Efficient Compression of SIDH Public Keys

Craig Costello[1(✉)], David Jao[2,3], Patrick Longa[1], Michael Naehrig[1], Joost Renes[4], and David Urbanik[2]

[1] Microsoft Research, Redmond, WA, USA
{craigco,plonga,mnaehrig}@microsoft.com
[2] Centre for Applied Cryptographic Research, University of Waterloo, Waterloo, ON, Canada
{djao,dburbani}@uwaterloo.ca
[3] evolutionQ, Inc., Waterloo, ON, Canada
david.jao@evolutionq.com
[4] Digital Security Group, Radboud University, Nijmegen, The Netherlands
j.renes@cs.ru.nl

Abstract. Supersingular isogeny Diffie-Hellman (SIDH) is an attractive candidate for post-quantum key exchange, in large part due to its relatively small public key sizes. A recent paper by Azarderakhsh, Jao, Kalach, Koziel and Leonardi showed that the public keys defined in Jao and De Feo's original SIDH scheme can be further compressed by around a factor of two, but reported that the performance penalty in utilizing this compression blew the overall SIDH runtime out by more than an order of magnitude. Given that the runtime of SIDH key exchange is currently its main drawback in relation to its lattice- and code-based post-quantum alternatives, an order of magnitude performance penalty for a factor of two improvement in bandwidth presents a trade-off that is unlikely to favor public-key compression in many scenarios.

In this paper, we propose a range of new algorithms and techniques that accelerate SIDH public-key compression by more than an order of magnitude, making it roughly as fast as a round of standalone SIDH key exchange, while further reducing the size of the compressed public keys by approximately 12.5%. These improvements enable the practical use of compression, achieving public keys of only 330 bytes for the concrete parameters used to target 128 bits of quantum security and further strengthens SIDH as a promising post-quantum primitive.

Keywords: Post-quantum cryptography · Diffie-Hellman key exchange · Supersingular elliptic curves · Isogenies · SIDH · Public-key compression · Pohlig-Hellman algorithm

D. Jao and D. Urbanik—Partially supported by NSERC, CryptoWorks21, and Public Works and Government Services Canada.

J. Renes—Partially supported by the Technology Foundation STW (project 13499 – TYPHOON & ASPASIA), from the Dutch government. Part of this work was done while Joost was an intern at Microsoft Research.

J.-S. Coron and J.B. Nielsen (Eds.): EUROCRYPT 2017, Part I, LNCS 10210, pp. 679–706, 2017.
DOI: 10.1007/978-3-319-56620-7_24

1 Introduction

In their February 2016 report on post-quantum cryptography [6], the United States National Institute of Standards and Technology (NIST) stated that *"It seems improbable that any of the currently known [public-key] algorithms can serve as a drop-in replacement for what is in use today,"* citing that one major challenge is that quantum resistant algorithms have larger key sizes than the algorithms they will replace. While this statement is certainly applicable to many of the lattice- and code-based schemes (e.g., LWE encryption [24] and the McEliece cryptosystem [19]), Jao and De Feo's 2011 supersingular isogeny Diffie-Hellman (SIDH) proposal [15] is one post-quantum candidate that could serve as a drop-in replacement to existing Internet protocols. Not only are high-security SIDH public keys smaller than their lattice- and code-based counterparts, they are even smaller than some of the traditional (i.e., finite field) Diffie-Hellman public keys.

SIDH Public-Key Compression. The public keys defined in the original SIDH papers [8,15] take the form

$$\mathtt{PK} = (E, P, Q),$$

where $E/\mathbb{F}_{p^2} : y^2 = x^3 + ax + b$ is a supersingular elliptic curve, $p = n_A n_B \pm 1$ is a large prime, the cardinality of E is $\#E(\mathbb{F}_{p^2}) = (p \mp 1) = (n_A n_B)^2$, and depending on whether the public key corresponds to Alice or Bob, the points P and Q either both lie in $E(\mathbb{F}_{p^2})[n_A]$, or both lie in $E(\mathbb{F}_{p^2})[n_B]$. Since P and Q can both be transmitted via their x-coordinates (together with a sign bit that determines the correct y-coordinate), and the curve can be transmitted by sending the two $\mathbb{F}_{p^2}$ elements a and b, the original SIDH public keys essentially consist of four $\mathbb{F}_{p^2}$ elements, and so are around $8 \log p$ bits in size.

A recent paper by Azarderakhsh, Jao, Kalach, Koziel and Leonardi [2] showed that it is possible to compress the size of SIDH public keys to around $4 \log p$ bits as follows. Firstly, to send the supersingular curve E, they pointed out that one can send the j-invariant $j(E) \in \mathbb{F}_{p^2}$ rather than $(a, b) \in \mathbb{F}_{p^2}^2$, and showed how to recover a and b (uniquely, up to isomorphism) from $j(E)$ on the other side. Secondly, for $n \in \{n_A, n_B\}$, they showed that since $E(\mathbb{F}_{p^2})[n] \cong \mathbb{Z}_n \times \mathbb{Z}_n$, an element in $E(\mathbb{F}_{p^2})[n]$ can instead be transmitted by sending two scalars $(\alpha, \beta) \in \mathbb{Z}_n \times \mathbb{Z}_n$ that determine its representation with respect to a basis of the torsion subgroup. This requires that Alice and Bob have a way of arriving at the same basis for $E(\mathbb{F}_{p^2})[n]$. Following [2], we note that it is possible to decompose points into their $\mathbb{Z}_n \times \mathbb{Z}_n$ representation since for well-chosen SIDH parameters, $n = \ell^e$ is always smooth, which means that discrete logarithms in order n groups can be solved in polynomial time using the Pohlig-Hellman algorithm [23]. Given that such SIDH parameters have $n_A \approx n_B$ (see [15]), it follows that $n \approx \sqrt{p}$ and that sending elements of $E(\mathbb{F}_{p^2})[n]$ as two elements of $\mathbb{Z}_n$ (instead of an element in $\mathbb{F}_{p^2}$) cuts the bandwidth required to send torsion points in half.

Although passing back and forth between (a, b) and $j(E)$ to (de)compress the curve is relatively inexpensive, the compression of the points P and Q requires three computationally intensive steps:

- *Step 1 – Constructing the n-torsion basis.* During both compression and decompression, Alice and Bob must, on input of the curve E, use a deterministic method to generate the same two-dimensional basis $\{R_1, R_2\} \in E(\mathbb{F}_{p^2})[n]$. The method used in [2] involves systematically sampling candidate points $R \in E(\mathbb{F}_{p^2})$, performing cofactor multiplication by h to move into $E(\mathbb{F}_{p^2})[n]$, and then testing whether or not $[h]R$ has "full" order n (and, if not, restarting).
- *Step 2 – Pairing computations.* After computing a basis $\{R_1, R_2\}$ of the group $E(\mathbb{F}_{p^2})[n]$, the task is to decompose the point P (and identically, Q) as $P = [\alpha_P]R_1 + [\beta_P]R_2$ and determine (α_P, β_P). While this could be done by solving a two-dimensional discrete logarithm problem (DLP) directly on the curve, Azarderakhsh *et al.* [2] use a number of Weil pairing computations to transform these instances into one-dimensional finite field DLPs in $\mu_n \subset \mathbb{F}^*_{p^2}$.
- *Step 3 – Solving discrete logarithms in μ_n.* The last step is to repeatedly use the Pohlig-Hellman algorithm [23] to solve DLPs in μ_n, and to output the four scalars α_P, β_P, α_Q and β_Q in $\mathbb{Z}_n$.

Each one of these steps presents a significant performance drawback for SIDH public-key compression. Subsequently, Azarderakhsh *et al.* report that, at interesting levels of security, each party's individual compression latency is more than a factor of ten times the latency of a full round of uncompressed key exchange [2, Sect. 5].

Our Contributions. We present a range of new algorithmic improvements that decrease the total runtime of SIDH compression and decompression by an order of magnitude, bringing its performance close to that of a single round of SIDH key exchange. We believe that this makes it possible to consider public-key compression a default choice for SIDH, and it can further widen the gap between the key sizes resulting from practical SIDH key exchange implementations and their code- and lattice-based counterparts.

We provide a brief overview of our main improvements with respect to the three compression steps described above. All known implementations of SIDH (e.g., [1,7,8]) currently choose $n_A = \ell_A^{e_A} = 2^{e_A}$ and $n_B = \ell_B^{e_B} = 3^{e_B}$ for simplicity and efficiency reasons, so we focus on $\ell \in \{2, 3\}$ below; however, unless specified otherwise, we note that all of our improvements will readily apply to other values of ℓ.

- *Step 1 – Constructing the n-torsion basis.* We make use of some results arising from explicit 2- and 3-descent of elliptic curves to avoid the need for the expensive cofactor multiplication that tests the order of points. These results characterize the images of the multiplication-by-2 and multiplication-by-3 maps on E, and allow us to quickly generate points that are elements of $E(\mathbb{F}_{p^2}) \setminus [2]E(\mathbb{F}_{p^2})$ and $E(\mathbb{F}_{p^2}) \setminus [3]E(\mathbb{F}_{p^2})$. Therefore, we no longer need

to check the order of (possibly multiple!) points using a full-length scalar multiplication by $n_A n_B$, but instead are *guaranteed* that one half-length cofactor multiplication produces a point of the correct order. For our purposes, producing points in $E \setminus [2]E$ is as easy as generating elliptic curve points whose x-coordinates are non-square (this is classical, e.g., [14, Chap. 1 (Sect. 4), Theorem 4.1]). On the other hand, to efficiently produce points in $E \setminus [3]E$, we make use of the analogous characteristic described in more recent work on explicit 3-descent by Schaefer and Stoll [26]. Combined with a tailored version of the Elligator 2 encoding [5] for efficiently generating points on E, this approach gives rise to highly efficient n-torsion basis generation. This is described in detail in Sect. 3.

– *Step 2 – Pairing computations.* We apply a number of optimizations from the literature on elliptic curve pairings in order to significantly speed up the runtime of all pairing computations. Rather than using the Weil pairing (as was done in [2]), we use the more efficient Tate pairing [4,10]. We organize the five pairing computations that are required during compression in such a way that only two Miller functions are necessary. Unlike all of the prior work done on optimized pairing computation, the pairings used in SIDH compression cannot take advantage of torsion subgroups that lie in subfields, which means that fast explicit formulas for point operations and Miller line computations are crucial to achieving a fast implementation. Subsequently, we derive new and fast inversion-free explicit formulas for computing pairings on supersingular curves, specific to the scenario of SIDH compression. Following the Miller loops, we compute all five final exponentiations by exploiting a fast combination of Frobenius operations together with either fast repeated cyclotomic squarings (from [31]) or our new formulas for enhanced cyclotomic cubing operations. The pairing optimizations are described in Sect. 4.
– *Step 3 – Solving discrete logarithms in μ_n.* All computations during the Pohlig-Hellman phase take place in the subgroup μ_n of the multiplicative group $G_{p+1} \subset \mathbb{F}_{p^2}^*$ of order $p+1$, where we take advantage of the fast cyclotomic squarings and cubings mentioned above, as well as the fact that $\mathbb{F}_{p^2}$ inversions are simply conjugations, so come almost for free (see Sect. 5.1). On top of this fast arithmetic, we build an improved version of the Pohlig-Hellman algorithm that exploits windowing methods to solve the discrete logarithm instances with lower asymptotic complexity than the original algorithm. For the concrete parameters, the new algorithm is approximately 14× (resp. 10×) faster in $\mu_{2^{372}}$ (resp. $\mu_{3^{239}}$), while having very low memory requirements (see Tables 1 and 2). This is all described in more detail in Sect. 5.
– *Improved compression.* By normalizing the representation of P and Q in $\mathbb{Z}_n^4$, we are able to further compress this part of the public key representation into $\mathbb{Z}_n^3$. Subsequently, our public keys are around $\frac{7}{2}\log p$ bits, rather than the $4\log p$ bits achieved in [2]. To the best of our knowledge, this is as far as SIDH public keys can be compressed in practice. This is explained in Sect. 6.1.
– *Decompression.* The decompression algorithm – which involves only the first of the three steps above and a double-scalar multiplication – is also accelerated in this work. In particular, on top of the faster torsion basis generation,

we show that the double-scalar multiplications can be absorbed into the shared secret computation. This makes them essentially free of cost. This is described in Sect. 6.2.

