x-only point addition formula and faster torsion basis generation in compressed SIKE

Geovandro C. C. F. Pereira[1,3], Javad Doliskani[2], and David Jao[1,3]

[1] Institute for Quantum Computing, University of Waterloo, Canada
[2] Ryerson University, Toronto, Canada
[3] evolutionQ, Inc.
{geovandro.pereira,djao}@uwaterloo.ca
javad.doliskani@ryerson.ca

Abstract. The optimization of the main key compression bottlenecks of the supersingular isogeny key encapsulation mechanism (SIKE) has been a target of research in the last few years. Significant improvements were introduced in the recent works of Costello et al. [6] and Zanon et al. [17,18]. The combination of the techniques in [17,18] reduced the running time of binary torsion basis generation in decompression by a factor of 29 compared to previous work [6]. On the other hand, generating such a basis still takes almost a million cycles on an Intel Core i5-6267U. In this paper, we continue the work of [18] and introduce a technique that drops the complexity of binary torsion basis generation by a factor $\log p$ in the number of underlying field multiplications. In particular, our experimental results show that a basis can be generated in about $1.3k$ cycles, attaining an improvement by a factor more than 600. Although this result eliminates one of the key compression bottlenecks, many other bottlenecks remain. In addition, we give further improvements for the ternary torsion generation with significant impact on the related decompression procedure. Moreover, a new trade-off between ciphertext sizes vs decapsulation speed and storage is introduced and achieves a 2 times faster decapsulation.

Keywords: Post-quantum cryptography, Supersingular elliptic curves, Public-key compression, Diffie-Hellman key exchange

1 Introduction

Public-key cryptosystems based on elliptic curve isogenies are conjectured to be secure against quantum attacks, and as a result have attracted some interest in the post-quantum cryptography community. One particular such cryptosystem, Supersingular Isogeny Key Encapsulation (SIKE) [16], has been proposed as a candidate for the NIST post-quantum standardization process [15]. SIKE is based on the Supersingular Isogeny Diffie-Hellman (SIDH) construction of Jao and De Feo [11], whose security relies on the hardness of the supersingular isogeny graph path-finding problem introduced by Charles et al. [5].

An especially attractive feature of SIKE is its small public key size. Of all the public-key cryptosystems submitted to the NIST standardization process in the first round, SIKE has the smallest proposed public keys at each of its supported security levels. Furthermore, the public keys can actually be made *even smaller*: in 2016, Azarderakhsh et. al. [2] introduced techniques to compress SIDH public keys by a factor of 2 in size. Unfortunately, these techniques are quite expensive in terms of performance, and they were not included in the SIKE first round NIST submission. Subsequent work [6,17,18] has led to a series of performance improvements in key compression, and an option to use key compression was added into the second round submission for SIKE, incorporating all of the aforementioned performance optimizations. For example, public keys occupy 196 bytes and ciphertexts 209 bytes for SIKEp434 [16, Table 2.2]. This work is about further improving the performance of key compression in SIKE beyond what was achieved in the SIKE second round submission and other prior work.

We remark that other isogeny-based cryptosystems, notably CSIDH [4], are able to achieve even smaller public keys than SIKE at the lowest security levels defined by NIST (namely, NIST category 1, equivalent to AES-128 in security). At higher security levels, in the post-quantum setting, CSIDH eventually requires larger keys than SIKE, since CSIDH admits a known quantum subexponential-time attack whereas SIKE does not; to our knowledge the exact location of the crossover has not been identified in prior literature. In any case, CSIDH is not a NIST candidate, and the focus of this paper is on SIKE.

Our contributions. SIKE is a key encapsulation mechanism (KEM) based on the combination of SIDH together with the Kirkwood variant [13] of the Fujisaki–Okamoto transform [9]. The latter transform is necessary in any setting where one party's public key is re-used, because of the GPST attack [10] against SIDH. Public keys in SIDH and SIKE are identical, but as we will see, key compression in SIKE involves different considerations than in SIDH because of variations in how the keys are used.

The results in this paper affect the speed of encapsulation and decapsulation which both involve torsion basis generation when decompressing keys and ciphertexts, respectively. We propose two types of performance improvements. Some of our optimizations apply equally well to SIDH or SIKE. Others arise from optimizing the interactions between key compression and the Fujisaki–Okamoto transform in SIKE. In many cases, performance gains are subject to some sort of time-space tradeoff, where the space being traded off refers not only to runtime memory usage, but also to key size. The specific improvements that we propose are as follows, in summary form:

1. Extend the shared elligator technique introduced in [18]. During key generation, add two extra bits of information to the public key indicating the correct ternary basis generators which are elligator points. This prevents repeating two quadratic residuosity tests during encapsulation. Moreover, instead of multiplying both ternary basis elements by a cofactor 2^{e_2} and then computing the linear combination corresponding to the secret kernel point, reverse the order of these computations, saving one scalar multiplication by the cofactor. Two finite field inversions can be saved due to this reordering since the `Ladder3pt` algorithm (introduced in [12] and improved in [8]) now takes two affine cofactor-unreduced points. The combination of all the previous optimizations applied to SIKEp751 gives a 6.2× faster ternary basis generation and a 1.9× faster decompression step in encapsulation.
2. During decapsulation, we save a square root evaluation by using an x-only point addition formula[4] to complete the `Ladder3pt` algorithm. The latter is used in the computation of the kernel point for the shared secret computation step. The use of x-only formula is possible here because the basis elements produced by entangled basis generation [17] are of a special form. In particular, the binary torsion basis generation applied to SIKEp751 is experimentally improved by a factor 662 (during decompression) and decompression itself by a factor 1.44.
3. An optimization proposed in [6] involves scaling the coordinate vectors in a compressed key so that one of the four coordinates equals 1, saving an additional 12.5% of key size (the coordinate vector is only half of the compressed key). In SIDH, this optimization is essentially free, since scaling the coordinate vector results in scaling the secret kernel point, leaving the kernel subgroup unchanged. However, in SIKE, it is no longer free, since Fujisaki–Okamoto key validation

[4] This formula was dubbed **entangled addition** due to the fact that one of the basis generator is intrinsically related to the other following the nomenclature introduced in [17]. This formula was independently discovered in [14].

during decapsulation requires comparing the transmitted key with a re-computed key, which is difficult if the two keys are scaled differently. We propose to eliminate this key size optimization, making the compressed ciphertexts larger, but the comparison easier. We also propose a new optimization for performing this comparison more quickly than naively. The experimental results show a speedup by a factor of ≈ 2 for the overall decapsulation operation.

Our software was integrated into the SIDHv3.2 library and will be available at `https://github.com/microsoft/PQCrypto-SIDH`.

Remark. We should stress that part of contribution 1 (saving 1 scalar multiplication) and the contribution 2 above were independently discovered by Naehrig and Renes [14]. Contribution 3 is fully independent from [14] and provides the best result of this paper (2 times speed up in the decapsulation operation). Comparing these contributions to [14], we give further detailed analysis of these improvements, Section 3 explains and gives algorithmic description of the improved decompression, including a complexity analysis of the new basis generation. Moreover, Section 7 provides tailored experiments giving precise figures of the impact of the entangled addition formula combined with basis generation.

1.1 Notations and conventions

For simplicity, we assume that finite field arithmetic is carried out in a base field $\mathbb{F}_p$ and its quadratic extension $\mathbb{F}_{p^2}$ for a prime p of form $p := 2^{e_2} \cdot 3^{e_3} - 1$ for some $e_2 > 2$ and $e_3 > 1$, so that $p \equiv 3 \pmod 4$. The quadratic extension $\mathbb{F}_{p^2}/\mathbb{F}_p$ is represented as $\mathbb{F}_{p^2} = \mathbb{F}_p[i]/\langle i^2+1\rangle$, and arithmetic closely mimics that of the complex numbers.

All curves are represented using the Montgomery model unless otherwise specified. We follow the convention of using subscripts 2 and 3 for Alice and Bob, respectively. For example, the secret isogeny ϕ_2 is computed by Alice and her public parameters are denoted by the points P_2, Q_2 and the curve E_2.

1.2 Key compression and Entangled basis review

Key (De)Compression. Given a Montgomery curve $E_3 : y^2 = x^3 + Ax^2 + x$ defined over $\mathbb{F}_{p^2}$ for $p = 2^{e_2}3^{e_3} - 1$, the public key of Bob in SIDH consists of two points on E_3, denoted by $\phi_3(P_2), \phi_3(Q_2) \in E_3$, where ϕ_3 is Bob's private isogeny.

