapsulation
https://sike.org/
April 17, 2019

Principal submitter: David Jao
E-mail address: djao@uwaterloo.ca
Telephone: +1-519-888-4567 x32493
Affiliation: University of Waterloo & evolutionQ, Inc.
Postal address:
Department of Combinatorics & Optimization
University of Waterloo
200 University Ave. W
Waterloo, Ontario N2L 3G1
Canada

Auxiliary submitters:
- Reza Azarderakhsh, Florida Atlantic University & PQSecure Technologies, LLC, USA
- Matthew Campagna, Amazon Inc., USA
- Craig Costello, Microsoft Research, USA
- Luca De Feo, UVSQ & Inria, Université de Paris-Saclay, France
- Basil Hess, InfoSec Global, Switzerland
- Amir Jalali, LinkedIn Corporation, USA
- Brian Koziel, Texas Instruments, USA
- Brian LaMacchia, Microsoft Research, USA
- Patrick Longa, Microsoft Research, USA
- Michael Naehrig, Microsoft Research, USA
- Geovandro Pereira, University of Waterloo & evolutionQ, Inc., Canada
- Joost Renes, Radboud University, The Netherlands
- Vladimir Soukharev, InfoSec Global, Canada
- David Urbanik, University of Toronto, Canada

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

Owner: Same as submitters.

Signature:
Contents

1 The SIKE protocol specification ................................. 1
   1.1 Mathematical Foundations .................................. 1
   1.2 Data types and conversions .................................. 7
   1.3 Detailed protocol specification ............................. 11
   1.4 Symmetric primitives ....................................... 18
   1.5 Public key compression ..................................... 18
   1.6 Parameter sets ............................................. 21

2 Detailed performance analysis ................................. 29
   2.1 Reference implementation ................................... 30
   2.2 Optimized and x64 assembly implementations ............... 31
   2.3 Compressed SIKE implementation ........................... 34
   2.4 64-bit ARM assembly implementation ......................... 35
   2.5 VHDL hardware implementation ............................... 35

3 Known Answer Test values ..................................... 38

4 Expected security strength ..................................... 39
   4.1 Security .................................................. 39
   4.2 Other attacks ............................................. 40
   4.3 Security proofs ........................................... 41

5 Analysis with respect to known attacks ...................... 43
   5.1 Classical security ......................................... 43
   5.2 Quantum security ......................................... 44
   5.3 Side-channel attacks ...................................... 45
6 Advantages and Limitations

A Explicit algorithms for $\text{isogen}_\ell$ and $\text{isoox}_\ell$: Optimized implementation

B Explicit algorithms for $\text{isogen}_\ell$ and $\text{isoox}_\ell$: Reference implementation

C Computing optimized strategies for fast isogeny computation
  C.1 Strategies for SIKEp434
  C.2 Strategies for SIKEp503
  C.3 Strategies for SIKEp610
  C.4 Strategies for SIKEp751

D Explicit algorithms for compressed SIKE: Optimized implementation

E Changes made in the 2nd round

F Notation
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 [20], and subsequently improved in various ways by numerous authors [7, 8, 11, 27]. 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 F.

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 \( \text{char}(\mathbb{F}_q) \) 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 $(a_0 + a_1 \cdot i)(b_0 + b_1 \cdot i) = (a_0b_0 - a_1b_1) + (a_0b_1 + a_1b_0) \cdot i$ in $\mathbb{F}_{p^2}$, where the addition, additive inverse and multiplication operations take place in $\mathbb{F}_p$.
- **Additive Inverse:** If $a \in \mathbb{F}_{p^2}$, then $(-a_0) + (-a_1) \cdot i \in \mathbb{F}_{p^2}$ is the additive inverse $(-a)$, where the values $(-a_i)$ are computed in the field $\mathbb{F}_p$.
- **Multiplicative Inversion:** If $a \in \mathbb{F}_p$, $a \neq 0$, then $(a_0(a_0^2 + a_1^2)^{-1} + ((-a_1)(a_0^2 + a_1^2)^{-1}) \cdot i) \in \mathbb{F}_{p^2}$ is the multiplicative inverse $a^{-1}$, where the operations take place in $\mathbb{F}_p$.
- **Square root:** If there exists an $r = \alpha + \beta \cdot i \in \mathbb{F}_{p^2}$ with $\alpha, \beta \in \mathbb{F}_p$ such that $r^2 = s$, then we define $\sqrt{s} = r$ if either $\alpha \neq 0$ is an even integer or $\alpha = 0$ and $\beta$ is an even integer, otherwise $\sqrt{s} = -r$. 

2
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.
- \( E_A \) when the underlying field \( \mathbb{F}_q \) is fixed by context, or unspecified, and when \( B \) (which specifies the quadratic twist) is presumed to either be \( B = 1 \) or irrelevant.

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

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

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

1.1.5 Point addition

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

\[
x_R = BA^2 - (x_P + x_Q) - A
\]
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

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

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

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

and

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

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 \( 0 \leq k_i < \text{ord}(P_i) \) 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 \([29]\) can assist in determining whether or not a set forms a basis, since for \( n = \text{ord}(P) = \text{ord}(Q) \), the order-\( n \) Weil pairing \( e_n \) has the property that \( \text{ord}(e_n(P, Q)) = n \) if and only if \( \langle P \rangle \cap \langle Q \rangle = \{O\} \).

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 \( E(\mathbb{F}_q) \) such that \( [m]P = O \).

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 $\phi: E_1 \to E_2$ 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 $\phi$ 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 $\phi: E_1 \to E_2$ we define the kernel of $\phi$ as follows:

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

For any finite subgroup $H$ of $E(\mathbb{F}_q)$, there is a unique isogeny (up to isomorphism) $\phi: E \to E'$ 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 [41] can be used to find the isogeny $\phi$ and isogenous curve $E/H$. Vélu’s formula is computationally impractical for arbitrary subgroups. SIKE uses isogenies over subgroups that are powers of 2, 3 and 4.

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

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

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

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

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

and

$$y_{\phi_2(P)} = y_P \cdot \frac{x_P^2x_2 - 2x_px_2^2 + x_2}{(x_P - x_2)^2}.$$ 

Observe that $x_{\phi_2(P)}$ only depends on $x_P$ and $x_2$. The inversion-free algorithm that takes advantage of this is given in Algorithm 12 of Appendix A.
4-isogenies  Let \((x_4, y_4) \in E_{A,B}\) be a point of order 4 with \(x_4 \neq \pm 1\) and let \(\phi_4 : E_{A,B} \to E_{A',B'}\) be the unique (up to isomorphism) 4-isogeny with kernel \((x_4, y_4)\). Then \(E_{A',B'}\) can be computed as

\[
(A', B') = \left( 4x_4^4 - 2, -x_4(x_4^3 + 1) \cdot B/2 \right)
\]

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

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

\[
x_{\phi_4(P)} = \frac{-(x_Px_4^2 + x_P - 2x_4)x_P(x_Px_4 - 1)^2}{(x_P - x_4)^2(2x_Px_4 - x_4^2 - 1)},
\]

and

\[
y_{\phi_4(P)} = y_P \cdot \frac{-2x_4^2(x_Px_4 - 1)(x_P^2x_4^2 + 1) - 4x_P^3(x_4^3 + x_4) + 2x_P^2(x_4^4 + 5x_4^2) - 4x_P(x_4^3 + x_4)x_4^2 + 1)}{(x_P - x_4)^3(2x_Px_4 - x_4^2 - 1)^2}.
\]

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

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

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

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

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

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

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 16 of Appendix A.

The SIKE protocol defines secret keys from two separate key spaces, \(\mathcal{K}_2\) and \(\mathcal{K}_3\) (cf. §1.3.8). A secret key \(sk\) defines a subgroup \(H\) of \(E(\mathbb{F}_q)\), which in turn defines an isogeny \(\phi_{sk} : E \to E/H\). The public key is determined by the isogeny \(\phi_{sk}\) 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 \((E/H, \phi_{sk}(P), \phi_{sk}(Q))\). The points \(P\) and \(Q\) are chosen so that \([P, Q]\) forms a basis for \(E[l']\). In our implementations, for efficiency reasons we represent a public key as a triplet of field elements, namely the three \(x\)-coordinates \(\{x_{\phi_{sk}(P)}, x_{\phi_{sk}(Q)}, x_{\phi_{sk}(P-Q)}\}\) of three points under the isogeny. It is possible to convert between
representations using the methods given in [8]. For example, the Montgomery curve coefficient \( A \) of \( E/H \) can be recovered by the three \( x \)-coordinates of a public key \( \{ x_{\phi_{sk}(P)}, x_{\phi_{sk}(Q)}, x_{\phi_{sk}(P-Q)} \} \) using the equation

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

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

\[
y_{\phi_{sk}(P)} = \sqrt{x_{\phi_{sk}(P)}^2 + A x_{\phi_{sk}(P)}^2 + x_{\phi_{sk}(P)}}
\]

and

\[
y_{\phi_{sk}(Q)} = \sqrt{x_{\phi_{sk}(Q)}^2 + A x_{\phi_{sk}(Q)}^2 + x_{\phi_{sk}(Q)}}
\]

and if

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

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

### 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 $m\text{len}$ and interprets the octet string in base $2^8$ of an integer.

**Input:** An octet string $M$ of length $m\text{len}$.

**Output:** An integer $a$.

**Action:** Convert $M$ to an integer $a$ as follows:

1. Parse $M = M_0M_1\ldots M_{m\text{len}-1}$ into $m\text{len}$-many octets.
2. Interpret each octet $M_i$ as an integer in $[0, 255]$.
3. Compute $a = \sum_{i=0}^{m\text{len}-1} M_i2^{8i}$.
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 $M = M_0M_1$ 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 $M = M_1M_2M_3$, where each $M_i$ is an octet string of length $2N_p$.
2. For $i \in \{1, 2, 3\}$ convert $M_i$ to a field element $x_i$ (cf. §1.2.4) or return FAIL.
3. Output $pk_\ell = (x_1, x_2, x_3)$.

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 $2^{8\cdot mlen} > a$.

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

**Output:** An octet string $M$ of length $mlen$ octets.

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

1. Convert $a = a_{mlen-1}2^{8(\cdot mlen-1)} + a_{mlen-2}2^{8(\cdot mlen-2)} + \cdots + a_12^8 + a_0$ represented in base $2^8$.
2. For $0 \leq i < mlen$, set $M_i = a_i$.
3. Form $M = M_0M_1\ldots M_{mlen-1}$.
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 $N_p = \lceil \log_2 p \rceil / 8$.

**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 $N_p = \lceil \log_2 p \rceil / 8$.

**Actions:** Compute the octet string as follows:

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

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

Compressed public keys $(\text{bit}, t_1, t_2, t_3, A, s, r) \in \mathbb{Z}_2 \times (\mathbb{Z}_{p^2})^3 \times \mathbb{F}_{p^2} \times \mathbb{Z}_{256}^2$ should be converted to octet strings as described in this section. This routine converts each component as an octet string encoding and concatenates them to form the output octet string.

In portions of the spec we will refer to a public key pk_comp in octet string format without explicitly referencing the compressed-public-key-to-octet-string conversion.
Input: A compressed public key \((bit, t_1, t_2, t_3, A, s, r)\) consisting of a bit, 3 elements in \(\mathbb{Z}_{\ell e}\), one element of the finite field and 2 bytes.

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

Actions: Compute the octet string as follows:

1. Convert \(t_1, t_2, t_3\) into the octet strings \(M_1, M_2, M_3\) respectively, each of length \(N_z\) (cf. §1.2.8).
2. Convert \(A\) into an octet string \(M_4\) of the length \(2 \cdot N_p\) (cf. §1.2.7).
3. Let \(M_5 = r\) of \(bit = 0\) and \(M_5 = r \lor 0x80\) if \(bit = 1\), i.e., \(bit\) is encoded in the most significant bit of the octet \(r\).
4. Set \(M_6 = s\), an octet.
5. Form \(M = M_1M_2M_3M_4M_5M_6\).
6. 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 [2, 7] 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 \(\mathbb{F}_{p^{e_2 e_3}}\) where \(p = 2^{e_2 e_3} - 1\),
- A starting supersingular elliptic curve \(E_0/\mathbb{F}_{p^{e_2}}\),
- 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
\[ E_0/F_p^2 : y^2 = x^3 + 6x^2 + x, \]
with \#\(E_0(F_p^2) = (2^e 3^e)^2\) and \(j\)-invariant equal to \(j(E_0) = 287496\). This is the special instance of the Montgomery curve \(B\) by \(y^2 = x^3 + Ax^2 + x\), where \(A = 6\) and \(B = 1\). Note that this has been updated since the initial proposal, for reasons that are further explained in [9, §5]. The original curve had \(j = 1728\), for which \(E_0\) above is the only non-isomorphic 2-isogenous curve, meaning an attacker would have known for certain the first step taken away from this starting point in the 2-isogeny graph, regardless of the secret. There also exist only two (as opposed to four generally) isomorphism classes that are 3-isogenous to \(E\) which the initial proposal, for reasons that are further explained in [9, §5]. The original curve had had \(j = 1728\), so that distinct kernels can lead to isomorphic isogenies. Starting on \(E_0\) avoids both of these problems. Moreover, the combination of the basis points on \(E_0\) (defined in §1.3.3) and the computation of the secret kernel subgroups (defined in §1.3.5) ensures that the first 2-isogeny taken from \(E_0\) is not in the direction of the curve with \(j = 1728\), but rather to one of the two other 2-isogenous curves.

1.3.3 Public generator points

The three \(x\)-coordinates in the public parameters corresponding to points in \(E_0[2^e]\) are specified as follows. We first specify two points
\[ P_2 \in E_0(F_p^2) \quad \text{and} \quad Q_2 \in E_0(F_p^2) \]
such that both points have exact order \(2^e\), and \([P_2, Q_2]\) forms a basis for \(E_0(F_p^2)[2^e]\), i.e., the order-\(2^e\) Weil pairing \(e_{2^e}(P_2, Q_2) \in F_p^\ast\) has full order, or equivalently, \(e_2([2^e-1]P_2, [2^e-1]Q_2) \in F_p^\ast\) is not equal to 1. Similarly, we specify two points
\[ P_3 \in E_0(F_p^2) \setminus E_0(F_p) \quad \text{and} \quad Q_3 \in E_0(F_p) \]
such that both points have exact order \(3^e\), and \([P_3, Q_3]\) forms a basis for \(E_0(F_p^2)[3^e]\).

Let \(f := x^3 + 6x^2 + x\) and recall \(F_p^2 \cong F_p(i)\) with \(i^2 + 1\). The points \(P_2, Q_2, P_3, Q_3\) are determined according to the following procedure:

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

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

In this section we fix $\ell, m \in \{2, 3\}$ such that $\ell \neq m$. The two fundamental isogeny algorithms described are $\text{isogen}_\ell$ and $\text{isotex}_\ell$. On input of the public parameters and a secret key, $\text{isogen}_\ell$ outputs the public key corresponding to the input secret key. On input of a secret key and a public key, $\text{isotex}_\ell$ 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 $\ell^e\ell$-degree isogeny via the composition of $e\ell$ individual $\ell$-degree isogenies; these $\ell$-degree isogenies are evaluated on at least one point lying on the domain curve. Following [8, 11], rather than evaluating the image of an isogeny on a point $R = (x_R, y_R)$, it is more efficient to evaluate its image under the $x$-only projection $(x_R, y_R) \mapsto x_R$. Since the coordinate maps for an isogeny $\psi: E \to E'$, $R \mapsto \psi(R)$ can always be written such that $x_{\psi(R)} = f(x_R)$ for some function $f$ [41], the $\text{isogen}_\ell$ and $\text{isotex}_\ell$ algorithms will assume the $i$-th $\ell$-degree isogeny $\phi_i$ adheres to this framework by writing $\phi_i: (x, -) \mapsto (f_i(x), -)$.

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: $\text{isogen}_\ell$

A supersingular isogeny key pair consists of a secret key $sk_\ell$, which is an integer, and a set of three $x$-coordinates $pk_\ell = (x_P, x_Q, x_R)$.

Public parameters. A prime $p = 2^e 3^e - 1$, the starting curve $E_0/F_{p^2}$, 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 $sk_\ell$.

Output. A public key $pk_\ell$.

Actions. Compute a public key $pk_\ell$, as follows:

1. Set $x_S \leftarrow x_{P_\ell + \ell [sk_\ell]Q_\ell}$;
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 $\ell$-isogeny $\phi_i: E_i \to E'$
      $(x, -) \mapsto (f_i(x), -)$
      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 $x_S \leftarrow f_i(x_S)$;
(d) Set $(x_1, x_2, x_3) \leftarrow (f_i(x_1), f_i(x_2), f_i(x_3))$;

4. Output $pk_{\ell} = (x_1, x_2, x_3)$.

1.3.6 Establishing shared keys: isoex$\ell$

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

Input. A public key $pk_m = (x_{P_m}, x_{Q_m}, x_{R_m})$ and a secret key $sk_{\ell}$.


Actions. Compute a shared secret $j$, as follows:

1. Compute $E'_0$ from $pk_m$ using cfpk (cf. §1.2.1);
2. Set $x_S \leftarrow x_{P_m} + [sk_{\ell}]Q_m$;
3. For $i$ from 0 to $e_{\ell} - 1$ do
   (a) Compute the $x$ portion for an $\ell$-isogeny
      \[
      \phi_i : E'_i \to E'
      \]
      \[
      (x, \quad) \longmapsto (f_i(x), \quad)
      \]
      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) $e_{\ell}$ multiplications by the scalar $\ell$, and is repeated $e_{\ell}$ times, for a total of $O(e_{\ell}^2)$ elementary operations.

In optimized implementations, following [11], it is recommended to replace the for loops by a recursive decomposition of the isogeny computation into elementary operations, requiring only $O(e_{\ell} \log e_{\ell})$ multiplications by the scalar $\ell$, and a similar amount of evaluations of $\ell$-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
leaves on one border, then the path length of the tree is proportional to the computational effort required by the strategy. See Figure 1.1 for an example, and [11, §4] for a more formal definition.

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.

\[
\text{Linearization: } (3, 2, 1, 1, 2, 1)
\]

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 \((x_1, x_2, \ldots)\) on \( E \). A strategy \((s_1, \ldots, s_{t-1})\) 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 \((\psi(x_1), \psi(x_2), \ldots)\) on \( E' \).

**Actions.**

1. If \( t = 1 \) (i.e., the strategy is empty) then
   (a) Compute an \( \ell \)-isogeny

\[
\phi : E \to E' \\
(x, \ldots) \mapsto (f(x), \ldots)
\]

such that \( \ker \phi = \langle S \rangle \);

(b) Return \((E', f(x_1), f(x_2), \ldots)\);

2. Let \( n = s_1 \); \footnote{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 $L = (s_2, \ldots, s_{t-n})$ and $R = (s_{t-n+1}, \ldots, s_{t-1})$;
4. Set $x_T \leftarrow x_{[\ell]S}$;
5. Set $(E, (x_U, x_1, x_2, \ldots)) \leftarrow$ Recurse on $(E, x_T, (x_S, x_1, x_2, \ldots))$ with strategy $L$;
6. Set $(E, (x_1, x_2, \ldots)) \leftarrow$ Recurse on $(E, x_U, (x_1, x_2, \ldots))$ with strategy $R$;
7. Return $(E, (x_1, x_2, \ldots))$.

A similar algorithm, without the inputs $(x_1, x_2, \ldots)$, can be replaced inside $\text{isoex}_\ell$ to obtain the same speedup. Remark that the simple algorithms of Sections 1.3.5 and 1.3.6 correspond to the strategy $(e_\ell - 1, \ldots, 2, 1)$. 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 $(s_1, \ldots, s_{t-1})$ correctly defines a strategy, i.e. that it belongs to the language $S$, defined by the following grammar:

$$S_1 ::= \epsilon,$$

$$S_{a+b} ::= b \cdot S_a \cdot S_b.$$

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

**Actions.**

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

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

### 1.3.8 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 $\mathbb{N}_{sk_2} = \lceil e_2/8 \rceil$. Secret keys $sk_2$ correspond to integers in the range $\{0, 1, \ldots, 2^{e_2} - 1\}$, encoded as an octet string of length $\mathbb{N}_{sk_2}$ using $\text{itoos}$ (cf. §1.2.6). The corresponding keyspace is denoted $K_2$.

Let $s = \lceil \log_2 3^{e_3} \rceil$ and $\mathbb{N}_{sk_3} = \lceil s/8 \rceil$. Secret keys $sk_3$ correspond to integers in the range $\{0, 1, \ldots, 2^s - 1\}$, encoded as an octet string of length $\mathbb{N}_{sk_3}$ using $\text{itoos}$ (cf. §1.2.6). The corresponding keyspace is denoted $K_3$. 

16
1.3.9 Public-key encryption

Algorithm 1 defines a public-key encryption scheme PKE = (Gen, Enc, Dec) [11, §3.3]. The two keyspaces \( \mathcal{K}_2 \) and \( \mathcal{K}_3 \) are defined in 1.3.8. The size of the message space \( M = \{0, 1\}^n \), 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 \( \text{Enc} \) generates randomness \( \text{sk}_2 \). In the case of the key encapsulation mechanism we want to pass this randomness as input, in which case we write \((c_0, c_1) \leftarrow \text{Enc}(pk_3, m; \text{sk}_2)\) (see Line 7 of Algorithm 2).