The combination of the three main improvements mentioned above, along with a number of further optimizations described in the rest of this paper, yields enhanced compression software that is an order of magnitude faster than the initial software benchmarked in [2].

The Compression Software. We wrote the new suite of algorithms in plain C and incorporated the compression software into the SIDH library recently made available by Costello, Longa and Naehrig [7]; their software uses a curve with $\log p = 751$ that currently offers around 192 bits of classical security and 128 bits of quantum security. The public keys in their uncompressed software were $6 \log p = 564$ bytes, while the compressed public keys resulting from our software are $\frac{7}{2} \log p = 330$ bytes. The software described in this paper can be found in the latest release of the SIDH library (version 2.0) at https://www.microsoft.com/en-us/research/project/sidh-library/.

Although our software is significantly faster than the previous compression benchmarks given by Azarderakhsh *et al.* [2], we believe that the most meaningful benchmarks we can present are those that compare the latency of our optimized SIDH compression to the latency of the state-of-the-art key generation and shared secret computations in [7]. This gives the reader (and the PQ audience at large) an idea of the cost of public-key compression when both the raw SIDH key exchange and the optional compression are optimized to a similar level. We emphasize that although the SIDH key exchange software from [7] targeted one isogeny class at one particular security level, and therefore so does our compression software, all of our improvements apply identically to curves used for SIDH at other security levels, especially if the chosen isogeny degrees remain (powers of) 2 and 3. Moreover, we expect that the relative cost of compressed SIDH to uncompressed SIDH will stay roughly consistent across different security levels, and that our targeted benchmarks therefore give a good gauge on the current state-of-the-art.

It is important to note that, unlike the SIDH software from [7] that uses private keys and computes shared secrets, by definition our public-key compression software only operates on public data[1]. Thus, while we call several of their constant-time functions when appropriate, none of our functions need to run in constant-time.

Remark 1 (Ephemeral SIDH). A recent paper by Galbraith, Petit, Shani and Ti [11] gives, among other results, a realistic and serious attack on instantiations of SIDH that reuse static private/public key pairs. Although direct public-key validation in the context of isogeny-based cryptography is currently non-trivial, there are methods of indirect public-key validation (see, e.g., [11,17]) that

[1] There is a minor caveat here in that we absorb part of the decompression into the shared secret computation, which uses the constant-time software from [7] – see Sect. 6.

mirror the same technique proposed by Peikert [22, Sect. 5–6] in the context of lattice-based cryptography, which is itself a slight modification of the well-known Fujisaki-Okamoto transform [9]. At present, the software from [7] only supports secure *ephemeral* SIDH key exchange, and does not yet include sufficient (direct or indirect) validation that allows the secure use of static keys. Thus, since our software was written around that of [7], we note that it too is only written for the target application of ephemeral SIDH key exchange. In this case attackers are not incentivized to tamper with public keys, so we can safely assume throughout this paper that all public keys are well-formed. Nevertheless, we note that the updated key exchange protocols in [9,11,17,22] still send values that can be compressed using our algorithms. On a related note, we also point out that our algorithms readily apply to the other isogeny-based cryptosystems described in [8] for which the compression techniques were detailed in [2]. In all of these other scenarios, however, the overall performance ratios and relative bandwidth savings offered by our compression algorithms are likely to differ from those we report for ephemeral SIDH.

Remark 2 (Trading speed for simplicity and space). Since the compression code in our software library only runs on public data, and therefore need not run in constant-time, we use a variable-time algorithm for field inversions (a variant of the extended binary GCD algorithm [16]) that runs faster than the typical exponentiation method (via Fermat's little theorem). Although inversions are used sparingly in our code and are not the bottleneck of the overall compression runtime, we opted to add a single variable-time algorithm in this case. However, during the design of our software library, we made several decisions in the name of simplicity that inevitably hampered the performance of the compression algorithms.

One such performance sacrifice is made during the computation of the torsion basis points in Sect. 3, where tests of quadratic and cubic residuosity are performed using field exponentiations. Here we could use significantly faster, but more complicated algorithms that take advantage of the classic quadratic and cubic reciprocity identities. Such algorithms require intermediate reductions modulo many variable integers, and a reasonably optimized generic reduction routine would increase the code complexity significantly. These tests are also used sparingly and are not the bottleneck of public-key compression, and in this case, we deemed the benefits of optimizing them to be outweighed by their complexity. A second and perhaps the most significant performance sacrifice made in our software is during the Pohlig-Hellman computations, where our windowed version of the algorithm currently fixes small window sizes in the name of choosing moderate space requirements. If larger storage is permitted, then Sutherland's analysis of an optimized version of the Pohlig-Hellman algorithm [32] shows that this phase could be sped up significantly (see Sect. 5). But again, the motivation to push the limits of the Pohlig-Hellman phase is stunted by the prior (pairing computation) phase being the bottleneck of the overall compression routine. Finally, we note that the probabilistic components of the torsion basis generation phase (see Sect. 3) lend themselves to an amended definition of the compressed

public keys, where the compressor can send a few extra bits or bytes in their public key to make for a faster and deterministic decompression. For simplicity (and again due to this phase not being the bottleneck of compression), we leave this more complicated adaptation to future consideration.

2 Preliminaries

Here we restrict only to the background that is necessary to understand this paper, i.e., only what is needed to define SIDH public keys. We refer to the extended paper by De Feo, Jao and Plût [8] for a background on the SIDH key exchange computations, and for the rationale behind the parameters given below.

SIDH Public Keys. Let $p = n_A n_B \pm 1$ be a large prime and $E/\mathbb{F}_{p^2}$ be a supersingular curve of cardinality $\#E(\mathbb{F}_{p^2}) = (p \mp 1)^2 = (n_A n_B)^2$. Let $n_A = \ell_A^{e_A}$ and $n_B = \ell_B^{e_B}$. Henceforth we shall assume that $\ell_A = 2$ and $\ell_B = 3$, which is the well-justified choice made in all known implementations to date [1,7,8]; however, unless specified otherwise, we note that the optimizations in this paper will readily apply to other reasonable choices of ℓ_A and ℓ_B. When the discussion is identical irrespective of ℓ_A or ℓ_B, we will often just use ℓ; similarly, we will often just use n when the discussion applies to both n_A and n_B. In general, $E/\mathbb{F}_{p^2}$ is specified using the short Weierstrass model $E/\mathbb{F}_{p^2} : y^2 = x^3 + ax + b$, so is defined by the two $\mathbb{F}_{p^2}$ elements a and b.

During one round of SIDH key exchange, Alice computes her public key as the image E_A of her secret degree-n_A isogeny ϕ_A on a fixed public curve E_0, for example $E_0/\mathbb{F}_{p^2} : y^2 = x^3 + x$, along with the images of ϕ_A on the two public points P_B and Q_B of order n_B, i.e., the points $\phi_A(P_B)$ and $\phi_A(Q_B)$. Bob performs the analogous computation applying his secret degree-n_B isogeny ϕ_B to E_0 to produce the image curve E_B and to produce the images of the public points P_A and Q_A, both of order n_A. In both cases, the public keys are of the form $\mathtt{PK} = (E, P, Q)$, where $E/\mathbb{F}_{p^2}$ is a supersingular elliptic curve transmitted as two $\mathbb{F}_{p^2}$ elements, and P and Q are points on E that are each transmitted as one $\mathbb{F}_{p^2}$ element corresponding to the x-coordinate, along with a single bit that specifies the choice of the corresponding y-coordinate. Subsequently, typical SIDH public keys are specified by 4 $\mathbb{F}_{p^2}$ elements (and two sign bits), and are therefore around $8 \log p$ bits in length.

General SIDH Compression. We now recall the main ideas behind the SIDH public key compression recently presented by Azarderakhsh, Jao, Kalach, Koziel and Leonardi [2]. Their first idea involves transmitting the j-invariant $j(E) \in \mathbb{F}_{p^2}$ of E, rather than the two curve coefficients, and recomputing a and b from $j(E)$ on the other side. However, since $\ell_A = 2$ and therefore $4 \mid \#E$, all curves in the isogeny class can also be written in Montgomery form as $E/\mathbb{F}_{p^2} : By^2 = x^3 + Ax^2 + x$; moreover, since $j(E)$ is independent of B, the implementation described in [7] performs all computations and transmissions ignoring the Montgomery B

coefficient. Although the Weierstrass curve compression in [2] applies in general, the presence of $\ell_A = 2$ in our case allows for the much simpler method of curve compression that simply transmits the coefficient $A \in \mathbb{F}_{p^2}$ of E in Montgomery form.

The second idea from [2], which is the main focus of this paper, is to transmit each of the two points $P, Q \in E(\mathbb{F}_{p^2})[n]$ as their two-dimensional scalar decomposition with respect to a fixed basis $\{R_1, R_2\}$ of $E(\mathbb{F}_{p^2})[n]$. Both of these decompositions are in $\mathbb{Z}_n^2$, requiring $2\log n$ bits each. But $n \approx \sqrt{p}$ (see [8]), so $2\log n \approx \log p$ is around half the size of the $2\log p$ bits needed to transmit a coordinate in $\mathbb{F}_{p^2}$. Of course, the curve in each public key is different, so there is no public basis that can be fixed once-and-for-all, and moreover, to transmit such a basis is as expensive as transmitting the points P and Q in the first place. The whole idea therefore firstly relies on Alice and Bob being able to, on input of a given curve E, arrive at the same basis $\{R_1, R_2\}$ for $E(\mathbb{F}_{p^2})[n]$. In [2] it is proposed to try successive points that result from the use of a deterministic pseudo-random number generator, checking the order of the points each time until two points of exact order n are found. In Sect. 3 we present alternative algorithms that deterministically compute a basis much more efficiently.