The main idea to achieve key compression [2,6] is the following: instead of transmitting points $\phi_3(P_2)$, $\phi_3(Q_2) \in E_3[2^{e_2}]$, which are represented by two abscissas in $\mathbb{F}_{p^2}$ and consume $4\log p$ bits, Bob computes a canonical basis $R_1, R_2 \in E_3[2^{e_2}]$ and expresses the expanded public key in that basis as $\phi_3(P_2) = a_0R_1 + b_0R_2$ and $\phi_3(Q_2) = a_1R_1 + b_1R_2$.

This representation consists of four smaller integers $(a_0, b_0, a_1, b_1) \in (\mathbb{Z}/2^{e_2}\mathbb{Z})^4$ of total size $2\log p$ bits as suggested in [2]. This was improved in [6] by transmitting only the triple $(a_0^{-1}b_0, a_0^{-1}a_1, a_0^{-1}b_1) \in (\mathbb{Z}/2^{e_2}\mathbb{Z})^3$ or $(b_0^{-1}a_0, b_0^{-1}a_1, b_0^{-1}b_1) \in (\mathbb{Z}/2^{e_2}\mathbb{Z})^3$ depending on whether a_0 or b_0 is invertible. Therefore, only $(3/2)\log p$, plus one bit indicating the invertibility of a_0 or b_0 modulo 2^{e_2}, is needed.

The major Alice's goal in decompression is to obtain the kernel point of her second isogeny, which is given by $ker(\phi_{23}) = \langle \phi_3(P_2) + sk_2 \cdot \phi_3(Q_2)\rangle$. Assuming a_0 to be invertible, the kernel of ϕ_{23} can be rewritten as

$$\begin{aligned}
&\langle \phi_3(P_2) + sk_2 \cdot \phi_3(Q_2)\rangle \equiv \\
&\langle (a_0R_1 + b_0R_2) + sk_2 \cdot (a_1R_1 + b_1R_2)\rangle \equiv \\
&\langle R_1 + a_0^{-1}b_0R_2 + sk_2 \cdot (a_0^{-1}a_1R_1 + a_0^{-1}b_1R_2))\rangle
\end{aligned}$$

Essentially, the decompression step consists of obtaining the basis R_1, R_2 and performing the related scalar multiplications in the last expression above upon reception of $a_0^{-1}b_0$, $a_0^{-1}a_1$ and $a_0^{-1}b_1$.

Entangled Basis. Zanon *et al.* introduced the idea of *entangled bases*, a faster technique to generate a basis $\{R_1, R_2\}$ for the 2^{e_2}-torsion subgroup of E_3, denoted $E_3[2^{e_2}] = \langle R_1, R_2\rangle$ where $R_1, R_2 \in E_3(\mathbb{F}_{p^2})$ [17]. Roughly speaking, the idea is inspired by the Elligator technique [3], which finds a point on the curve deterministically. The difference is that the authors in [17] tweaked the original Elligator to get not one but two points that are on the curve simultaneously with probability 50%. This was combined with the 2-descent technique [6] to get not only points on the curve but of maximal order 2^{e_2}. In particular, they proved that if the two points are picked in a special way (see Theorem 1) they not only are on the curve but they are linearly independent, therefore forming a basis for $E_3[2^{e_2}]$. To be precise, the following theorem was proved:

Theorem 1. *[17] Given a Montgomery supersingular elliptic curve $E_3/\mathbb{F}_{p^2} : y^2 = x(x^2 + Ax + 1)$ where $p = 2^{e_2} \cdot 3^{e_3} - 1$, $\#E_3(\mathbb{F}_{p^2}) = (p+1)^2$, and $A \neq 0$, let $t \in \mathbb{F}_{p^2}$ be a field element such that $t^2 \in \mathbb{F}_{p^2} \setminus \mathbb{F}_p$, and let $x_1 := -A/(1+t^2)$ be a quadratic non-residue that defines the abscissa of a point $R_1 \in E_3(\mathbb{F}_{p^2})$. Then $x_2 := -x_1 - A$ defines the abscissa of another point $R_2 \in E_3(\mathbb{F}_{p^2})$ such that $\langle [h]R_1, [h]R_2\rangle = E_3[2^{e_2}]$, where $h := 3^{e_3}$ is the cofactor of the 2^{e_2}-torsion group.*

Note that in the theorem above points R_1 and R_2 may have order $k_i \cdot 2^{e_2}$, i.e., a multiple of 2^{e_2} for some k_i dividing 3^{e_3} for $i \in \{1, 2\}$. We also say that points R_i are cofactor-unreduced since they contain the cofactor k_i. The multiplication by the cofactor $h = 3^{e_3}$ reduces the points R_i and ensures that the result is a point of order exactly 2^{e_2}.

2 x-only entangled addition formula

Let $E : y^2 = x^3 + A \cdot x^2 + x$ be a Montgomery supersingular elliptic curve over $\mathbb{F}_{p^2}$. Zanon *et al.* showed that if $x_1 = -A \cdot v$ is a quadratic non-residue (QNR) and the abscissa of a point on E, where $v = 1/(1 + u \cdot r^2) \in \mathbb{F}_{p^2}$, $u \in \mathbb{F}_{p^2}$ is a quadratic residue (QR) and $y_1 = \sqrt{x_1}$, then $x_2 := u \cdot r^2 \cdot x_1 = -A - x_1$ and $y_2 = u_0 \cdot r \cdot y_1$[5] are automatically the coordinates of a point $S_2 \in E$ such that the points $\langle [3^{e_3}]S_1, [3^{e_3}]S_2\rangle = E[2^{e_2}]$ generate a basis for the binary torsion subgroup [17, Theorem 1].

In the SIKE protocol, the kernel point generator computed in the first step of decapsulation is of the form $K := S_1 + t \cdot S_2 \in E[2^{e_2}]$ for some $t \in \mathbb{Z}_{2^{e_2}}$ as explained in Section 5 (Equation 7). In order to evaluate the expression $S_1 + t \cdot S_2$, a `Ladder3pt` algorithm B.2 is used. Such algorithm takes as input the affine representation of S_1, S_2 and the x coordinate of $S_2 - S_1$. The latter can be computed using the traditional Montgomery addition formula

$$x(S_2 + (-S_1)) = ((y_2 - (-y_1))/(x_2 - x_1))^2 - A - x_1 - x_2 \tag{1}$$

assuming that the Montgomery coefficient is $B = 1$. Observe that when S_1 and S_2 are entangled points, the linear relation $y(S_2) = y_2 = u_0 \cdot r \cdot y_1$ is satisfied. Substituting such relation in 1 one gets the x-only addition formula for $x(S_2 + (-S_1))$

$$\begin{aligned} x(S_2 - S_1) &= ((y_2 + y_1)/(x_2 - x_1))^2 - A - x_1 - x_2 \\ &= ((u_0 \cdot r \cdot y_1 + y_1)/(x_2 - x_1))^2 - A \\ &\quad - x_1 - (-A - x_1) \\ &= (y_1(u_0 \cdot r + 1)/(x_2 - x_1))^2 \\ &= y_1^2(u_0 \cdot r + 1)^2/(x_2 - x_1)^2 \\ &= (x_1^3 + Ax_1^2 + x_1)(u_0 r + 1)^2/(x_2 - x_1)^2. \end{aligned} \tag{2}$$

[5] Note that $u_0 = \sqrt{u} \in \mathbb{F}_{p^2} \backslash \mathbb{F}_p$ as defined in the original work.

Interestingly, the `Ladder3pt` Algorithm B.2 can be modified to accept $x(S_2 - S_1)$ in its projective representation, and consequently the inversion in Equation 2 can be avoided by taking $x = X/Z$ where $X = (x_1^3 + A \cdot x_1^2 + x_1)(u_0 \cdot r + 1)^2$ and $Z = (x_2 - x_1)^2$. Overall, the above formula avoids the need for the y coordinates of S_1 and S_2 and does not add any extra inversion. Algorithm 2.1 illustrates the sequence of steps of the addition. It is adapted to receive the inverse of the second point which is necessary for SIKE's use cases.