Algorithm 1: PKE = (Gen, Enc, Dec)

<table>
<thead>
<tr>
<th>function Gen</th>
<th>function Enc</th>
<th>function Dec</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong></td>
<td><strong>Input:</strong></td>
<td><strong>Input:</strong></td>
</tr>
<tr>
<td>()</td>
<td>( pk, m \in M )</td>
<td>( \text{sk}, (c_0, c_1) )</td>
</tr>
<tr>
<td><strong>Output:</strong></td>
<td><strong>Output:</strong></td>
<td><strong>Output:</strong></td>
</tr>
<tr>
<td>( \text{pk}_3, \text{sk}_3 )</td>
<td>( (c_0, c_1) )</td>
<td>( m )</td>
</tr>
<tr>
<td>( \text{sk}_3 \leftarrow \mathcal{K}_3 )</td>
<td>( \text{sk}_2 \leftarrow \mathcal{K}_2 )</td>
<td>( j \leftarrow \text{isogen}_3(c_0, \text{sk}_3) )</td>
</tr>
<tr>
<td>( \text{pk}_3 \leftarrow \text{isogen}_3(\text{sk}_3) )</td>
<td>( c_0 \leftarrow \text{isogen}_2(\text{sk}_2) )</td>
<td>( h \leftarrow F(j) )</td>
</tr>
<tr>
<td>return</td>
<td>( j \leftarrow \text{isoex}_3(\text{pk}_3, \text{sk}_2) )</td>
<td>( m \leftarrow h \oplus c_1 )</td>
</tr>
<tr>
<td></td>
<td>( h \leftarrow F(j) )</td>
<td>return</td>
</tr>
<tr>
<td></td>
<td>( c_1 \leftarrow h \oplus m )</td>
<td>( m )</td>
</tr>
<tr>
<td></td>
<td>return</td>
<td></td>
</tr>
</tbody>
</table>