Assuming that Alice or Bob have computed the basis $\{R_1, R_2\}$ for $E(\mathbb{F}_{p^2})[n]$, the idea is to now write $P = [\alpha_P]R_1 + [\beta_P]R_2$ and $Q = [\alpha_Q]R_1 + [\beta_Q]R_2$, and to solve these equations for $(\alpha_P, \beta_P, \alpha_Q, \beta_Q) \in \mathbb{Z}_n^4$. To compute α_P and β_P, Azarderakhsh *et al.* [2] propose first using the Weil pairing $e\colon E(\mathbb{F}_{p^2})[n] \times E(\mathbb{F}_{p^2})[n] \to \mu_n$ to set up the two discrete logarithm instances that arise from the three pairings $e_0 = e(R_1, R_2)$, $e(R_1, P) = e_0^{\beta_P}$, and $e(R_2, Q) = e_0^{-\alpha_P}$; computing α_Q and β_Q then requires two additional pairings, since e_0 can be reused. In Sect. 4 we exploit the fact that these five Weil pairings can be replaced by the much more efficient Tate pairing, and we give an optimized algorithm that computes them all simultaneously.

To finalize compression, it remains to use the Pohlig-Hellman algorithm [23] to solve the four DLP instances in μ_n. In Sect. 5 we present an efficient version of the Pohlig-Hellman algorithm that exploits windowing methods to solve the discrete logarithm instances with lower complexity than the original algorithm. In Sect. 6 we show that one of the four scalars in $\mathbb{Z}_n$ need not be transmitted, since it is always possible to normalize the tuple $(\alpha_P, \beta_P, \alpha_Q, \beta_Q)$ by dividing three of the elements by a determinstically chosen invertible one. The public key is then transmitted as 3 scalars in $\mathbb{Z}_n$ and the curve coefficient $A \in \mathbb{F}_{p^2}$.

SIDH Decompression. The first task of the recipient of a compressed public key is to compute the basis $\{R_1, R_2\}$ in the same way as was done during compression. Once the recipient has computed the basis $\{R_1, R_2\}$, two double-scalar multiplications can be used to recover P and Q. In Sect. 6.2, we show that these double-scalar multiplications can be ommitted by absorbing these scalars into the secret SIDH scalars used for shared secret computations. This further enhances the decompression phase.

Concrete Parameters. As mentioned above, we illustrate our techniques by basing our compression software on the SIDH library recently presented in [7]. This library was built using a specific supersingular isogeny class defined by $p = n_A n_B - 1$, with $n_A = \ell_A^{e_A} = 2^{372}$ and $n_B = \ell_B^{e_B} = 3^{239}$, chosen such that all curves in this isogeny class have order $(n_A n_B)^2$. In what follows we will assume that our parameters correspond to this curve, but reiterate that these techniques will be equally as applicable for any supersingular isogeny class with $\ell_A = 2$ and $\ell_B = 3$.

3 Constructing Torsion Bases

For a given $A \in \mathbb{F}_{p^2}$ corresponding to a supersingular curve $E/\mathbb{F}_{p^2} : y^2 = x^3 + Ax^2 + x$ with $\#E(\mathbb{F}_{p^2}) = (n_A n_B)^2$, the goal of this section is to produce a basis for $E(\mathbb{F}_{p^2})[n]$ (with $n \in \{n_A, n_B\}$) as efficiently as possible. This amounts to computing two order n points R_1 and R_2 whose Weil pairing $e_n(R_1, R_2)$ has exact order n. Checking the order of the Weil pairing either comes for free during subsequent computations, or requires the amendments discussed in Remark 3 at the end of this section. Thus, for now our goal is simplified to efficiently computing points of order $n \in \{n_A, n_B\}$ in $E(\mathbb{F}_{p^2})$.

Let $\{n, n'\} = \{n_A, n_B\}$, write $n = \ell^e$ and $n' = \ell'^{e'}$, and let $\mathcal{O}$ be the identity in $E(\mathbb{F}_{p^2})$. The typical process of computing a point of exact order n is to start by computing $R \in E(\mathbb{F}_{p^2})$ and multiplying by the cofactor n' to compute the candidate output $\tilde{R} = [n']R$. Note that the order of $\tilde{R}$ divides n, but might not be n. Thus, we multiply $\tilde{R}$ by ℓ^{e-1}, and if $[\ell^{e-1}]\tilde{R} \neq \mathcal{O}$, we output $\tilde{R}$, otherwise we must pick a new R and restart.

In this section we use explicit results arising from 2- and 3-descent to show that the cofactor multiplications by n' and by ℓ^{e-1} can be ommitted by making use of elementary functions involving points of order 2 and 3 to check whether points are (respectively) in $E \setminus [2]E$ or $E \setminus [3]E$. In both cases this guarantees that the subsequent multiplication by n' produces a point of exact order n, avoiding the need to perform full cofactor multiplications to check order prior to the pairing computation, and avoiding the need to restart the process if the full cofactor multiplication process above fails to output a point of the correct order (which happens regularly in practice). This yields much faster algorithms for basis generation than those that are used in [2].

We discuss the 2^e-torsion basis generation in Sect. 3.2 and the 3^e-torsion basis generation in Sect. 3.3. We start in Sect. 3.1 by describing some arithmetic ingredients.

3.1 Square Roots, Cube Roots, and Elligator 2

In this section we briefly describe the computation of square roots and that of testing cubic residuosity in $\mathbb{F}_{p^2}$, as well as our tailoring of the Elligator 2 method [5] for efficiently producing points in $E(\mathbb{F}_{p^2})$.

Computing Square Roots and Checking Cubic Residuosity in $\mathbb{F}_{p^2}$. Square roots in $\mathbb{F}_{p^2}$ are most efficiently computed via two square roots in the base field $\mathbb{F}_p$. Since $p \equiv 3 \bmod 4$, write $\mathbb{F}_{p^2} = \mathbb{F}_p(i)$ with $i^2+1=0$. Following [27, Sect. 3.3], we use the simple identity

$$\sqrt{a + b \cdot i} = \pm\left(\alpha + \beta \cdot i\right), \tag{1}$$

where $\alpha = \sqrt{(a \pm \sqrt{a^2+b^2})/2}$ and $\beta = b/(2\alpha)$; here $a, b, \alpha, \beta \in \mathbb{F}_p$. Both of $(a + \sqrt{a^2+b^2})/2$ and $(a - \sqrt{a^2+b^2})/2$ will not necessarily be square, so we make the correct choice by assuming that $z = (a + \sqrt{a^2+b^2})/2$ is square and setting $\alpha = z^{(p+1)/4}$; if $\alpha^2 = z$, we output a square root as $\pm(\alpha + \beta i)$, otherwise we can output a square root as $\pm(\beta - \alpha i)$.

In Sect. 3.3 we will need to efficiently test whether elements $v \in \mathbb{F}_{p^2}$ are cubic residues or not. This amounts to checking whether $v^{(p^2-1)/3} = 1$ or not, which we do by first computing $v' = v^{p-1} = v^p/v$ via one application of Frobenius (i.e., $\mathbb{F}_{p^2}$ conjugation) and one $\mathbb{F}_{p^2}$ inversion. We then compute $v'^{(p+1)/3}$ as a sequence of $e_A = 372$ repeated squarings followed by $e_B - 1 = 238$ repeated cubings. Both of these squaring and cubing operations are in the order $p+1$ cyclotomic subgroup of $\mathbb{F}^*_{p^2}$, so can take advantage of the fast operations described in Sect. 5.1.

Elligator 2. The naïve approach to obtaining points in $E(\mathbb{F}_{p^2})$ is to sequentially test candidate x-coordinates in $\mathbb{F}_{p^2}$ until $f(x) = x^3 + Ax^2 + x$ is square. Each of these tests requires at least one exponentiation in $\mathbb{F}_p$, and a further one (to obtain the corresponding y) if $f(x)$ is a square. The Elligator 2 construction deterministically produces points in $E(\mathbb{F}_{p^2})$ using essentially the same operations, so given that the naïve method can fail (and waste exponentiations), Elligator 2 performs significantly faster on average.

The idea behind Elligator 2 is to let u be any non-square in $\mathbb{F}_{p^2}$, and for any $r \in \mathbb{F}_{p^2}$, write

$$v = -\frac{A}{1+ur^2} \qquad \text{and} \qquad v' = \frac{A}{1+ur^2} - A. \tag{2}$$

Then either v is an x-coordinate of a point in $E(\mathbb{F}_{p^2})$, or else v' is [5]; this is because $f(v)$ and $f(v')$ differ by the non-square factor ur^2.

In our implementation we fix $u = i+4$ as a system parameter and precompute a public table consisting of the values $-1/(1+ur^2) \in \mathbb{F}_{p^2}$ where r^2 ranges from 1 to 10. This table is fixed once-and-for-all and can be used (by any party) to efficiently generate torsion bases as A varies over the isogeny class. Note that the size of the table here is overkill, we very rarely need to use more than 3 or 4 table values to produce basis points of the correct exact order.

The key to optimizing the Elligator 2 construction (see [5, Sect. 5.5]) is to be able to efficiently modify the square root computation in the case that $f(v)$ is non-square, to produce $\sqrt{f(v')}$. This update is less obvious for our field than in the case of prime fields, but nevertheless achieves the same result. Referring back to (1), we note that whether or not $a + b \cdot i$ is a square in $\mathbb{F}_{p^2}$ is determined

solely by whether or not $a^2 + b^2$ is a square in $\mathbb{F}_p$ [27, Sect. 3.3]. Thus, if this check deems that $a + bi$ is non-square, we multiply it by $ur^2 = (i + 4)r^2$ to yield a square, and this is equivalent to updating $(a, b) = (r(4a - b), r(a + 4b))$, which is trivial in the implementation.

3.2 Generating a Torsion Basis for $E(\mathbb{F}_{p^2})[2^{e_A}]$

The above discussion showed how to efficiently generate candidate points R in $E(\mathbb{F}_{p^2})$. In this subsection we show how to efficiently check that R is in $E \setminus [2]E$, which guarantees that $[3^{e_B}]R$ is a point of exact order 2^{e_A}, and can subsequently be used as a basis element.

Since the supersingular curves $E/\mathbb{F}_{p^2} : y^2 = x(x^2 + Ax + 1)$ in our isogeny class have a full rational 2-torsion, we can always write them as $E/\mathbb{F}_{p^2} : y^2 = x(x - \gamma)(x - \delta)$. A classic result (cf. [14, Chap. 1 (Sect. 4), Theorem 4.1]) says that, in our case, any point $R = (x_R, y_R)$ in $E(\mathbb{F}_{p^2})$ is in $[2]E(\mathbb{F}_{p^2})$, i.e., is the result of doubling another point, if and only if x_R, $x_R - \gamma$ and $x_R - \delta$ are all squares in $\mathbb{F}_{p^2}$. This means that we do not need to find the roots δ and γ of $x^2 + Ax + 1$ to test for squareness, since we want the x_R such that at least one of x_R, $x_R - \gamma$ and $x_R - \delta$ are a non-square. We found it most efficient to simply ignore the latter two terms and reject any x_R that is square, since the first non-square x_R we find corresponds to a point R such that $[3^{e_B}]R$ has exact order 2^{e_A}, and further testing square values of x_R is both expensive and often results in the rejection of R anyway.

In light of the above, we note that for the 2-torsion basis generation, the Elligator approach is not as useful as it is in the next subsection. The reason here is that we want to only try points with a non-square x-coordinate, and there is no exploitable relationship between the squareness of v and v' in (2) (such a relation only exists between $f(v)$ and $f(v')$). Thus, the best approach here is to simply proceed by trying candidate v's as consecutive elements of a list $L = [u, 2u, 3u, \dots]$ of non-squares in $\mathbb{F}_{p^2}$ until $(v^3 + Av^2 + v)$ is square; recall from above that this check is performed efficiently using one exponentiation in $\mathbb{F}_p$.