Algorithm 2.1 `EntangledAddition` [**this work**]: add an entangled basis generator with the inverse of the other without using their y-coordinates.

```
INPUT: Affine points $P_1 = (x_1, y_1), P_2 = (x_2, y_2)$ s.t. $y_2 = u_0 \cdot r \cdot y_1 \in \mathbb{F}_{p^2}$
    – Montgomery curve coefficient $A \in \mathbb{F}_{p^2}$
    – public parameter $u_0 \in \mathbb{F}_{p^2}$
OUTPUT: A projective representation $(X : Z)$ of the addition $P_1 + (-P_2)$
$t_1 \leftarrow u_0 \cdot r + 1$
$t_1 \leftarrow t_1^2$
$X \leftarrow x_1 + A$
$X \leftarrow x_1 \cdot X + 1$
$X \leftarrow x_1 \cdot X$
$X \leftarrow t_1 \cdot X$   // $X = y_1^2(u_0 \cdot r + 1)^2 = (x_1^3 + A \cdot x_1^2 + x_1)(u_0 \cdot r + 1)^2$
$Z \leftarrow x_2 - x_1$
$Z \leftarrow z^2$   // $Z = (x_2 - x_1)^2$
return $(X, Z)$
```

3 Faster entangled basis generation

This section explains how SIKE decapsulation can benefit from the point addition formula introduced in Section 2. Zanon et al. introduced a faster (entangled) binary torsion basis generation in [17], which was further improved in [18] with a shared elligator technique. The idea of shared elligator is to share the small counters obtained during key compression for getting basis generator points. These counters tell which are the correct points on the curve and the user performing decompression can simply recover the points deterministically if counters are shared. The combined improvements of [17,18] dropped the original cycle count from 26M to about a million for SIKEp751 on an Intel i5 processor at 2.9 GHz.

In particular, Algorithm B.3 from [18] generates an entangled basis during compression and stores both the bit representing the quadraticity of the curve coefficient A and the elligator counter r that gives the correct entry of a precomputed table T. This extra information learned during compression is then transmitted to make decompression faster. Typically the counter r does not exceed one byte. During decompression, binary basis generation is then performed deterministically by consuming the shared information using the tailored Algorithm B.4.

An immediate consequence of the **entangled addition** formula is that the expensive square root computation taken at Steps 4–14 of Algorithm B.4 can be automatically avoided, since the y coordinates become unnecessary for the subsequent steps. As a result, an extremely fast entangled basis generation can be achieved. In particular, the major cost of the entangled basis is reduced to one single multiplication in $\mathbb{F}_{p^2}$ independently of the field size as opposed to the previous $O(\log p)$ multiplications due to the square root needed for recovering the y-coordinate. This represents roughly a $\log p$ factor improvement. A more precise complexity analysis based on the underlying operation counts is provided later in this section.

The faster entangled basis generation is shown in Algorithm 3.2. In practice, this algorithm revealed to be more than 662 times faster than the previous work from our experiments for the

Algorithm 3.1 `EntangledBasisDecompression` [18]: Entangled basis generation with shared Elligator for $E[2^{e_2}](\mathbb{F}_{p^2}) : y^2 = x^3 + Ax^2 + x$

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 $u_0 \in \mathbb{F}_{p^2} : u = u_0^2 \in \mathbb{F}_{p^2} \backslash \mathbb{F}_p$; 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 $\langle [3^{e_3}]S_1, [3^{e_3}]S_2 \rangle = E[2^{e_2}](\mathbb{F}_{p^2})$

```
$T \leftarrow (bit \overset{?}{=} 1)\ T_1 : T_2$   // Table $T_1$ is picked if $A$ is QR and $T_2$ otherwise
look up entry $v$ corresponding to $r$ in $T$
$x \leftarrow -A \cdot v$
$t \leftarrow x \cdot (x \cdot (x + A) \cdot x + 1)$
test quadraticity of $t = c + di$
$z \leftarrow c^2 + d^2$
$s \leftarrow z^{(p+1)/4}$
if $s^2 \neq z$ then
    Abort: invalid input parameters $(bit, r)$ received
end if
$z \leftarrow (c + s)/2$
$\alpha \leftarrow z^{(p+1)/4}$
$\beta \leftarrow d \cdot (2\alpha)^{-1}$
$y \leftarrow (\alpha^2 \overset{?}{=} z)\ \alpha + \beta i : -\beta - \alpha i$   // $y \leftarrow \sqrt{x^3 + A \cdot x^2 + x}$
return $S_1 \leftarrow (x, y)$, $S_2 \leftarrow (u \cdot r^2 \cdot x, u_0 \cdot r \cdot y)$
```

prime p751. The actual cycle counts are of 0.85×10^3 and 1.25×10^3 for SIKEp434 and SIKEp751, respectively, showing that binary torsion basis generation during decompression is not a bottleneck any more (see Section 7). The integrated version of the new entangled addition with the fast basis generation is given by the decompression Algorithm 3.3. Note that the decompression procedure is a step of the SIKE decapsulation (`Decaps` function of Algorithm 6.1) and has its cost slightly amortized by the remaining steps.

Algorithm 3.2 `EntangledBasisDecompression` **[this work]**: Entangled basis generation with shared elligator and **entangled addition** for $E[2^{e_2}](\mathbb{F}_{p^2}) : y^2 = x^3 + A \cdot x^2 + x$

INPUT: $A = a + bi \in \mathbb{F}_{p^2}$
- a bit bit indicating A's quadraticity
- an elligator counter $r \in \mathbb{F}_p$
- public 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 v's, respectively

OUTPUT: $\{x(S_1), x(S_2)\}$ such that $\langle [3^{e_3}]S_1, [3^{e_3}]S_2 \rangle = E[2^{e_2}]$

```
$T \leftarrow (bit \overset{?}{=} 1)\ T_1 : T_2$   // select proper table according to $A$'s quadraticity
lookup entry $v$ corresponding to $r$ on $T$
$x_1 \leftarrow -A \cdot v$
$x_2 \leftarrow -x_1 - A$
return $(x_1, x_2)$
```

3.1 Complexity analysis of entangled basis generation

In order to evaluate the theoretical expected improvement for entangled basis generation during Decompression (ran within SIKE Decapsulation) we analyze the previous Algorithm B.4 and the proposed Algorithm 3.2 with respect to the underlying operation counts in terms of base field multiplications ($\mathbb{F}_p$).

For this analysis, we denote by **i**, **c**, **m** and **s** the costs of inverting, cubing, multiplying, squaring, and adding/subtracting/shifting in $\mathbb{F}_p$, respectively, and by **I**, **C**, **M** and **S** the costs of the corre-

Algorithm 3.3 `Decompress_2` **[this work]**: decompress the public key and compute a kernel generator for the last 2^{e_2}-isogeny

```
INPUT: Secret key $sk_2 \in \mathbb{Z}_{2^{e_2}}$,
    – compressed public key $pk_2 = \{bit, (t_1, t_2, t_3) \in (\mathbb{Z}_{2^{e_2}})^3, A \in \mathbb{F}_{p^2}, ent\_bit, r \in \mathbb{Z}_{256}\}$
    – $ent\_bit$ indicates if $A$ is a QR or not
    – $bit$ indicates which one of the possible scalars is invertible
OUTPUT: A kernel generator $(x', z')$ for the last $2^{e_2}$-isogeny
$(x_1, x_2) \leftarrow$ EntangledBasisDecompression($A, ent\_bit, r$)   // Alg. 3.2
$X, Z \leftarrow$ ADDEntangled($x_1, x_2, A$)   // Alg. 2.1
if $bit = 0$ then
    $scal \leftarrow (t_1 + sk_2 \cdot t_3)(1 + sk_2 \cdot t_2))^{-1}$
    $(x, z) \leftarrow$ Ladder3pt($scal, x_1, x_2, X, Z, A$)   // Alg. B.2
else
    $scal \leftarrow (t_1 + sk_2 \cdot t_2)(1 + sk_2 \cdot t_3))^{-1}$
    $(x, z) \leftarrow$ Ladder3pt($scal, x_2, x_1, X, Z, A$)   // Alg. B.2
end if
$(x', z') \leftarrow [3^{e_3}](x, z)$
return $(x', z')$
```

sponding operations in $\mathbb{F}_{p^2}$. We disregard the cost of changing a sign (for instance, when handling the conjugate of a field element) and additions. The squaring over the base field **s** is implemented by simply calling a multiplication **m** and thus $\mathbb{F}_p$-squarings are replaced by $\mathbb{F}_p$-multiplications in the analysis. The costs of the $\mathbb{F}_{p^2}$ operations relative to the costs of operations in $\mathbb{F}_p$ can be approximated by $1\mathbf{M} \approx 3\mathbf{m}$ and $1\mathbf{S} \approx 2\mathbf{m}$.