1.3.10 Key encapsulation mechanism

Algorithm 2 defines a key encapsulation mechanism KEM = (KeyGen, Encaps, Decaps) by applying a transformation of Hofheinz, Hövelmanns and Kiltz [19] to the PKE defined in §1.3.9. We slightly modify this transformation by including \( pk_3 \) in the input to \( G \) (as in [4]), and by simplifying “re-encryption” (see the proof of Theorem 1). Again, The two keyspaces \( \mathcal{K}_2 \) and \( \mathcal{K}_3 \) are defined in 1.3.8. The size of \( M = \{0, 1\}^n \) 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:

```c
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)**

<table>
<thead>
<tr>
<th>function KeyGen</th>
<th>function Encaps</th>
<th>function Decaps</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> ()</td>
<td><strong>Input:</strong> pk</td>
<td><strong>Input:</strong> (s, sk₃, pk₃)</td>
</tr>
<tr>
<td><strong>Output:</strong> (s, sk₃, pk₃)</td>
<td><strong>Output:</strong> (c, K)</td>
<td><strong>Output:</strong> (c, K)</td>
</tr>
<tr>
<td>1 sk₃ ←₉ 𝐾₃</td>
<td>5 m ←₉ {0, 1}ⁿ</td>
<td>10 m' ← Dec(sk₃, (c₀, c₁))</td>
</tr>
<tr>
<td>2 pk₃ ← isogen₃(sk₃)</td>
<td>6 r ← G(m</td>
<td></td>
</tr>
<tr>
<td>3 s ←₉ {0, 1}ⁿ</td>
<td>7 (c₀, c₁) ← Enc(pk₃, m; r)</td>
<td>12 c₀' ← isogen₂(r')</td>
</tr>
<tr>
<td>4 return (s, sk₃, pk₃)</td>
<td><strong>return</strong> ((c₀, c₁), K)</td>
<td><strong>if</strong> c₀' = c₀ <strong>then</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>13 K ← H(m'</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>else</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>16 K ← H(s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>17 return K</td>
</tr>
</tbody>
</table>

The public key pk is given by pk₃. The secret key sk consists of the concatenation of s, sk₃ and pk₃². The ciphertext ct consists of the concatenation of c₀ and c₁. 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 SHAKE256 as specified by NIST in [13].

Specifically, the function G hashes the random bit string m ∈ M = {0, 1}ⁿ concatenated with the public key pk₃. It is instantiated with SHAKE256, taking m || pk₃ as the input, requesting e₂ output bits. In the notation of [13], this means G(m || pk₃) = SHAKE256(m || pk₃, e₂). The value n corresponds to n ∈ {128, 192, 256}.

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

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

### 1.5 Public key compression

Recall that uncompressed SIKE public keys are of the form (xₚ, xₐ, xₗ) ∈ (𝔽ₚⁿ)³, which correspond to three points P, Q, R ∈ Eₐ(𝔽ₚⁿ) of exact order ℓⁿ, for ℓ ∈ {2, 3}. Following [2] (and the further improvements described in [7, 42]), the idea of public key compression is to instead represent these points as elements of ²Since NIST’s decapsulation API does not include an input for the public key, it needs to be included as part of the secret key.
Below we describe the compression and decompression algorithms, \texttt{compress}_\ell and \texttt{decompress}_\ell. Note that, since all of the operations in both algorithms are performed on public data, side-channel countermeasures (e.g., constant-time routines) are irrelevant except by the last step of \texttt{decompress}_\ell that computes the last kernel generator and employs the ladder3pt in Algorithm 8. We point out that several alternatives for subroutines in both compression and decompression are possible. For example, Step 5 of \texttt{compress}_\ell requires the solutions of 2-dimensional discrete logarithm problems in $E(\mathbb{F}_{p^2})[\ell^e]$, which can be solved directly in $E(\mathbb{F}_{p^2})[\ell^e]$ (cf. [36]), but for improved performance our optimized implementation instead transports the problems to multiple 1-dimensional discrete logarithms in $\mathbb{F}_p^\times$, by way of the Tate pairing $[2, 7, 42]$.

### 1.5.1 Public key compression: \texttt{compress}_\ell

**Input.** Three $x$-coordinates $pk_\ell = (x_p, x_Q, x_R)$ where $P, Q, R$ are $\ell^e$-torsion points and $\ell^e$ means the complementary torsion to $\ell^e$.

**Output.** A compressed public key $\text{PK} = (\text{bit}, t_1, t_2, t_3, A, s, r) \in \mathbb{Z}_2 \times (\mathbb{Z}_{\ell^e})^3 \times \mathbb{F}_{p^2} \times \mathbb{Z}_{2^{56}}^2$, encoded as in 1.2.10.

**Actions.**

1. Compute $E_A$ from $(x_p, x_Q, x_R)$ using cfpk (see §1.2.1).
2. Recover the $y$-coordinates of $\pm P = (x_p, \pm y_p)$ and $\pm Q = (x_Q, \pm y_Q)$, and set $P$ and $Q$ such that $R = Q - P$ via the expressions at the end of §1.1.9.
3. If $\ell = 3$: Compute an entangled basis $\{U, V\}$ for $E_A[2^{c_2}]$ as follows:
   (a) Select table $T$ containing only QNR or QR values $v := 1/(1 + ur^2) \in \mathbb{F}_p$, depending on whether $A$ is a QR or QNR, respectively. We have $u = u_0^2 \in \mathbb{F}_p \backslash \mathbb{F}_p^\times$, $u_0 \in \mathbb{F}_p \backslash \mathbb{F}_p$ and $r > 0$ is a small counter that can be seen as a small element in $\mathbb{F}_p$.
   (b) Compute the first abscissa candidate $x = -A \cdot v$ which is non-square by construction.
   (c) If $x^3 + Ax^2 + x$ is a non-square, increment counter $r$ and try another $v$ until a point on the curve is found. Otherwise, the point $U := (x, \sqrt{x^3 + Ax^2 + x}) \in E_A$ has full order $2^{c_2}$ by the 2-descent result.
   (d) For $U = (x, y)$, the other generator is automatically defined as $V := (u_0 \cdot r \cdot x, u \cdot r^2 \cdot y)$ and $E_A[2^{c_2}] = \langle [3^{c_2}]U, [3^{c_2}]V \rangle$ (see Theorem 1 of [42]).
   (e) Store the information learned so that entangled basis generation is faster during decompression (Algorithm 49). The quadraticity of $A$ can be transmitted as a bit, and the counter $r$ is smaller than 256 with very high probability, and thus can be transmitted as a byte.
4. If $\ell = 2$: Compute a basis $\{U, V\}$ of $E_A[3^{c_2}]$ using a general Algorithm 54.

---

Note that here $u$ is a square instead of a non-square as in the original elligator.
(a) First, find a candidate point \((x_1, z_1)\) using the conventional elligator technique and scalar multiplication to test for full order correctness. Store the respective elligator counter \(s\).

(b) Once the first candidate is found, get a second candidate \((x_2, z_2)\) and check for linear independence using scalar multiplication (removal of cofactors needed) until a basis is found. Store the respective elligator counter \(s\) from the final second candidate.

5. Find \((\alpha P, \beta P) \in \mathbb{Z}_\ell \times \mathbb{Z}_\ell\) such that \(P = [\alpha P] U + [\beta P] V\) and \((\alpha Q, \beta Q) \in \mathbb{Z}_\ell \times \mathbb{Z}_\ell\) such that \(Q = [\alpha Q] U + [\beta Q] V\).

6. Compute \(PK \in (\mathbb{Z}_\ell)^3 \times \mathbb{F}_p^2 \times \mathbb{Z}_{256}^2\) as

\[
PK = \begin{cases} 
(0, \alpha^{-1} \beta, \alpha^{-1} \alpha \beta, A, s, r) & \text{if } \alpha \in \mathbb{Z}_\ell^* \\
(1, \beta^{-1} \alpha \beta, \beta^{-1} \alpha \beta, \beta^{-1} \beta, A, s, r) & \text{if } \beta \in \mathbb{Z}_\ell^*. 
\end{cases}
\]

7. Encode the compressed public key as an array of octets according to §1.2.10.

1.5.2 Public key decompression: \texttt{decompress}\(_\ell\)

**Input.** A public key \(PK\) encoded as in §1.2.10.

**Output.** A kernel generator \(R = (x, y)\) for the last isogeny computed by Algorithm 44 or 45.

**Actions.**

1. Decode \(PK\) into \((\text{bit}, t_1, t_2, t_3, A, s, r) \in \mathbb{Z}_2 \times (\mathbb{Z}_\ell)^3 \times \mathbb{F}_p^2 \times \mathbb{Z}_{256}^2\) as in 1.2.10.

2. If \(\ell = 2\), find entangled basis \(\{U, V\}\) from \(A, s\) and \(r\) using Algorithm 55.

3. If \(\ell = 3\), find entangled basis \(\{U, V\}\) from \(A, \text{entang\_bit}\) and \(r\) using Algorithm 49.

4. Project the secret key and coefficients \(t_i\) into the basis \(\{U, V\}\) in order to recover the kernel generator \(R = (x, y)\) for the last isogeny. Algorithms 68 and 69 are tailored for this task.

**Remark 1.** It is worth mentioning that both SIKE public keys and ciphertexts are compressible. Due to the asymmetry in the original SIDH construction (binary and ternary torsions), compression techniques are faster in the binary torsion, and therefore torsions in Algorithms 2 are swapped for compression. This implies that the most frequently used operation (Encapsulation) performs the fastest compression \(\text{compress}_\ell\) for \(\ell = 3\) which compressed points that are in the \(2^e\)-torsion subgroup.

**Remark 2.** Due to an optimization used by Algorithms 68 and 69 introduced in [7] that can save one scalar multiplication, compression techniques require the secret key \(sk_2\) to be an even number and \(sk_3\) a multiple of 3. This is incorporated in the optimized additional implementation.

\[\text{Note that the points } pk, \text{ are in the complementary torsion other than } \ell.\]
1.6 Parameter sets

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

The four sets of parameters are SIKEp434, SIKEp503, SIKEp610, and SIKEp751, 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 $x_{Q,0}$ and $x_{Q,1}$ such that $x_Q = x_{Q,0} + x_{Q,1} \cdot i$; the values $x_{P,0}$ and $x_{P,1}$ such that $x_P = x_{P,0} + x_{P,1} \cdot i$; the values $x_{R,0}$ and $x_{R,1}$ such that $x_R = x_{R,0} + x_{R,1} \cdot i$.

1.6.1 SIKEp434

\[
p = \text{0002341F 27177344 6CFC5FD6 81C52056 7BC65C78 3158AE3A FDC1767AEEFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFF

\[
e_2 = \text{000000D8}
e_3 = \text{00000089}
\]

\[
x_{Q,0} = \text{0000C746 1738340E FCF09CE3 88F666EB 38F7F3AF D42DC0B6 64D9F461}
F31AA2ED C6B4AB71 BD42F4D7 C058E13F 64B237EF 7DDD2ABC 0DEB0C6C
\]

\[
x_{Q,1} = \text{000025DE 37157F50 D75D320D D0682AB4 A67E4715 86FBC2D3 1AA32E69}
57FA2B26 14C4CD40 A1E27283 EAAF4272 AE517847 197432E2 D61C85F5
\]

\[
y_{Q,0} = \text{0001D407 B70B01E4 AEE172ED F491F4EF 32144F03 F5E054CE F9FDEA53}
5EFA3642 A1181790 5ED0D4F1 93F31124 264924A5 F64EFE14 B6EC97E5
\]

\[
y_{Q,1} = \text{0000E7DE C83C2F50 A4E735A8 39DCDB89 FE0763A1 84C525F7 B7D0EBC0}
E84E9D83 E9AC53A5 72A25D19 E1464B50 9D97272A E761657B 4765B3D6
\]

\[
x_{P,0} = \text{00003CCF C5E1F050 630636E6 920A0F7A 4C6C71E6 3DE63A0E 6475AF62}
1995705F 7CB45BC0 C2BB61E9 50E19EAB 8661254C 4A50E27 9646C4B8
\]

\[
x_{P,1} = \text{0001AD1C 1CAE7840 EDADA6D8A 924520F6 0E573D3B 9DFAC6D1 89941CB2}
2326D2A8 48816CC4 249410FE 80D68047 D823C97D 705246F8 69E3EAA5
\]

\[
y_{P,0} = \text{0001AB06 6B849495 82E3F666 88452B92 55E72A01 7C451B14 D719D9A6}
3CDB7BE6 F48C812E 33B68161 D5AB3A0A 36906F04 A6A6957E 6F4FB2E0
\]

\[
y_{P,1} = \text{0000DF87 F67EA576 CE97FF65 BF9FF4F76 88C4C752 DCE9F8BD 2B36AD66}
E04249AA F8337C01 E6E4E1A8 44267BA1 A1887B43 3729E1DD 90C7DD2F
\]

\[
x_{R,0} = \text{0000F37A B34BA0CE AD94F43C DC5DE96 AD19C67C E4928346 E829CB92}
580DA84D 7C36506A 2516696B BE3AEB52 3AD7172A 6D239513 C5FD2516
\]

\[
x_{R,1} = \text{000196CA 2ED606A5 7E90A735 43F3902C 208F4108 95B49CF8 4CD89BE9}
ED6E4EE7 E8DF90B0 5F3FDB8B DFE489D1 B3558E98 7013F980 60365AC
1.6.2 SIKEp503

\[ p = \text{004066F5 418111E1E 6045C6BD DA77A4D0 1B9BF6C8 7B7E7DAF 13085BDA 2211E7A0 ABFFFFF7 FFFFFF FFFFFF FFFFFF FFFFFF FFFFFF FFFFFF FFFFFF} \]

\[ e2 = \text{000000FA} \]

\[ e3 = \text{0000009F} \]

\[ xQ20 = \text{00325CF6 A8E2C618 3A8B9932 198039A7 F965BA85 87B67925 D08D809D BF9A69DE 1B621F7F 134FA2DA B82FF5A2 615F92CC 71419FFF AAF86A29 0D604AB1 67616461} \]

\[ xQ21 = \text{003E7B04 94C8E60A 8B72308A E09ED348 45B34EA0 911E356B 77A11872 CF7FEEFF 745D98D0 624097BC 1AD7CD2A DF7FCC2C 1AA5BA3C 6684B964 FA555A07 15E57DB1} \]

\[ yQ20 = \text{003A3465 4000BD4C B2612017 BD5A1965 A9F89FE1 1C55D517 DF91B89B 94F4F9C5 8B9A9D05 56915573 FEDC09CC D4997E82 378759E0 0A2DE225 CE04589D 201FD754} \]

\[ yQ21 = \text{0019DEF0 E8930E51 23A22E34 6B1FFBD3 5EB01451 647D8708 A4835473} \]
xP20  =  0002ED31 A03825FA F280A99B 7BF86A1C E05D55BD 603C3BA9
       D7C08FD8 DE796B84 9A78851F FBC6D0A1 1CB2FA1B 57F3B4BE F87720DD
       9A489B55 81F915D2

yP20  =  00244D5F 814B6253 688138E3 17F24975 E596B09B B156418E E5295A6F
       73BA7F96 EFED145D FAE1B21A 8B7B121F EFA1B6E8 B52F0047 8218589E
       604B9735 9B8A6E0F

xP21  =  001EE4E4 E944F8BB AB45B5AE F280A99B 7BF86A1C E05D55BD 603C3BA9
       D7C08FD8 DE796B84 9A78851F FBC6D0A1 1CB2FA1B 57F3B4BE F87720DD
       9A489B55 81F915D2

yP21  =  00181CCC 9F0CE8E3 90CC1414 9E8DE88E E79992DA 32230DED B25F40FA
       DE07F242 A9057366 060CB599 27DB6DCC 820E61B1 747156E3 C5300545
       E96764A7 AB393CA7

xR20  =  003D24CF 1F347F1D A54C1696 442E6AFC 192CEE5E 320905E0 EAB3CD93
       FB595CA2 6C154F39 427A0416 A9F36337 354CF1E6 E5AED73D DF80C710
       026D4955 0AC8CE04

xR21  =  0006869E A28E4CEE 05DCE8EB 08ACD597 75D03ADA 0DC80B94 C85156C2
       12C23C72 CB2AB2D2 D90D4637 5AA6D666 58E4F8F8 219431D3 006FDED7
       993F5164 9CD029498

xQ30  =  0039014A 74763076 675D24CF 3FA28318 DAC75BCB 04E54ADD C6496493
       F72EBBD7 A7DC6A3B BCD188DA D5B6EC9D 6BB4ABDD 05DB38C5 FBE52D98
       5DCF744 12C2D453

xQ31  =  00000000

yQ31  =  00000000

yQ31  =  00255120 12C90A68 69C4B29B 9A757A03 006BC7DF 0BF7A252 6A071393
       9FA48018 AE3E249B D63699BE B3B8DEA2 15B7AE1B 5A30FEE7 1B64C5F1
       B0BF051A 0AC8CE04

xP30  =  0032D03F D1E99ED0 CB05C070 7AF74617 CBEA5AC6 B75905B4 B54B1B0C
       2D736978 40155E7B 1005F90B 2B5D0279 7A8666A5 D258C76A 3C9EF745
       CECE11E9 A178BADF

xP31  =  00000000

yP30  =  0022D810F 828E3DC0 24D1BBBC 7D6FA6E3 02CC5D45 8571763B 7CDD0E4D
       BC9FA116 3F0C1F8F 4AE32A57 F89DF8D2 586D2A06 E9FA3044 2B94A725
       266358C4 5236ADF3

yP31  =  00000000

xR30  =  0000C146 5FD048FF B8BF2158 ED57F0CF FF0C4D5A 4397C754 2D722567
1.6.3 SIKEp610

\[ p = 00000002 \text{7BF6A768 819010C2 51E7D88C B255B2FA 10C4252A 9AE7BF45 048FF9AB B1784DE8 AA5AB02E 6E01FFF} \]
\[ \text{FFFFFFFF FFFFFFFFF FFFFFFFFF FFFFFFFFF FFFFFFFFF FFFFFFFFF FFFFFFFFF FFFFFFFFF FFFFFFFFF FFFFFFFFF FFFFFFFFF FFFFFFFFF FFFFFFFFF FFFFFFFFF FFFFFFFFF FFFFFFFFF FFFFFFFFF FFFFFFFFF FFFFFFFFF FFFFFFFFF FFFFFFFFF} \]

\[ e2 = 000000131 \]

\[ e3 = 000000C0 \]

\[ xQ20 = 25DA39EC 90CDFB9B C0F772CD A52CB8B5 A9F478D7 AF8DBBA0 AEB3E524 32822DD8 8C38F4E3 AEC0746E 56149F1F E89707C7 7F8BA413 45686297 24F4A8E5 4B06BFE5 C5E66E08 67EC38B2 83798B8A \]

\[ xQ21 = 00000002 250E1959 256AE502 428338CB 47153995 51AEC7B8D 8935B2DC 73FCDCFB DB1A0118 A2D3EF03 489BA6F6 37B1C7FE E7E5F313 40A1A537 B76B5B73 6B4CDD28 4918918E 8C986FC0 2741FB8C 98F0A0ED \]

\[ yQ20 = A4FD5539 025C0611 E4B1CEC3 C36F0D75 90C035D3 A25AD930 22849CCE B3F67E4B 1DBE9884 64290D8B 878B8D5E 69ED3B0C 5DBCA24 8DC90D74 CF762012 CFE2D725 CFD92057 F2DFB8B0 4C7B12CC \]

\[ yQ21 = 00000002 01C807BD 738624E2 2B87554A 2E053A46 A9573B8A 63D4A9D3 09533E30 B27BF7DD 8137F5CE 0F79C263 D9D05054 1D69817A 839085A7 6395F879 315F6999 E3441FC8 FB3936DE E1BEF5B4 E0E25096 \]

\[ xP20 = 00000001 B368BC60 19B46C0D 02129209 B3E65B98 BC64A92B C4DB2F9F 3AC96B97 A1B9C124 DF549B52 8F18BECC B1666D27 D4753043 5E842212 72F3A97F B80527D8 F8A359F8 F1598D36 5744CA30 70A5F26C \]

\[ xP21 = 00000001 459685DC A7112D1F 6030D8C0 98F2CCB9 41617B6A D913E652 3416CCBD 8ED9C784 1D97DF83 092B59B3 2AF00D62 E08DAD8F A743CBBCC CC1782BE 0186A343 2D3C97C3 7CA16873 BDE01F0 637C1AA2 \]

\[ yP20 = 00000001 CD75CF51 2FFA9DF8 78E4950 01A57ABC 07FC7CE9 BB48B8B5 2DDD727 2D8A4FD1 1DD258ED 3F844C86 2CF48803 B9AC2668 C7CB79C3 96128763 B578080C 30D14CA7 EB709F98 E3E682A3 91FB35A7 \]
\[ y_{P21} = 00000002 \ 001062A6 \ 289E4082 \ CED88402 \ 9207A1AC \ DEC525D7 \ BC165A6C \ FF88B469 \ A8588950 \ A416DB99 \ 24D2D673 \ E3D6C32D \ 232F6E6A \ DA62B376 \ 08F652C0 \ B8628827 \ B304BF13 \ 65D82113 \ 46207B24 \ EFF09458 \]

\[ x_{R20} = 00000001 \ B36A006D \ 05F9E370 \ D5078CCA \ 54A16845 \ B2BFF737 \ C8653687 \ 97C0DBBE \ 9F5A62A9 \ B9C79ADF \ 11932A9F \ A4806210 \ E25C92DB \ 019CC146 \ 706DFBC7 \ FA2638EC \ C4343C1E \ 390426FA \ A7F2F07F \ DA163FB5 \]

\[ x_{R21} = 00000001 \ 83C9ABF2 \ 297CA696 \ 99357F58 \ FED92553 \ 436BBEBA \ 2C3600D8 \ 9522E700 \ 9D19EA5D \ B9C79ADF \ 11932A9F \ 3923ED93 \ 6F7FEFEB \ 0282876B \ 2F2FC263 \ 3CA548C3 \ AB0C45CC \ 991417A5 \ 6F7FEFEB \]

\[ x_{Q30} = 00000001 \ 4E647CB1 \ 9B7EAAAC \ 640A9C26 \ B9C79ADF \ DEDA8FC9 \ 399F4F8C \ E620D2B2 \ 200480F4 \ 338755AE \ 16D0E090 \ F15EA188 \ 2166836A \ 478C6E16 \ 1C938E4E \ B8C2DD77 \ 9B45FFDD \ 17DCDF15 \ 8AF48DE1 \ 26B3A047 \]

\[ x_{Q31} = 00000000 \]

\[ y_{Q30} = 00000000 \]

\[ y_{Q31} = E674067F \ 5EA6DE85 \ 545C0A99 \ E9E71E64 \ FABFDC28 \ 1D1E540F \ EDA47A56 \ ED3ACDDE \ E1841083 \ FABC7954 \ B467C71A \ C6349B04 \ 974A7F9B \ 688C5F73 \ 5632EBC3 \ 94146B0A \ C8088323 \ 4EDF795B \]

\[ x_{P30} = 00000001 \ 587822E6 \ 47707ED4 \ 313D3BEE \ 6A811A694 \ FB201561 \ 111838A0 \ 816BFB5D \ EC6252D3 \ 772DE48A \ 2678C04 \ EEB26CA4 \ A571C67C \ E4DC4C62 \ 0282876B \ 2F2FC263 \ 3CA548C3 \ AB0C45CC \ 991417A5 \ 6F7FEFEB \]

\[ x_{P31} = 00000000 \]

\[ y_{P30} = 14F29511 \ 4B69D476 \ 9AC06DD0 \ 7F051AD1 \ 1148CB7F \ A6B6DEE1 \ 9F840969 \ AFD56FD1 \ F728907D \ 3320A030 \ 9462A944 \ 4D24FEE7 \ 4666DB24 \ 70080951 \ B31C2AC5 \ 9704ABC7 \ 670C3C3A \ 992C3C16 \ 29791F30 \]

\[ y_{P31} = 00000000 \]

\[ x_{R30} = 00000001 \ DB73BC2D \ E666D24E \ 59AF5E23 \ B79251BA \ 0D189629 \ EF87E56C \ 38778A44 \ 8FACE312 \ D08EDFB8 \ 76C3FD45 \ ECF3746D \ 96E2CADB \ BA08B1A2 \ 06C47DDD \ 93137659 \ E34C90E2 \ E42E10F3 \ 0F6E5F52 \ DED74222 \]

\[ x_{R31} = 00000001 \ B2C30180 \ DAF5D918 \ 71555CE8 \ EFEC76A4 \ D521F877 \ B7543112 \ 28C7180A \ 3E2318B4 \ E7A00341 \ FF9F34E \ 35BF7A10 \ 53CA76FD \ 77C0AFAE \ 38E20918 \ 62AB4F1D \ D4C8D9C8 \ 3DE37ACB \ A6646EDB \ 4C238B48 \]

### 1.6.4 SIKEp751

\[
p = 00006FE5 \ D541F71C \ 0E12909F \ 97BADC66 \ 8562B504 \ 5CB25748 \ 084E9867\]
Chapter 2

Detailed performance analysis

The submission package includes:

1. A generic reference implementation written exclusively in portable C with simple algorithms to compute isogeny and field operations, using GMP for multi-precision arithmetic,

2. An optimized implementation written exclusively in portable C that includes efficient algorithms to compute isogeny and field operations,

3. An additional, optimized implementation for x64 platforms that exploits x64 assembly,

4. An additional, optimized implementation for x64 platforms that exploits x64 assembly and public key compression (§1.5),

5. An additional, optimized implementation for ARM64 platforms that exploits ARMv8 assembly,

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


All implementations except implementations number 1 and 7 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, optimized implementation, and x64 assembly-optimized implementation (numbers 1 to 3) support all four parameter sets, namely SIKEp434, SIKEp503, SIKEp610 and SIKEp751. The version with public key compression (number 4) uses the same public parameters as the uncompressed version, but requires different KAT files, because the output formats are different. Therefore, we formally assign to this implementation a different collection of parameter sets, denoted by SIKEpXXX_compressed for XXX ∈ {434, 503, 610, 751}.

The ARM64 assembly-optimized implementation (number 5) supports the parameter sets SIKEp503, SIKEp751, SIKEp503_compressed, and SIKEp751_compressed.
The VHDL implementation (number 6) supports the SIKEp751 parameter set. Because of time constraints, this implementation has not yet been updated for the second round. In particular, the change of starting curve from $A = 0$ to $A = 6$ (cf. Appendix E) is not reflected in this implementation, and the KAT files for the 2nd round do not pass on this implementation. Updated implementations will be posted on the SIKE web page (https://sike.org/) when available. In the meantime, the KAT files from the 1st round submission may be used.

The Weierstrass implementation (number 7) supports the same prime sizes as the main implementation (namely, 434, 503, 610, and 751 bits). However, it is not directly compatible with any of the parameter sets, because its main purpose is to illustrate isogeny computations using textbook formulas over elliptic curves in short Weierstrass form, whereas the parameter sets are defined using Montgomery curves. Converting between curves in short Weierstrass form and the curves of Montgomery form used in the parameter sets would defeat the purpose of having a simple textbook implementation.

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

### 2.1 Reference implementation

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

1. Multiprecision arithmetic using GMP.

2. Finite field arithmetic over $\mathbb{F}_p$ is implemented with a generic API, hiding the underlying GMP functions. The same API is used for any prime. The function headers are available in fp.h.

3. Quadratic extension field arithmetic over $\mathbb{F}_{p^2}$ is built on top of the $\mathbb{F}_p$ API. The function headers are available in fp2.h.

4. Montgomery elliptic curve arithmetic uses the $\mathbb{F}_{p^2}$ code and implements point addition, point doubling, point tripling, 2/3/4-isogeny generation and evaluation, scalar multiplication and $j$-invariant computation. For simplicity reasons, the scalar multiplication algorithm is not safe against side-channel attacks, but could be protected with well known countermeasures against side-channel attacks for ECC. The headers for Montgomery curve arithmetic and 2/3/4-isogeny generation are available in montgomery.h and isogeny.h, respectively.

5. The SIDH key agreement scheme is implemented with the key-generation algorithm (corresponding to $\text{isogeny}_\ell$) and the shared secret algorithm (corresponding to $\text{isoex}_\ell$). The function headers are available in sidh.h.
6. The SIKE key encapsulation protocol is built on top of SIDH and implements PKE encryption, PKE decryption, KEM encapsulation and KEM decapsulation. The function headers are available in sike.h and api_generic.h.

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

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

2.2 Optimized and x64 assembly implementations

The optimized implementation, which is written in portable C only, uses efficient algorithms for isogeny and elliptic curve computations using projective coordinates on Montgomery curves, the Montgomery ladder, and efficient tree traversal strategies for fast isogeny computation. Specifically, this implementation makes use of Algorithms 3–24 listed in Appendix A. The optimal tree traversal strategies used in Algorithms 19 and 20 are given in Appendix 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 radix-r Montgomery reduction and exploiting the MULX and ADX instructions (when available), as done in [14].

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

2.2.1 Performance on x64 Intel

To evaluate the performance of the optimized and x64-assembly implementations, we ran our benchmarking suite on a machine powered by a 3.4GHz Intel Core i7-6700 (Skylake) processor, running Ubuntu
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.

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 7.2.0.

Table 2.1 details the performance of the reference, optimized, x64-assembly, and compressed implementations of SIKE. As we can see, the constant-time optimized implementation is about 14-18 times faster than the variable-time reference implementation, thanks to the use of more efficient elliptic curve arithmetic and optimal strategies for isogeny computation. The use of assembly optimizations further improves performance greatly. Compilers still do a poor job of generating efficient code for multiprecision operations, especially multiprecision multiplication and reduction. Thus, our best performance for SIKEp434, SIKEp503, SIKEp610 and SIKEp751 (i.e., 6.3 msec., 9.0 msec., 16.8 msec. and 25.8 msec., respectively, obtained by adding the times for encapsulation and decapsulation) is achieved with the use of hand-tuned x64 assembly.
<table>
<thead>
<tr>
<th>Scheme</th>
<th>secret key</th>
<th>public key</th>
<th>ciphertext</th>
<th>shared secret</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIKEp434</td>
<td>(44+330) 374</td>
<td>330</td>
<td>346</td>
<td>16</td>
</tr>
<tr>
<td>SIKEp503</td>
<td>(56+378) 434</td>
<td>378</td>
<td>402</td>
<td>24</td>
</tr>
<tr>
<td>SIKEp610</td>
<td>(62+462) 524</td>
<td>462</td>
<td>486</td>
<td>24</td>
</tr>
<tr>
<td>SIKEp751</td>
<td>(80+564) 644</td>
<td>564</td>
<td>596</td>
<td>32</td>
</tr>
<tr>
<td>SIKEp434_compressed</td>
<td>(43+196) 239</td>
<td>196</td>
<td>209</td>
<td>16</td>
</tr>
<tr>
<td>SIKEp503_compressed</td>
<td>(56+224) 280</td>
<td>224</td>
<td>248</td>
<td>24</td>
</tr>
<tr>
<td>SIKEp610_compressed</td>
<td>(62+273) 336</td>
<td>273</td>
<td>297</td>
<td>24</td>
</tr>
<tr>
<td>SIKEp751_compressed</td>
<td>(79+334) 413</td>
<td>331</td>
<td>363</td>
<td>32</td>
</tr>
</tbody>
</table>

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/](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](https://github.com/lnishan/massif-cherrypick)), which is an extension that outputs memory usage per function:

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

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

It can also be seen that the static library sizes can grow relatively high (see option compiled for speed). However, it is possible to reduce the library sizes significantly, to around 60KB, at little performance cost: compiling the additional implementations for size more than halves the library sizes and reduces speed by less than 5%. It should be noted that the reference implementation is a single library for all SIKE instantiations, and that GMP attributes to its size because of static linking. The stack memory usage is relatively low due to GMP’s internal memory management.
<table>
<thead>
<tr>
<th>Scheme</th>
<th>KeyGen (stack)</th>
<th>Encaps (stack)</th>
<th>Decaps (stack)</th>
<th>static library</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>speed (-O3)</td>
<td>size (-Os)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reference Implementation</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SIKEp434</td>
<td>448</td>
<td>448</td>
<td>448</td>
<td>89,148</td>
</tr>
<tr>
<td>SIKEp503</td>
<td>512</td>
<td>512</td>
<td>512</td>
<td>89,148</td>
</tr>
<tr>
<td>SIKEp610</td>
<td>640</td>
<td>640</td>
<td>640</td>
<td>89,148</td>
</tr>
<tr>
<td>SIKEp751</td>
<td>768</td>
<td>768</td>
<td>768</td>
<td>89,148</td>
</tr>
<tr>
<td>Optimized Implementation</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SIKEp434</td>
<td>8,040</td>
<td>8,360</td>
<td>8,744</td>
<td>105,474</td>
</tr>
<tr>
<td>SIKEp503</td>
<td>8,072</td>
<td>8,456</td>
<td>8,904</td>
<td>120,202</td>
</tr>
<tr>
<td>SIKEp610</td>
<td>12,008</td>
<td>12,408</td>
<td>12,936</td>
<td>163,312</td>
</tr>
<tr>
<td>SIKEp751</td>
<td>13,912</td>
<td>14,040</td>
<td>14,696</td>
<td>164,810</td>
</tr>
<tr>
<td>Additional implementation using x64 assembly</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SIKEp434</td>
<td>8,136</td>
<td>8,456</td>
<td>8,840</td>
<td>116,192</td>
</tr>
<tr>
<td>SIKEp503</td>
<td>8,152</td>
<td>8,536</td>
<td>8,984</td>
<td>131,872</td>
</tr>
<tr>
<td>SIKEp610</td>
<td>13,536</td>
<td>12,512</td>
<td>12,112</td>
<td>155,864</td>
</tr>
<tr>
<td>SIKEp751</td>
<td>14,064</td>
<td>14,192</td>
<td>14,960</td>
<td>188,800</td>
</tr>
<tr>
<td>Compressed SIKE implementation using x64 assembly</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SIKEp434</td>
<td>16,920</td>
<td>15,640</td>
<td>17,000</td>
<td>1,875,070</td>
</tr>
<tr>
<td>SIKEp503</td>
<td>18,872</td>
<td>17,560</td>
<td>19,128</td>
<td>5,508,190</td>
</tr>
<tr>
<td>SIKEp610</td>
<td>23,824</td>
<td>22,048</td>
<td>24,144</td>
<td>4,342,086</td>
</tr>
<tr>
<td>SIKEp751</td>
<td>28,024</td>
<td>27,936</td>
<td>28,320</td>
<td>5,114,334</td>
</tr>
</tbody>
</table>

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

2.3 Compressed SIKE implementation

We provide an implementation of SIKE that supports public-key compression (§1.5). The compressed SIKE implementation uses the same codebase as the optimized x64 implementation, but additionally performs public key compression along with key encapsulation. Compression is performed both for the static public key and for the component of the ciphertext corresponding to the ephemeral public key.

In SIDH (§4.3.1), the public key pk₂ encodes $3^e$-torsion points, and the public key pk₃ encodes $2^e$-torsion points. Key compression is faster on $2^e$-torsion points than on $3^e$-torsion points. If we assume that one public key may be used to encrypt multiple ciphertexts over its lifespan, then we must designate pk₂ for static public keys and pk₃ for ephemeral public keys in order to achieve optimal performance in the KEM in the key compression setting. Accordingly, the **compressed SIKE implementation swaps the roles** of the subscripts ₂ and ₃ in Algorithm 2.
Tables 2.1 and 2.2 respectively indicate the performance penalty and key size improvements offered by compressed SIKE. The performance penalties range from 139% to 161% for key generation, 66% to 90% for encapsulation, and 59% to 68% for decapsulation, depending on parameter size. The size of the public key is reduced by 41%, and the size of the ciphertext by 39%.

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

2.4 64-bit ARM assembly implementation

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

To evaluate the performance of this implementation, we ran our benchmarking suite on a Google Pixel 3 device, powered by a 2.45GHz 64-bit ARM Cortex-A75 processor, running Android version 9. For compilation we used clang version 8.0.0 with the command `clang -O3`.

Table 2.4 compares the performance of the additional ARMv8-assembly implementation of SIKE to the (portable) optimized implementation of SIKE. As we can see, the specialized implementation is roughly 4 to 5 times faster than the generic optimized implementation, thanks to the use of assembly routines. Our best performance for SIKEp503 and SIKEp751 on the targeted platform is 37.4 ms and 135.3 ms, respectively, corresponding to the total time that it takes to compute the encapsulation and decapsulation operations.

2.5 VHDL hardware implementation

The optimized VHDL hardware implementation accelerates SIKE operations by using Algorithms 3–24 listed in Appendix A. Thus, this hardware implementation uses projective coordinates on Montgomery curves, an efficient double-point multiplication ladder, and an efficient tree traversal algorithm for isogeny computation. A separate tree traversal strategy was computed with Algorithm 46 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 ($\text{isogen}_2$, $\text{isogen}_3$, $\text{isoex}_2$, and $\text{isoex}_3$) for the public parameters listed in SIKEp751.
### Table 2.4: Performance (in thousands of cycles) of SIKE on a 2.45GHz 64-bit ARM Cortex-A75 processor. Results are measured in ns and scaled to cycles using the nominal processor frequency. Cycle counts are rounded to the nearest $10^3$ cycles.

<table>
<thead>
<tr>
<th>Scheme</th>
<th>KeyGen</th>
<th>Encaps</th>
<th>Decaps</th>
<th>total (Encaps + Decaps)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Optimized implementation (portable)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SIKEp503</td>
<td>121,048</td>
<td>199,093</td>
<td>211,271</td>
<td>410,364</td>
</tr>
<tr>
<td>SIKEp751</td>
<td>415,085</td>
<td>677,698</td>
<td>723,416</td>
<td>1,401,114</td>
</tr>
<tr>
<td>SIKEp503_compressed</td>
<td>301,663</td>
<td>365,259</td>
<td>340,315</td>
<td>997,237</td>
</tr>
<tr>
<td>SIKEp751_compressed</td>
<td>992,640</td>
<td>1,241,449</td>
<td>1,158,156</td>
<td>2,399,605</td>
</tr>
<tr>
<td><strong>Additional implementation using ARMv8 assembly</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SIKEp503</td>
<td>26,928</td>
<td>44,388</td>
<td>47,377</td>
<td>91,764</td>
</tr>
<tr>
<td>SIKEp751</td>
<td>90,012</td>
<td>147,589</td>
<td>159,517</td>
<td>307,118</td>
</tr>
<tr>
<td>SIKEp503_compressed</td>
<td>67,424</td>
<td>81,424</td>
<td>75,631</td>
<td>224,506</td>
</tr>
<tr>
<td>SIKEp751_compressed</td>
<td>217,710</td>
<td>271,493</td>
<td>252,955</td>
<td>542,165</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>#Multipliers</th>
<th>Scheme</th>
<th>Cycle counts ($cc \times 10^3$)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>KeyGen</td>
<td>Encaps</td>
</tr>
<tr>
<td>2</td>
<td>SIKEp751</td>
<td>3,920</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>2,464</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>1,941</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>1,798</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>1,698</td>
</tr>
</tbody>
</table>

2.5.1 Performance

The SIKE hardware accelerator can perform KEM functions for the public parameters listed in SIKEp751. There is a high 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.

2.5.2 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
Table 2.6: FPGA implementation results of SIKE accelerator over SIKEp751 on a Xilinx Virtex-7 FPGA.

<table>
<thead>
<tr>
<th># Mults</th>
<th>Area (kGE)</th>
<th>Freq (MHz)</th>
<th>Time (msec)</th>
</tr>
</thead>
<tbody>
<tr>
<td># FFs</td>
<td># LUTs</td>
<td># Slices</td>
<td># DSPs</td>
</tr>
<tr>
<td>8</td>
<td>51,914</td>
<td>44,822</td>
<td>16,756</td>
</tr>
<tr>
<td>198</td>
<td>KeyGen</td>
<td>Encaps</td>
<td>Decaps</td>
</tr>
<tr>
<td>9.08</td>
<td>16.27</td>
<td>17.08</td>
<td>33.35</td>
</tr>
<tr>
<td></td>
<td>(Encaps + Decaps)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.

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.

2.5.3 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 $\mu m^2$, so this 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 SIKEp434, SIKEp503, SIKEp610, SIKEp751, SIKEp434_compressed, SIKEp503_compressed, SIKEp610_compressed and SIKEp751_compressed. The KAT files can be found in the media folder of the submission: \KAT\PQKemKAT_374.rsp, \KAT\PQKemKAT_434.rsp, \KAT\PQKemKAT_524.rsp and \KAT\PQKemKAT_644.rsp, \KAT\PQKemKAT_239.rsp, \KAT\PQKemKAT_280.rsp, \KAT\PQKemKAT_336.rsp and \KAT\PQKemKAT_413.rsp for SIKEp434, SIKEp503, SIKEp610, SIKEp751, SIKEp434_compressed, SIKEp503_compressed, SIKEp610_compressed and SIKEp751_compressed, respectively.

In addition, we provide a test suite that can be used to verify the KAT values against any of the implementations. Instructions to compile and run the KAT test suite can be found in the README file in the top-level directory of the media folder (see Section 2, “Quick Instructions”).
Chapter 4

Expected security strength

4.1 Security

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

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

Over $\mathbb{F}_{p^2}$, there is a unique isogeny class of supersingular elliptic curves (up to twist), and it has size roughly $p/12$. Thus, the algorithm of Delfs and Galbraith would find an isogeny between the starting curve $E_0$ and a public curve $E'$ in $O(\sqrt{p})$ time. 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 $\ell^e$ between $E_0$ and $E'$, an attacker builds a tree of all curves isogenous to $E_0$ via isogenies of degree $\ell^{e/2}$, and a similar tree of all curves isogenous to $E'$ of degree $\ell^{e/2}$. Since we suppose that an isogeny of degree $\ell^e$ 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 $O(\sqrt[4]{\ell^e})$ elementary steps, or $O(\sqrt[4]{p})$, as announced.

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

\footnote{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 $\text{isogeny}_\ell$. However, a complete reduction of the security of SIKE to the isogeny walk problem is presented in [17].}
and looking for hits for \( g(b) \) where \( b \in B \). However, one can do better with a quantum computer using Tani’s claw-finding algorithm \([37]\), which only uses \( O(\sqrt{|A||B|}) \) invocations to quantum oracles for \( f \) and \( g \). These complexities are optimal for a black-box claw attack \([43]\). For given supersingular curves \( E, E' \) we could, for example, let \( A \) resp. \( B \) be the set of points of order exactly \( \ell^{e/2} \) on \( E \) resp. \( E' \), and \( C \) the set of supersingular \( j \)-invariants. The functions \( f \) and \( g \) compute \( \ell^{e/2} \)-isogenies which have kernels generated by their input points and return the \( j \)-invariant of the final curve. Classically this is exactly the \( O(\sqrt{\ell^{e}}) \) attack described above, and applying Tani’s algorithm to SIKE gives an attack requiring \( O(\sqrt{\ell^{e}}) = O(\sqrt{p}) \) invocations of a quantum isogeny computation oracle.

While the generic algorithms described above (and their asymptotic complexities) were used for the security analysis in the initial SIKE proposal, a series of subsequent papers beginning with \([1]\) have since argued that the parallel collision finding algorithm of van Oorschot and Wiener \([40]\) is the best classical claw-finding attack on SIKE, and \([21]\) even argues that the above query-optimal instantiation of Tani’s algorithm is outperformed by the classical van Oorschot and Wiener algorithm. We further discuss the concrete security of SIKE in Chapter 5.

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

### 4.2 Other attacks

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

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

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

- **Galbraith, Petit, Shani and Ti** \([17]\) show how a secret \( j \)-invariant can be recovered from some partial knowledge of it.

- **Ti** \([39]\) explains how a random perturbation to the inputs of \texttt{isogen}_\ell yields to a key recovery with very high probability in most protocols derived from SIDH. It is not clear, however, how the technique can be used against the public key format specified in 1.2.9.

- **Gélin and Wesolowski** \([18]\) present a loop-abort fault attack that potentially leads to an efficient key recovery against the “simple” version of \texttt{isogen}_\ell given in Algorithms 17 and 18. However their attack is efficiently countered by the additional checks in Algorithms 19 and 20.
A recent preprint by Petit [30] 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 [31] showed that the isogeny walk problem used to construct the Charles-Goren-Lauter hash function [5] is equivalent to the problem of computing endomorphism rings of supersingular elliptic curves, which is possibly (but not yet shown to be) harder than the SIDH problem. However, it does not appear to be possible to extend the Charles-Goren-Lauter hash construction to yield key exchange.

4.3 Security proofs

The PKE scheme in §1.3.9 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 [11, Problem 5.3].

**Problem 1.** Let $sk_2 \in K_2$ and $sk_3 \in K_3$. Let $pk_2 = \text{isogen}_2(sk_2)$ and $pk_3 = \text{isogen}_3(sk_3)$. Given $(E_0, pk_2, pk_3)$, compute $j = \text{isoex}_2(pk_3, sk_2) = \text{isoex}_3(pk_2, sk_3)$.

4.3.2 IND-CPA PKE

Define the IND-CPA security of a public-key encryption scheme in the standard way (e.g. see [3, 23]). 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 [23, Thm 5], [16, Thm 20.4.11] or [22, Thm 11.21].

**Remark 3.** There exist alternative proofs of security in the standard model, reducing the security to a decisional variant of SIDH [11, Problem 5.4] instead of SIDH (see [11, Thm 6.2], based on [34, Thm 2] and [33, §3.4]).
4.3.3 IND-CCA KEM

Theorem 1 ([19]). For any IND-CCA adversary B against 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 PKE with

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

Proof. This is the bound obtained by combining the results from Theorem 3.2 and Theorem 3.4 from [19], setting $\text{KEM} = U^L[T[\text{PKE}, G], H]$.

Note that Decaps slightly deviates from the definition in [19]. Instead of full “re-encryption” ($c'_0, c'_1 \leftarrow \text{Enc}(pk_3, m'; G(m' \parallel pk_3))$, we only re-compute $c'_0$. However, full computation would yield

$$c'_1 = m' \oplus F(\text{isoex}_2(pk_3, G(m' \parallel pk_3))) = m' \oplus F(\text{isoex}_3(c'_0, sk_3)),$$

while $c_1 = m' \oplus F(\text{isoex}_3(c_0, sk_3))$. Hence it is clear that $c'_0 = c_0$ implies $c'_1 = c_1$, making the computation of $c'_1$ redundant. \qed
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 ∈ {434, 610, 751}, requires respective resources comparable to those required for key search on a k-bit (ideal) block cipher $B$, where $k \in \{128, 192, 256\}$. In addition, our goal is to ensure that the computational cost of breaking SIKEp503 requires resources comparable to those required for collision search on a 256-bit (ideal) hash function.

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

5.1 Classical security

Following the submission of SIKE to the NIST call in November of 2017, a series of papers have emerged that have scrutinized the application of generic meet-in-the-middle attacks described in §4.1. The work of Adj, Cervantes-Vázquez, Chi-Domínguez, Menezes and Rodríguez-Henríquez [1] was the first paper to argue that the parallel collision-finding algorithm of van Oorschot and Wiener (vOW) [40] is actually the attack that should be used to evaluate the security of SIKE. The reason is that the $O(p^{1/4})$ memory that is required to mount the generic meet-in-the-middle attack — that which runs in $O(p^{1/4})$ time — is far beyond feasible for SIKE parameters in the ranges of interest. Since the best known generic attack against ideal block ciphers (e.g., AES) use only a moderate amount of memory, in deriving SIKE parameters for which the computational resources are comparable to AES instantiations, the most appropriate model is to fix an upper bound on the classical memory available, and to evaluate the runtime of the best known attacks subject to this limit.

Under the assumption that the memory available permits the storage of $2^{80}$ units, Adj et al. [1] conclude that SIKEp434 and SIKEp610 meet the respective security requirements of NIST’s categories 2 and 4. A subsequent paper by Jaques and Schanck [21] — which is largely geared towards the analysis of quantum algorithms, but also considers vOW — further endorses the classical complexity claims of Adj et al with respect to these two curves and the NIST requirements they satisfy. And, in addition to a further endorsement of these two curves, a recent paper by Costello, Longa, Naehrig, Renes and Virdia [9] argues that SIKEp751, which was initially proposed to meet level 3, actually meets NIST’s category 5 requirements.
We refer to these three papers (and the original vOW paper) for the in-depth analyses, but we summarize their application to three SIKE parameterizations in Table 5.1, noting that they use slightly different memory assumptions and/or cost metrics in order to estimate the complexity of vOW against SIKE parameters. Adj et al. assume that the memory permits the storage of $2^{80}$ units, and present their results in “total time”, where the unit of time is actually the time complexity of a degree $\ell^{1/2}$-isogeny; thus, although their times fall slightly below NIST’s required gate counts, the corresponding conversion to gate counts would see these parameters comfortably exceed NIST’s requirements. The classical analysis of Jaques and Schanck uses the PRAM model and estimates the number of classical gates under the assumption that the memory is $2^{96}$ bits. Their model does incorporate the cost of the isogeny computations, but is still rather conservative. Finally, the vOW analysis of Costello et al. estimates the total number of x64 instructions required to mount the vOW attack, and argues that this is also a conservative lower bound on the true classical gate count. In particular, for the SIKEp751 parameterization, they conclude that the true gate count corresponding to their estimated $2^{262}$ x64 instructions would exceed NIST’s $2^{272}$ gate count requirement.

<table>
<thead>
<tr>
<th>Target level</th>
<th>Classical gate requirement</th>
<th>Total time</th>
<th>Classical security estimates</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Classical security estimates</td>
<td>[38]</td>
<td>[1]</td>
</tr>
<tr>
<td></td>
<td>Total time</td>
<td>Gates</td>
<td>x64 instructions</td>
</tr>
<tr>
<td></td>
<td>memory $2^{80}$ units</td>
<td>memory $2^{96}$ bits</td>
<td>memory $2^{80}$ units</td>
</tr>
<tr>
<td>SIKEp434</td>
<td>1</td>
<td>143</td>
<td>128</td>
</tr>
<tr>
<td>SIKEp503</td>
<td>2</td>
<td>146</td>
<td>152</td>
</tr>
<tr>
<td>SIKEp610</td>
<td>3</td>
<td>207</td>
<td>189</td>
</tr>
<tr>
<td>SIKEp751</td>
<td>5</td>
<td>272</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 5.1: Classical security estimates of the three SIKE parameterizations according to Adj et al. [1], Jaques and Schanck [21], and Costello et al. [9]. Gate requirements and classical security estimates are all expressed as their base-2 logarithms. The values marked with (*) are not found in the actual papers. In the case of [9], we obtained the numbers for SIKEp503 using their scripts, where (for the half-sized isogenies used in vOW) the optimal strategy for the 2-torsion resulted in 362 doublings and 189 4-isogenies, and the optimal strategy for the 3-torsion yielded 229 triplings and 275 3-isogenies. In the former scenario, a vOW isogeny required over $2^{22}$ x64 instructions, and in the latter, over $2^{23}$ x64 instructions. In the case of [21], the RAM operations for SIKEp503 and SIKEp751 were taken from the width-restricted table in §5.2.

5.2 Quantum security

The initial SIKE proposal used the asymptotic complexity of Tani’s quantum claw-finding algorithm [37] together with crude lower bounds for the number of quantum gates used in an $\mathbb{F}_p$-multiplication and the number of such multiplications in a typical isogeny computation to provide conservative resource estimates for the cost of quantum cryptanalysis of SIKE. The recent paper by Jaques and Schanck [21] conducts a much more detailed analysis of the best known quantum algorithms to solve the computational supersingular isogeny problem. Jaques and Schanck propose a model of quantum computation that allows a direct comparison between quantum and classical algorithms. They treat qubit registers as memory.
peripherals for classical control processors, which run quantum circuits through RAM operations on qubit memory peripherals. This allows them to use the number of RAM operations as the algorithm’s cost function, derived either from the quantum gate count in a quantum circuit or the product of its depth and width. The crucial difference to previous cost estimates lies in considering the complexity of implementing and querying quantum memory.

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

Figure 4 in [21] provides concrete cost estimates for solving the computational supersingular isogeny problem in different scenarios for the parameters SIKEp434 and SIKEp610. The relevant constraint for matching the NIST security categories is imposing a depth restriction on quantum circuits between $2^{64}$ and $2^{96}$ (corresponding to the MAXDEPTH parameter in the NIST call for proposals [38]). Allowing depth $2^{96}$, Jaques and Schanck conclude that no known quantum algorithm can break SIKE in their model of computation with less than $2^{143}$ classical gates and $2^{207}$ classical gates for SIKEp434 and SIKEp610, respectively. Therefore, these two parameter sets are suitable for NIST categories 1 and 3. Running the scripts accompanying [21] to produce the same tables for SIKEp503 and SIKEp751 suggest that no quantum algorithm can break those with less than $2^{146}$ and $2^{272}$ classical gates, respectively, which confirms that these parameter sets are suitable for NIST categories 2 and 5.

Tables 5.2, 5.3, 5.4 and 5.5 were obtained with the methodology from [21], including all SIKE parameter sets. They show the base-2 logarithm of the classical gate count costs ($G$) and the corresponding depth ($D$) and width ($W$) for a specific parameterisation of a given algorithm. The algorithms considered are a direct application of Grover search, Tani’s claw-finding algorithm and the classical van Oorschot-Wiener collision search algorithm (vOW). Table 5.2 shows results when the depth is restricted to either $2^{64}$ or $2^{96}$. This corresponds to the NIST model for quantum computation. In this case, the classical vOW algorithm does not show the optimal gate count, but instead minimizes the memory (width) with the given depth restriction. Table 5.3 instead restricts the width of the algorithm to either $2^{64}$ or $2^{96}$. Tables 5.4 and 5.5 show the gate cost optimal and the depth-width cost optimal settings. It should be noted that these parameterizations either violate a reasonable depth or width constraint.

5.3 Side-channel attacks

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

---

1Sam Jaques kindly produced these tables for us with the scripts used to generate the tables in [21].
### Table 5.2: Cost estimates for algorithms to solve the computational supersingular isogeny problem on SIKE parameter sets with depth constraints. The first three lines restrict to maximal depth close to $2^{64}$, the last three to $2^{96}$.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>SIKEp434</th>
<th>SIKEp503</th>
<th>SIKEp610</th>
<th>SIKEp751</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$G$</td>
<td>$D$</td>
<td>$W$</td>
<td>$G$</td>
</tr>
<tr>
<td>Grover</td>
<td>190</td>
<td>64</td>
<td>127</td>
<td>226</td>
</tr>
<tr>
<td>Tani</td>
<td>175</td>
<td>63</td>
<td>126</td>
<td>210</td>
</tr>
<tr>
<td>vOW</td>
<td>145</td>
<td>64</td>
<td>91</td>
<td>162</td>
</tr>
<tr>
<td>Grover</td>
<td>158</td>
<td>96</td>
<td>63</td>
<td>194</td>
</tr>
<tr>
<td>Tani</td>
<td>143</td>
<td>95</td>
<td>62</td>
<td>178</td>
</tr>
<tr>
<td>vOW</td>
<td>155</td>
<td>95</td>
<td>70</td>
<td>173</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>SIKEp434</th>
<th>SIKEp503</th>
<th>SIKEp610</th>
<th>SIKEp751</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$G$</td>
<td>$D$</td>
<td>$W$</td>
<td>$G$</td>
</tr>
<tr>
<td>Grover</td>
<td>159</td>
<td>95</td>
<td>64</td>
<td>177</td>
</tr>
<tr>
<td>Tani</td>
<td>144</td>
<td>94</td>
<td>64</td>
<td>162</td>
</tr>
<tr>
<td>vOW</td>
<td>158</td>
<td>104</td>
<td>64</td>
<td>185</td>
</tr>
<tr>
<td>Grover</td>
<td>175</td>
<td>79</td>
<td>96</td>
<td>193</td>
</tr>
<tr>
<td>Tani</td>
<td>160</td>
<td>78</td>
<td>96</td>
<td>178</td>
</tr>
<tr>
<td>vOW</td>
<td>142</td>
<td>56</td>
<td>96</td>
<td>169</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>SIKEp434</th>
<th>SIKEp503</th>
<th>SIKEp610</th>
<th>SIKEp751</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$G$</td>
<td>$D$</td>
<td>$W$</td>
<td>$G$</td>
</tr>
<tr>
<td>Grover</td>
<td>132</td>
<td>122</td>
<td>10</td>
<td>150</td>
</tr>
<tr>
<td>Tani</td>
<td>124</td>
<td>114</td>
<td>25</td>
<td>142</td>
</tr>
<tr>
<td>vOW</td>
<td>132</td>
<td>14</td>
<td>128</td>
<td>150</td>
</tr>
</tbody>
</table>
power analysis (DPA) statistically analyzes many power runs of a device. Timing analysis targets timing information of various portions of the computation. Electromagnetic radiation can be seen as an extension of power analysis attacks by analyzing electromagnetic emissions instead of power.

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

1. Reveal parts of the hidden kernel point,
2. Reveal secret isogeny walks during the isogeny computation.

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

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

In targeting these parts of the SIKE cryptosystem, an attacker no doubt has access to a wide range of power, timing, fault, and various other side-channels. Constant-time implementations using a constant set of operations has been shown to be a good countermeasure against SPA and timing attacks. Higher level differential power analysis attacks and fault injection attacks are much harder to defend against. Papers and publications describing side-channel attacks against SIKE and countermeasures include [18, 25, 26, 39]. 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.

---

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

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>$SIKE_{p434}$</th>
<th>$SIKE_{p503}$</th>
<th>$SIKE_{p610}$</th>
<th>$SIKE_{p751}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grover</td>
<td>$132$</td>
<td>$122$</td>
<td>$10$</td>
<td>$150$</td>
</tr>
<tr>
<td>Tani</td>
<td>$131$</td>
<td>$122$</td>
<td>$10$</td>
<td>$149$</td>
</tr>
<tr>
<td>VW</td>
<td>$132$</td>
<td>$14$</td>
<td>$128$</td>
<td>$150$</td>
</tr>
</tbody>
</table>
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 [24]. Although there exists a subexponential quantum algorithm [6] that can solve the analogue of SIDH that uses ordinary curves (this scheme was first suggested by Couveignes in 1997 [10] and later published by Rostovtsev and Stolbunov in 2006 [32, 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-finding algorithms, then we have known lower bounds on the respective classical and quantum complexities in the asymptotic case (cf. §4.1). Moreover, two recent works [1, 9] have both shown that implementations of the vOW algorithm in the context of the computational supersingular isogeny problem essentially perform in exact accordance with the theoretical predictions made by van Oorschot and Wiener [40]. Coupled with the recent quantum cryptanalysis performed by Jaques and Schanck [21], these works provide confidence in the concrete security of the parameterizations in the present proposal.

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

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

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

The ease of partnering supersingular isogeny-based public-key cryptography with strong classical elliptic curve cryptography (ECC) is discussed in [8, §8]. In particular, a sound SIKE software library contains all of the ingredients necessary to securely implement elliptic curve Diffie-Hellman in a hybrid key exchange scheme, with a minimal amount of additional coding effort required. As in the case of high-performance ECC implementations, a large portion of the code is dedicated to tailored arithmetic in the underlying finite field. Strong, well-chosen Montgomery curves (like those recently chosen for adoption in TLS [28]) can be defined over any large enough prime field, and (beyond the field arithmetic) are essentially implemented in the same way. Even when defined over the 434 and 503-bit prime fields corresponding to SIKEp434 and SIKEp503, this technique gives rise to respective SIKE+ECDH hybrids that offer around 216 and 251 bits of ECDLP security, the latter being comparable to the NISTp521 curve. The corresponding (uncompressed) SIKE+ECDH public keys inflate by a factor of no more than 1.17x relative to SIKE alone, and the benchmarks reported in [8, Table 3] show that the performance slowdown is even less than this factor.

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


[38] The National Institute of Standards and Technology (NIST). Submission requirements and evaluation criteria for the post-quantum cryptography standardization process, December, 2016. 44, 45


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 3–24 can compute isogen_{\ell} and isoex_{\ell} for \( \ell \in \{2, 3\} \) in their entirety for the three sets of parameters SIKEp434, SIKEp503, and SIKEp751. In the case of SIKEp610, the exponent of 2 is odd, meaning the algorithm needs to start or finish with a single 2-isogeny; for simplicity, we have presented the algorithms using 4-isogenies only, but refer to the codebase(s) in the case of SIKEp610 for the scenario where a 2-isogeny is needed.

The notation \((X : Z)\) with \(Z \neq 0\) is used for the projective tuple in \( \mathbb{P}^1(\mathbb{F}_p^2) \) representing the Montgomery \( x \)-coordinate \(x = X/Z\); 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 [14].

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

---

**Algorithm 3: Coordinate doubling**