To summarize the computation of a basis $\{R_1, R_2\}$ for $E(\mathbb{F}_{p^2})[2^{e_A}]$, we compute R_1 by letting v be the first element in L where $(v^3 + Av^2 + v)$ is square. We do not compute the square root of $(v^3 + Av^2 + v)$, but rather use e_B repeated x-only tripling operations starting from v to compute x_{R_1}. We then compute y_{R_1} as the square root of $x_{R_1}^3 + Ax_{R_1}^2 + x_{R_1}$. Note that either choice of square root is fine, so long as Alice and Bob take the same one. The point R_2 is found identically, i.e., using the second element in L that corresponds to an x-coordinate of a point on $E(\mathbb{F}_{p^2})$, followed by e_B x-only tripling operations to arrive at x_{R_2}, and the square root computation to recover y_{R_2}. Note that the points R_1 and R_2 need not be normalized before their input into the pairing computation; as we will see in Sect. 4, the doubling-only and tripling-only pairings do not ever perform additions with the original input points, so the input points are essentially forgotten after the first iteration.

3.3 Generating a Torsion Basis for $E(\mathbb{F}_{p^2})[3^{e_B}]$

The theorem used in the previous subsection was a special case of more general theory that characterizes the action of multiplication-by-m on E. We refer to Silverman's chapter [29, Chap. X] and to [26] for the deeper discussion in the general case, but in this work we make use of the explicit results derived in the case of $m = 3$ by Schaefer and Stoll [26], stating only what is needed for our purposes.

Let $P_3 = (x_{P_3}, y_{P_3})$ be any point of order 3 in $E(\mathbb{F}_{p^2})$ (recall that the entire 3-torsion is rational here), and let $g_{P_3}(x, y) = y - (\lambda x + \mu)$ be the usual tangent function to E at P_3. For any other point $R \in E(\mathbb{F}_{p^2})$, the result we use from [26] states that $R \in [3]E$ if and only if $g_{P_3}(R)$ is in $(\mathbb{F}_{p^2})^3$ (i.e., is a cube) for all of the 3-torsion points[2] P_3.

Again, since we do not want points in $[3]E$, but rather points in $E \setminus [3]E$, we do not need to test that R gives a cube for all of the $g_{P_3}(R)$, we simply want to compute an R where *any* one of the $g_{P_3}(R)$ is not a cube. In this case the test involves both coordinates of R, so we make use of Elligator 2 as it is described in Sect. 3.1 to produce candidate points $R \in E(\mathbb{F}_{p^2})$.

Unlike the previous case, where the 2-torsion point $(0, 0)$ is common to all curves in the isogeny class, in this case it is computing a 3-torsion point P_3 that is the most difficult computation. We attempted to derive an efficient algorithm that finds x_{P_3} as any root of the (quartic) 3-division polynomial $\Phi_3(A, x)$, but this solution involved several exponentiations in both $\mathbb{F}_{p^2}$ and $\mathbb{F}_p$, and was also hampered by the lack of an efficient enough analogue of (1) in the case of cube roots[3]. We found that a much faster solution was to compute the initial 3-torsion point via an x-only cofactor multiplication: we use the first step of Elligator 2 to produce an x-coordinate x_R, compute $x_{\tilde{R}} = x_{[2^{e_A}]R}$ via e_A repeated doublings, and then apply k repeated triplings until the result of a tripling is $(X : Z) \in \mathbb{P}^1$ with $Z = 0$, which corresponds to the point $\mathcal{O}$, at which point we can set out x_{P_3}, the x-coordinate of a 3-torsion point P_3, as the last input to the tripling function. Moreover, if the number of triplings required to produce $Z = 0$ was $k = e_B$, then it must be that $\tilde{R}$ is a point of exact order 3^{e_B}. If this is the case, we can use a square root to recover $y_{\tilde{R}}$ from $x_{\tilde{R}}$, and we already have one of our two basis points.

At this stage we either need to find one more point of order 3^{e_B}, or two. In either case we use the full Elligator routine to obtain candidate points R exactly as described in Sect. 3.1, use our point P_3 (together with our efficient test of cubic residuosity above) to test whether $g_{P_3}(R) = y_R - (\lambda x_R + \mu)$ is a cube, and if it is not, we output $\pm[2^{e_A}]R$ as a basis point; this is computed via a sequence of x-only doublings and one square root to recover $y_{[2^{e_A}]R}$ at the end. On the other hand, if $g_{P_3}(R)$ is a cube, then $R \in [3]E$, so we discard it and proceed to generate the next R via the tailored version of Elligator 2 above.

[2] The astute reader can return to Sect. 3.2 and see that this is indeed a natural analogue of [14, Chap. 1 (Sect. 4), Theorem 4.1].

[3] A common subroutine when finding roots of quartics involve solving the so-called *depressed cubic*.

We highlight the significant speed advantage that is obtained by the use of the result of Schaefer and Stoll [26]. Testing that points are in $E\backslash[3]E$ by cofactor multiplication requires e_A point doubling operations and e_B point tripling operations, while the same test using the explicit results from 3-descent require one field exponentiation that tests cubic residuosity. Moreover, this exponentiation only involves almost-for-free Frobenius operations and fast cyclotomic squaring and cubing operations (again, see Sect. 5.1).

Remark 3 (Checking the order of the Weil pairing). As mentioned at the beginning of this section, until now we have simplified the discussion to focus on generating two points R_1 and R_2 of exact order n. However, this does not mean that $\{R_1, R_2\}$ is a basis for $E(\mathbb{F}_{p^2})[n]$; this is the case if and only if the Weil pairing $e_n(R_1, R_2)$ has full order n. Although the Weil pairing will have order n with high probability for random R_1 and R_2, the probability is not so high that we do not encounter it in practice. Thus, baked into our software is a check that this is indeed the case, and if not, an appropriate backtracking mechanism that generates a new R_2. We note that, following [7, Sect. 9] and [11, Sect. 2.5], checking whether or not the Weil pairing $e_n(R_1, R_2)$ has full order is much easier than computing it, and can be done by comparing the values $[n/\ell]R_1$ and $[n/\ell]R_2$.

4 The Tate Pairing Computation

Given the basis points R_1 and R_2 resulting from the previous section, and the two points P and Q in the (otherwise uncompressed) public key, we now have four points of exact order n. As outlined in Sect. 2, the next step is to compute the following five pairings to transfer the discrete logarithms to the multiplicative group $\mu_n \subset \mathbb{F}^*_{p^2}$:

$$
\begin{aligned}
e_0 &:= e(R_1, R_2) = f_{n,R_1}(R_2)^{(p^2-1)/n} \\
e_1 &:= e(R_1, P) = f_{n,R_1}(P)^{(p^2-1)/n} \\
e_2 &:= e(R_1, Q) = f_{n,R_1}(Q)^{(p^2-1)/n} \\
e_3 &:= e(R_2, P) = f_{n,R_2}(P)^{(p^2-1)/n} \\
e_4 &:= e(R_2, Q) = f_{n,R_2}(Q)^{(p^2-1)/n}.
\end{aligned}
$$

As promised in Sect. 1, the above pairings are already defined by the order n reduced Tate pairing $e\colon E(\mathbb{F}_{p^2})[n] \times E(\mathbb{F}_{p^2})/nE(\mathbb{F}_{p^2}) \mapsto \mu_n$, rather than the Weil pairing that was used in [2]. The rationale behind this choice is clear: the lack of special (subfield) groups inside the n-torsion means that many of the tricks used in the pairing literature cannot be exploited in the traditional sense. For example, there does not seem to be a straight-forward way to shorten the Miller loop by using efficiently computable maps arising from Frobenius (see, e.g., [3,12,13]), our denominators lie in $\mathbb{F}_{p^2}$ so cannot be eliminated [4], and, while distortion maps exist on all supersingular curves [33], finding efficiently computable and therefore useful maps seems hard for random curves in the

isogeny class. The upshot is that the Miller loop is far more expensive than the final exponentiation in our case, and organizing the Tate pairings in the above manner allows us to get away with the computation of only two Miller functions, rather than the four that were needed in the case of the Weil pairing [2].

In the case of ordinary pairings over curves with a larger embedding degree[4], the elliptic curve operations during the Miller loop take place in a much smaller field than the extension field; in the Tate pairing the point operations take place in the base field, while in the loop-shortened ate pairing [13] (and its variants) they take place in a small-degree subfield. Thus, in those cases the elliptic curve arithmetic has only a minor influence on the overall runtime of the pairing.

In our scenario, however, we are stuck with elliptic curve points that have both coordinates in the full extension field. This means that the Miller line function computations are the bottleneck of the pairing computations (and, as it turns out, this is the main bottleneck of the whole compression routine). The main point of this section is to present optimized explicit formulas in this case; this is done in Sect. 4.1. In Sect. 4.2 we discuss how to compute the five pairings in parallel and detail how to compute the final exponentiations efficiently.

4.1 Optimized Miller Functions

We now present explicit formulas for the point operations and line computations in Miller's algorithm [20]. In the case of the order-2^{e_A} Tate pairings inside $E(\mathbb{F}_{p^2})[2^{e_A}]$, we only need the doubling-and-line computations, since no additions are needed. In the case of the order-3^{e_B} Tate pairings inside $E(\mathbb{F}_{p^2})[3^{e_B}]$, we investigated two options: the first option computes the pairing in the usual "double-and-add" fashion, reusing the doubling-and-line formulas with addition-and-line formulas, while the second uses a simple sequence of e_B tripling-and-parabola operations. The latter option proved to offer much better performance and is arguably more simple than the double-and-add approach[5].

We tried several coordinate systems in order to lower the overall number of field operations in both pairings, and after a close inspection of the explicit formulas in both the doubling-and-line and tripling-and-parabola operations, we opted to use the coordinate tuple $(X^2 : XZ : Z^2 : YZ)$ to represent intermediate projective points $P = (X : Y : Z) \in \mathbb{P}^2$ in $E(\mathbb{F}_{p^2})$. Note that all points in our routines for which we use this representation satisfy $XYZ \neq 0$, as their orders are strictly larger than 2. This ensures that the formulas presented below do not contain exceptional cases[6].

[4] This has long been the preferred choice of curve in the pairing-based cryptography literature.

[5] An earlier version of this article claimed that the performance was favourable in the former case, but further optimization in the tripling-and-parabola scenario since proved this option to be significantly faster.

[6] The input into the final iteration in the doubling-only pairing is a point of order 2, but (as is well-known in the pairing literature) this last iterate is handled differently than all of the prior ones.