Complexity of Algorithm B.4. The relative cost of Steps 1 and 2 is negligible as they involve a conditional test and a table entry lookup. Step 3 involves $1\mathbf{M} \approx 3\mathbf{m}$. Step 4 involves $3\mathbf{M}$, or equivalently, $\approx 9\mathbf{m}$. Step 5 is just a comment for the following steps. Note that from Step 6 and further (except by the last one), operations take place over the base field. Step 6 involves $2\mathbf{s} \approx 2\mathbf{m}$. Step 7 involves an exponentiation to the $(p+1)/4$-power or, equivalently, to the $2^{e_2-2}3^{e_3}$-power. This amounts to $(e_2 - 2)\mathbf{s} \approx (e_2 - 2)\mathbf{m}$ and $e_3\mathbf{c} \approx 2e_3\mathbf{m}$. Step 8 amounts to $1\mathbf{s} \approx 1\mathbf{m}$. Step 12 involves an exponentiation similar to Step 7 and thus takes $(e_2 - 2)\mathbf{m} + 2e_3\mathbf{m} = (e_2 + 2e_3 - 2)\mathbf{m}$. Step 13 involves $1\mathbf{i}$ and $1\mathbf{m}$. Step 14 carries $1\mathbf{s} \approx 1\mathbf{m}$. Finally, in Step 15, both expressions $u_0 \cdot r$ and $u \cdot r^2$ can be computed each with $2\mathbf{m}$ for small constant r. Two extra multiplications over the quadratic field are then performed, i.e., $6\mathbf{m}$. Adding up all the steps together one gets:

$$(2e_2 + 4e_3 + 23)\mathbf{m} + \mathbf{i} \tag{3}$$

Complexity of Algorithm 3.2. Equivalently to Algorithm B.4 Steps 1 and 2 have negligible cost. Step 3 involves $1\mathbf{M} \approx 3\mathbf{m}$. Adding up all the steps together one gets:

$$3\mathbf{m} \tag{4}$$

For instance, the expected improvement for SIKEp751 parameter set can be theoretically estimated as

$$\left(\frac{(2 \cdot 372 + 4 \cdot 239 + 23)\mathbf{m} + 150\mathbf{m}}{3\mathbf{m}}\right) \approx 624 \tag{5}$$

For the above estimate we verified experimentally that our inversion operation **i**, implemented as a binary GCD algorithm, costs $\approx 150\mathbf{m}$.

4 Extend shared elligator for faster ternary basis generation

In [18], the authors introduced the shared elligator technique to speed-up basis generation during decompression. This section focuses on the ternary basis generation which impacts the encapsulation operation of SIKE (`Encaps` function of Algorithm 6.1).

For speeding up ternary basis generation during decompression, the entity who performs key compression transmits not only the compressed key but also a small counter r storing the correct entry in an auxiliary table T. This table is used for getting basis generator points on a Montgomery curve $E : y^2 = x^3 + A \cdot x^2 + x, A \in \mathbb{F}_{p^2}$. In particular, it stores precomputed values of the form $v := 1/(1 + Ur^2) \in \mathbb{F}_{p^2}$ for small values of $r \in \mathbb{F}_p$ and a constant $U \in \mathbb{F}_{p^2}$. The values v in T are used to search for point candidates whose abscissa can be either

$$x := -A \cdot v \quad \text{or} \quad x := -A + A \cdot v.$$

A quadraticity test is conducted to decide which one of the abscissas corresponds to a point on the curve [3]. Once two of the points found are full order 3^{e_3} and L.I., a ternary basis has been generated. The values of r that lead to the basis points typically fit one byte and thus it is worth it to share them at the cost of a small increase in the public key.

Although shared elligator allows for a deterministic recovery of the correct table entries r, there are still two expensive quadraticity tests to be performed by the user decompressing the keys. We suggest an extension of the elligator idea that transmits two extra bits indicating the correct abscissas and therefore avoids the expensive quadraticity tests. The two extra bits can be accommodated in one extra byte in the public key. This modification adds a very small overhead in size compared to speed up gains. The faster (and fully) deterministic ternary basis generation algorithm is illustrated in Algorithm 4.1. Moreover, the full (ternary) decompression including the extended elligator is described in Algorithm 5.2.

Algorithm 4.1 `BasePoint3nDecompression` **[this work]**: Deterministic affine construction of a point of order 3^{e_3} in the Montgomery curve $E : y^2 = x^3 + A \cdot x^2 + x$ from r and bit

```
INPUT: curve coefficient $A \in \mathbb{F}_{p^2}$
    – elligator counter $r \in \mathbb{Z}$ points the correct table entry
    – elligator $bit$ indicating the correct elligator point
    – public table $T$ of elligator values $v = 1/(1 + Ur^2) \in \mathbb{F}_{p^2}$ where $U \in \mathbb{F}_{p^2} \backslash \mathbb{F}_p$ is a QNR
OUTPUT: Abscissa $x$ of a point of order $3^{e_3}$ from $r$
lookup entry $v$ corresponding to $r$ on $T$
$x \leftarrow -A \cdot v$
if $bit$ then
    $x \leftarrow -x - A$
end if
return $x$
```

5 Eliminate a multiplication by cofactor in the ternary decompression

We briefly explain how one can save a cofactor multiplication in the decompression of the ternary basis. The same technique was originally used for the binary basis in [18]. The key compression idea of [2,6] for Alice's public key $\phi_2(P_3), \phi_2(Q_3) \in E_2[3^{e_3}]$ is as follows. Let $R_1, R_2 \in E_2[3^{e_3}]$ be a canonical basis. Write $\phi_2(P_3) = a_0 R_1 + b_0 R_2$ and $\phi_2(Q_3) = a_1 R_1 + b_1 R_2$ for unique a_i, b_i, $i = 0, 1$. Then, Alice only needs to transmit the triple $(a_0^{-1} b_0, a_0^{-1} a_1, a_0^{-1} b_1) \in (\mathbb{Z}/3^{e_3}\mathbb{Z})^3$ (or $(b_0^{-1} a_0$, $b_0^{-1} a_1, b_0^{-1} b_1) \in (\mathbb{Z}/3^{e_3}\mathbb{Z})^3)$ if a_0 (or b_0) is invertible $\mod 3^{e_3}$. The coefficients a_0, a_1, b_0, b_1 can be computed using Tate pairings and discrete logs in $\mathbb{Z}/3^{e_3}\mathbb{Z}$.

In [18], the authors proposed *reverse basis decomposition* to save one pairing computation. The idea is to write R_1, R_2 as linear combinations of $\phi_2(P_3), \phi_2(Q_3)$. Let $R_1 = c_0\phi_2(P_3) + d_0\phi_2(Q_3)$ and $R_2 = c_1\phi_2(P_3) + d_1\phi_2(Q_3)$. Then the coefficients c_0, d_0, c_1, d_1 can be computed by first computing the pairings

$$\begin{aligned}
h_0 &= \mathsf{e}_{3^{e_3}}(\phi_2(P_3), \phi_2(Q_3)) \\
h_1 &= \mathsf{e}_{3^{e_3}}(\phi_2(P_3), R_1) \\
&= \mathsf{e}_{3^{e_3}}(\phi_2(P_3), c_0\phi_2(P_3) + d_0\phi_2(Q_3)) = h_0^{d_0} \\
h_2 &= \mathsf{e}_{3^{e_3}}(\phi_2(P_3), R_2) \\
&= \mathsf{e}_{3^{e_3}}(\phi_2(P_3), c_1\phi_2(P_3) + d_1\phi_2(Q_3)) = h_0^{d_1} \\
h_3 &= \mathsf{e}_{3^{e_3}}(\phi_2(Q_3), R_1) \\
&= \mathsf{e}_{3^{e_3}}(\phi_2(Q_3), c_0\phi_2(P_3) + d_0\phi_2(Q_3)) = h_0^{-c_0} \\
h_4 &= \mathsf{e}_{3^{e_3}}(\phi_2(Q_3), R_2) \\
&= \mathsf{e}_{3^{e_3}}(\phi_2(Q_3), c_1\phi_2(P_3) + d_1\phi_2(Q_3)) = h_0^{-c_1},
\end{aligned} \tag{6}$$

and then computing the discrete logs $c_0 = -\log_{h_0} h_3$, $d_0 = \log_{h_0} h_1, c_1 = -\log_{h_0} h_4$ and $d_1 = \log_{h_0} h_2$ in $\mathbb{Z}/3^{e_3}\mathbb{Z}$. The first pairing h_0 can be computed once and for all using public parameters, so only the last four pairings need to be computed at runtime. Assume that d_1 is a unit mod 3^{e_3}, then Alice transmits the triple $(-d_1^{-1}d_0, -d_1^{-1}c_1, d_1^{-1}c_0) = (a_0^{-1}b_0, a_0^{-1}a_1, a_0^{-1}b_1)$. After receiving this compressed key, Bob first computes the basis R_1, R_2, and then computes the kernel $\langle R_1 + (1 + sk_3a_0^{-1}a_1)^{-1}(a_0^{-1}b_0 + sk_3a_0^{-1}b_1)R_2\rangle$ of his secret isogeny $\phi_{2,3}$.