```plaintext
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. \(t_0^2 \leftarrow t_0^2\)
4. \(t_1^2 \leftarrow t_1^2\)
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 ∈ Z
    Output: (X_{[2^e]P} : Z_{[2^e]P})

1. (X' : Z') ← (X_P : Z_P)
2. for i = 1 to e do
3.     (X' : Z') ← xDBL((X' : Z'), (A_{24}^+ : C_{24})) // 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) ~ (A + 2C : 4C)
    Output: (X_{[2]P} : Z_{[2]P}), (X_{P+Q} : Z_{P+Q})

1. t_0 ← X_P + Z_P
2. t_1 ← X_P - Z_P
3. X_{[2]P} ← t_0^2
4. t_2 ← X_Q - Z_Q
5. X_{P+Q} ← X_Q + Z_Q
6. t_0 ← t_0 · t_2
7. Z_{[2]P} ← t_1^2
8. t_1 ← t_1 · X_{P+Q}
9. t_2 ← X_{[2]P} - Z_{[2]P}
11. X_{P+Q} ← a_{24}^+ · t_2
12. Z_{P+Q} ← t_0 - t_1
13. Z_{[2]P} ← X_{P+Q} + Z_{[2]P}
14. X_{P+Q} ← t_0 + t_1
15. Z_{[2]P} ← Z_{[2]P} · t_2
16. Z_{P+Q} ← Z_{P+Q}^2
17. X_{P+Q} ← X_{P+Q}^2
18. Z_{P+Q} ← X_{Q-P} · Z_{P+Q}
19. X_{P+Q} ← Z_{Q-P} · 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 ← X_P - Z_P
2. t_2 ← t_0^2
3. t_1 ← X_P + Z_P
4. t_3 ← t_1^2
5. t_4 ← t_1 + t_0
6. t_0 ← t_1 - t_0
7. t_1 ← t_4^2
8. t_1 ← t_1 - t_3
9. t_1 ← t_1 - t_2
10. t_5 ← t_3 · A_{24}^+
11. t_3 ← t_5 · t_3
12. t_6 ← t_2 · A_{24}^-
13. t_2 ← t_2 · t_6
14. t_3 ← t_2 - t_3
15. t_2 ← t_5 - t_6
16. t_1 ← t_2 · t_1
17. t_2 ← t_3 + t_1
18. t_2 ← t_5^2
19. X_{[3]P} ← t_2 · t_4
20. t_1 ← t_3 - t_1
21. t_1 ← t_1^2
22. Z_{[3]P} ← t_1 · t_0
23. return (X_{[3]P} : Z_{[3]P})

55
Algorithm 7: Repeated coordinate tripling

function xTPL e
   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 \((X' : Z') \leftarrow xTPL((X' : Z'), (A_{24}^+: A_{24}^-))\) // Alg. 6
4 return \((X' : Z')\)

Algorithm 8: Three point ladder

function Ladder3pt
   Input: \(m = (m_{\ell-1}, \ldots, m_0) \in \mathbb{Z}, (x_P, x_Q, x_{Q-P}), \) and \((A : 1)\)
   Output: \((X_{P+[m]Q} : Z_{P+[m]Q})\)
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 \(a_{24}^- \leftarrow (A + 2)/4\)
3 for \(i = 0 \) to \(\ell - 1 \) do
4    if \(m_i = 1\) then
5       \(((X_0 : Z_0), (X_1 : Z_1)) \leftarrow \text{xDBLADD}((X_0 : Z_0), (X_1 : Z_1), (X_2 : Z_2), (a_{24}^+ : 1))\) // Alg. 5
6    else
7       \(((X_0 : Z_0), (X_2 : Z_2)) \leftarrow \text{xDBLADD}((X_0 : Z_0), (X_2 : Z_2), (X_1 : Z_1), (a_{24}^+ : 1))\) // Alg. 5
8 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 \text{return } j
Algorithm 10: Recovering the Montgomery curve coefficient

```python
function get_A
  Input: xp, xQ and xQ-P corresponding to points on E_A: \( y^2 = x^3 + Ax^2 + x \)
  Output: A ∈ 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 2-isogenous curve

```python
function 2_iso_curve
  Input: (X_{P_2} : Z_{P_2}), where P_2 has exact order 2 on E_{A/C}
  Output: (A_{24}^+ : C_{24}) ~ (A' + 2C' : 4C') corresponding to E_{A'/C'} = E_{A/C}/\langle P_2 \rangle

1. A_{24}^+ \leftarrow X_{P_2}^2
2. C_{24} \leftarrow Z_{P_2}^2
3. A_{24}^+ \leftarrow A_{24}^+ - C_{24}
4. return A_{24}^+, C_{24}
```

Algorithm 12: Evaluating a 2-isogeny at a point

```python
function 2_iso_eval
  Input: (X_{P_2} : Z_{P_2}), where P_2 has exact order 2 on E_{A/C}, and (X_Q : Z_Q) where Q ∈ E_{A/C}
  Output: (X_{Q'} : Z_{Q'}) corresponding to Q' ∈ E_{A'/C'}, where E_{A'/C'} is the curve 2-isogenous to E_{A/C} output from 2_iso_curve

1. \( t_0 \leftarrow X_{P_2} + Z_{P_2} \)
2. \( t_1 \leftarrow X_{P_2} - Z_{P_2} \)
3. \( t_2 \leftarrow X_Q + Z_Q \)
4. \( t_3 \leftarrow X_Q - Z_Q \)
5. \( t_0 \leftarrow t_0 \cdot t_3 \)
6. \( t_1 \leftarrow t_1 \cdot t_2 \)
7. \( t_2 \leftarrow t_0 + t_1 \)
8. \( t_3 \leftarrow t_0 - t_1 \)
9. \( X_{Q'} \leftarrow X_Q \cdot t_2 \)
10. \( Z_{Q'} \leftarrow Z_Q \cdot t_3 \)
11. return (X_{Q'} : Z_{Q'})
```

Algorithm 13: Computing the 4-isogenous curve