Doubling-and-Line Operations. The doubling step in Miller's algorithm takes as input the tuple $(U_1, U_2, U_3, U_4) = (X^2, XZ, Z^2, YZ)$ corresponding to the point $P = (X : Y : Z) \in \mathbb{P}^2$ in $E(\mathbb{F}_{p^2})$, and outputs the tuple $(V_1, V_2, V_3, V_4) = (X_2^2 : X_2Z_2 : Z_2^2 : Y_2Z_2)$ corresponding to the point $[2]P = (X_2 : Y_2 : Z_2) \in \mathbb{P}^2$, as well as the 5 coefficients in the Miller line function $l/v = (l_x \cdot x + l_y \cdot y + l_0)/(v_x x + v_0)$ with divisor $2(P) - ([2]P) - (\mathcal{O})$ in $\mathbb{F}_q[x, y](E)$. The explicit formulas are given as:

$$\begin{aligned} l_x &= 4U_4^3 + 2U_2U_4(U_1 - U_3), \\ l_y &= 4U_2U_4^2, \\ l_0 &= 2U_1U_4(U_1 - U_3), \\ v_x &= 4U_2U_4^2, \\ v_0 &= U_2(U_1 - U_3)^2, \end{aligned}$$

together with

$$\begin{aligned} V_1 &= (U_1 - U_3)^4, \\ V_2 &= 4U_4^2(U_1 - U_3)^2, \\ V_3 &= 16U_4^4, \\ V_4 &= 2U_4(U_1 - U_3)((U_1 - U_3)^2 + 2U_2(4U_2 + A(U_1 + U_3))). \end{aligned}$$

The above point doubling-and-line function computation can be computed in $9\mathbf{M} + 5\mathbf{S} + 7\mathbf{a} + 1\mathbf{s}$. The subsequent evaluation of the line function at the second argument of a pairing, the squaring that follows, and the absorption of the squared line function into the running paired value costs $5\mathbf{M} + 2\mathbf{S} + 1\mathbf{a} + 2\mathbf{s}$.

Tripling-and-Parabola Operations. The tripling-and-parabola operation has as input the tuple $(U_1, U_2, U_3, U_4) = (X^2, XZ, Z^2, YZ)$ corresponding to the point $P = (X : Y : Z) \in \mathbb{P}^2$ in $E(\mathbb{F}_{p^2})$, and outputs the tuple $(V_1, V_2, V_3, V_4) = (X_3^2 : X_3Z_3 : Z_3^2 : Y_3Z_3)$ corresponding to the point $[3]P = (X_3 : Y_3 : Z_3) \in \mathbb{P}^2$, as well as the 6 coefficients in the Miller parabola function $l/v = (l_y \cdot y + l_{x,2} \cdot x^2 + l_{x,1}x + l_{x,0})/(v_x x + v_0)$ with divisor $3(P) - ([3]P) - 2(\mathcal{O})$ in $\mathbb{F}_q[x, y](E)$. The explicit formulas are given as:

$$\begin{aligned} l_y &= 8U_4^3, \\ l_{x,2} &= U_3(3U_1^2 + 4U_1AU_2 + 6U_1U_3 - U_3^2), \\ l_{x,1} &= 2U_2(3U_1^2 + 2U_1U_3 + 3U_3^2 + 6U_1AU_2 + 4A^2U_2^2 + 6AU_2U_3), \\ l_{x,0} &= U_1(-U_1^2 + 6U_1U_3 + 3U_3^2 + 4AU_2U_3), \\ v_x &= 8U_3U_4^3(3U_1^2 + 4U_1AU_2 + 6U_1U_3 - U_3^2)^4, \\ v_0 &= -8U_2U_4^3(3U_1^2 + 4U_1AU_2 + 6U_1U_3 - U_3^2)^2(U_1^2 - 6U_1U_3 - 3U_3^2 - 4AU_2U_3)^2, \end{aligned}$$

together with

$$
\begin{aligned}
V_1 &= 8U_4^3U_1(-U_1^2 + 6U_1U_3 + 3U_3^2 + 4AU_2U_3)^4,\\
V_2 &= 8U_2U_4^3(3U_1^2 + 4U_1AU_2 + 6U_1U_3 - U_3^2)^2(U_1^2 - 6U_1U_3 - 3U_3^2 - 4AU_2U_3)^2,\\
V_3 &= 8U_3U_4^3(3U_1^2 + 4U_1AU_2 + 6U_1U_3 - U_3^2)^4,\\
V_4 &= -8U_3(3U_1^2 + 4U_1AU_2 + 6U_1U_3 - U_3^2)U_1(-U_1^2 + 6U_1U_3 + 3U_3^2 + 4AU_2U_3)\\
&\quad \cdot \big(16U_1U_3A^2U_2^2 + 28U_1^2AU_2U_3 + 28U_1U_3^2AU_2 + 4U_3^3AU_2 + 4U_1^3AU_2\\
&\quad + 6U_1^2U_3^2 + 28U_1^3U_3 + U_3^4 + 28U_1U_3^3 + U_1^4\big)(U_3 + U_1 + AU_2)^2.
\end{aligned}
$$

The above point tripling-and-parabola function computation can be computed in $19\mathbf{M} + 6\mathbf{S} + 15\mathbf{a} + 6\mathbf{s}$. The subsequent evaluation of the line function at the second argument of a pairing, the cubing that follows, and the absorption of the cubed line function into the running paired value costs $10\mathbf{M} + 2\mathbf{S} + 4\mathbf{a}$.

Remark 4 (No irrelevant factors). It is common in the pairing literature to abuse notation and define the order-n Tate pairing as $e_n(P,Q) = f_P(Q)^{(p^k-1)/n}$, where k is the embedding degree (in our case $k = 2$), and f_P has divisor $(f_P) = n(P) - n(\mathcal{O})$ in $\mathbb{F}_{p^k}[x,y](E)$. This is due to an early result of Barreto, Kim, Lynn and Scott [4, Theorem 1], who showed that the actual definition of the Tate pairing, i.e., $e_n(P,Q) = f_P(D_Q)^{(p^k-1)/n}$ where D_Q is a divisor equivalent to $(Q)-(\mathcal{O})$, could be relaxed in practical cases of interest by replacing the divisor D_Q with the point Q. This is due to the fact that the evaluation of f_P at $\mathcal{O}$ in such scenarios typically lies in a proper subfield of $\mathbb{F}^*_{p^k}$, so becomes an *irrelevant factor* under the exponentiation to the power of $(p^k-1)/n$. In our case, however, this is generally not the case because the coefficients in our Miller functions lie in the full extension field $\mathbb{F}^*_{p^2}$. Subsequently, our derivation of explicit formulas replaces Q with the divisor $D_Q = (Q) - (\mathcal{O})$, and if the evaluation of the Miller functions at $\mathcal{O}$ is ill-defined, we instead evaluate them at the divisor $(Q+T)-(T)$ that is linearly equivalent to D_Q, where we fixed $T = (0,0)$ as the (universal) point of order 2. If $Q = (x_Q, y_Q)$, then $Q + T = (1/x_Q, -y_Q/x_Q^2)$, so evaluating the Miller functions at the translated point amounts to a permutation of the coefficients, and evaluating the Miller functions at $T = (0,0)$ simply leaves a quotient of the constant terms. These modifications are already reflected in the operation counts quoted above.

Remark 5. In the same vein as Remark 2, there is another possible speed improvement within the pairing computation that is not currently exploited in our library. Recall that during the generation of the torsion bases described in Sect. 3, the candidate basis point R is multiplied by the cofactor $n \in \{n_A, n_B\}$ to check whether it has the correct (full) order, and if so, R is kept and stored as one of the two basis points. Following the idea of Scott [27, Sect. 9], the intermediate multiples of R (and partial information about the corresponding Miller line functions) that are computed in this cofactor multiplication could be stored in anticipation for the subsequent pairing computation, should R indeed be one of the two basis points. Another alternative here would be to immediately compute

the pairings using the first two candidate basis points and to absorb the point order checks inside the pairing computations, but given the overhead incurred if either or both of these order checks fails, this could end up being too wasteful (on average).

4.2 Parallel Pairing Computation and the Final Exponentiation

In order to solve the discrete logarithms in the subgroup μ_n of n-th roots of unity in $\mathbb{F}_{p^2}^*$, we compute the five pairings $e_0 := e(R_1, R_2)$, $e_1 := e(R_1, P)$, $e_2 := e(R_1, Q)$, $e_3 := e(R_2, P)$, and $e_4 := e(R_2, Q)$. The first argument of all these pairings is either R_1 or R_2, i.e., all are of the form $f_{n,R_i}(S)^{(p^2-1)/n}$ for $i \in \{1, 2\}$ and $S \in \{R_2, P, Q\}$. This means that the only Miller functions we need are f_{n,R_1} and f_{n,R_2}, and we get away with computing only those two functions for the five pairing values. The two functions are evaluated at the points R_2, P, Q during the Miller loop to obtain the desired combinations. It therefore makes sense to accumulate all five Miller values simultaneously.

Computing the pairings simultaneously also becomes advantageous when it is time to perform inversions. Since we cannot eliminate denominators due to the lack of a subfield group, we employ the classic way of storing numerators and denominators separately to delay all inversions until the very end of the Miller loop. At this point, we have ten values (five numerators and five denominators), all of which we invert using Montgomery's inversion sharing trick [21] at the total cost of one inversion and 30 $\mathbb{F}_{p^2}$ multiplications. The five inverted denominators are then multiplied by the corresponding numerators to give the five unreduced paired values. The reason we not only invert the denominators, but also the numerators, is because these inverses are needed in the easy part of the final exponentiation.

The final exponentiation is an exponentiation to the power $(p^2 - 1)/n = (p - 1)\frac{p+1}{n}$. The so-called *easy part*, i.e., raising to the power $p - 1$, is done by one application of the Frobenius automorphism and one inversion. The Frobenius is simply a conjugation in $\mathbb{F}_{p^2}$, and the inversion is actually a multiplication since we had already computed all required inverses as above. The so-called *hard part* of the final exponentiation has exponent $(p + 1)/n$ and needs to be done with regular exponentiation techniques.

A nice advantage that makes the hard part quite a little easier is the fact that after a field element $a = a_0 + a_1 \cdot i \in \mathbb{F}_{p^2}$ has been raised to the power $p-1$, it has order $p + 1$, which means it satisfies $1 = a^p \cdot a = a_0^2 + a_1^2$. This equation can be used to deduce more efficient squaring and cubing formulas that speed up this final part of the pairing computation (see Sect. 5.1 for further details).

Finally, in the specific setting of SIDH, where $p = n_A n_B - 1$, we have that $(p + 1)/n_A = n_B$ and $(p + 1)/n_B = n_A$. When n_A and n_B are powers of 2 and 3, respectively, the hard part of the final exponentiation consists of only squarings or only cubings, respectively. These are done with the particularly efficient formulas described in Sect. 5.1 below.

5 Efficient Pohlig-Hellman in μ_{ℓ^e}

In this section, we describe how we optimize the Pohlig-Hellman [23] algorithm to compute discrete logarithms in the context of public-key compression for supersingular-isogeny-based cryptosystems, and we show that we are able to improve on the quadratic complexity described in [23]. A similar result has already been presented in the more general context of finite abelian p-groups by Sutherland [32]. However, our software employs a different optimization of Pohlig-Hellman, by choosing small memory consumption over a more efficient computation, which affects parameter choices. We emphasize that there are different time-memory trade-offs that could be chosen, possibly speeding up the Pohlig-Hellman computation by another factor of two (see Remark 2).

Following the previous sections, the two-dimensional discrete logarithm problems have been reduced to four discrete logarithm problems in the multiplicative group $\mu_{\ell^e} \subset \mathbb{F}_{p^2}^*$ of ℓ^e-th roots of unity, where $\ell, e \in \mathbb{Z}$ are positive integers and ℓ is a (small) prime. Before elaborating on the details of the windowed Pohlig-Hellman algorithm, we note that the condition $\ell^e \mid p+1$ makes various operations in μ_{ℓ^e} more efficient than their generic counterpart in $\mathbb{F}_{p^2}^*$.