The same public key triple can be obtained without R_1, R_2 being necessarily a basis. More precisely, suppose there is an algorithm that generates two linearly independent points $S_1, S_2 \in E_2(\mathbb{F}_{p^2})$ of orders $2^{k_1}3^{e_3}, 2^{k_2}3^{e_3}$ for some $k_1, k_2 \geq 0$. Without loss of generality, assume that $(R_1, R_2) = (2^{e_2}S_1, 2^{e_2}S_2)$ is our canonical basis. By definition, the Tate pairing in $E_2[3^{e_3}]$ allows using an arbitrary point of $E(\mathbb{F}_{p^2})$ in the second argument. We have

$$\begin{aligned}
\hat{h}_1 &= \mathsf{e}_{3^{e_3}}(\phi_2(P_3), S_1) = \mathsf{e}_{3^{e_3}}(\phi_2(P_3), S_1)^{2^{e_2}2^{-e_2}} \\
&= \mathsf{e}_{3^{e_3}}(\phi_2(P_3), 2^{e_2}S_1)^{2^{-e_2}} = \mathsf{e}_{3^{e_3}}(\phi_2(P_3), R_1)^{2^{-e_2}} \\
&= h_1^{2^{-e_2}}.
\end{aligned}$$

Repeating the same thing for the other pairings we obtain $\hat{h}_i = h_i^{2^{-m}}$ for all $1 \leq i \leq 4$. Now, solving discrete logs gives $\hat{c}_0 = 2^{-e_2}c_0, \hat{d}_0 = 2^{-e_2}d_0, \hat{c}_1 = 2^{-e_2}c_1, \hat{d}_1 = 2^{-e_2}d_1$, and assuming $\hat{d}_1$ is a unit mod 3^{e_3}, Alice transmits

$$\begin{aligned}
&(-\hat{d}_1^{-1}\hat{d}_0, -\hat{d}_1^{-1}\hat{c}_1, \hat{d}_1^{-1}\hat{c}_0) = \\
&= (-d_1^{-1}2^{e_2}2^{-e_2}d_0, -d_1^{-1}2^{e_2}2^{-e_2}c_1, d_1^{-1}2^{e_2}2^{-e_2}c_0) \\
&= (-d_1^{-1}d_0, -d_1^{-1}c_1, d_1^{-1}c_0) \\
&= (a_0^{-1}b_0, a_0^{-1}a_1, a_0^{-1}b_1).
\end{aligned}$$

To decompress, Bob uses the same algorithm to generate S_1, S_2 and computes

$$S = S_1 + (1 + sk_3a_0^{-1}a_1)^{-1}(a_0^{-1}b_0 + sk_3a_0^{-1}b_1)S_2. \tag{7}$$

The kernel of his secret isogeny will then be $\ker(\phi_{2,3}) = \langle 2^{e_2}S\rangle$. In summary, instead of computing R_1, R_2, which takes two scalar multiplication by the cofactor 2^{e_2}, and then computing $\ker(\phi_{2,3})$, Bob only needs to compute $2^{e_2}S$, which takes one scalar multiplication by 2^{e_2}, and then $\ker(\phi_{2,3})$.

Note that since the scalar multiplication by cofactor is postponed, Steps 1 and 2 in the general ternary basis generation Algorithm 5.2 will retrieve the abscissas of affine points, and consequently Algorithm 5.1 gets projective points with $z = 1$ in Steps 3 and 4. This means that inversions and some extra multiplications and squarings in the `CompleteMPoint` can be avoided in those steps.

Algorithm 5.1 `CompleteMPoint`: given a xz-only representation on a Montgomery curve E, compute the affine representation.

```
INPUT: Montgomery curve coefficient $A$, point $P = (x, z) \in E$
OUTPUT: $(x', y', z')$, the affine representation of $P$
if $z \neq 0$ then
    $xz \leftarrow x \cdot z$
    $ss \leftarrow (x + i \cdot z)(x - i \cdot z)$
    $rr \leftarrow xz(A \cdot xz + ss)$
    $sr \leftarrow \sqrt{rr}$
    $x' \leftarrow x$
    $y' \leftarrow sr$
    $z' \leftarrow 1$
else
    $x' \leftarrow 0$; $y' \leftarrow 1$; $z' \leftarrow 0$
end if
return $x', y', z'$
```

Algorithm 5.2 `BuildE3nBasisDecompression` **[this work]**: deterministically generating a basis for $E[3^{e_3}](\mathbb{F}_{p^2}) : y^2 = x^3 + Ax^2 + x$ from A and elligator counters r_1, r_2 and bits bit_1, bit_2

```
INPUT: Montgomery curve coefficient $A$, elligator bits $bit_1, bit_2$ and elligator counters $(r_1, r_2) \in \mathbb{Z}^2_{256}$
OUTPUT: $(x_1, y_1), (x_2, y_2)$: a basis for $E[3^{e_3}]$
$x_1 \leftarrow$ BasePoint3nDecompression($A, bit_1, r_1$)   // Alg. 4.1
$x_2 \leftarrow$ BasePoint3nDecompression($A, bit_2, r_2$)   // Alg. 4.1
$x_1, y_1 \leftarrow$ CompleteMPoint($A, x_1, 1$)   // Alg. 5.1
$x_2, y_2 \leftarrow$ CompleteMPoint($A, x_2, 1$)   // Alg. 5.1
return $(x_1, y_1), (x_2, y_2)$
```

6 A new tradeoff between ciphertext size and speed

In the SIKE specification submitted to NIST [1], a variant with compressed keys and ciphertexts was described. The compression techniques follow closely [6] and the subsequent performance optimizations [17,18].

An important observation is that during decapsulation, the Fujisaki–Okamoto transform requires an extra expensive validation (Steps 3–9 of Decaps Algorithm 6.1). This validation is implemented by regenerating and recompressing the ciphertext component c_0' so that it can be compared against the received ciphertext c_0. This approach poses two major drawbacks 1) it is more computationally costly and 2) it involves discrete log computations due to recompression. Unfortunately, the discrete log computation employs precomputed tables on the order of megabytes to achieve high speed. Many real applications typically have a memory-constrained IoT device that receives data and performs decapsulation. Deploying those tables would pose a big challenge in terms of flash and stack memory consumption.

We now show how to eliminate those tables while making decapsulation faster. The main idea is to avoid recompressing c_0' by generating a partially compressed ciphertext c_0 during encapsulation. This will allow for a comparison between the ciphertexts in their uncompressed form. The main observation is that instead of sending 3 coefficients in $\mathbb{Z}/2^{e_2}\mathbb{Z}$ as suggested in [6], function `Encaps` will send the 4 coefficients (a_0, a_1, b_0, b_1) as originally proposed in [2] and described in Section 5. In this case, the points $c_0' = (\phi_3'(P_2), \phi_3'(Q_2))$ are left uncompressed and compared directly against the points reconstructed from the ciphertext (c_0, c_1), which contains the 4 coefficients and the information necessary to recover the entangled basis $\{S_1, S_2\}$.