```python
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}) ~ (A' + 2C' : 4C') corresponding to E_{A'/C'} = E_{A/C}/\langle P_4 \rangle, and constants (K_1, K_2, K_3) ∈ (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. \( A_{24}^+ \leftarrow A_{24}^+ + A_{24}^+ \) (K_1, K_2, K_3)
9. \( A_{24}^+ \leftarrow (A_{24}^+)^2 \)
10. return A_{24}^+, C_{24},
```
Algorithm 14: Evaluating a 4-isogeny at a point

\[
\text{function } 4\_iso\_eval \\
\text{Input: Constants } (K_1, K_2, K_3) \in (\mathbb{F}_p)^3 \text{ from } 4\_iso\_curve, \text{ and } (X_Q : Z_Q) \text{ where } Q \in E_{A/C} \\
\text{Output: } (X_{Q'} : Z_{Q'}) \text{ corresponding to } Q' \in E_{A'/C'}, \text{ where } E_{A'/C'} \text{ is the curve 4-isogenous to } E_{A/C} \text{ output from } 4\_iso\_curve
\]

\[
\begin{align*}
1. & \quad t_0 \leftarrow X_Q + Z_Q, \\
2. & \quad t_1 \leftarrow X_Q - Z_Q, \\
3. & \quad X_Q \leftarrow t_0 \cdot K_2, \\
4. & \quad Z_Q \leftarrow t_1 \cdot K_3, \\
5. & \quad t_0 \leftarrow t_0 \cdot t_1, \\
6. & \quad t_0 \leftarrow t_0 \cdot K_1, \\
7. & \quad t_1 \leftarrow X_Q + Z_Q, \\
8. & \quad Z_Q \leftarrow X_Q - Z_Q, \\
9. & \quad t_1 \leftarrow t_1^2, \\
10. & \quad Z_Q \leftarrow Z_Q^2, \\
11. & \quad X_Q \leftarrow t_0 + t_1, \\
12. & \quad t_0 \leftarrow Z_Q - t_0, \\
13. & \quad X_{Q'} \leftarrow X_Q \cdot t_1, \\
14. & \quad Z_{Q'} \leftarrow Z_Q \cdot t_0, \\
15. & \quad \text{return } (X_{Q'} : Z_{Q'})
\end{align*}
\]

Algorithm 15: Computing the 3-isogenous curve

\[
\text{function } 3\_iso\_curve \\
\text{Input: } (X_{P_3} : Z_{P_3}), \text{ where } P_3 \text{ has exact order 3 on } E_{A/C} \\
\text{Output: Curve constant } (A_{24}^+ : A_{24}^-) = (A' + 2C' : A' - 2C') \text{ corresponding to } E_{A'/C'} = E_{A/C}/\langle P_3 \rangle, \text{ and } \\
\text{constants } (K_1, K_2) \in (\mathbb{F}_p)^2
\]

\[
\begin{align*}
1. & \quad K_1 \leftarrow X_{P_3} - Z_{P_3}, \\
2. & \quad t_0 \leftarrow K_1^2, \\
3. & \quad K_2 \leftarrow X_{P_3} + Z_{P_3}, \\
4. & \quad t_1 \leftarrow K_2^2, \\
5. & \quad t_2 \leftarrow t_0 + t_1, \\
6. & \quad t_3 \leftarrow K_1 + K_2, \\
7. & \quad t_3 \leftarrow t_3^2, \\
8. & \quad t_3 \leftarrow t_3 - t_2, \\
9. & \quad t_2 \leftarrow t_1 + t_3, \\
10. & \quad t_4 \leftarrow t_1 + t_4, \\
11. & \quad t_4 \leftarrow t_5 + t_0, \\
12. & \quad t_4 \leftarrow t_4 + t_4, \\
13. & \quad t_4 \leftarrow t_1 + t_4, \\
14. & \quad A_{24}^- \leftarrow t_2 \cdot t_4, \\
15. & \quad A_{24}^+ \leftarrow A_{24}^- + t_0, \\
16. & \quad t_4 \leftarrow t_4 + t_4, \\
17. & \quad t_4 \leftarrow t_0 + t_4, \\
18. & \quad t_4 \leftarrow t_3 \cdot t_4, \\
19. & \quad t_0 \leftarrow t_4 - A_{24}^+, \\
20. & \quad A_{24}^+ \leftarrow A_{24}^- + t_0, \\
21. & \quad \text{return } (A_{24}^+ : A_{24}^-), \\
& \quad (K_1, K_2) \in (\mathbb{F}_p)^2
\end{align*}
\]

Algorithm 16: Evaluating a 3-isogeny at a point

\[
\text{function } 3\_iso\_eval \\
\text{Input: Constants } (K_1, K_2) \in (\mathbb{F}_p)^3 \text{ output from } 3\_iso\_curve \text{ together with } (X_Q : Z_Q) \text{ corresponding to } Q \in E_{A/C} \\
\text{Output: } (X_{Q'} : Z_{Q'}) \text{ corresponding to } Q' \in E_{A'/C'}, \text{ where } E_{A'/C'} \text{ is 3-isogenous to } E_{A/C}
\]

\[
\begin{align*}
1. & \quad t_0 \leftarrow X_Q + Z_Q, \\
2. & \quad t_1 \leftarrow X_Q - Z_Q, \\
3. & \quad t_0 \leftarrow K_1 \cdot t_0, \\
4. & \quad t_1 \leftarrow K_2 \cdot t_1, \\
5. & \quad t_2 \leftarrow t_0 + t_1, \\
6. & \quad t_0 \leftarrow t_1 - t_0, \\
7. & \quad t_2 \leftarrow t_2^2, \\
8. & \quad t_0 \leftarrow t_0^2, \\
9. & \quad X_{Q'} \leftarrow X_Q \cdot t_2, \\
10. & \quad Z_{Q'} \leftarrow Z_Q \cdot t_0, \\
11. & \quad \text{return } (X_{Q'} : Z_{Q'})
\end{align*}
\]

58
Algorithm 17: Computing and evaluating a 2\textsuperscript{e}-isogeny, simple version

function 2\_e\_iso

Static parameters: Integer e\textsubscript{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 \(2^{e_2}\) on \(E_{A/C}\)

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

Output: \((A_{24}^+ \cdot C_{24}^\prime)\) 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 \(e = e_2 - 2\) downto 0 by \(-2\) do

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

\(((A_{24}^+ : C_{24}), (K_1, K_2, K_3)) ← 4\_iso\_curve ((X_T : Z_T))\) // Alg. 13

\((X_S : Z_S) ← 4\_iso\_eval ((K_1, K_2, K_3), (X_S : Z_S))\) // Alg. 14

for \((X_j : Z_j)\) in optional input do

\((X_j : Z_j) ← 4\_iso\_eval ((K_1, K_2, K_3), (X_j : Z_j))\) // Alg. 14

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

Algorithm 18: Computing and evaluating a 3\textsuperscript{e}-isogeny, simple version

function 3\_e\_iso

Static parameters: Integer e\textsubscript{3} from the public parameters

Input: Constants \((A_{24}^+ : A_{24}^\prime)\) corresponding to a curve \(E_{A/C}, (X_S : Z_S)\) where \(S\) has exact order \(3^{e_3}\) on \(E_{A/C}\)

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

Output: \((A_{24}^+ : A_{24}^\prime)\) 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 \(e = e_3 - 1\) downto 0 by \(-1\) do

\((X_T : Z_T) ← xTPLe \(((X_S : Z_S), (A_{24}^+ : A_{24}^\prime), e)\) // Alg. 7

\[((A_{24}^+ : A_{24}^\prime), (K_1, K_2)) ← 3\_iso\_curve ((X_T : Z_T))\) // Alg. 15

\((X_S : Z_S) ← 3\_iso\_eval ((K_1, K_2), (X_S : Z_S))\) // Alg. 16

for \((X_j : Z_j)\) in optional input do

\((X_j : Z_j) ← 3\_iso\_eval ((K_1, K_2), (X_j : Z_j))\) // Alg. 16

7 return \((A_{24}^+ : A_{24}^\prime), [(X_1 : Z_1), (X_2 : Z_2), (X_3 : Z_3)]\)
Algorithm 19: Computing and evaluating a $2^e$-isogeny, optimized version

function $2_e$-iso

Static parameters: Integer $e_2$ from the public parameters, a strategy

$(s_1, \ldots, s_{e_2/2-1}) \in (\mathbb{N}^*)^{e_2/2-1}$

Input: Constants $(A^+_2 : C_{24})$ corresponding to a curve $E_{A/C}$, $(X_S : Z_S)$ where $S$ has exact order $2^{e_1}$ on $E_{A/C}$

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

Output: $(A^+_1 : C'_{24})$ 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. Initialize empty deque $S$
2. push($S, (e_2/2, (X_S : Z_S))$)
3. $i \leftarrow 1$
4. while $S$ not empty do
   5. $(h, (X : Z)) \leftarrow$ pop($S$)
   6. if $h = 1$ then
      7. $(A^+_2 : C_{24}), (K_1, K_2, K_3) \leftarrow 4$-iso.curve($(X : Z)$)  // Alg. 13
      8. Initialize empty deque $S'$
      9. while $S$ not empty do
         10. $(h, (X : Z)) \leftarrow$ pull($S$)
         11. $(X : Z) \leftarrow 4$-iso.eval($(K_1, K_2, K_3), (X : Z)$)  // Alg. 14
         12. push($S', (h - 1, (X : Z))$)
      13. $S \leftarrow S'$
      14. for $(X_j : Z_j)$ in optional input do
         15. $(X_j : Z_j) \leftarrow 4$-iso.eval($(K_1, K_2, K_3), (X_j : Z_j)$)  // Alg. 14
    else if $0 < s_i < h$ then
       16. push($S, (h, (X : Z)))$
       17. $(X : Z) \leftarrow xDBLe((X : Z), (A^+_2 : C_{24}), 2 \cdot s_i)$  // Alg. 4
       18. push($S, (h - s_i, (X : Z))$)
       19. $i \leftarrow i + 1$
    else
       20. Error: Invalid strategy
21. return $(A^+_2 : C_{24}), [(X_1 : Z_1), (X_2 : Z_2), (X_3 : Z_3)]$
Algorithm 20: Computing and evaluating a 3e-isogeny, optimized version

function $3_{e\text{-iso}}$

**Static parameters:** Integer $e_3$ from the public parameters, a strategy $(s_1, \ldots, s_{e_3-1}) \in (\mathbb{N}^*)^{e_3-1}$

**Input:** Constants $(A_{24}^+ : A_{24}^-)$ corresponding to a curve $E_{A/C}$, $(X_S : Z_S)$ where $S$ has exact order $3^{e_3}$ 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 $S$
2. $\text{push}(S, (e_3, (X_S : Z_S)))$
3. $i \leftarrow 1$
4. **while** $S$ **not empty** **do**
5. \hspace{1em} $(h, (X : Z)) \leftarrow \text{pop}(S)$
6. \hspace{1em} **if** $h = 1$ **then**
7. \hspace{2em} $((A_{24}^+ : A_{24}^-), (K_1, K_2)) \leftarrow 3\_\text{iso\_curve}((X : Z))$  \hspace{1em} // Alg. 15
8. \hspace{2em} Initialize empty deque $S'$
9. \hspace{2em} **while** $S'$ **not empty** **do**
10. \hspace{3em} $(h, (X : Z)) \leftarrow \text{pull}(S)$
11. \hspace{3em} $(X : Z) \leftarrow 3\_\text{iso\_eval}((K_1, K_2), (X : Z))$  \hspace{1em} // Alg. 16
12. \hspace{3em} $\text{push}(S', (h - 1, (X : Z)))$
13. \hspace{2em} $S \leftarrow S'$
14. \hspace{2em} **for** $(X_j : Z_j)$ **in** optional input **do**
15. \hspace{3em} $(X_j : Z_j) \leftarrow 3\_\text{iso\_eval}((K_1, K_2), (X_j : Z_j))$  \hspace{1em} // Alg. 16
16. **else if** $0 < s_i < h$ **then**
17. \hspace{2em} $\text{push}(S, (h, (X : Z)))$
18. \hspace{2em} $(X : Z) \leftarrow \text{xTPLe}\left((X : Z), (A_{24}^+ : A_{24}^-), s_i\right)$  \hspace{1em} // Alg. 7
19. \hspace{2em} $\text{push}(S, (h - s_i, (X : Z)))$
20. \hspace{2em} $i \leftarrow i + 1$
21. **else**
22. \hspace{2em} **Error:** invalid strategy
23. **return** $(A_{24}^+ : A_{24}^-), [(X_1 : Z_1), (X_2 : Z_2), (X_3 : Z_3)]$
Algorithm 21: Computing public keys in the 2-torsion

function isogen₂

Input: Secret key $sk_2 \in \mathbb{Z}$ (see §1.2.6) and public parameters
\{e₂, e₃, p, xP₂, xQ₂, xR₂, xP₃, xQ₃, xR₃\} (see §1.6)

Output: Public key $pk_2 = (x_1, x_2, x_3)$ equivalent to the output of Step 4 of isogen₇ (see §1.3.5)

1. $((A : C), (A_{24}^+ : C_{24})) \leftarrow ((6 : 1), (8 : 4))$
2. $((X_1 : Z_1), (X_2 : Z_2), (X_3 : Z_3)) \leftarrow ((xP₂ : 1), (xQ₂ : 1), (xR₂ : 1))$
3. $(X_5 : Z_5) \leftarrow \text{Ladder3pt}(sk_2, (xP₂, xQ₂, xR₂), (A : C))$  \hspace{1em} // Alg. 8
4. $((A_{24}^+ : C_{24}), (X_1 : Z_1), (X_2 : Z_2), (X_3 : Z_3)) \leftarrow$
   $2\_e\_iso((A_{24}^+ : C_{24}), (X_5 : Z_5), (X_1 : Z_1), (X_2 : Z_2), (X_3 : Z_3))$  \hspace{1em} // Alg. 17 or Alg. 19
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 $pk_2 = (x_1, x_2, x_3)$  \hspace{1em} // Encoded as in §1.2.9

Algorithm 22: Computing public keys in the 3-torsion

function isogen₃

Input: Secret key $sk₃ \in \mathbb{Z}$ (see §1.2.6) and public parameters
\{e₂, e₃, p, xP₂, xQ₂, xR₂, xP₃, xQ₃, xR₃\} (see §1.6)

Output: Public key $pk₃ = (x_1, x_2, x_3)$ equivalent to the output of Step 4 of isogen₇ (see §1.3.5)

1. $((A : C), (A_{24}^+ : A_{24}^-)) \leftarrow ((6 : 1), (8 : 4))$
2. $((X_1 : Z_1), (X_2 : Z_2), (X_3 : Z_3)) \leftarrow ((xP₂ : 1), (xQ₂ : 1), (xR₂ : 1))$
3. $(X_5 : Z_5) \leftarrow \text{Ladder3pt}(sk_3, (xP₃, xQ₃, xR₃), (A : C))$  \hspace{1em} // Alg. 8
4. $((A_{24}^+ : A_{24}^-), (X_1 : Z_1), (X_2 : Z_2), (X_3 : Z_3)) \leftarrow$
   $3\_e\_iso((A_{24}^+ : A_{24}^-), (X_5 : Z_5), (X_1 : Z_1), (X_2 : Z_2), (X_3 : Z_3))$  \hspace{1em} // Alg. 18 or Alg. 20
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 $pk₃ = (x_1, x_2, x_3)$  \hspace{1em} // Encoded as in §1.2.9
Algorithm 23: Establishing shared keys in the 2-torsion

function isoex₂

Input: Secret key sk₂ ∈ ℤ (see §1.2.6), public key pk₃ = (x₁, x₂, x₃) ∈ (𝔽ₚ²)³ (see §1.2.9), and parameter e₂ (see §1.6)

Output: A j-invariant j₂ equivalent to the output of Step 4 of isogenₓ (see §1.3.6)

1 (A : C) ← (get₁₋₂₄ₐ(x₁, x₂, x₃) : 1) // Alg. 10
2 (Xₛ : Zₛ) ← Ladder₃pt(sk₂, (x₁, x₂, x₃), (A : C)) // Alg. 8
3 (A₂₄⁺ : C₂₄) ← (A + 2 : A − 2)
4 (A₂₄⁻ : C₂₄) ← 2₋ₑ_iso((A₂₄⁺ : C₂₄), (Xₛ : Zₛ)) // Alg. 17 or Alg. 19
5 (A : C) ← (4A₂₄⁺ − 2C₂₄ : C₂₄)
6 j = j₋₂₄₋₂₄((A : C)) // Alg. 9
7 return j // Encoded as in §1.2.8

Algorithm 24: Establishing shared keys in the 3-torsion

function isoex₃

Input: Secret key sk₃ ∈ ℤ (see §1.2.6), public key pk₂ = (x₁, x₂, x₃) ∈ (𝔽ₚ²)³ (see §1.2.9), and parameter e₃ (see §1.6)

Output: A j-invariant j₃ equivalent to the output of Step 4 of isogenₓ (see §1.3.6)

1 (A : C) ← (get₁₋₃₋₃₄ₐ(x₁, x₂, x₃) : 1) // Alg. 10
2 (Xₛ : Zₛ) ← Ladder₃pt(sk₃, (x₁, x₂, x₃), (A : C)) // Alg. 8
3 (A₂₄⁺ : A₂₄⁻) ← (A + 2 : A − 2)
4 (A₂₄⁺ : A⁻₂₄⁻) ← 3₋ₑ_iso((A₂₄⁺ : A⁻₂₄⁻), (Xₛ : Zₛ)) // Alg. 18 or Alg. 20
5 (A : C) ← (2 · (A⁻₂₄⁻ + A⁺₂₄⁺) : A⁺₂₄⁺ − A⁻₂₄⁻)
6 j = j₋ₑ₋₂₄₋₂₄₋₂₄((A : C)) // Alg. 9
7 return j // Encoded as in §1.2.8
Appendix B

Explicit algorithms for \( \text{isogen}_\ell \) and \( \text{isoex}_\ell \): 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 25–45 can compute \( \text{isogen}_\ell \) and \( \text{isoex}_\ell \) for \( \ell \in \{2, 3\} \) in their entirety.

The notation \((x_P, y_P)\) is used for the affine tuple in \( \mathbb{F}^1(\mathbb{F}_{p^2}) \) 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 \(E_{a,b}/\mathbb{F}_{p^2} : by^2 = x^3 + ax^2 + x\).

Algorithm 25: Affine coordinate doubling

```
function xDBL

Input: \((x_P, y_P)\) and \((a, b)\)
Output: \((x_{[2]}P, y_{[2]}P)\)

1 if \(P = \infty\) then
2    return \(\infty\)