5.1 Arithmetic in the Cyclotomic Subgroup

Efficient arithmetic in μ_{ℓ^e} can make use of the fact that μ_{ℓ^e} is a subgroup of the multiplicative group $G_{p+1} \subset \mathbb{F}_{p^2}^*$ of order $p+1$. Various subgroup cryptosystems based on the hardness of the discrete logarithm problem have been proposed in the literature [18,30], which can be interpreted in the general framework of torus-based cryptography [25]. The following observations for speeding up divisions and squarings in G_{p+1} have been described by Stam and Lenstra [31, Sect. 3.23 and Lemma 3.24].

Division in μ_{ℓ^e}. Let $p \equiv 3 \pmod 4$ and $\mathbb{F}_{p^2} = \mathbb{F}_p(i)$, $i^2 = -1$. For any $a = a_0 + a_1 \cdot i \in G_{p+1}$, $a_0, a_1 \in \mathbb{F}_p$, we have that $a \cdot a^p = a^{p+1} = 1$, and therefore, the inverse $a^{-1} = a^p = a_0 - a_1 \cdot i$. This means that inversion in μ_{ℓ^e} can be computed almost for free by conjugation, i.e., a single negation in $\mathbb{F}_p$, and thus divisions become as efficient as multiplications in μ_{ℓ^e}.

Squaring in μ_{ℓ^e}. The square of $a = a_0 + a_1 \cdot i$ can be computed as $a^2 = (2a_0^2 - 1) + ((a_0 + a_1)^2 - 1) \cdot i$ by essentially two base field squarings. In the case where such squarings are faster than multiplications, this yields a speed-up over generic squaring in $\mathbb{F}_{p^2}$.

Cubing in μ_{ℓ^e}. As far as we know, a cubing formula in G_{p+1} has not been considered in the literature before. We make the simple observation that a^3 can be computed as $a^3 = (a_0 + a_1 \cdot i)^3 = a_0(4a_0^2 - 3) + a_1(4a_0^2 - 1) \cdot i$, which needs only one squaring and two multiplications in $\mathbb{F}_p$, and is significantly faster than a naïve computation via a squaring and a multiplication in μ_{ℓ^e}.

5.2 Pohlig-Hellman

We now discuss the Pohlig-Hellman algorithm as presented in [23] for the group μ_{ℓ^e}. Let $r, g \in \mu_{\ell^e}$ be such that $r = g^\alpha$ for some $\alpha \in \mathbb{Z}$. Given r and g, the goal is to determine the unknown scalar α. Denote α as

$$\alpha = \sum_{i=0}^{e-1} \alpha_i \ell^i \quad (\alpha_i \in \{0, \ldots, \ell - 1\}).$$

Now define $s = g^{\ell^{e-1}}$, which is an element of order ℓ, and let $r_0 = r$. Finally, define

$$g_i = g^{\ell^i} \quad (0 \leq i \leq e - 1)$$

and

$$r_i = \frac{r_{i-1}}{g_{i-1}^{\alpha_{i-1}}} \quad (1 \leq i \leq e - 1).$$

A straightforward computation then shows that for all $0 \leq i \leq e - 1$,

$$r_i^{\ell^{e-(i+1)}} = s^{\alpha_i}. \tag{3}$$

As proven in [23], this allows to inductively recover all α_i, by solving the discrete logarithms of Eq. (3) in the group $\langle s \rangle$ of order ℓ. This can be done by precomputing a table containing all elements of $\langle s \rangle$. Alternatively, if ℓ is not small enough, one can improve the efficiency by applying the Baby-Step Giant-Step algorithm [28], at the cost of some more precomputation. For small ℓ the computation has complexity $\mathcal{O}(e^2)$, while precomputing and storing the g_i requires $\mathcal{O}(e)$ memory.

5.3 Windowed Pohlig-Hellman

The original version of the Pohlig-Hellman algorithm reduces a single discrete logarithm in the large group μ_{ℓ^e} to e discrete logarithms in the small group μ_ℓ. In this section we consider an intermediate version, by reducing the discrete logarithm in μ_{ℓ^e} to $\frac{e}{w}$ discrete logarithms in μ_{ℓ^w}. Let r, g, α as in the previous section, and let $w \in \mathbb{Z}$ be such that $w \mid e$. Note that it is not necessary for e to be divisible by w. If it is not, we replace e by $e - (e \pmod w)$, and compute the discrete logarithm for the final $e \pmod w$ bits at the end. However the assumption $w \mid e$ improves the readability of the arguments with little impact on the results, so we focus on this case here. Write

$$\alpha = \sum_{i=0}^{\frac{e}{w}-1} \alpha_i \ell^{wi} \quad (\alpha_i \in \{0, \ldots, \ell^w - 1\}),$$

define $s = g^{\ell^{e-w}}$, which is an element of order ℓ^w, and let $r_0 = r$. Let

$$g_i = g^{\ell^{wi}} \quad (0 \leq i \leq \frac{e}{w} - 1)$$

and

$$r_i = \frac{r_{i-1}}{g_{i-1}^{\alpha_{i-1}}} \quad (1 \leq i \leq \frac{e}{w} - 1). \tag{4}$$

A analogous computation to the one in [23] proves that

$$r_i^{\ell^{e-w(i+1)}} = s^{\alpha_i} \quad (0 \leq i \leq \frac{e}{w} - 1). \tag{5}$$

Hence we inductively obtain α_i for all $0 \leq i \leq \frac{e}{w} - 1$, and thereby α. To solve the discrete logarithm in the smaller subgroup μ_{ℓ^w}, we consider two strategies as follows.

Baby-Step Giant-Step in $\langle s \rangle$. As before, for small ℓ and w we can compute a table containing all ℓ^w elements of $\langle s \rangle$, making the discrete logarithms in (5) trivial to solve. As explained in [28], the Baby-Step Giant-Step algorithm allows us to make a trade-off between the size of the precomputed table and the computational cost. That is, given some $v \leq w$, we can compute discrete logarithms in $\langle s \rangle$ with computational complexity $\mathcal{O}(\ell^v)$ and $\mathcal{O}(\ell^{w-v})$ memory. Note that the computational complexity grows exponentially with v, whereas the memory requirement grows exponentially with $w - v$. This means that if we want to make w larger, we need to grow v as well, as otherwise the table-size will increase. Therefore in order to obtain an efficient and compact algorithm, we must seemingly limit ourselves to small w. We overcome this limitation in the next section.

Pohlig-Hellman in $\langle s \rangle$. We observe that $\langle s \rangle$ has order ℓ^w, which is again smooth. This allows us to solve the discrete logarithms in $\langle s \rangle$ by using the original Pohlig-Hellman algorithm of Sect. 5.2. However, we can also choose to solve the discrete logarithm in $\langle s \rangle$ with a second windowed Pohlig-Hellman algorithm. Note the recursion that occurs, and we can ask what the optimal depth of this recursion is. We further investigate this question in Sect. 5.4.

5.4 The Complexity of Nested Pohlig-Hellman

We estimate the cost of an execution of the nested Pohlig-Hellman algorithm by looking at the cost of doing the computations in (4) and (5). Let F_n $(n \geq 0)$ denote the cost of an n-times nested Pohlig-Hellman algorithm, and set $F_{-1} = 0$. Let $w_0, w_1, \ldots, w_n, w_{n+1}$ be the window sizes, and set $w_0 = e$, $w_{n+1} = 1$ (note that $n = 0$ corresponds to the original Pohlig-Hellman algorithm). Again, assume that $w_n \mid w_{n-1} \mid \cdots \mid w_1 \mid e$, which is merely for ease of exposition. The first iteration has window size w_1, which means that the cost of the exponentiations in (5) is

$$\left(\sum_{i=0}^{\frac{e}{w_1}-1} w_1 i \right) \mathbf{L} = \frac{1}{2} w_1 \left(\frac{e}{w_1} - 1 \right) \frac{e}{w_1} \mathbf{L} = \frac{1}{2} e \left(\frac{e}{w_1} - 1 \right) \mathbf{L},$$

where $\mathbf{L}$ denotes the cost of an exponentiation by ℓ. The exponentiations in (4) are performed with a scalar of size $\log \alpha_i \approx w_1 \log \ell$, which on average costs $\frac{1}{2} w_1 \log \ell \, \mathbf{M} + w_1 \log \ell \, \mathbf{S}$. On average, to do all $\frac{e}{w_1}$ of them then costs

$$\frac{1}{2} e \log \ell \, \mathbf{M} + e \log \ell \, \mathbf{S}.$$

We emphasize that for small w_i and ℓ this is a somewhat crude estimation, yet it is enough to get a good feeling for how to choose our parameters (i.e., window sizes). We choose to ignore the divisions, since there are only a few (see Remark 6) and, as we showed in Sect. 5.1, they can essentially be done at the small cost of a multiplication. We also ignore the cost of the precomputation for the $g^{\ell^{w_i}}$, which is small as well (see Remark 7). To complete the algorithm, we have to finish the remaining $\frac{e}{w_1}$ $(n-1)$-times nested Pohlig-Hellman routines. In other words, we have shown that

$$F_n \approx \frac{1}{2} e \left(\frac{e}{w_1} - 1 \right) \mathbf{L} + \frac{1}{2} e \log \ell \, \mathbf{M} + e \log \ell \, \mathbf{S} + \frac{e}{w_1} F_{n-1}.$$

Now, by using analogous arguments on F_{n-1}, and induction on n, we can show that

$$F_n \approx \frac{1}{2} e \left(\frac{e}{w_1} + \ldots + \frac{w_{n-1}}{w_n} + w_n - n \right) \mathbf{L} + \frac{n+1}{2} e \log \ell \, \mathbf{M} + (n+1) e \log \ell \, \mathbf{S}. \tag{6}$$

To compute the optimal choice of $(w_1, \ldots, w_n)$, we compute the derivatives,

$$\frac{\partial F_n}{\partial w_i} = \frac{1}{2} e \left(\frac{1}{w_{i+1}} - \frac{w_{i-1}}{w_i^2} \right) \mathbf{L} \quad (1 \leq i \leq n)$$

and simultaneously equate them to zero to obtain the equations

$$w_i = \sqrt{w_{i-1} w_{i+1}} \quad (1 \leq i \leq n).$$

From this we can straightforwardly compute that the optimal choice is

$$(w_1, \ldots, w_n) = \left(e^{\frac{n}{n+1}}, e^{\frac{n-1}{n+1}}, \ldots, e^{\frac{2}{n+1}}, e^{\frac{1}{n+1}} \right). \tag{7}$$

Plugging this back into the Eq. (6), we conclude that

$$F_n \approx \frac{1}{2} e \, (n+1) \left(e^{\frac{1}{n+1}} - 1 \right) \mathbf{L} + \frac{n+1}{2} e \log \ell \, \mathbf{M} + (n+1) e \log \ell \, \mathbf{S}.$$

Observe that $F_0 \approx \frac{1}{2} e^2$, agreeing with the complexity described in [23]. However, as n grows, the complexity of the nested Pohlig-Hellman algorithm goes from quadratic to linear in e, giving a significant improvement.