Algorithm 6.1 SIKE KEM = (KeyGen, Encaps, Decaps) [1].
Function G below is SHAKE256.

```
function KeyGen
INPUT: ( )
OUTPUT: $(s, \mathrm{sk}_2, \mathrm{pk}_2)$
    $\mathrm{sk}_2 \leftarrow_R \mathbb{Z}_{2^{e_2}}$
    $\mathrm{pk}_2 \leftarrow \mathtt{isogen}_2(\mathrm{sk}_2)$   // Alg. in 1.3.5 of [1]
    $s \leftarrow_R \{0,1\}^n$
    return $(s, \mathrm{sk}_2, \mathrm{pk}_2)$

end function
```

```
function Encaps
INPUT: $\mathrm{pk}_2$
OUTPUT: $(c, K)$
    $m \leftarrow_R \{0,1\}^n$
    $r \leftarrow G(m \mid\mid \mathrm{pk}_2)$
    $(c_0, c_1) \leftarrow \mathtt{Enc}(\mathrm{pk}_2, m; r)$   // Alg. 1 of [1]
    $K \leftarrow H(m \mid\mid (c_0, c_1))$
    return $((c_0, c_1), K)$
end function
```

```
function Decaps
INPUT: $(s, \mathrm{sk}_2, \mathrm{pk}_2)$, $(c_0, c_1)$
OUTPUT: $K$
    $m' \leftarrow \mathtt{Dec}(\mathrm{sk}_2, (c_0, c_1))$
    $r' \leftarrow G(m' \mid\mid \mathrm{pk}_2)$
    $c_0' \leftarrow \mathtt{isogen}_2(r')$   // Alg. 1 of [1]
    if $c_0' = c_0$ then
        $K \leftarrow H(m' \mid\mid (c_0, c_1))$
    else
        $K \leftarrow H(s \mid\mid (c_0, c_1))$
    end if
    return $K$
end function
```

In particular, naively evaluating $c_0' \stackrel{?}{=} c_0$ using the 4-coefficient approach incurs two equality checks with ciphertexts in their uncompressed form[6]:

$$\begin{aligned} \phi'(P) &\stackrel{?}{=} 3^{e_3}(a_0 S_1 + b_0 S_2) \\ \phi'(Q) &\stackrel{?}{=} 3^{e_3}(a_1 S_1 + b_1 S_2) \end{aligned} \tag{8}$$

Note that the above equality checks involve six scalar multiplications in total (by a_i, b_i and 3^{e_3}) and the two point additions $\{a_i S_1 + b_i S_2\}$ require recovering the y coordinates of $a_i S_1$ and $b_i S_2$.

This can be optimized by adding up both equations to save two scalar multiplications and a point addition. One must be careful in doing that, since it is easy to see that if a plain addition of the equations is performed it becomes easy for an adversary to manipulate the 4 coefficients so that the equality check still holds. The intuition for this is that individual comparisons $\phi_3'(P_2) \stackrel{?}{=} 3^{e_3}(a_0 S_1 + b_0 S_2)$ and $\phi_3'(Q_2)) \stackrel{?}{=} 3^{e_3}(a_1 S_1 + b_1 S_2)$ check the unique decomposition of points $\phi_3'(P_2)$, $\phi_3'(Q_2)$ into the vector space generated by $\{S_1, S_2\}$. On the other hand, the point $\phi_3'(P_2) + \phi_3'(Q_2)$ has projection $a_0 + a_1$ into the subspace $\langle S_1 \rangle$ and $b_0 + b_1$ into the subspace $\langle S_2 \rangle$ and the coefficients (a_0, a_1) and (b_0, b_1) do not need to be unique (only their sum) in this case.

The problem above can be avoided by using a well known idea of randomizing one of the equations during verification (cf. batch signature verification). One can randomize one of the subspaces (e.g., $\langle \phi_3'(Q) \rangle$) before adding up the two equations. If the randomization scalar t is unknown to an adversary, then they don't know how much the values $a_0 + t a_1$ and $b_0 + t b_1$ add up to and are unlikely to fake the coefficients. This leads to the following check

[6] We omit the subscripts for ϕ, P and Q for the sake of simplicity. Note also that the entangled basis generators S_1, S_2 are not cofactor-reduced and extra multiplications by 3^{e_3} appear in Equation 8.

$$\phi'(P) + [t]\phi'(Q) \overset{?}{=} 3^{e_3}([a_0 + a_1 t]S_1 + [b_0 + t b_1]S_2) \tag{9}$$

An adversary will have negligible advantage on forging the coefficients (a_0, a_1, b_0, b_1) in Equation 9. To see this, expand out the LHS in terms of the basis $\{S_1, S_2\}$. Also, simplify the RHS by performing the 3^{e_3} multiplication by S_1 and S_2. This yields the equality $[a_0]R_1 + [b_0]R_2) + [t]([a_1]R_1 + [b_1]R_2 \overset{?}{=} [a_0 + a_1 t]R_1 + [b_0 + b_1 t]R_2$ which holds as long as the respective scalars $a_0 + ta_1$ of R_1 and $b_0 + tb_1$ of R_2 are equal in both sides, since R_1 and R_2 are L.I. Without loss of generality, assume that an adversary applies the replacement $(a_0, a_1, b_0, b_1) \mapsto (a_0 + \alpha, a_1 + \beta, b_0 + \gamma, b_1 + \delta)$ for non-zero $\alpha, \beta, \gamma, \delta$. The equality holds if $a_0 + ta_1 \equiv (a_0 + \alpha) + t(a_1 + \beta) \pmod{2^{e_2}}$ and $b_0 + tb_1 \equiv (b_0 + \gamma) + t(b_1 + \delta) \pmod{2^{e_2}}$. This implies that $-\alpha/\beta \equiv t$ and $-\gamma/\delta \equiv t$. The latter equivalences tell us that for any β (or δ) chosen by the adversary, there is only one single value of α (or γ) that will satisfy the equality for a given $t \in \mathbb{Z}/2^{e_2}\mathbb{Z}$ picked at random by the verifier. Therefore, the success probability of forging the coefficients is at most 2^{-e_2}.

For efficiency reasons and without loss of generality, note that Equation 9 can be rewritten as

Table 1: Benchmark of the **2^{e_2}-torsion basis generation** on an Intel Core i5-6267U clocked at 2.9 GHz (GCC compiler with `-O3` flag, and $\mathbf{s} = \mathbf{m}$ in this implementation).
(*) Benchmarks provided in [18].

technique	source	Kcycles	ratio (previous/current)
2-descent	SIDH v2.0 [6]	23,770*	–
ent. basis + shared ell	Zanon et al. [18]	830.00	29
ent. basis + shared ell + ent. add	**this work**	**1.25**	**662**

$$\begin{aligned} &\phi'(P + [t]Q) \overset{?}{=} \\ &3^{e_3}[a_0 + ta_1](S_1 + [(b_0 + tb_1)(a_0 + ta_1)^{-1}]S_2). \end{aligned} \tag{10}$$

where the LHS applies the linearity of the isogeny to move the randomization into the initial curve and the RHS assumes that $(a_0 + ta_1)$ is invertible and can be factored out. In order to perform the LHS evaluation, a key idea is that one could set the randomization scalar to be $t := sk_2$ since sk_2 is private and thus unpredictable by the adversary. Moreover, the point $K_2 := P + [sk_2]Q$ is exactly the kernel generator of Alice's isogeny, which was already computed during key generation (Step 3 of `KeyGen` Algorithm 6.1). Therefore, Alice can append the x-coordinate of K_2 to her secret key and reuse it during decapsulation. Overall, computing the LHS would boil down to computing the isogeny ϕ'_3 by only carrying the point K_2 (instead of carrying P and Q) and no scalar multiplication is needed so far.

Setting $t := sk_2$ also benefits the RHS as it becomes $3^{e_3}[a_0 + sk_2 a_1](S_1 + [(b_0 + sk_2 b_1)(a_0 + sk_2 a_1)^{-1}]S_2$, which coincides (up to a scalar factor $[a_0 + sk_2 a_1]$) with the generator point of Alice's final isogeny already computed in Step 2 of `Decaps` in Algorithm 6.1. Therefore, since the RHS is readily available, no computation at all is needed. Finally, for the equality check to hold, the LHS just needs to accomodate the scalar $a_0 + sk_2 a_1$ and one single scalar multiplication is needed overall (as opposed to six multiplications in SIKE Round 2 submission).

The experimental results show that the technique proposed in this section improves decapsulation speed by a factor ≈ 2 in exchange of a ≈ 12% larger ciphertext and a larger secret key by one field element.