3 \(t_0 \leftarrow xp^2\)
4 \(t_1 \leftarrow t_0 + t_0\)
5 \(t_2 \leftarrow 1\)
6 \(t_0 \leftarrow t_0 + t_1\)
7 \(t_1 \leftarrow a \cdot xp\)
8 \(t_1 \leftarrow t_1 + t_1\)
9 \(t_0 \leftarrow t_0 + t_1\)
10 \(t_0 \leftarrow t_0 + t_2\)
11 \(t_1 \leftarrow b \cdot y_P\)
12 \(t_1 \leftarrow t_1 + t_1\)
13 \(t_1 \leftarrow t_1^{-1}\)
14 \(t_0 \leftarrow t_0 \cdot t_1\)
15 \(t_1 \leftarrow t_1^2\)
16 \(t_2 \leftarrow t_1 \cdot t_1\)
17 \(t_2 \leftarrow t_2 - a\)
18 \(t_2 \leftarrow t_2 - xp\)
19 \(t_2 \leftarrow t_2 - x_P\)
20 \(t_1 \leftarrow t_0 \cdot t_1\)
21 \(t_1 \leftarrow b \cdot t_1\)
22 \(t_1 \leftarrow t_1 + y_P\)
23 \(y_{[2]}P \leftarrow x_P + x_P\)
24 \(y_{[2]}P \leftarrow y_{[2]}P + xp\)
25 \(y_{[2]}P \leftarrow y_{[2]}P + a\)
26 \(y_{[2]}P \leftarrow y_{[2]}P \cdot t_0\)
27 \(y_{[2]}P \leftarrow y_{[2]}P - t_1\)
28 \(x_{[2]}P \leftarrow t_2\)
29 return \((x_{[2]}P, y_{[2]}P)\)
```
Algorithm 26: Repeated affine coordinate doubling

function \texttt{xDBLe}

\begin{itemize}
\item \textbf{Input:} \((x_P, y_P), (a, b), \text{ and } e \in \mathbb{Z}\)
\item \textbf{Output:} \((x_{[2^e]P}, y_{[2^e]P})\)
\end{itemize}

1 \((x', y') \leftarrow (x_P, y_P)\)
2 \textbf{for} \(i = 1\) \textbf{to} \(e\) \textbf{do}
3 \((x', y') \leftarrow \texttt{xDBL}((x', y'), (a, b))\) \hspace{1cm} // Alg. 25
4 \textbf{return} \((x', y')\)

Algorithm 27: Affine coordinate addition

function \texttt{xADD}

\begin{itemize}
\item \textbf{Input:} \(P = (x_P, y_P), Q = (x_Q, y_Q), \text{ and } (a, b)\)
\item \textbf{Output:} \((x_{P+Q}, y_{P+Q})\)
\end{itemize}

1 \textbf{if} \(P = \infty\) \textbf{then}
2 \textbf{return} \((x_Q, y_Q)\)
3 \textbf{if} \(Q = \infty\) \textbf{then}
4 \textbf{return} \((x_P, y_P)\)
5 \textbf{if} \(P = Q\) \textbf{then}
6 \textbf{return} \texttt{xDBL}((x_P, y_P), (a, b))
7 \textbf{if} \(P = -Q\) \textbf{then}
8 \textbf{return} \(\infty\)
9 \(t_0 \leftarrow y_Q - y_P\)
10 \(t_1 \leftarrow x_Q - x_P\)
11 \(t_1 \leftarrow t_1^{-1}\)
12 \(t_0 \leftarrow t_0 \cdot t_1\)
13 \(t_1 \leftarrow t_0^2\)
14 \(t_2 \leftarrow x_P + x_Q\)
15 \(t_2 \leftarrow t_2 + x_0\)
16 \(t_2 \leftarrow t_2 + a\)
17 \(t_2 \leftarrow t_2 \cdot t_0\)
18 \(t_0 \leftarrow t_0 \cdot t_1\)
19 \(t_0 \leftarrow b \cdot t_0\)
20 \(t_0 \leftarrow t_0 + y_P\)
21 \(t_0 \leftarrow t_2 - t_0\)
22 \(t_1 \leftarrow b \cdot t_1\)
23 \(t_1 \leftarrow t_1 - a\)
24 \(t_1 \leftarrow t_1 - x_P\)
25 \(x_{P+Q} \leftarrow t_1 - x_Q\)
26 \(y_{P+Q} \leftarrow t_0\)
27 \textbf{return} \((x_{P+Q}, y_{P+Q})\)

Algorithm 28: Affine coordinate tripling

function \texttt{xTPL}

\begin{itemize}
\item \textbf{Input:} \((x_P, y_P)\) and \((a, b)\)
\item \textbf{Output:} \((x_{[3]P}, y_{[3]P})\)
\end{itemize}

1 \((x_{[2]P}, y_{[2]P}) \leftarrow \texttt{xDBL}((x_P, y_P), (a, b))\) \hspace{1cm} // Alg. 25
2 \((x_{[3]P}, y_{[3]P}) \leftarrow \texttt{xADD}((x_P, y_P), (x_{[2]P}, y_{[2]P}), (a, b))\) \hspace{1cm} // Alg. 27
3 \textbf{return} \((x_{[3]P}, y_{[3]P})\)
Algorithm 29: Repeated affine coordinate tripling

function $x$TPL$_e$

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 $e$
3 $(x', y') \leftarrow x$TPL$((x', y'), (a, b))$ // Alg. 28
4 return $(x', y')$

Algorithm 30: Double-and-add scalar multiplication

function double_and_add

Input: $m = (m_{\ell-1}, \ldots, m_0)_2 \in \mathbb{Z}$, $P = (x, y)$, and $(a, b)$

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

1 $(x_0, y_0) \leftarrow (0, 0)$
2 for $i = \ell - 1$ to $0$ by $-1$
3 $(x_0, y_0) \leftarrow x$DBL$((x_0, y_0), (a, b))$ // Alg. 25
4 if $m_i = 1$
5 $(x_0, y_0) \leftarrow x$ADD$((x_0, y_0), (x, y), (a, b))$ // Alg. 27
6 return $(x_0, y_0)$

Algorithm 31: Montgomery $j$-invariant computation

function $j$ _inv

Input: $a$

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

1 $t_0 \leftarrow a^2$ 6 $j \leftarrow j + j$
2 $j \leftarrow 3$ 7 $j \leftarrow j + j$
3 $j \leftarrow t_0 - j$ 8 $j \leftarrow j + j$
4 $t_1 \leftarrow j^2$ 9 $j \leftarrow j + j$
5 $j \leftarrow j \cdot t_1$ 10 $j \leftarrow j + j$
11 $j \leftarrow j + j$ 14 $t_1 \leftarrow 4$
12 $j \leftarrow j + j$ 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 32: Computing the 2-isogenous curve

function curve_2_iso
    Input: \( x_{P_2} \) and \( b \), where \( P_2 \) has exact order 2 on \( E_{a,b} \)
    Output: \((a', b')\) corresponding to \( E_{a',b'} = E_{a,b}/\langle P_2 \rangle \)

1. \( t_1 \leftarrow x_{P_2}^2 \)
2. \( t_1 \leftarrow t_1 + t_1 \)
3. \( t_1 \leftarrow 1 - t_1 \)
4. \( a' \leftarrow t_1 + t_1 \)
5. \( b' \leftarrow x_{P_2} \cdot b \)
6. return \((a', b')\)

Algorithm 33: Evaluating a 2-isogeny at a point

function eval_2_iso
    Input: \((x_Q, y_Q)\) and \( x_{P_2} \), where \( P \in E_{a,b} \), and \( P_2 \) has exact order 2 on \( E_{a,b} \)
    Output: \((x_{Q'}, y_{Q'})\) corresponding to \( Q' \in E_{a',b'} \), where \( E_{a',b'} \) is the curve 2-isogenous to \( E_{a,b} \), output from curve_2_iso

1. \( t_1 \leftarrow x_Q \cdot x_{P_2} \)
2. \( t_2 \leftarrow x_Q \cdot t_1 \)
3. \( t_3 \leftarrow t_1 \cdot x_{P_2} \)
4. \( t_3 \leftarrow t_3 + t_3 \)
5. \( t_3 \leftarrow t_3 - t_3 \)
6. \( t_2 \leftarrow t_2 - x_Q \)
7. \( t_2 \leftarrow t_2 - x_Q \)
8. \( t_1 \leftarrow t_1 \cdot t_1 \)
9. \( t_1 \leftarrow t_1 - t_1 \)
10. \( t_1 \leftarrow t_1 - t_1 \)
11. \( t_1 \leftarrow t_1 - t_1 \)
12. \( t_1 \leftarrow t_1 - t_1 \)
13. \( y_{Q'} \leftarrow t_3 \cdot t_1 \)
14. return \((x_{Q'}, y_{Q'})\)

Algorithm 34: Computing the 4-isogenous curve

function curve_4_iso
    Input: \( x_{P_4} \) and \( b \), where \( P_4 \) has exact order 4 on \( E_{a,b} \)
    Output: \((a', b')\) corresponding to \( E_{a',b'} = E_{a,b}/\langle P_4 \rangle \)

1. \( t_1 \leftarrow x_{P_4}^2 \)
2. \( a' \leftarrow t_1^2 \)
3. \( a' \leftarrow a' + a' \)
4. \( a' \leftarrow a' + a' \)
5. \( t_2 \leftarrow 2 \)
6. \( t_2 \leftarrow a' - t_2 \)
7. \( t_1 \leftarrow x_{P_4} \cdot t_1 \)
8. \( t_1 \leftarrow t_1 + x_{P_4} \)
9. \( t_1 \leftarrow t_1 \cdot b \)
10. \( t_2 \leftarrow t_2 \cdot t_2 \)
11. \( t_2 \leftarrow -t_2 \)
12. \( b' \leftarrow t_2 \cdot t_2 \)
13. return \((a', b')\)
Algorithm 35: Evaluating a 4-isogeny at a point

\[
\text{function } \text{eval} \_4 \_iso \\
\text{Input: } (x_Q, y_Q) \text{ and } x_{P_4}, \text{ where } P \in E_{a,b}, \text{ and } P_4 \text{ has exact order 4 on } E_{a,b} \\
\text{Output: } (x_{Q'}, y_{Q'}) \text{ corresponding to } Q' \in E_{a',b'}, \text{ where } E_{a',b'} \text{ is the curve 4-isogenous to } E_{a,b}, \text{ output from curve} \_4 \_iso
\]

\[
\begin{align*}
1 & \quad t_1 \leftarrow x_Q^2 \\
2 & \quad t_2 \leftarrow t_1^2 \\
3 & \quad t_3 \leftarrow x_{P_4}^2 \\
4 & \quad t_4 \leftarrow t_2 \cdot t_3 \\
5 & \quad t_5 \leftarrow t_2 + t_3 \\
6 & \quad t_4 \leftarrow t_1 \cdot t_3 \\
7 & \quad t_4 \leftarrow t_4 + t_5 \\
8 & \quad t_5 \leftarrow t_4 + t_4 \\
9 & \quad t_5 \leftarrow t_5 + t_5 \\
10 & \quad t_4 \leftarrow t_4 + t_5 \\
11 & \quad t_5 \leftarrow t_2 + t_4 \\
12 & \quad t_4 \leftarrow t_5^2 \\
13 & \quad t_5 \leftarrow t_1 \cdot t_4 \\
14 & \quad t_5 \leftarrow t_5 + t_5 \\
15 & \quad t_2 \leftarrow t_2 + t_5 \\
16 & \quad t_1 \leftarrow t_1 \cdot x_Q \\
17 & \quad t_4 \leftarrow x_{P_4} \cdot t_3 \\
18 & \quad t_5 \leftarrow t_1 \cdot t_4 \\
19 & \quad t_5 \leftarrow t_5 + t_5 \\
20 & \quad t_5 \leftarrow t_5 + t_5 \\
21 & \quad t_2 \leftarrow t_2 - t_5 \\
22 & \quad t_1 \leftarrow t_1 \cdot t_3 \\
23 & \quad t_1 \leftarrow t_1 + t_1 \\
24 & \quad t_1 \leftarrow t_1 + t_1 \\
25 & \quad t_1 \leftarrow t_2 - t_1 \\
26 & \quad t_2 \leftarrow x_Q \cdot t_4 \\
27 & \quad t_2 \leftarrow t_2 - t_2 \\
28 & \quad t_2 \leftarrow t_2 - t_2 \\
29 & \quad t_1 \leftarrow t_1 - t_2 \\
30 & \quad t_1 \leftarrow t_1 + t_1 \\
31 & \quad t_1 \leftarrow t_1 + 1 \\
32 & \quad t_2 \leftarrow x_Q \cdot x_{P_4} \\
33 & \quad t_4 \leftarrow t_2 - 1 \\
34 & \quad t_2 \leftarrow t_2 + t_2 \\
35 & \quad t_5 \leftarrow t_2 + t_2 \\
36 & \quad t_1 \leftarrow t_1 - t_5 \\
37 & \quad t_1 \leftarrow t_4 \cdot t_1 \\
38 & \quad t_1 \leftarrow t_3 \cdot t_1 \\
39 & \quad t_1 \leftarrow y_Q \cdot t_1 \\
40 & \quad t_1 \leftarrow t_1 + t_1 \\
41 & \quad y_{Q'} \leftarrow -t_1 \\
42 & \quad t_2 \leftarrow t_2 - t_3 \\
43 & \quad t_1 \leftarrow t_2 - 1 \\
44 & \quad t_2 \leftarrow x_Q \cdot x_{P_4} \\
45 & \quad t_1 \leftarrow t_2 \cdot t_1 \\
46 & \quad t_5 \leftarrow t_1^2 \\
47 & \quad t_5 \leftarrow t_5 \cdot t_2 \\
48 & \quad t_5 \leftarrow t_5^1 \\
49 & \quad y_{Q'} \leftarrow y_Q \cdot t_5 \\
50 & \quad t_1 \leftarrow t_1 \cdot t_2 \\
51 & \quad t_1 \leftarrow t_1^{-1} \\
52 & \quad t_4 \leftarrow t_4^2 \\
53 & \quad t_1 \leftarrow t_1 \cdot t_4 \\
54 & \quad t_1 \leftarrow x_Q \cdot t_1 \\
55 & \quad t_2 \leftarrow x_Q \cdot t_3 \\
56 & \quad t_2 \leftarrow t_2 + x_Q \\
57 & \quad t_3 \leftarrow x_{P_4} + x_{P_4} \\
58 & \quad t_2 \leftarrow t_2 - t_3 \\
59 & \quad t_2 \leftarrow -t_2 \\
60 & \quad x_{Q'} \leftarrow t_1 \cdot t_2 \\
61 & \quad \text{return } (x_{Q'}, y_{Q'})
\end{align*}
\]

Algorithm 36: Computing the 3-isogenous curve

\[
\text{function } \text{curve} \_3 \_iso \\
\text{Input: } x_{P_3}, \text{ and } (a, b), \text{ where } P_3 \text{ has exact order 3 on } E_{a,b} \\
\text{Output: } \text{Curve constant } (a', b') \text{ corresponding to } E_{a',b'} = E_{a,b}/(P_3)
\]

\[
\begin{align*}
1 & \quad t_1 \leftarrow x_{P_3}^2 \\
2 & \quad b' \leftarrow b \cdot t_1 \\
3 & \quad t_1 \leftarrow t_1 + t_1 \\
4 & \quad t_2 \leftarrow t_1 + t_1 \\
5 & \quad t_1 \leftarrow t_1 + t_2 \\
6 & \quad t_2 \leftarrow 6 \\
7 & \quad t_1 \leftarrow t_1 - t_2 \\
8 & \quad t_2 \leftarrow a \cdot x_{P_3} \\
9 & \quad t_1 \leftarrow t_2 - t_1 \\
10 & \quad a' \leftarrow t_1 \cdot x_{P_3} \\
11 & \quad \text{return } (a', b')
\end{align*}
\]
Algorithm 37: Evaluating a 3-isogeny at a point

function eval_3_iso
    Input: \((x_Q, y_Q)\) and \(x_{P_3}\), where \(P \in E_{a,b}\), and \(P_3\) has exact order 3 on \(E_{a,b}\)
    Output: \((x_{Q'}, y_{Q'})\) corresponding to \(Q' \in E_{a',b'}\), where \(E_{a',b'}\) is the curve 3-isogenous to \(E_{a,b}\) output from curve_3_iso

\[
\begin{align*}
1 & \quad t_1 \leftarrow x_Q^2 \\
2 & \quad t_1 \leftarrow t_1 \cdot x_{P_3} \\
3 & \quad t_2 \leftarrow x_{P_3}^2 \\
4 & \quad t_2 \leftarrow x_Q \cdot t_2 \\
5 & \quad t_3 \leftarrow t_2 + t_2 \\
6 & \quad t_2 \leftarrow t_2 + t_3 \\
7 & \quad t_1 \leftarrow t_1 - t_2 \\
8 & \quad t_1 \leftarrow t_1 + x_Q \\
9 & \quad t_1 \leftarrow t_1 + x_{P_3} \\
10 & \quad t_2 \leftarrow x_Q - x_{P_3} \\
11 & \quad t_2 \leftarrow t_2^{-1} \\
12 & \quad t_3 \leftarrow t_2^2 \\
13 & \quad t_2 \leftarrow t_2 \cdot t_3 \\
14 & \quad t_4 \leftarrow x_Q \cdot x_{P_3} \\
15 & \quad t_4 \leftarrow t_4 - 1 \\
16 & \quad t_1 \leftarrow t_4 \cdot t_1 \\
17 & \quad t_1 \leftarrow t_1 \cdot t_2 \\
18 & \quad t_2 \leftarrow t_4^2 \\
19 & \quad t_2 \leftarrow t_2 \cdot t_3 \\
20 & \quad x_{Q'} \leftarrow x_Q \cdot t_2 \\
21 & \quad y_{Q'} \leftarrow y_Q \cdot t_1 \\
22 & \quad return (x_{Q'}, y_{Q'})
\end{align*}
\]
Algorithm 38: Computing and evaluating a $2^e$-isogeny, simple version

function iso_2_e

Static parameters: Integer $e_2$ from the public parameters

Input: Constants $(a, b)$ corresponding to a curve $E_{a,b}$, $(x_S, y_S)$ where $S$ has exact order $2^{e_2}$ 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 $E_{a',b'}$

1. $(a', b') \leftarrow (a, b)$
2. $e'_2 \leftarrow e_2$
3. if $e'_2$ is odd then
4. \hspace{1em} $(x_T, y_T) \leftarrow \text{xDBLe}((x_S, y_S), (a', b'), e'_2 - 1)$ \hspace{1em} \hfill // Alg. 26
5. \hspace{1em} $(a', b') \leftarrow \text{curve}_2\_\text{iso}(x_T, b')$ \hspace{1em} \hfill // Alg. 32
6. \hspace{1em} $(x_S, y_S) \leftarrow \text{eval}_2\_\text{iso}((x_S, y_S), x_T)$ \hspace{1em} \hfill // Alg. 33
7. \hspace{1em} for $(x_j, y_j)$ in optional input do
8. \hspace{1em} \hspace{1em} $(x'_j, y'_j) \leftarrow \text{eval}_2\_\text{iso}((x_j, y_j), x_T)$ \hspace{1em} \hfill // Alg. 33
9. \hspace{1em} \hspace{1em} $e'_2 \leftarrow e'_2 - 1$
10. for $e = e'_2 - 2$ downto $0$ by $-2$ do
11. \hspace{1em} $(x_T, y_T) \leftarrow \text{xDBLe}((x_S, y_S), (a', b'), e)$ \hspace{1em} \hfill // Alg. 26
12. \hspace{1em} $(a', b') \leftarrow \text{curve}_4\_\text{iso}(x_T, b')$ \hspace{1em} \hfill // Alg. 34
13. \hspace{1em} $(x_S, y_S) \leftarrow \text{eval}_4\_\text{iso}((x_S, y_S), x_T)$ \hspace{1em} \hfill // Alg. 35
14. \hspace{1em} for $(x_j, y_j)$ in optional input do
15. \hspace{1em} \hspace{1em} $(x'_j, y'_j) \leftarrow \text{eval}_4\_\text{iso}((x_j, y_j), x_T)$ \hspace{1em} \hfill // Alg. 35
16. return $(a', b'), [ (x'_1, y'_1), ..., (x'_n, y'_n) ]$
Algorithm 39: Computing and evaluating a $3^e$-isogeny, simple version

function $iso_{3 \cdot e}$

- **Static parameters:** Integer $e_3$ from the public parameters
- **Input:** Constants $(a, b)$ corresponding to a curve $E_{a,b}$, $(x_S, y_S)$ where $S$ has exact order $3^{e_3}$ on $E_{a,b}$
- **Optional input:** $\{(x_1, y_1), \ldots, (x_n, y_n)\}$ on $E_{a,b}$
- **Output:** $(a', b')$ corresponding to the curve $E'_{a', b'} = E/\langle S \rangle$
- **Optional output:** $\{(x'_1, y'_1), \ldots, (x'_n, y'_n)\}$ on $E'_{a', b'}$

1. $(a', b') \leftarrow (a, b)$
2. for $e = e_3 - 1$ downto $0$ by $-1$ do
3.   $(x_T, y_T) \leftarrow xTPL_e ((x_S, y_S), (a', b'), e)$ // Alg. 29
4.   $(a', b') \leftarrow \text{curve}_3\_\text{iso} (x_T, y_T)$ // Alg. 36
5.   $(x_S, y_S) \leftarrow \text{eval}_3\_\text{iso} ((x_S, y_S), x_T)$ // Alg. 37
6.   for $(x_j, y_j)$ in optional input do
7.     $(x'_j, y'_j) \leftarrow \text{eval}_3\_\text{iso} ((x_j, y_j), x_T)$ // Alg. 37
8. return $(a', b'), [(x'_1, y'_1), \ldots, (x'_n, y'_n)]$

Algorithm 40: Recovering the $x$-coordinate of $R$

function $\text{get}_{\text{-xR}}$

- **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. $(x_R, y_R) \leftarrow \text{xADD} ((x_P, y_P), (x_Q, -y_Q), (a, b))$ // Alg. 27
2. return $x_R$
Algorithm 41: Recovering the y-coordinates of $P$ and $Q$, and the Montgomery curve coefficient $a$

```plaintext
function get_yP_yQ_A_B
  Input: $pk = (x_P, x_Q, x_R)$  // Encoded as in §1.2.8
  Output: $(y_P, y_Q, a, b)$  // Alg. 10

1. $a \leftarrow$ get_A($x_P, x_Q, x_R$)
2. $b \leftarrow 1$
3. $t_1 \leftarrow x_P^2$
4. $t_2 \leftarrow x_P \cdot t_1$
5. $t_1 \leftarrow a \cdot t_1$
6. $t_1 \leftarrow t_2 + t_1$
7. $t_2 \leftarrow t_1 + x_P$
8. $y_P \leftarrow \sqrt{t_1}$
9. $t_1 \leftarrow x_Q^2$
10. $t_2 \leftarrow x_0 \cdot t_1$
11. $t_1 \leftarrow t_1 + x_Q$
12. $t_1 \leftarrow t_2 + t_1$
13. $t_1 \leftarrow t_1 + x_Q$
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. 27
16. if $x_T \neq x_R$ then
17.    $y_Q \leftarrow -y_Q$
18. return $(y_P, y_Q, a, b)$
```

Algorithm 42: Computing public keys in the 2-torsion

```plaintext
function isogeny2
  Input: Secret key $sk_2 \in \mathbb{Z}$ (see §1.2.6) and public parameters
         $\{e_2, e_3, p, (x_{P2}, y_{P2}), (x_{Q2}, y_{Q2}), (x_{P3}, y_{P3}), (x_{Q3}, y_{Q3})\}$ (see §1.6)
  Output: Public key $pk_2 = (x_{P3}', x_{Q3}', x_{R3}')$ equivalent to the output of Step 4 of isogeny
         (see §1.3.5)

1. $(a, b) \leftarrow (6, 1)$
2. $(x_S, y_S) \leftarrow$ double_and_add($sk_2, (x_{Q2}, y_{Q2}), (a, b)$)  // Alg. 30
3. $(x_S, y_S) \leftarrow$ xADD($((x_{P2}, y_{P2}), (x_S, y_S), (a, b)$)  // Alg. 27
4. $\left((a', b'), (x_{P3}', y_{P3}'), (x_{Q3}', y_{Q3}')\right) \leftarrow$ isogeny2($\left((a, b), (x_S, y_S), (x_{P3}, y_{P3}), (x_{Q3}, y_{Q3})\right)$)  // Alg. 38
5. $x_{R3}' \leftarrow$ get_xR($\left((a', b'), (x_{P3}', y_{P3}'), (x_{Q3}', y_{Q3}')\right)$)  // Alg. 40
6. return $pk_2 = (x_{P3}', x_{Q3}', x_{R3}')$  // Encoded as in §1.2.9
```
Algorithm 43: Computing public keys in the 3-torsion

function isogen3

Input: Secret key sk₃ ∈ ℤ (see §1.2.6) and public parameters
{e₂, e₃, p, (xₑ₂, yₑ₂), (xₑ₃, yₑ₃), (xₚ, yₚ), (xₚ₃, yₚ₃), (xₚ₅, yₚ₅)} (see §1.6)

Output: Public key pk₃ = (xⱼ₂', xⱼ₃₂', xⱼ₃₂) equivalent to the output of Step 4 of isogen₅
(see §1.3.5)

1. (a, b) ← (6, 1)
2. (x₅, y₅) ← double_and_add(sk₃, (xₑ₃, yₑ₃), (a, b)) // Alg. 30
3. (x₅, y₅) ← xADD((xₑ₅, yₑ₅), (x₅, y₅), (a, b)) // Alg. 27
4. (a', b'), (xⱼ₂', yⱼ₂'), (xⱼ₃₂', yⱼ₃₂') ← iso_3_e((a, b), (x₅, y₅), (xₑ₂, yₑ₂), (xₑ₃, yₑ₃)) // Alg. 39
5. xⱼ₃₂ ← get_xR((a', b'), (xⱼ₂', yⱼ₂'), (xⱼ₃₂', yⱼ₃₂')) // Alg. 40
6. return pk₃ = (xⱼ₂', xⱼ₃₂', xⱼ₃₂) // Encoded as in §1.2.9

Algorithm 44: Establishing shared keys in the 2-torsion

function isoex₂

Input: Secret key sk₂ ∈ ℤ (see §1.2.6), public key pk₃ = (xⱼ₂', xⱼ₃₂', xⱼ₃₂') ∈ (𝔽ₚ²)³ (see §1.2.9),
and parameter e₂ (see §1.6)

Output: A j-invariant j₂ equivalent to the output of Step 4 of isogen₅ (see §1.3.6)

1. (yⱼ₂', yⱼ₃₂', a, b) ← get_yP_yQ_A_B(xⱼ₂', xⱼ₃₂', xⱼ₃₂') // Alg. 41
2. (x₅, y₅) ← mult_double_add(sk₂, (xⱼ₂', yⱼ₂'), (a, b)) // Alg. 30
3. (x₅, y₅) ← xADD((xⱼ₃₂', yⱼ₃₂'), (x₅, y₅), (a, b)) // Alg. 27
4. (a, b) ← 2_e_iso ((a, b), (x₅, y₅)) // Alg. 38
5. j₂ = j_inv(a) // Alg. 31
6. return j₂ // Encoded as in §1.2.8

73
Algorithm 45: Establishing shared keys in the 3-torsion

function \texttt{isoex}_3

\textbf{Input:} Secret key \( sk_3 \in \mathbb{Z} \) (see §1.2.6), public key \( pk_2 = (x_3', x_3', x_3') \in (\mathbb{F}_{p^2})^3 \) (see §1.2.9), and parameter \( e_3 \) (see §1.6)

\textbf{Output:} A \( j \)-invariant \( j_3 \) equivalent to the output of Step 4 of \texttt{isogen}_\ell (see §1.3.6)

1. \( (y_{P3}', y_{Q3}', a, b) \leftarrow \texttt{get}_\_yP\_yQ\_A\_B(x_{P3}', x_{Q3}', x_{R3}') \) // Alg. 41
2. \( (x_S, y_S) \leftarrow \texttt{mult\_double\_add}(sk_3, (x_{Q3}', y_{Q3}'), (a, b)) \) // Alg. 30
3. \( (x_S, y_S) \leftarrow \texttt{xADD}((x_{P3}', y_{P3}'), (x_S, y_S), (a, b)) \) // Alg. 27
4. \( (a, b) \leftarrow \texttt{3\_e\_iso}((a, b), (x_S, y_S)) \) // Alg. 39
5. \( j_3 = j\_\text{inv}(a) \) // Alg. 31
6. return \( j_3 \) // Encoded as in §1.2.8
Appendix C

Computing optimized strategies for fast isogeny computation

Algorithms 19 and 20 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 \texttt{xDBL} operations, \(q_4\) the cost of computation and evaluation of a 4-isogeny, i.e. of the functions \texttt{4_iso_curve} and \texttt{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 \texttt{xTPL} operation, and \(q_3\) the cost of computation and evaluation of a 3-isogeny, i.e. of the functions \texttt{3_iso_curve} and \texttt{3_iso_eval}. We denote the respective strategies by \(S_4\) and \(S_3\), respectively.

**Algorithm 46:** Computing optimized strategy