Remark 6. We observe that for every two consecutive windows w_i and w_{i+1}, we need less than $\frac{w_i}{w_{i+1}}$ divisions for (4). Breaking the full computation down, it is easy to show that the total number of divisions is less than

$$\frac{e}{w_1} + \frac{e}{w_1}\left(\frac{w_1}{w_2} + \frac{w_1}{w_2}\left(\cdots + \frac{w_{n-2}}{w_{n-1}}\left(\frac{w_{n-1}}{w_n} + \frac{w_{n-1}}{w_n}w_n\right)\right)\right),$$

which can be rewritten as

$$e\left(\frac{1}{w_1} + \frac{1}{w_2} + \ldots + \frac{1}{w_n} + \frac{w_n}{w_{n-1}}\right).$$

Now we note that $w_{i+1} \mid w_i$, while $w_{i+1} \neq w_i$, for all $0 \leq i \leq n$. As $w_{n+1} = 1$, it follows that $w_{n+1-i} \geq 2^i$ for all $0 \leq i \leq n$. Therefore

$$e\left(\frac{1}{w_1} + \frac{1}{w_2} + \ldots + \frac{1}{w_n} + \frac{w_n}{w_{n-1}}\right) \leq e\left(\frac{1}{2^n} + \frac{1}{2^{n-1}} + \ldots + \frac{1}{2} + 1\right) < 2e.$$

Remark 7. As every table element is of the form g^{ℓ^i}, where i is an integer such that $0 \leq i \leq e-1$, we conclude that we need at most $(e-1)\mathbf{L}$ to pre-compute all tables.

5.5 Discrete Logarithms in $\mu_{2^{372}}$

For this section we fix $\ell = 2$ and $e = 372$. In this case $\mathbf{L}$ is the cost of a squaring, i.e., $\mathbf{L} = \mathbf{S}$. To validate the approach, we present estimates for the costs of the discrete logarithm computations in $\mu_{2^{372}}$ through a Magma implementation. In this implementation we count every multiplication, squaring and division operation; on the other hand, some of these were ignored for the estimation of F_n above. The results are shown in Table 1 for $0 \leq n \leq 4$, choosing the window sizes as computed in (7). The improved efficiency as well as the significantly smaller table sizes are clear, and we observe that in the group $\mu_{2^{372}}$ it is optimal to choose $n = 3$.

Table 1. Estimations of F_n in $\mu_{2^{372}}$ via a Magma implementation. Here $\mathbf{m}$ and $\mathbf{s}$ are the cost of multiplications and squarings in $\mathbb{F}_p$, while $\mathbf{M} = 3 \cdot \mathbf{m}$ and $\mathbf{S} = 2 \cdot \mathbf{s}$ are the cost of multiplications and squarings in $\mathbb{F}_{p^2}$. The costs are averaged over 100 executions of the algorithm.

#	Windows				$\mathbb{F}_{p^2}$		$\mathbb{F}_p$		Table size
n	w_1	w_2	w_3	w_4	**M**	**S**	**m**	**s**	$\mathbb{F}_{p^2}$
0	–	–	–	–	372	69 378	1 116	138 756	375
1	19	–	–	–	375	7 445	1 125	14 891	43
2	51	7	–	–	643	4 437	1 931	8 847	25
3	84	21	5	–	716	3 826	2 150	7 652	25
4	114	35	11	3	1 065	3 917	3 197	7 835	27

Table 2. Estimations of F_n in $\mu_{3^{239}}$ via a Magma implementation. Here $\mathbf{m}$ and $\mathbf{s}$ are the cost of multiplications and squarings in $\mathbb{F}_p$, while $\mathbf{M} = 3 \cdot \mathbf{m}$, $\mathbf{S} = 2 \cdot \mathbf{s}$ and $\mathbf{C} = \mathbf{m} + 2 \cdot \mathbf{s}$ are the cost of multiplications, squarings and cubings in $\mathbb{F}_{p^2}$ respectively. The costs are averaged over 100 executions of the algorithm.

#	Windows				$\mathbb{F}_{p^2}$			$\mathbb{F}_p$		Table size
n	w_1	w_2	w_3	w_4	**M**	**S**	**C**	**m**	**s**	$\mathbb{F}_{p^2}$
0	–	–	–	–	239	78	28 680	58 077	28 837	242
1	15	–	–	–	349	341	3 646	8 340	4 328	35
2	39	6	–	–	612	660	2 192	6 220	3 512	22
3	61	15	3	–	656	836	1 676	5 320	3 349	17
4	79	26	8	3	954	1 252	1 427	5 717	3 932	16

5.6 Discrete Logarithms in $\mu_{3^{239}}$

We now fix $\ell = 3$ and $e = 239$ and present estimates for the costs of the discrete logarithm computations in $\mu_{3^{239}}$. Here $\mathbf{L}$ is now the cost of a cubing in $\mu_{3^{239}}$. As explained in Sect. 5.1, this is done at the cost of one multiplication and two squarings in $\mathbb{F}_p$. As shown in Table 2, the optimal case in $\mu_{3^{239}}$ is also $n = 3$.

6 Final Compression and Decompression

In this section we explain how to further compress a public key $\mathtt{PK}$ from $\mathbb{F}_{p^2} \times \mathbb{Z}_n^4$ to $\mathbb{F}_{p^2} \times \mathbb{Z}_2 \times \mathbb{Z}_n^3$. Moreover, we also show how to merge the key decompression with one of the operations of the SIDH scheme, making much of the decompression essentially free of cost. For ease of notation we follow the scheme described in [7], but everything that follows in this section generalizes naturally to the theory as originally described in [8].

6.1 Final Compression

Using the techniques explained in all previous sections, we can compress a triple $(E_A, P, Q) \in \mathbb{F}_{p^2}^3$ to a tuple $(A, \alpha_P, \beta_P, \alpha_Q, \beta_Q) \in \mathbb{F}_{p^2} \times \mathbb{Z}_n^4$ such that

$$(P, Q) = (\alpha_P R_1 + \beta_P R_2, \alpha_Q R_1 + \beta_Q R_2)\,,$$

where $\{R_1, R_2\}$ is a basis of $E_A[n]$. As described in [7], the goal is to compute $\langle P + \ell m Q \rangle$, for $\ell \in \{2, 3\}$ and a secret key m. Again, we note that the original proposal expects to compute $\langle n_1 P + n_2 Q \rangle$, for secret key (n_1, n_2), but we emphasize that all that follows can be generalized to this case.

Now, since P is an element of order n, either $\alpha_P \in \mathbb{Z}_n^*$ or $\beta_P \in \mathbb{Z}_n^*$. It immediately follows that

$$\langle P + \ell m Q \rangle = \begin{cases} \langle \alpha_P^{-1} P + \ell m \alpha_P^{-1} Q \rangle & \text{if} \quad \alpha_P \in \mathbb{Z}_n^* \\ \langle \beta_P^{-1} P + \ell m \beta_P^{-1} Q \rangle & \text{if} \quad \beta_P \in \mathbb{Z}_n^* \end{cases}.$$

Hence, to compute $\langle P + \ell m Q \rangle$, we do not necessarily have to recompute (P, Q). Instead, we can compute

$$\left(\alpha_P^{-1} P, \alpha_P^{-1} Q\right) = \left(R_1 + \alpha_P^{-1}\beta_P R_2, \alpha_P^{-1}\alpha_Q R_1 + \alpha_P^{-1}\beta_Q R_2\right)$$

or

$$\left(\beta_P^{-1} P, \beta_P^{-1} Q\right) = \left(\beta_P^{-1}\alpha_P R_1 + R_2, \beta_P^{-1}\alpha_Q R_1 + \beta_P^{-1}\beta_Q R_2\right).$$

Note that in both cases we have normalized one of the scalars. We conclude that we can compress the public key to $\mathtt{PK} \in \mathbb{F}_{p^2} \times \mathbb{Z}_2 \times \mathbb{Z}_n^3$, where

$$\mathtt{PK} = \begin{cases} \left(A, 0, \alpha_P^{-1}\beta_P, \alpha_P^{-1}\alpha_Q, \alpha_P^{-1}\beta_Q\right) & \text{if} \quad \alpha_P \in \mathbb{Z}_n^* \\ \left(A, 1, \beta_P^{-1}\alpha_P, \beta_P^{-1}\alpha_Q, \beta_P^{-1}\beta_Q\right) & \text{if} \quad \beta_P \in \mathbb{Z}_n^* \end{cases}.$$

6.2 Decompression

Let $\left(A, b, \widetilde{\alpha}_P, \widetilde{\alpha}_Q, \widetilde{\beta}_Q\right) \in \mathbb{F}_{p^2} \times \mathbb{Z}_2 \times \mathbb{Z}_n^3$ be a compressed public key. Note that, by the construction of the compression, there exists a $\gamma \in \mathbb{Z}_n^*$ such that

$$\left(\gamma^{-1} P, \gamma^{-1} Q\right) = \begin{cases} \left(R_1 + \widetilde{\alpha}_P R_2, \widetilde{\alpha}_Q R_1 + \widetilde{\beta}_Q R_2\right) & \text{if} \quad b = 0, \\ \left(\widetilde{\alpha}_P R_1 + R_2, \widetilde{\alpha}_Q R_1 + \widetilde{\beta}_Q R_2\right) & \text{if} \quad b = 1. \end{cases} \tag{8}$$

The naïve strategy, analogous to the one originally explained in [2], would be to generate the basis $\{R_1, R_2\}$ of $E_A[n]$, use the public key to compute $\left(\gamma^{-1} P, \gamma^{-1} Q\right)$ via (8), and finally compute

$$\langle P + \ell m Q \rangle = \langle \gamma^{-1} P + \ell m \gamma^{-1} Q \rangle,$$

where $m \in \mathbb{Z}_n$ is the secret key. The cost is approximately a 1-dimensional and a 2-dimensional scalar multiplication on E_A, while the final 1-dimensional scalar multiplication is part of the SIDH scheme.

Instead, we use (8) to observe that

$$\begin{aligned} \langle P + \ell m Q \rangle &= \langle \gamma^{-1} P + \ell m \gamma^{-1} Q \rangle \\ &= \begin{cases} \langle \left(1 + \ell m \widetilde{\alpha}_Q\right) R_1 + \left(\widetilde{\alpha}_P + \ell m \widetilde{\beta}_Q\right) R_2 \rangle & \text{if} \quad b = 0, \\ \langle \left(\widetilde{\alpha}_P + \ell m \widetilde{\alpha}_Q\right) R_1 + \left(1 + \ell m \widetilde{\beta}_Q\right) R_2 \rangle & \text{if} \quad b = 1. \end{cases} \end{aligned}$$

Thus, since $1 + \ell m \widetilde{\alpha}_Q, 1 + \ell m \widetilde{\beta}_Q \in \mathbb{Z}_n^*$ (recall that $n = \ell^e$), we conclude that

$$\langle P + \ell m Q \rangle = \begin{cases} \langle R_1 + \left(1 + \ell m \widetilde{\alpha}_Q\right)^{-1} \left(\widetilde{\alpha}_P + \ell m \widetilde{\beta}_Q\right) R_2 \rangle & \text{if} \quad b = 0, \\ \langle \left(1 + \ell m \widetilde{\beta}_Q\right)^{-1} \left(\widetilde{\alpha}_P + \ell m \widetilde{\alpha}_Q\right) R_1 + R_2 \rangle & \text{if} \quad b = 1. \end{cases}$$

Decompressing in this way costs only a handful of field operations in $\mathbb{F}_{p^2}$ in addition to a 1-dimensional scalar multiplication on E_A. Since the scalar multiplication is already part of the SIDH scheme, this makes the cost of decompression essentially the cost of generating $\{R_1, R_2\}$. This is done exactly as explained in Sect. 3.