Table 2: Benchmark of the **3^{e_3}-torsion basis generation** on an Intel Core i5-6267U clocked at 2.9 GHz (GCC compiler with `-O3` flag, and $\mathbf{s} = \mathbf{m}$ in this implementation).
(*) Benchmarks provided in [18].

technique	source	Kcycles	ratio (previous/current)
Costello *et. al.* (based on 3-descent)	SIDH v2.0 [6]	19,980*	–
shared ell.	Zanon et al. [18]	7,260	2.8
ext. shared ell. + avoid mult. by cof.	**this work**	**1,175**	**6.2**

7 Implementation and experimental results

The techniques introduced in the previous sections have been implemented on top of the official SIKE optimized C + ASM implementation submitted to the second round of NIST [1]. The implementation provided in [1] offers a faster field arithmetic due to recent improvements in assembly and our results also get benefit from that when compared to [18], which builds on top of a previous implementation of SIKE. Our methodology consists of measuring the cost of the operations in cycle counts which allows us to compare directly against [17,18] as we run the algorithms over the same processor model. The cycle count is an average over 5 thousand executions.

Table 1 shows the results for the fast binary basis generation during decompression which runs internally to SIKE's decapsulation procedure (`Decaps` in Algorithm 6.1). The results here are particularly impacted by the techniques of entangled addition introduced in Section 2 and of faster entangled basis generation introduced in Section 3. Note that binary basis generation was originally a considered bottleneck with about 24M cycles. The reduction to about 1k cycles (or even less for smaller SIKE primes) can be considered as an effective solution for this bottleneck.

We also show the results for the faster ternary basis generation during decompression in Table 2, which impacts SIKE's encapsulation procedure (`Encaps` in Algorithm 6.1). The new figures are impacted by the techniques of extended elligator with extra bits introduced in Section 4 and removal of a cofactor multiplication introduced in Section 5.

It is worth mentioning that although we achieve a much better speed-up factor in the binary torsion basis generation saving almost a million cycles, we were able to save even more cycles for the ternary case ($\approx 6M$ in SIKEp751), since the latter case seems to not have been fully optimized by previous works.

Table 3 gives a complete comparison that includes high-level operations of SIKE. The experimental results for the ciphertext size *vs* decapsulation speed tradeoff introduced in Section 6 is also given. In this case, we were able to get a factor 2 speedup in exchange for a $\approx 12.5\%$ larger ciphertext and one extra field element in the secret key. One can also see that although a big improvement was achieved for individual basis generation, they get amortized in the presence of even more expensive operations related to key compression, i.e., discrete logs and pairings. The overall decapsulation is improved by a factor of 1.27 to 2.0 depending on the technique employed and the overall encapsulation is improved by a factor of 1.19 to 1.21 for SIKEp751. For the sake of completeness we also give extra comparisons for the binary basis generation including the new SIKE primes in Table 4 of Appendix A.

Table 3: Benchmarks in 10^6 cycles on an Intel Core i5-6267U clocked at 2.9 GHz (GCC compiler with `-O3` flag, and $\mathbf{s} = \mathbf{m}$ in this implementation).
(*) A few megabytes saved in storage for the entity running decapsulation since it avoids discrete log computations.

	2^{e_2}-torsion			**3^{e_3}-torsion**		
operation	Zanon et al. [18]	ours	ratio	Zanon et al. [18]	ours	ratio
basis generation	0.830	0.00125	**662.0**	7.260	1.175	**6.2**
decompression	8.970	6.223	**1.44**	12.910	6.888	**1.87**
encapsulation	89.110	74.684	**1.19**	–	–	–
decapsulation	90.130	70.856	**1.27**	–	–	–
encapsulation tradeoff (*)	–	73.897	**1.21**	–	–	–
decapsulation tradeoff (*)	–	46.077	**2.04**	–	–	–

8 Conclusion

In this work, we fully remove one of the SIDH/SIKE's bottlenecks, i.e., the binary torsion basis generation during decompression. This has an impact on SIKE decapsulation algorithm. We also provide a faster ternary basis generation where about 6M cycles are saved for SIKEp751. This impacts the encapsulation algorithm. Furthermore, we introduce a tradeoff where ciphertexts are increased by about 12% and decapsulation speed improves by a factor of ≈ 2 compared to SIKE Round 2 submission and more importantly get rid of megabytes from precomputed tables used in discrete log computation required in previous SIKE decapsulation. This makes the decapsulation operation more friendly to IoT devices. A natural next step is to combine the results proposed here with the ones in [14], which achieves a much faster pairing computation.

Acknowledgement

This work is supported in part by NSERC, CryptoWorks21, Canada First Research Excellence Fund, Public Works and Government Services Canada, and the Royal Bank of Canada.

References

1. R Azarderakhsh, M Campagna, C Costello, LD Feo, B Hess, A Jalali, D Jao, B Koziel, B LaMacchia, P Longa, M Naehrig, G Pereira, J Renes, V Soukharev, and D Urbanik. Supersingular isogeny key encapsulation. *Submission to the 2nd Round of the NIST Post-Quantum Standardization project*, 2019.
2. R. Azarderakhsh, D. Jao, K. Kalach, B. Koziel, and C. Leonardi. Key compression for isogeny-based cryptosystems. In *Proceedings of the 3rd ACM International Workshop on ASIA Public-Key Cryptography*, pages 1–10. ACM, 2016.
3. D. J. Bernstein, M. Hamburg, A. Krasnova, and T. Lange. Elligator: Elliptic-curve points indistinguishable from uniform random strings. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 967–980. ACM, 2013.
4. W. Castryck, T. Lange, C. Martindale, L. Panny, and J. Renes. Csidh: An efficient post-quantum commutative group action. In Thomas Peyrin and Steven Galbraith, editors, *Advances in Cryptology – ASIACRYPT 2018*, pages 395–427, Cham, 2018. Springer International Publishing.

5. Denis X. Charles, Kristin E. Lauter, and Eyal Z. Goren. Cryptographic hash functions from expander graphs. *Journal of Cryptology*, 22(1):93–113, Jan 2009.
6. C. Costello, D. Jao, P. Longa, M. Naehrig, J. Renes, and D. Urbanik. Efficient compression of SIDH public keys. In *Advances in Cryptology – Eurocrypt 2017*, number 10210 in Lecture Notes in Computer Science, pages 679–706, Paris, France, 2017. Springer.
7. L. De Feo, D. Jao, and J. Plût. Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. *Journal of Mathematical Cryptology*, 8(3):209–247, 2014.
8. Armando Faz-Hernández, Julio López, Eduardo Ochoa-Jiménez, and Francisco Rodríguez-Henríquez. A faster software implementation of the supersingular isogeny diffie-hellman key exchange protocol. *IEEE Transactions on Computers*, 2017.
9. E. Fujisaki and T. Okamoto. Secure integration of asymmetric and symmetric encryption schemes. *Journal of Cryptology*, 26(1):80–101, Jan 2013.
10. S.D̃. Galbraith, C. Petit, B. Shani, and Y. Ti. On the security of supersingular isogeny cryptosystems. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *Advances in Cryptology – ASIACRYPT 2016*, pages 63–91, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.
11. D. Jao and L. De Feo. Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. In *Post-quantum cryptography*, volume 7071 of *Lecture Notes in Comput. Sci.*, pages 19–34. Springer, Heidelberg, 2011.
12. D. Jao and L. De Feo. Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. In *Post-Quantum Cryptography – PQCrypto 2011*, number 7071 in Lecture Notes in Computer Science, pages 19–34, Taipei, Taiwan, 2011. Springer.
13. D. Kirkwood, B. Lackey, J. McVey, M. Motley, J. Solinas, and D. Tuller. Failure is not an option: Standardization issues for post-quantum key agreement. https://csrc.nist.gov/groups/ST/post-quantum-2015/presentations/session7-motley-mark.pdf, 2015.
14. Michael Naehrig and Joost Renes. Dual isogenies and their application to public-key compression for isogeny-based cryptography. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 243–272. Springer, 2019.
15. National Institute of Standards and Technology. Post-quantum cryptography. https://www.nist.gov/pqcrypto/, 2019.
16. SIKE. Supersingular isogeny key encapsulation, 2017. https://sike.org.
17. G. Zanon, M. Simplicio Jr., G. Pereira, J. Doliskani, and P. Barreto. Faster isogeny-based compressed key agreement. In *International Workshop on Post-Quantum Cryptography*, pages 248–268. Springer, 2018.
18. G. Zanon, M. Simplicio Jr., G. Pereira, J. Doliskani, and P. Barreto. Faster key compression for isogeny-based cryptosystems. *IEEE Transactions on Computers*, 68:688–701, 2018.