```plaintext
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 \( b \leftarrow \arg\min_{0 < b < i} (C[i - b] + C[b] + bp + (i - b)q) \)
    5 Set \( S[i] \leftarrow b \cdot S[i - b] \cdot S[b] \)
    6 Set \( C[i] \leftarrow C[i - b] + C[b] + bp + (i - b)q \)
  7 return \( S[n + 1] \)
```

75
C.1 Strategies for SIKEp434

C.1.1 2-torsion

\[ \begin{align*}
    n_4 &= 107 \\
    p_4 &= 5633 \\
    q_4 &= 5461 \\
    S_4 &= (48, 28, 16, 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 3, 7, 4, 2, 1, 1, 2, 1, 3, 2, 1, 1, 1, 5, 4, 2, 1, 1, 2, 1, 1, 2, 1, 1, 1, 2, 1, 1, 9, 5, 3, 2, 1, 1, 1, 2, 1, 1, 1, 4, 2, 1, 1, 2, 1, 1)
\end{align*} \]

C.1.2 3-torsion

\[ \begin{align*}
    n_3 &= 136 \\
    p_3 &= 5322 \\
    q_3 &= 5282 \\
    S_3 &= (66, 33, 17, 9, 5, 3, 2, 1, 1, 1, 1, 2, 1, 1, 1, 4, 2, 1, 1, 2, 1, 1, 8, 4, 2, 1, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 8, 4, 2, 1, 1, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 1, 16, 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 1, 16, 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1)
\end{align*} \]

C.2 Strategies for SIKEp503

C.2.1 2-torsion

\[ \begin{align*}
    n_4 &= 124 \\
    p_4 &= 7490 \\
    q_4 &= 7278 \\
    S_4 &= (61, 32, 16, 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 16, 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 13, 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 5, 4, 2, 1, 1, 2, 1, 1, 2, 1, 1, 1)
\end{align*} \]
C.3.2 3-torsion

\[ n_3 = 158 \]
\[ p_3 = 7189 \]
\[ q_3 = 7051 \]
\[ S_3 = (71, 38, 21, 13, 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 5, 4, 2, 1, 1, 2, 1, 1, 2, 1, 1, 9, 5, 3, 2, 1, 1, 1, 2, 1, 1, 1, 4, 2, 1, 1, 2, 1, 1, 17, 9, 5, 3, 2, 1, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 2, 1, 1, 33, 17, 9, 5, 3, 2, 1, 1, 1, 1, 1, 4, 2, 1, 1, 2, 1, 1, 8, 4, 2, 1, 1, 2, 1, 1, 16, 8, 4, 2, 1, 1, 1, 2, 1, 1, 4, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1) \]

C.3 Strategies for SIKEp610

C.3.1 2-torsion

\[ n_4 = 151 \]
\[ p_4 = 10370 \]
\[ q_4 = 10096 \]
\[ S_4 = (67, 37, 21, 12, 7, 4, 2, 1, 1, 2, 1, 1, 3, 2, 1, 1, 1, 5, 3, 2, 1, 1, 1, 2, 1, 1, 1, 9, 5, 3, 2, 1, 1, 1, 2, 1, 1, 1, 4, 2, 1, 1, 1, 2, 1, 1, 6, 9, 5, 3, 2, 1, 1, 1, 2, 1, 1, 1, 4, 2, 1, 1, 2, 1, 1, 33, 16, 8, 5, 2, 1, 1, 2, 1, 1, 1, 4, 2, 1, 1, 2, 1, 1, 8, 4, 2, 1, 1, 2, 1, 1, 16, 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 7, 4, 2, 1, 1, 2, 1, 1, 2, 1, 1, 2, 1, 1, 1, 2, 1, 1, 2, 1, 1) \]

C.3.2 3-torsion

\[ n_3 = 191 \]
\[ p_3 = 10084 \]
\[ q_3 = 9794 \]
\[ S_3 = (86, 48, 27, 15, 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 7, 4, 2, 1, 1, 2, 1, 1, 3, 2, 1, 1, 1, 2, 1, 1, 1, 5, 3, 2, 1, 1, 1, 2, 1, 1, 1, 21, 12, 7, 4, 2, 1, 1, 2, 1, 1, 3, 2, 1, 1, 1, 5, 3, 2, 1, 1, 1, 2, 1, 1, 1, 9, 5, 3, 2, 1, 1, 1, 2, 1, 1, 1, 4, 2, 1, 1, 1, 2, 1, 1, 3, 2, 1, 1, 1, 5, 3, 2, 1, 1, 1, 2, 1, 1, 1, 9, 5, 3, 2, 1, 1, 1, 2, 1, 1, 1, 4, 2, 1, 1, 1, 2, 1) \]
C.4 Strategies for SIKEp751

C.4.1 2-torsion

\[ n_4 = 185 \]
\[ p_4 = 14166 \]
\[ q_4 = 13810 \]
\[ S_4 = \{ 80, 48, 27, 15, 8, 4, 2, 1, 1, 1, 4, 2, 1, 1, 1, 7, 4, 2, 1, 1, 2, 1, 1, 3, 2, 1, 1, 1, \\
12, 7, 4, 2, 1, 1, 2, 1, 1, 3, 2, 1, 1, 1, 5, 3, 2, 1, 1, 1, 2, 1, 1, 9, 5, 3, 2, 1, 1, 1, 2, 1, 1, 4, 2, 1, \\
1, 1, 2, 1, 1, 33, 20, 12, 7, 4, 2, 1, 1, 2, 1, 3, 2, 1, 1, 1, 5, 3, 2, 1, 1, 1, 2, 1, 1, 1, \\
8, 5, 3, 2, 1, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 16, 8, 4, 2, 1, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, \\
1, 1, 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1 \} \]

C.4.2 3-torsion

\[ n_3 = 238 \]
\[ p_3 = 13898 \]
\[ q_3 = 13409 \]
\[ S_3 = \{ 112, 63, 32, 16, 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, \\
1, 1, 16, 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, \\
31, 16, 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, \\
1, 15, 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 7, 4, 2, 1, 1, 2, 1, 1, 3, 2, 1, 1, 1, 49, 31, 16, 8, 4, 2, 1, 1, 2, \\
1, 1, 4, 2, 1, 1, 2, 1, 1, 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1, 5, 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, \\
1, 1, 2, 1, 1, 7, 4, 2, 1, 1, 2, 1, 1, 3, 2, 1, 1, 1, 21, 12, 8, 4, 2, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, \\
1, 1, 5, 3, 2, 1, 1, 1, 2, 1, 1, 1, 9, 5, 3, 2, 1, 1, 1, 2, 1, 1, 4, 2, 1, 1, 2, 1, 1 \} \]
Appendix D

Explicit algorithms for compressed SIKE: Optimized implementation

The major algorithms underlying the current key compression techniques [42] are listed next. In this section we assume that the public parameters include the following data.

- The torsion basis generation algorithms 48 and 54 use elligator-like techniques to get points on the curve. This involves computing values of the form $v = 1/(1 + u \cdot r^2) \in \mathbb{F}_{p^2}$. In order to avoid multiple inversions, a precomputed table $T$ is employed in the optimized implementation. Experimentally, less than 20 elements are enough for storage.

- Optimal traversal paths $\{\text{path}_k = (s_1, \ldots, s_{e_k/2-1}) \in (\mathbb{N}^+)^{e_k/2-1}\}$ where $k \in \{2, 3\}$ for solving discrete logarithms via Pohlig-Hellman. Optimal paths for discrete logarithms can be generated by $\text{compute\_strategy}$ (Algorithm 46), the same that allows for computing smooth degree isogenies in complexity $O(e_{\ell} \log e_{\ell})$. In this context the input parameters $p, q$ will consist of the costs of powering an element of $\mathbb{F}_{p^2}$ to $\ell$ (or $\ell^w$ in the general case) and of multiplying two elements in $\mathbb{F}_{p^2}$, respectively.

- When computing a discrete logarithm of the form $\log_g(r)$ with $r = g^d$ and $d = (d_{k-1} \cdots d_1 d_0)e_\ell$, the base $g \in \mathbb{F}_{p^2}$ is fixed and can be included in the public parameters. In particular, precomputed tables are employed to speed up computations. The compression algorithms uses tables $T_1[u][d] := g^{-e_{\ell} u \cdot d}$ and $T_2[0][d_i] := g^{-d_i}$ and $T_2[u][d_i] := g^{-d_i \cdot e_{\mod w(u-1)}}$ for $0 < u < [e/w]$ and 0 as described in [42] to speed up computations. The parameter $w$ can be seen as a tradeoff between speed vs. storage. Larger $w$ implies smaller trees to be traversed but larger discrete logarithm instances at the leaves.
**Algorithm 47:** $x$-only tripling $k$ times on the Montgomery curve $E_A : y^2 = x^3 + Ax^2 + x$

```plaintext
function xTPLLe_fast
  Input: $P = (x, z) \in E_A$, the coefficient $A_2 = A/2 \in \mathbb{F}_p$ and the number of triplings $k$.
  Output: $[3^k]P = (x', z')$

1 for $j = 1$ to $k$
  2 $t_1 \leftarrow x^2$
  3 $t_2 \leftarrow z^2$
  4 $t_3 \leftarrow t_1 + t_2$
  5 $t_4 \leftarrow A_2 \cdot ((x + z)^2 - t_3) + t_3$
  6 $t_3 \leftarrow (t_1 - t_2)^2$
  7 $t_1 \leftarrow (t_1 \cdot t_4 - t_3)^2$
  8 $t_2 \leftarrow (t_2 \cdot t_4 - t_3)^2$
  9 $x' \leftarrow x \cdot t_2$
10 $z' \leftarrow z \cdot t_1$
11 return $(x', z')$
```
Algorithm 48: Entangled basis generation for $E[2^{e_2}](\mathbb{F}_{p^2})$: $y^2 = x^3 + Ax^2 + x$

function `get_2_torsion_entangled_basis_compression`

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

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

1. $z \leftarrow a^2 + b^2$
2. $s \leftarrow z^{(p+1)/4}$
3. $T \leftarrow (s^2 \equiv z) \ T_1 : T_2$ select proper table by testing quadraticity of $A$
4. repeat
5. lookup next entry $(r, v)$ from $T$
6. $x \leftarrow -A \cdot v$
7. $t \leftarrow x \cdot (x^2 + A \cdot x + 1)$
8. test quadraticity of $t = c + di$
9. $z \leftarrow c^2 + d^2,$
10. $s \leftarrow z^{(p+1)/4}$
11. until $s^2 = z$
12. compute $y \leftarrow \sqrt{x^3 + A \cdot x^2 + x}$
13. $z \leftarrow (c + s)/2$
14. $\alpha \leftarrow z^{(p+1)/4}$
15. $\beta \leftarrow d \cdot (2\alpha)^{-1}$
16. $y \leftarrow (\alpha^2 \equiv z) \alpha + \beta i : -\beta - \alpha i$
17. return $S_1 \leftarrow (x, y), \ S_2 \leftarrow (ur^2x, u_0ry), \ bit \leftarrow (T \equiv T_1) \ 1 : 0, \ r$
Algorithm 49: Entangled basis generation with shared Elligator for $E[2^{e^2}](\mathbb{F}_p^2) : y^2 = x^3 + Ax^2 + x$

function get_2_torsion_entangled_basis_decompression
  Input: $A = a + bi \in \mathbb{F}_p^2$, a bit $bit$ indicating $A$’s quadraticity, a counter $r \in \mathbb{F}_p$ and the public parameters $u_0 \in \mathbb{F}_p^2 : u = u_0^2 \in \mathbb{F}_p^2 \setminus \mathbb{F}_p$; tables $T_1, T_2$ of pairs $(r \in \mathbb{F}_p, v = 1/(1 + ur^2) \in \mathbb{F}_p^2)$ of QNR and QR.
  Output: $\{S_1, S_2\}$ such that $\langle [3e]S_1, [3e]S_2 \rangle = E[2^{e^2}](\mathbb{F}_p^2)$

1. $T \leftarrow (bit = 1)$ $T_1 : T_2$ select proper table according to $A$’s quadraticity
2. lookup entry $v$ corresponding to $r$ on $T$
3. $x \leftarrow -A \cdot v$
4. $t \leftarrow x \cdot (x^2 + A \cdot x + 1)$
5. test quadraticity of $t = c + di$
6. $z \leftarrow c^2 + d^2$
7. $s \leftarrow z^{(p+1)/4}$
8. if $s^2 \neq z$ then
   - Abort: invalid input parameters ($bit, r$) received
9. compute $y \leftarrow \sqrt{x^3 + A \cdot x^2 + x}$
10. $z \leftarrow (c + s)/2$,
11. $\alpha \leftarrow z^{(p+1)/4}$,
12. $\beta \leftarrow d \cdot (2\alpha)^{-1}$
13. $y \leftarrow (\alpha^2 \div z) \alpha + \beta i : -\beta - \alpha i$
14. return $S_1 \leftarrow (x, y)$, $S_2 \leftarrow (ur^2 x, u_0 ry)$

Algorithm 50: $x$-only doubling ($k$ times) on a Montgomery curve $E : y^2 = x^3 + Ax^2 + x$

function Double
  Input: Curve coefficient in the form $A24 = (A + 2)/4$, point $(x, z)$ and integer $k$
  Output: Point $(x', z') = [2^k](x, z) \in E.$
1. for $j = 1$ to $k$ do
2.   $a \leftarrow x + z$
3.   $b \leftarrow x - z$
4.   $aa \leftarrow a^2$
5.   $bb \leftarrow b^2$
6.   $c \leftarrow aa - bb$
7.   $x \leftarrow aa \cdot bb$
8.   $z \leftarrow c(bb + A24 \cdot c)$
9. return $x, z$
Algorithm 51: $xz$-only construction of a point of order $3^{e_3}$ in the Montgomery curve $E : y^2 = x^3 + Ax^2 + x$ from counter $r$

function BasePoint3n

Input: Curve coefficient $A$, $r \in \mathbb{Z}_{256}$, nd public table $T$ of elligator values $v = 1/(1 + ur^2) \in \mathbb{F}_{p^2}$

Output: Chosen table entry $r$, point $(x, z)$ of order $3^{e_3}$ and $(t, w) = [3^{e_3-1}](x, y)$

repeat

2 \hspace{1em} r ← r + 1

3 \hspace{1em} v ← T[r]

4 \hspace{1em} x ← −Av

5 \hspace{1em} yy ← x((x + A)x + 1)

6 \hspace{1em} N ← norm(yy) \hspace{1em} // NB: norm(a + bi) = a^2 + b^2

7 \hspace{1em} z ← N^{(p+1)/4}

8 \hspace{1em} if $z^2 \neq N$ then

9 \hspace{2em} x ← −x − A

10 \hspace{1em} x, z = DOUBLE(x, 1, (A + 2)/4, e_2) \hspace{1em} // Alg. 50

11 \hspace{1em} t, w ← xTPlle_fast(A/2, x, z, e_3 − 1) \hspace{1em} // Alg. 47

12 \hspace{1em} until $w \neq 0$

13 \hspace{1em} return $(r, x, z, t, w)$
Algorithm 52: Deterministic $xz$-only construction of a point of order $3^{e_3}$ in the Montgomery curve $E: y^2 = x^3 + Ax^2 + x$ from $r$

function BasePoint3n_decompression

\[\textbf{Input:}\] Curve coefficient $A$, $r \in \mathbb{Z}$ and public table $T$ of elligator values $v = 1/(1 + ur^2) \in \mathbb{F}_{p^2}$

\[\textbf{Output:}\] Point $(x, z)$ of order $3^{e_3}$

1. $r \leftarrow r + 1$
2. $v \leftarrow T[r]$
3. $x \leftarrow -Av$
4. $yy \leftarrow x((x + A)x + 1)$
5. $N \leftarrow \text{norm}(yy)$ \hspace{1cm} // NB: $\text{norm}(a + bi) = a^2 + b^2$
6. $z \leftarrow N^{(p+1)/4}$
7. if $z^2 \neq N$ then
   8. \hspace{1cm} $x \leftarrow -x - A$
9. $x, z = DOUBLE(x, 1, (A + 2)/4, e_2)$ \hspace{1cm} // Alg. 50
10. return $(x, z)$

Algorithm 53: Given a $xz$-only representation on a Montgomery curve $E$, compute the affine representation.

function CompleteMPoint

\[\textbf{Input:}\] Montgomery curve coefficient $A$, point $P = (x, z) \in E$

\[\textbf{Output:}\] $(x', y', z')$, the affine representation of $P$

1. if $z \neq 0$ then
   2. $xz \leftarrow x \cdot z$
   3. $ss \leftarrow (x + i \cdot z)(x - i \cdot z)$
   4. $rr \leftarrow xz(A \cdot xz + ss)$
   5. $yz \leftarrow \sqrt{rr}$
   6. $invz \leftarrow z^{-1}$
   7. $x' \leftarrow x \cdot invz$
   8. $y' \leftarrow yz \cdot invz^2$
   9. $z' \leftarrow 1$
10. else
   11. \hspace{1cm} $x' \leftarrow 0; y' \leftarrow 1; z' \leftarrow 0$
12. return $x', y', z'$
Algorithm 54: Generating a basis for $E[3^e](\mathbb{F}_{p^2})$: $y^2 = x^3 + Ax^2 + x$

```plaintext
function BuildOrdinaryE3nBasis
    \textbf{Input:} Montgomery curve coefficient $A$
    \textbf{Output:} $(x_1, y_1), (x_2, y_2)$: a basis for $E[3^e]$ in affine representation and the elligator counters $(r_1, r_2) \in \mathbb{Z}_{256}^2$
    \begin{align*}
    &1 \quad r \leftarrow 0 \\
    &2 \quad r, x_1, z_1, t_1, w_1 \leftarrow BasePoint3n(A, r) \quad \text{\textit{Alg. 51}} \\
    &3 \quad r_1 \leftarrow r \text{ \textbf{repeat}} \\
    &4 \quad r, x_2, z_2, t_2, w_2 \leftarrow BasePoint3n(A, r) \quad \text{\textit{Alg. 51}} \\
    &5 \quad \textbf{until } t_2 \cdot w_1 = t_1 \cdot w_2 \\
    &6 \quad r_2 \leftarrow r \\
    &7 \quad x_1, y_1 \leftarrow CompleteMPoint(A, x_1, z_1) \quad \text{\textit{Alg. 53}} \\
    &8 \quad x_2, y_2 \leftarrow CompleteMPoint(A, x_2, z_2) \quad \text{\textit{Alg. 53}} \\
    &9 \quad \textbf{return} (x_1, y_1), (x_2, y_2), r_1, r_2
    \end{align*}
```

Algorithm 55: Deterministically generating a basis for $E[3^e](\mathbb{F}_{p^2})$: $y^2 = x^3 + Ax^2 + x$ from $A$ and elligator counters $r_1, r_2$

```plaintext
function BuildOrdinaryE3nBasis_decompression
    \textbf{Input:} Montgomery curve coefficient $A$ and elligator counters $(r_1, r_2) \in \mathbb{Z}_{256}^2$
    \textbf{Output:} $(x_1, y_1), (x_2, y_2)$: a basis for $E[3^e]$
    \begin{align*}
    &1 \quad x_1, z_1, t_1, w_1 \leftarrow BasePoint3n\_decompression(A, r_1) \quad \text{\textit{Alg. 52}} \\
    &2 \quad x_2, z_2, t_2, w_2 \leftarrow BasePoint3n\_decompression(A, r_2) \quad \text{\textit{Alg. 52}} \\
    &3 \quad x_1, y_1 \leftarrow CompleteMPoint(A, x_1, z_1) \quad \text{\textit{Alg. 53}} \\
    &4 \quad x_2, y_2 \leftarrow CompleteMPoint(A, x_2, z_2) \quad \text{\textit{Alg. 53}} \\
    &5 \quad \textbf{return} (x_1, y_1), (x_2, y_2)
    \end{align*}
```
Algorithm 56: Tate2($P, [Q_j])$: reduced Tate pairing of order $r = 2^{e_2}$

function Tate_pairings_2_torsion

Input: Weierstrass Curve $E: y^2 = x^3 + ax + b$, point $P = [X_P : Y_P : Z_P]$ on $E$ of order $2^{e_2}$ and $t$ points $Q_j = [X_{Q_j} : Y_{Q_j} : Z_{Q_j}]$ on $E, Z_{Q_j} \in \{0, 1\}$

Output: List of $t$ values $e_2^2 e_2 e_2 e_2(P, Q_j)$

1. $X \leftarrow X_P; Y \leftarrow Y_P; Z \leftarrow Z_P; T \leftarrow Z^2; U \leftarrow a \cdot T^2$
2. for $j \leftarrow 0$ to $t - 1$ do
3.   $f_j \leftarrow 1$
4.     $h_j \leftarrow T \cdot X_{Q_j} - X$
5. for $k \leftarrow 0$ to $e_2 - 1$ do
6.     point doubling and line function construction:
7.     $X_2 \leftarrow X^2; Y_2 \leftarrow Y^2; W \leftarrow 2Y_2; W_2 \leftarrow W^2$
8.     $M \leftarrow 3X_2 + U; S \leftarrow (X + W)^2 - X_2 - W_2$
9.     $X' \leftarrow M^2 - 2S; Y' \leftarrow M \cdot (S - X') - 2W_2$
10.    $Z' \leftarrow (Y + Z)^2 - Y_2 - T; T' \leftarrow (Z)^2$
11.    $U' \leftarrow 4W_2 \cdot U; L \leftarrow Z' \cdot T$
12.     if $Z' = 0$ then
13.         exception for points in $[2]E$
14.         $X' \leftarrow 1$
15.         $Y' \leftarrow 1$
16.     line function evaluation and accumulation:
17.     for $j \leftarrow 0$ to $t - 1$ do
18.         if $Z' \neq 0$ then
19.             $g \leftarrow M \cdot h_j + W - L \cdot Y_{Q_j}$
20.             $h_j \leftarrow T' \cdot X_{Q_j} - X'$
21.             $g \leftarrow g \cdot \bar{h}_j$
22.         else
23.             exception for points in $[2]E$
24.             $g \leftarrow h_j$
25.             $f_j \leftarrow f_j^2$
26.             $f_j \leftarrow f_j \cdot g$
27.       $X \leftarrow X'; Y \leftarrow Y'; Z \leftarrow Z'; T \leftarrow T'; U \leftarrow U'$
28. a dedicated final exponentiation should be used next:
29. return $[(Z_{Q_j} \neq 0) f_j^{(p^2-1)/r} : 1 | j = 0 \ldots t - 1]$
Algorithm 57: Tate3\((P, [Q_j])\): reduced Tate pairing of order \(r = 3^e\)

```
function Tate_pairings_3_torsion

  Input: Weierstrass Curve \(E: y^2 = x^3 + ax + b\), point \(P = [X_P : Y_P : Z_P]\) on \(E\) of order \(3^e\) and \(t\) points \(Q_j = [X_{Q_j} : Y_{Q_j} : Z_{Q_j}]\) on \(E\), \(Z_{Q_j} \in \{0, 1\}\)

  Output: List of \(t\) values \(e_{3^e}(P, Q_j)\)