7 Implementation Details

To evaluate the performance of the new compression and decompression, we implemented the proposed algorithms in plain C and wrapped them around the SIDH software from [7]. This library supports a supersingular isogeny class defined over $p = 2^{372} \cdot 3^{239} - 1$, which contains curves of order $(2^{372} \cdot 3^{239})^2$. These parameters target 128 bits of post-quantum security.

Table 3 summarizes the results after benchmarking the software with the clang compiler v3.8.0 on a 3.4 GHz Intel Core i7-4770 Haswell processor running Ubuntu 14.04 LTS with TurboBoost turned off. The details in the table include the size of compressed and uncompressed public keys, the performance of Alice's and Bob's key exchange computations using compression, the performance of the proposed compression and decompression routines, and the total costs of SIDH key exchange with and without the use of compression. These results are compared with those from the prior work by Azarderakhsh et al. [2], which uses a supersingular isogeny class defined over $p = 2^{387} \cdot 3^{242} - 1$.

Table 3. Comparison of SIDH key exchange and public key compression implementations targeting the 128-bit post-quantum and 192-bit classical security level. Benchmarks for our implementation were done on a 3.4 GHz Intel Core i7-4770 Haswell processor running Ubuntu 14.04 LTS with TurboBoost disabled. Results for [2], obtained on a 4.0 GHz Intel Core i7-4790K Haswell processor, were scaled from seconds to cycles using the CPU frequency; the use of TurboBoost is not specified in [2]. The performance results, expressed in millions of clock cycles, were rounded to the nearest 10^6 cycles.

Implementation		This work	Prior work [2]
Public key (bytes)	Uncompressed	564	768
	Compressed	328 (Alice) 330 (Bob)	385
Cycles (cc $\times 10^6$)	Alice's keygen + shared key	80	NA
	Alice's compression	109	6,081
	Alice's decompression	42	539
	Bob's keygen + shared key	92	NA
	Bob's compression	112	7,747
	Bob's decompression	34	493
	Total (no compression)	192	535
	Total (compression)	469	15,395

As can be seen in Table 3, the new algorithms for compression and decompression are significantly faster than those from [2]: compression is up to 66 times faster, while decompression is up to 15 times faster. Similarly, the full key exchange with compressed public keys can be performed about 30 times faster.

Even though part of these speedups can indeed be attributed to the efficiency of the original SIDH library, this only represents a very small fraction of the performance difference (note that the original key exchange from the SIDH library is only 2.8 times faster than the corresponding result from [2]).

Our experimental results show that the use of compressed public keys introduces a factor-2.4 slowdown to SIDH. However, the use of compact keys (in this case, of 330 bytes) can now be considered practical; e.g., one round of SIDH key exchange is computed in only 150 ms on the targeted platform.

Acknowledgements. We thank Drew Sutherland for his helpful discussions concerning the optimization of the Pohlig-Hellman algorithm, and the anonymous Eurocrypt reviewers for their useful comments.

References

1. Azarderakhsh, R., Fishbein, D., Jao, D.: Efficient implementations of a quantum-resistant key-exchange protocol on embedded systems. Technical report (2014). http://cacr.uwaterloo.ca/techreports/2014/cacr2014-20.pdf
2. Azarderakhsh, R., Jao, D., Kalach, K., Koziel, B., Leonardi, C.: Key compression for isogeny-based cryptosystems. In: Emura, K., Hanaoka, G., Zhang, R. (eds.) Proceedings of the 3rd ACM International Workshop on ASIA Public-Key Cryptography, AsiaPKC@AsiaCCS, Xi'an, China, 30 May–03 June 2016, pp. 1–10. ACM (2016)
3. Barreto, P.S.L.M., Galbraith, S.D., O'Eigeartaigh, C., Scott, M.: Efficient pairing computation on supersingular abelian varieties. Des. Codes Crypt. **42**(3), 239–271 (2007)
4. Barreto, P.S.L.M., Kim, H.Y., Lynn, B., Scott, M.: Efficient algorithms for pairing-based cryptosystems. In: Yung, M. (ed.) CRYPTO 2002. LNCS, vol. 2442, pp. 354–369. Springer, Heidelberg (2002). doi:10.1007/3-540-45708-9_23
5. Bernstein, D.J., Hamburg, M., Krasnova, A., Lange, T.: Elligator: elliptic-curve points indistinguishable from uniform random strings. In: Sadeghi, A.-R., Gligor, V.D., Yung, M. (eds.) 2013 ACM SIGSAC Conference on Computer and Communications Security, CCS 2013, Berlin, Germany, 4–8 November 2013, pp. 967–980. ACM (2013)
6. Chen, L., Jordan, S., Liu, Y.-K., Moody, D., Peralta, R., Perlner, R., Smith-Tone, D.: Report on post-quantum cryptography. NISTIR 8105, DRAFT (2016). http://csrc.nist.gov/publications/drafts/nistir-8105/nistir_8105_draft.pdf
7. Costello, C., Longa, P., Naehrig, M.: Efficient algorithms for supersingular isogeny Diffie-Hellman. In: Robshaw, M., Katz, J. (eds.) CRYPTO 2016. LNCS, vol. 9814, pp. 572–601. Springer, Heidelberg (2016). doi:10.1007/978-3-662-53018-4_21
8. De Feo, L., Jao, D., Plût, J.: Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. J. Math. Cryptol. **8**(3), 209–247 (2014)
9. Fujisaki, E., Okamoto, T.: Secure integration of asymmetric and symmetric encryption schemes. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 537–554. Springer, Heidelberg (1999). doi:10.1007/3-540-48405-1_34
10. Galbraith, S.D., Harrison, K., Soldera, D.: Implementing the tate pairing. In: Fieker, C., Kohel, D.R. (eds.) ANTS 2002. LNCS, vol. 2369, pp. 324–337. Springer, Heidelberg (2002). doi:10.1007/3-540-45455-1_26

11. Galbraith, S.D., Petit, C., Shani, B., Ti, Y.B.: On the security of supersingular isogeny cryptosystems. In: Cheon, J.H., Takagi, T. (eds.) ASIACRYPT 2016. LNCS, vol. 10031, pp. 63–91. Springer, Heidelberg (2016). doi:10.1007/978-3-662-53887-6_3
12. Hess, F.: Pairing lattices. In: Galbraith, S.D., Paterson, K.G. (eds.) Pairing 2008. LNCS, vol. 5209, pp. 18–38. Springer, Heidelberg (2008). doi:10.1007/978-3-540-85538-5_2
13. Hess, F., Smart, N.P., Vercauteren, F.: The eta pairing revisited. IEEE Trans. Inf. Theory **52**(10), 4595–4602 (2006)
14. Husemöller, D.: Elliptic Curves. Graduate Texts in Mathematics, vol. 111. Springer, Heidelberg (2004)
15. Jao, D., Feo, L.: Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. In: Yang, B.-Y. (ed.) PQCrypto 2011. LNCS, vol. 7071, pp. 19–34. Springer, Heidelberg (2011). doi:10.1007/978-3-642-25405-5_2
16. Kaliski Jr., B.S.: The Montgomery inverse and its applications. IEEE Trans. Comput. **44**(8), 1064–1065 (1995)
17. Kirkwood, D., Lackey, B.C., McVey, J., Motley, M., Solinas, J.A., Tuller, D.: Failure is not an option: standardization issues for post-quantum key agreement. In: Talk at NIST Workshop on Cybersecurity in a Post-Quantum World, April 2015. http://www.nist.gov/itl/csd/ct/post-quantum-crypto-workshop-2015.cfm
18. Lenstra, A.K., Verheul, E.R.: The XTR public key system. In: Bellare, M. (ed.) CRYPTO 2000. LNCS, vol. 1880, pp. 1–19. Springer, Heidelberg (2000). doi:10.1007/3-540-44598-6_1
19. McEliece, R.J.: A public-key cryptosystem based on algebraic coding theory. Coding Thv **4244**, 114–116 (1978)
20. Miller, V.S.: The Weil pairing, and its efficient calculation. J. Cryptol. **17**(4), 235–261 (2004)
21. Montgomery, P.L.: Speeding the Pollard and elliptic curve methods of factorization. Math. Comput. **48**(177), 243–264 (1987)
22. Peikert, C.: Lattice cryptography for the internet. In: Mosca, M. (ed.) PQCrypto 2014. LNCS, vol. 8772, pp. 197–219. Springer, Cham (2014). doi:10.1007/978-3-319-11659-4_12
23. Pohlig, S.C., Hellman, M.E.: An improved algorithm for computing logarithms over GF(p) and its cryptographic significance. IEEE Trans. Inf. Theory **24**(1), 106–110 (1978)
24. Regev, O.: On lattices, learning with errors, random linear codes, and cryptography. In: Gabow, H.N., Fagin, R. (eds.) Proceedings of the 37th Annual ACM Symposium on Theory of Computing, Baltimore, MD, USA, 22–24 May 2005, pp. 84–93. ACM (2005)
25. Rubin, K., Silverberg, A.: Torus-based cryptography. In: Boneh, D. (ed.) CRYPTO 2003. LNCS, vol. 2729, pp. 349–365. Springer, Heidelberg (2003). doi:10.1007/978-3-540-45146-4_21
26. Schaefer, E., Stoll, M.: How to do a p-descent on an elliptic curve. Trans. Am. Math. Soc. **356**(3), 1209–1231 (2004)
27. Scott, M.: Implementing cryptographic pairings. In: Takagi, T., Okamoto, E., Okamoto, T., Okamoto, T. (eds.) Pairing 2007. LNCS, vol. 4575, pp. 177–196. Springer, Heidelberg (2007)
28. Shanks, D.: Class number, a theory of factorization, and genera. In: Proceedings of the Symposium in Pure Mathematics, vol. 20, pp. 415–440 (1971)
29. Silverman, J.H.: The Arithmetic of Elliptic Curves. Graduate Texts in Mathematics, 2nd edn. Springer, Heidelberg (2009)

30. Smith, P., Skinner, C.: A public-key cryptosystem and a digital signature system based on the Lucas function analogue to discrete logarithms. In: Pieprzyk, J., Safavi-Naini, R. (eds.) ASIACRYPT 1994. LNCS, vol. 917, pp. 355–364. Springer, Heidelberg (1995). doi:10.1007/BFb0000447
31. Stam, M., Lenstra, A.K.: Efficient subgroup exponentiation in quadratic and sixth degree extensions. In: Kaliski, B.S., Koç, K., Paar, C. (eds.) CHES 2002. LNCS, vol. 2523, pp. 318–332. Springer, Heidelberg (2003). doi:10.1007/3-540-36400-5_24
32. Sutherland, A.V.: Structure computation and discrete logarithms in finite abelian p-groups. Math. Comput. **80**(273), 477–500 (2011)
33. Verheul, E.R.: Evidence that XTR is more secure than supersingular elliptic curve cryptosystems. J. Cryptol. **17**(4), 277–296 (2004)