A Additional performance experiments

In order to illustrate our techniques for different SIKE primes that are not present in the previous compression works, we give extra benchmarks in Table 4.

Table 4: Benchmark for 2^{e_2}-torsion basis generation in cycles on an Intel Core i5-6267U clocked at 2.9 GHz (GCC compiler with `-O3` flag, and $\mathbf{s} = \mathbf{m}$ in this implementation).
(*) Cycle counts based on our implementation of [18] since they only provide results for p751.

prime	binary basis generation		ratio
	Zanon et al. [18]	this work	
p434	186,192*	852	**219**
p503	249,483*	1083	**230**
p610	420,331*	1183	**355**
p751	830,000	1254	**662**

B Auxiliary algorithms

Algorithm B.1 `xDBLADD`: combined coordinate doubling and differential addition [16]

```
INPUT: $(X_P : Z_P)$, $(X_Q : Z_Q)$, $(X_{Q-P} : Z_{Q-P})$, and $A_{24} = A + 2/4$
OUTPUT: $(X_{[2]P} : Z_{[2]P})$, $(X_{P+Q} : Z_{P+Q})$
$t_0 \leftarrow X_P + Z_P$
$t_1 \leftarrow X_P - Z_P$
$X_{[2]P} \leftarrow t_0^2$
$t_2 \leftarrow X_Q - Z_Q$
$X_{P+Q} \leftarrow X_Q + Z_Q$
$t_0 \leftarrow t_0 \cdot t_2$
$Z_{[2]P} \leftarrow t_1^2$
$t_1 \leftarrow t_1 \cdot X_{P+Q}$
$t_2 \leftarrow X_{[2]P} - Z_{[2]P}$
$X_{[2]P} \leftarrow X_{[2]P} \cdot Z_{[2]P}$
$X_{P+Q} \leftarrow A_{24} + t_2$
$Z_{P+Q} \leftarrow t_0 - t_1$
$Z_{[2]P} \leftarrow X_{P+Q} + Z_{[2]P}$
$X_{P+Q} \leftarrow t_0 + t_1$
$Z_{[2]P} \leftarrow Z_{[2]P} \cdot t_2$
$Z_{P+Q} \leftarrow Z_{P+Q}^2$
$X_{P+Q} \leftarrow X_{P+Q}^2$
$Z_{P+Q} \leftarrow X_{Q-P} \cdot Z_{P+Q}$
$X_{P+Q} \leftarrow Z_{Q-P} \cdot X_{P+Q}$
return $\{(X_{[2]P} : Z_{[2]P}), (X_{P+Q} : Z_{P+Q})\}$
```

Algorithm B.2 Ladder3pt: Three point ladder [16,7]

INPUT: $m = (m_{\ell-1}, \ldots, m_0)_2 \in \mathbb{Z}$, (x_P, x_Q, x_{Q-P}), and A
OUTPUT: $(X_{P+[m]Q} : Z_{P+[m]Q})$

```
$\big((X_0 : Z_0), (X_1 : Z_1), (X_2 : Z_2)\big) \leftarrow \big((x_Q : 1), (x_P : 1), (x_{Q-P} : 1)\big)$
$A_{24} \leftarrow (A+2)/4$
for $i = 0$ to $\ell - 1$ do
    if $m_i = 1$ then
        $\big((X_0 : Z_0), (X_1 : Z_1)\big) \leftarrow \texttt{xDBLADD}\big((X_0 : Z_0), (X_1 : Z_1), (X_2 : Z_2), A_{24}\big)$
    else
        $\big((X_0 : Z_0), (X_2 : Z_2)\big) \leftarrow \texttt{xDBLADD}\big((X_0 : Z_0), (X_2 : Z_2), (X_1 : Z_1), A_{24}\big)$
    end if
end for
return $(X_1 : Z_1)$
```

Algorithm B.3 Entangled basis generation for $E[2^{e_2}](\mathbb{F}_{p^2}) : y^2 = x^3 + Ax^2 + x$ [18]

INPUT: $A = a + bi \in \mathbb{F}_{p^2}$ and the public parameters $u_0 \in \mathbb{F}_{p^2} : u = u_0^2 \in \mathbb{F}_{p^2} \backslash \mathbb{F}_p$; tables T_1, T_2 of pairs $(r \in \mathbb{F}_p, v = 1/(1+ur^2) \in \mathbb{F}_{p^2})$ of QNR and QR.
OUTPUT: $\{S_1, S_2\}$ such that $\langle [3^{e_3}]S_1, [3^{e_3}]S_2 \rangle = E[2^{e_2}](\mathbb{F}_{p^2})$, a bit bit indicating the quadraticity of A and the table entry for r

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

$s \leftarrow z^{(p+1)/4}$
 // select proper table by testing quadraticity of $A$
if $s^2 \overset{?}{=} z$ then
    $bit \leftarrow 1$;   $T \leftarrow T_1$   // $A$ is a QR
else
    $bit \leftarrow 0$;   $T \leftarrow T_2$   // $A$ is a QNR
end if
repeat
    lookup next entry $(r, v)$ from $T$
    $x \leftarrow -A \cdot v$
    $t \leftarrow x \cdot (x^2 + A \cdot x + 1)$   // test quadraticity of $t = c + di$
    $z \leftarrow c^2 + d^2$,   $s \leftarrow z^{(p+1)/4}$
until $s^2 = z$
$z \leftarrow (c+s)/2$
$\alpha \leftarrow z^{(p+1)/4}$
$\beta \leftarrow d \cdot (2\alpha)^{-1}$
$y \leftarrow (\alpha^2 \overset{?}{=} z)\ \alpha + \beta i : -\beta - \alpha i$   // $y \leftarrow \sqrt{x^3 + A \cdot x^2 + x}$
return $S_1 \leftarrow (x, y)$, $S_2 \leftarrow (ur^2 x, u_0 r y)$, $bit, r$
```

Algorithm B.4 EntangledBasisDecompression [18]: Entangled basis generation with shared Elligator for $E[2^{e_2}](\mathbb{F}_{p^2}) : y^2 = x^3 + Ax^2 + x$

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 $u_0 \in \mathbb{F}_{p^2} : u = u_0^2 \in \mathbb{F}_{p^2} \backslash \mathbb{F}_p$; 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 $\langle [3^{e_3}]S_1, [3^{e_3}]S_2 \rangle = E[2^{e_2}](\mathbb{F}_{p^2})$

```
$T \leftarrow (bit \overset{?}{=} 1)\ T_1 : T_2$   // select proper table according to $A$'s quadraticity
lookup entry $v$ corresponding to $r$ on $T$
$x \leftarrow -A \cdot v$
$t \leftarrow x \cdot (x^2 + A \cdot x + 1)$
test quadraticity of $t = c + di$
$z \leftarrow c^2 + d^2$
$s \leftarrow z^{(p+1)/4}$
**if** $s^2 \neq z$ **then**
    Abort: invalid input parameters $(bit, r)$ received
**end if**
$z \leftarrow (c + s)/2$
$\alpha \leftarrow z^{(p+1)/4}$
$\beta \leftarrow d \cdot (2\alpha)^{-1}$
$y \leftarrow (\alpha^2 \overset{?}{=} z)\ \alpha + \beta i : -\beta - \alpha i$   // $y \leftarrow \sqrt{x^3 + A \cdot x^2 + x}$
**return** $S_1 \leftarrow (x, y)$, $S_2 \leftarrow (u \cdot r^2 \cdot x, u_0 \cdot r \cdot y)$
```

Algorithm B.5 *BasePoint3n_decompression* [18]: 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

INPUT: Curve coefficient $A \in \mathbb{F}_{p^2}$

- Elligator counter $r \in \mathbb{Z}$ informing the correct table entry
- Public table T of elligator values $v = 1/(1 + ur^2) \in \mathbb{F}_{p^2}$

OUTPUT: Projective point (x, z) of order 3^{e_3} from r

```
$v \leftarrow T[r + 1]$
$x \leftarrow -A \cdot v$
$yy \leftarrow x + A$
$yy \leftarrow x \cdot yy + 1$
$yy \leftarrow x \cdot yy$   // $yy = x^3 + Ax^2 + x = a + bi$
$N \leftarrow a^2 + b^2$
$z \leftarrow N^{(p+1)/4}$
**if** $z^2 \neq N$ **then**
    $x \leftarrow -x - A$
**end if**
$x, z \leftarrow [2^{e_2}](x, 1)$
**return** $(x, z)$
```