1. \(X \leftarrow X_P; Y \leftarrow Y_P; Z \leftarrow Z_P; T \leftarrow Z^2; U \leftarrow a \cdot T^2\)
2. for \(j \leftarrow 0\) to \(t - 1\) do
   3. \(f_j \leftarrow 1;\)
   4. \(h_j \leftarrow T \cdot X_{Q_j} - X\)
3. for \(k \leftarrow 0\) to \(e_{3} - 1\) do
   4. point tripling and parabola function construction:
   5. \(X_2 \leftarrow X^2; Y_2 \leftarrow Y^2; Y_4 \leftarrow Y_2^2\)
   6. \(M \leftarrow 3X_2 + U; M_2 \leftarrow M^2\)
   7. \(D \leftarrow (X + Y_2)^2 - X_2 - Y_4; F \leftarrow 6D - M_2\)
   8. \(F_2 \leftarrow F^2; W \leftarrow 2Y_2; W' \leftarrow 2W; S \leftarrow 16Y_4\)
   9. \(G \leftarrow (M + F)^2 - M_2 - F_2 - S; G' \leftarrow S - G\)
   10. \(H \leftarrow 2F_2; H_2 \leftarrow H^2; H' \leftarrow 4G; F' \leftarrow 2F\)
   11. \(X' \leftarrow (X + H)^2 - X_2 - H_2 - W' \cdot H'\)
   12. \(Y' \leftarrow 2Y \cdot (H' \cdot G' - F' \cdot H)\)
   13. \(Z' \leftarrow (Z + F)^2 - T - F_2\)
   14. \(T' \leftarrow (Z')^2; U' \leftarrow 4H_2 \cdot U\)
   15. \(L \leftarrow ((Y + Z)^2 - Y_2 - T) \cdot T\)
4. if \(Z' = 0\) exception for points in \([3]E\) then
   5. \(X' \leftarrow 1; Y' \leftarrow 1\)
6. parabola function evaluation and accumulation:
7. for \(j \leftarrow 0\) to \(t - 1\) do
   8. \(d \leftarrow W - L \cdot Y_{Q_j}\) if \(Z' \neq 0\) then
     9. \(g \leftarrow (M \cdot h_j + d)(G' \cdot h_j + F' \cdot d)(W' \cdot h_j + F)^{-1}\)
   10. \(h_j \leftarrow T' \cdot X_{Q_j} - X'; g \leftarrow g \cdot \bar{h}_j\)
   11. else
   12. exception for points in \([3]E\)
   13. \(g \leftarrow (M \cdot h_j + d)\)
   14. \(f \leftarrow f_3^3\)
   15. \(f \leftarrow f \cdot g\)
   16. \(X \leftarrow X'; Y \leftarrow Y'; Z \leftarrow Z'; T \leftarrow T'; U \leftarrow U'\)
7. a dedicated final exponentiation should be used next:
8. return \([Z_{Q_j}^2 \neq 0] f_j^{(p^2 - 1)/r} : 1 \mid j = 0 \ldots t - 1\)
```
Algorithm 58: Compute $\log_g(r)$ for a fixed $g \in \mathbb{F}_{p^2}$ using an optimal traversal strategy

function Traverse_w_div_e

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

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

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

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

3 if $z > 1$ then

4 \hspace{1em} $t \leftarrow P[z]$

5 \hspace{1em} $r' \leftarrow r^{\ell w(z-t)}$ \hspace{1em} // go left $w(z-t)$ times

6 \hspace{1em} Traverse_w_div_e($r'$, $j + (z - t), k, t, \ell, w, P, T, d$)

7 \hspace{1em} $r' \leftarrow r' \cdot \prod_{h=k}^{t+k-1} T[j + h][d_h]$ \hspace{1em} // go right $t$ times

8 \hspace{1em} Traverse_w_div_e($r'$, $j, k + t, z - t, \ell, w, P, T, d$)

9 else

10 \hspace{1em} // leaf

11 \hspace{1em} find $t \in \{0, \ldots, \ell^w - 1\}$ such that $r = T[e/w - 1][t]$

12 \hspace{1em} $d_k \leftarrow t$ \hspace{1em} // recover $k$-th digit $d_k$
Algorithm 59: Compute $\log_g(r)$ for a fixed $g \in \mathbb{F}_{p^2}$ using an optimal traversal strategy

**function** Traverse_w_notdiv_e

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

**Output:** $d$: digits (radix $\ell^w$) of $\log_g r_0$

Remark: initial call is Traverse_w_notdiv_e($r_0,0,0,len(P)-1,\ell,w,P,T_1,T_2,d$).

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

1. if $z > 1$ then
2.   $t \leftarrow P[z]$  // $z$ leaves
3.   if $j > 0$ then
4.       $r' \leftarrow r^{P[(z-1)]}$  // go left $w(z-t)$ times
5.       Traverse_w_notdiv_e($r',j+(z-t),k,t,\ell,w,P,T_1,T_2,d$)
6.   else
7.       $r' \leftarrow r^{P[w(z-t-1)]}$  // go left $e \mod w + w(z-t-1)$ times
8. Traverse_w_notdiv_e($r',j+(z-t),k,t,\ell,w,P,T_1,T_2,d$)
9. if $j = 0$ then
10.   $r' \leftarrow r \cdot \prod_{h=k}^{k+t-1} T_1[j+h][d_h]$  // go right $t$ times
11. else
12.   $r' \leftarrow r \cdot \prod_{h=k}^{k+t-1} T_2[j+h][d_h]$
13. Traverse_w_notdiv_e($r',j,k,t,z-t,\ell,w,P,T_1,T_2,d$)
14. else  // leaf
15.   if $j = 0$ and $k = [e/w] - 1$ then
16.     find $0 \leq t < \ell w \mod w$ s.t. $r = T_1[[e/w] - 1][t]$  // recover the $k$-th digit $d_k$
17.   else
18.     find $0 \leq t < \ell^w \mod w$ s.t. $r = T_2[[e/w] - 1][t]$
Algorithm 60: Convert a point on a Montgomery curve $E : y^2 = x^3 + Ax^2 + x$ into the corresponding point on its short Weierstrass form $E_W : y^2 = x^3 + ax + b$.

function PointMonty2Weier
    Input: Point $(x, y, z) \in E$ and $A$
    Output: Affine point $(x', y') \in E_W$
1 if $z = 0$ then
2    $x' \leftarrow 0$; $y' \leftarrow 1$; $z' \leftarrow 0$
3 else
4    $x' \leftarrow x + A/3$
5    $y' \leftarrow y$
6    $z' \leftarrow 1$
7 return $(x', y', z')$

Algorithm 61: Convert a Montgomery curve $E : y^2 = x^3 + Ax^2 + x$ into the corresponding short Weierstrass form $E_W : y^2 = x^3 + ax + b$.

function Monty2Weier
    Input: Montgomery curve coefficient $A$
    Output: Weierstrass coefficients $(a, b)$
1 $a \leftarrow 1 - A^2/3$
2 $b \leftarrow (2 \cdot A^3 - 9 \cdot A)/27$
3 return $(a, b)$
Algorithm 62: Compute 4 reduced Tate pairings simultaneously:

\[ e_2^{(4)}(P, S_1), e_2^{(4)}(P, S_2), e_2^{(4)}(Q, S_1), e_2^{(4)}(Q, S_2) \]

function Tate_4_pairings_2_torsion

\[ \text{Input: Points } P = (x_{P_w}, y_{P_w}), Q = (x_{Q_w}, y_{Q_w}), S_1 = (x_{S_1}, y_{S_1}), S_2 = (x_{S_2}, y_{S_2}) \text{ on } E : y^2 = x^3 + Ax^2 + x \]

\[ \text{Output: Reduced Tate pairing values } (n_1, n_2, n_3, n_4) \in (\mathbb{F}_p')^4 \]

1. \[ a, b \leftarrow \text{Monty2Weier}(A) \quad \text{// Alg. 61} \]
2. \[ (x_{P_w}, y_{P_w}) \leftarrow \text{PointMonty2Weier}(x_{P_w}, y_{P_w}, A) \quad \text{// Alg. 60} \]
3. \[ (x_{Q_w}, y_{Q_w}) \leftarrow \text{PointMonty2Weier}(x_{Q_w}, y_{Q_w}, A) \quad \text{// Alg. 60} \]
4. \[ (x_{S_1w}, y_{S_1w}) \leftarrow \text{PointMonty2Weier}(x_{S_1}, y_{S_1}, A) \quad \text{// Alg. 60} \]
5. \[ (x_{S_2w}, y_{S_2w}) \leftarrow \text{PointMonty2Weier}(x_{S_2}, y_{S_2}, A) \quad \text{// Alg. 60} \]
6. \[ n_1, n_2 \leftarrow \text{Tate_pairings_2_torsion}(x_{P_w}, y_{P_w}, [(x_{S_1w}, y_{S_1w}), (x_{S_2w}, y_{S_2w})], a, 2) \quad \text{// Alg. 56} \]
7. \[ n_3, n_4 \leftarrow \text{Tate_pairings_2_torsion}(x_{Q_w}, y_{Q_w}, [(x_{S_1w}, y_{S_1w}), (x_{S_2w}, y_{S_2w})], a, 2) \quad \text{// Alg. 56} \]
8. return \( (n_1, n_2, n_3, n_4) \in (\mathbb{F}_p')^4 \)

Algorithm 63: Compute 4 reduced Tate pairings simultaneously:

\[ e_3^{(3)}(P, S_1), e_3^{(3)}(P, S_2), e_3^{(3)}(Q, S_1), e_3^{(3)}(Q, S_2) \]

function Tate_4_pairings_3_torsion

\[ \text{Input: Points } P = (x_{P_w}, y_{P_w}), Q = (x_{Q_w}, y_{Q_w}), S_1 = (x_{S_1}, y_{S_1}), S_2 = (x_{S_2}, y_{S_2}) \text{ on } E : y^2 = x^3 + Ax^2 + x \]

\[ \text{Output: Reduced Tate pairing values } (n_1, n_2, n_3, n_4) \in (\mathbb{F}_p')^4 \]

1. \[ a, b \leftarrow \text{Monty2Weier}(A) \quad \text{// Alg. 61} \]
2. \[ (x_{P_w}, y_{P_w}) \leftarrow \text{PointMonty2Weier}(x_{P_w}, y_{P_w}, A) \quad \text{// Alg. 60} \]
3. \[ (x_{Q_w}, y_{Q_w}) \leftarrow \text{PointMonty2Weier}(x_{Q_w}, y_{Q_w}, A) \quad \text{// Alg. 60} \]
4. \[ (x_{S_1w}, y_{S_1w}) \leftarrow \text{PointMonty2Weier}(x_{S_1}, y_{S_1}, A) \quad \text{// Alg. 60} \]
5. \[ (x_{S_2w}, y_{S_2w}) \leftarrow \text{PointMonty2Weier}(x_{S_2}, y_{S_2}, A) \quad \text{// Alg. 60} \]
6. \[ n_1, n_2 \leftarrow \text{Tate_pairings_3_torsion}(x_{P_w}, y_{P_w}, [(x_{S_1w}, y_{S_1w}), (x_{S_2w}, y_{S_2w})], a, 2) \quad \text{// Alg. 57} \]
7. \[ n_3, n_4 \leftarrow \text{Tate_pairings_3_torsion}(x_{Q_w}, y_{Q_w}, [(x_{S_1w}, y_{S_1w}), (x_{S_2w}, y_{S_2w})], a, 2) \quad \text{// Alg. 57} \]
8. return \( (n_1, n_2, n_3, n_4) \in (\mathbb{F}_p')^4 \)
Algorithm 64: Compute the discrete logarithm (optimal Pohlig-Hellman traversal strategy) $d = \log_g(r)$ where $g = e_{C_{k}}(P_k, Q_k)_{\overline{e_k}} \in \mathbb{F}_{p^2}$ and $\overline{k}$ is the complement of the torsion $k \in \{2, 3\}$.

```
function solve_dlog
    Input: Element $r \in \mathbb{F}_{p^2}$, the corresponding torsion $k$ and the following public parameters corresponding to torsion $k$: optimal Pohlig-Hellman traversal path $\overline{\text{path}}_k \in \mathbb{Z}^{\text{plen}_k}$, tables $(T_1)_k, (T_2)_k$ of precomputed values in $\mathbb{F}_{p^2}$, and exponent $w_k$.
    Output: The discrete logarithm $d \in \mathbb{Z}_{\overline{e_k}k}$

1. if $e_k \mod w = 0$ then
2.     $d \leftarrow \text{Traverse}_w_{\text{div}}(r, 0, 0, \text{plen}_k - 1, \ell, w_k, \overline{\text{path}}_k, (T_1)_k)$  // Alg. 58
3. else
4.     $d \leftarrow \text{Traverse}_w_{\text{notdiv}}(r, 0, 0, \text{plen}_k - 1, \ell, w_k, \overline{\text{path}}_k, (T_1)_k, (T_2)_k)$  // Alg. 59
5. return $d \in \mathbb{Z}_{\overline{e_k}k}$
```

Algorithm 65: Compute 4 discrete logarithms (optimal Pohlig-Hellman strategy) on the multiplicative subgroup of order $\ell_k^{e_k}$

```
function ph
    Input: Points $P = (x_{P_M}, y_{P_M}), Q = (x_{Q_M}, y_{Q_M}), S_1 = (x_{S_1}, y_{S_1}), S_2 = (x_{S_2}, y_{S_2})$ on $E : y^2 = x^3 + Ax^2 + x$, the coefficient $A$, and the corresponding torsion $k \in \{2, 3\}$
    Output: The discrete logs $(c_0, c_1, d_0, d_1) \in \mathbb{Z}_{\overline{e_k}k}$ such that $P = \lfloor c_0 \rfloor S_1 + \lfloor c_1 \rfloor S_2$ and $Q = \lfloor d_0 \rfloor S_1 + \lfloor d_1 \rfloor S_2$

1. $n_1, n_2, n_3, n_4 \leftarrow \text{Tate}_4_{\text{pairings}}_{\text{k torsion}}(P, Q, S_1, S_2, A)$  // Alg. 62 or 63
2. $d_0 \leftarrow \text{solve_dlog}(n_1, k)$  // Alg. 64
3. $c_0 \leftarrow \text{solve_dlog}(n_3, k)$  // Alg. 64
4. $d_1 \leftarrow \text{solve_dlog}(n_2, k)$  // Alg. 64
5. $c_1 \leftarrow \text{solve_dlog}(n_4, k)$  // Alg. 64
6. return $c_0, d_0, c_1, d_1 \in (\mathbb{Z}_{\overline{e_k}k})^4$
```
Algorithm 66: Computing compressed public keys in the $3^{e_3}$-torsion

function PublicKeyCompression_2
  Input: Public key $pk_2 = (x_1, x_2, x_3)$
  Output: Compressed public key $pk_{comp2} = (bit, t_1, t_2, t_3, A, r_1, r_2)$ according to compressed encoding

1. $y_P, y_Q, A \leftarrow \text{get}_yP_yQ_A_B(x_1, x_2, x_3)$  // Alg. 41
2. $x_1, y_1, x_2, y_2, r_1, r_2 \leftarrow \text{BuildOrdinaryE}3nBasis(A)$  // Alg. 54
3. $c_0, d_0, c_1, d_1 \leftarrow \text{ph}(y_P, y_Q, x_1, y_1, x_2, y_2, A, 3)$  // Alg. 65

4. if $d_1 \pmod{3^{e_3}} \neq 0$ then
   5. $bit \leftarrow 0$
   6. $t_1 \leftarrow -d_0 \cdot d_1^{-1}$
   7. $t_2 \leftarrow -c_1 \cdot d_1^{-1}$
   8. $t_3 \leftarrow c_0 \cdot d_1^{-1}$
5. else
   6. $bit \leftarrow 1$
   7. $t_1 \leftarrow -d_1 \cdot d_0^{-1}$
   8. $t_2 \leftarrow c_1 \cdot d_0^{-1}$
   9. $t_3 \leftarrow -c_0 \cdot d_0^{-1}$
14. return $(bit, t_1, t_2, t_3, A, r_1, r_2)$  // Encoded as in §1.2.10
Algorithm 67: Computing compressed public keys in the $2^{e_2}$-torsion

function PublicKeyCompression_3
    Input: Public key $pk_3 = (x_1, x_2, x_3)$
    Output: Compressed public key $pk_{comp_3} = (bit, t_1, t_2, t_3, entang_bit, r)$ // Encoded as in §1.2.10

1. $y_P, y_Q, A \leftarrow \text{get}_yP_yQ_A_B(x_1, x_2, x_3)$ // Alg. 41
2. $x_1, y_1, x_2, y_2, entang_bit, r \leftarrow \text{get}_2_torsion_entangled_basis_compression(A)$ // Alg. 48
3. $c_0, d_0, c_1, d_1 \leftarrow \text{ph}(y_P, y_Q, x_1, y_1, x_2, y_2, A, 2)$ // Alg. 65

4. if $d_1 \pmod{2^{e_2}} \neq 0$ then
   5. $bit \leftarrow 0$
   6. $t_1 \leftarrow -d_0 \cdot d_1^{-1}$
   7. $t_2 \leftarrow -c_1 \cdot d_1^{-1}$
   8. $t_3 \leftarrow c_0 \cdot d_1^{-1}$

else
   9. $bit \leftarrow 1$
   10. $t_1 \leftarrow -d_1 \cdot d_0^{-1}$
   11. $t_2 \leftarrow c_1 \cdot d_0^{-1}$
   12. $t_3 \leftarrow -c_0 \cdot d_0^{-1}$

14. return $(bit, t_1, t_2, t_3, A, entang_bit, r)$ // Encoded as in §1.2.10
Algorithm 68: Compute a kernel generator for the last $2^{e_2}$-isogeny

```plaintext
function PublicKeyDecompression_2
    Input: Secret key $sk_2 \in \mathbb{Z}_{2^{e_2}}$ and compressed public key 
           $\{bit, (t_1, t_2, t_3) \in (\mathbb{Z}_{2^{e_2}})^3, A, entang_bit, r\}$
    Output: A kernel generator $(x', z') \in E[2^{e_2}]$ of the last $2^{e_2}$-isogeny
1. $(x_1, y_1)_{P_1}, (x_2, y_2)_{P_2} \leftarrow \text{BuildEntangledE2mBasis-Decompression}(A, entang_bit, r)$  // Alg. 49
2. if $bit = 0$ then
3.     $scal \leftarrow (t_1 + sk_2 \cdot t_3)(1 + sk_2 \cdot t_2)^{-1}$
4.     $(x, z) \leftarrow \text{Ladder3pt}(scal, x_1, x_2, x(P_1 - P_2), (A : 1))$  // Alg. 8
5. else
6.     $scal \leftarrow (t_1 + sk_2 \cdot t_2)(1 + sk_2 \cdot t_3)^{-1}$
7.     $(x, z) \leftarrow \text{Ladder3pt}(scal, x_2, x_1, x(P_1 - P_2), (A : 1))$  // Alg. 8
8. $(x', z') \leftarrow xTPLE\_fast(x, z, A/2, e_3)$  // Alg. 47
9. return $(x', z')$
```

Algorithm 69: Compute a kernel generator for the last $3^{e_3}$-isogeny

```plaintext
function PublicKeyDecompression_3
    Input: Secret key $sk_3 \in \mathbb{Z}_{3^{e_3}}$ and compressed public key $\{bit, (t_1, t_2, t_3) \in (\mathbb{Z}_{3^{e_3}})^3, A, r_1, r_2\}$
    Output: A kernel generator $(x', z') \in E_A[3^{e_3}]$ of the last $3^{e_3}$-isogeny
1. $(x_1, y_1)_{P_1}, (x_2, y_2)_{P_2} \leftarrow \text{BuildOrdinaryE3mBasis-decompression}(A, bit, r_1, r_2)$  // Alg. 55
2. if $bit = 0$ then
3.     $scal \leftarrow (t_1 + sk_3 \cdot t_3)(1 + sk_3 \cdot t_2)^{-1}$
4.     $(x, z) \leftarrow \text{Ladder3pt}(scal, x_1, x_2, x(P_1 - P_2), (A : 1))$  // Alg. 8
5. else
6.     $scal \leftarrow (t_1 + sk_3 \cdot t_2)(1 + sk_3 \cdot t_3)^{-1}$
7.     $(x, z) \leftarrow \text{Ladder3pt}(scal, x_2, x_1, x(P_2 - P_1), (A : 1))$  // Alg. 8
8. $(x', z') \leftarrow xTPLE\_fast(x, z, A/2, e_3)$  // Alg. 47
9. return $(x', z')$
```
Appendix E

Changes made in the 2nd round

The main differences between the first round and second round SIKE submissions are as follows.

- Two new parameter sets have been added: SIKEp434 (§1.6.1) and SIKEp610 (§1.6.4).
- One parameter set (SIKEp964) has been removed.
- Security categories for parameter sets have been adjusted upward. Chapter 5 presents the rationale for this change.
- The starting curve has been changed from $A = 0$ to $A = 6$. §1.3.2 presents the rationale for this change.
- An additional implementation including public key compression has been added (§1.5, §2.3).
Appendix F

Notation

\( \ell, m \) \quad \text{Integers } \ell, m \in \{2, 3\} \text{ such that } \ell \neq m \n
\( e_\ell \) \quad \text{The index of } \ell \text{ in the degree of the } \ell\text{-power isogeny} \n
\( \text{sk}_\ell \) \quad \text{The secret key corresponding to points in the } \ell^{e_\ell}\text{-torsion} \n
\( \text{pk}_\ell \) \quad \text{The public key corresponding to points in the } \ell^{e_\ell}\text{-torsion} \n
\( \phi_\ell \) \quad \text{The secret } \ell^{e_\ell}\text{-isogeny corresponding to } \text{sk}_\ell \n
\( P_\ell \) \quad \text{A point of exact order } \ell^{e_\ell} \text{ in } E_0(\mathbb{F}_{p^2}) \setminus E_0(\mathbb{F}_p), \text{ such that the order-} \ell^{e_\ell} \text{ Weil pairing, } e_{\ell^{e_\ell}}(P_\ell, Q_\ell), \text{ has exact order } \ell^{e_\ell} \n
\( Q_\ell \) \quad \text{A point of exact order } \ell^{e_\ell} \text{ in } E_0(\mathbb{F}_p) \n
\( R_\ell \) \quad \text{The point defined as } R_\ell = Q_\ell - P_\ell \n
\( \text{isogen}_\ell \) \quad \text{The algorithm that computes public keys — see §1.3.5} \n
\( \text{isoex}_\ell \) \quad \text{The algorithm that establishes shared keys — see §1.3.6} \n
\( \text{compress}_\ell \) \quad \text{The algorithm that compresses public keys — see §1.5.1} \n
\( \text{decompress}_\ell \) \quad \text{The algorithm that decompresses public keys — see §1.5.2} \n
\( E_a \) \quad \text{The Montgomery curve defined by } E_\alpha/\mathbb{F}_{p^2} : y^2 = x^3 + ax^2 + x \n
\( p \) \quad \text{The prime field characteristic defined as } p = 2^{e_23^{e_3}} - 1 \n
\( x_P \) \quad \text{The } x\text{-coordinate of the point } P \n
\( y_P \) \quad \text{The } y\text{-coordinate of the point } P \n
\( \mathcal{K}_2 \) \quad \text{The keyspace corresponding to points in the } 2^{e_2}\text{-torsion} \n
\( \mathcal{K}_3 \) \quad \text{The keyspace corresponding to points in the } 3^{e_3}\text{-torsion} \n
\( N_p \) \quad \text{The number of bytes used to represent elements in } \mathbb{F}_p \n
\( N_{sk} \) \quad \text{The number of bytes used to represent secret keys} \n
\( N_{pk} \) \quad \text{The number of bytes used to represent public keys} \n
\( \mathbb{Z} \) \quad \text{The ring of integers} \n
\( \mathbb{F}_q \) \quad \text{The finite field with } q \text{ elements} \n
\( \overline{\mathbb{F}}_q \) \quad \text{The algebraic closure of the finite field with } q \text{ elements} \n
\( \mathbb{F}_p \) \quad \text{The prime field with } p \text{ elements} \n
\( \mathbb{F}_{p^2} \) \quad \text{The quadratic extension field } \mathbb{F}_{p^2}, \text{ constructed over the prime field } \mathbb{F}_p \text{ as } \mathbb{F}_{p^2} = \mathbb{F}_p(i) \text{ with } i^2 + 1 = 0 \n
\( \mathbb{P}^n(K) \) \quad \text{The projective space of dimension } n \text{ over the field } K \n
\( Q_2 \) \quad \text{A point of exact order } 2^{e_2} \text{ in } E_0(\mathbb{F}_p) \n
97
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 $2^e_2$

The point defined as $R_2 = Q_2 - P_2$

A point of exact order $3^e_3$ in $E_0(\mathbb{F}_p)$

A point of exact order $3^e_3$ in $E_0(\mathbb{F}_{p^2}) \setminus E_0(\mathbb{F}_p)$, such that the order-$3^e_3$ Weil pairing, $e_3^{e_3}(P_3, Q_3)$, has exact order $3^e_3$

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
⊕ The binary exclusive or (XOR) of equal-length bitstrings
$I$ An oracle computing isogenies of degree $\ell^{e_\ell/2}$
$\mathcal{B}$ A block cipher
$G^c$ The number of gates of a classical circuit
$G^Q$ The number of gates of a quantum circuit
$D^c$ The depth of a classical circuit
$D^Q$ The depth of a quantum circuit
AES Advanced Encryption Standard
PKE An isogeny-based public-key encryption scheme
KEM An isogeny-based key encapsulation mechanism
Gen Key generation algorithm for PKE
Enc Encryption algorithm for PKE
Dec Decryption algorithm for PKE
KeyGen Key generation algorithm for KEM
Encaps Encapsulation algorithm for KEM
Decaps Decapsulation algorithm for KEM
$F$ A random oracle
$G$ A random oracle
$H$ A random oracle
SHAKE256 A customizable extendable-output function standardized by NIST
$c_0$ First part of an encapsulation of KEM
$c_1$ Second part of an encapsulation of KEM