# Quantum circuits for the CSIDH: optimizing quantum evaluation of isogenies

Daniel J. Bernstein[1], Tanja Lange[2], Chloe Martindale[2], Lorenz Panny[2]

[1] Department of Computer Science
University of Illinois at Chicago
Chicago, IL 60607–7045, USA
djb@cr.yp.to

[2] Department of Mathematics and Computer Science
Technische Universiteit Eindhoven
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
tanja@hyperelliptic.org
chloemartindale@gmail.com
lorenz@yx7.cc

**Abstract.** Choosing safe post-quantum parameters for the new CSIDH isogeny-based key-exchange system requires concrete analysis of the cost of quantum attacks. The two main contributions to attack cost are the number of queries in hidden-shift algorithms and the cost of each query. This paper analyzes algorithms for each query, introducing several new speedups while showing that some previous claims were too optimistic for the attacker. This paper includes a full computer-verified simulation of its main algorithm down to the bit-operation level.

**Keywords:** Elliptic curves, isogenies, circuits, constant-time computation, reversible computation, quantum computation, cryptanalysis.

## 1 Introduction

Castryck, Lange, Martindale, Panny, and Renes recently introduced CSIDH [15], an isogeny-based key exchange that runs efficiently and permits non-interactive key exchange. Like the original CRS [20, 64, 68] isogeny-based cryptosystem, CSIDH has public keys and ciphertexts only about twice as large as traditional

* Author list in alphabetical order; see https://www.ams.org/profession/leaders/culture/CultureStatement04.pdf. This work was supported in part by the Commission of the European Communities through the Horizon 2020 program under project number 643161 (ECRYPT-NET), 645622 (PQCRYPTO), 645421 (ECRYPT-CSA), and CHIST-ERA USEIT (NWO project 651.002.004); the Netherlands Organisation for Scientific Research (NWO) under grants 628.001.028 (FASOR) and 639.073.005; and the U.S. National Science Foundation under grant 1314919. "Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation" (or other funding agencies). Permanent ID of this document: 9b88023d7d9ef3f55b11b6f009131c9f. Date of this document: 2019.02.28.

elliptic-curve keys and ciphertexts for a similar security level against all pre-quantum attacks known. CRS was accelerated recently by De Feo, Kieffer, and Smith [23]; CSIDH builds upon this and chooses curves in a different way, obtaining much better speed.

For comparison, the SIDH (and SIKE) isogeny-based cryptosystems [37, 22, 36] are somewhat faster than CSIDH, but they do not support non-interactive key exchange, and their public keys and ciphertexts are 6 times larger[3] than in CSIDH. Furthermore, there are concerns that the extra information in SIDH keys might allow attacks; see [58].

These SIDH disadvantages come from avoiding the commutative structure used in CRS and now in CSIDH. SIDH deliberately avoids this structure because the structure allows *quantum* attacks that asymptotically take subexponential time; see below. The CRS/CSIDH key size thus grows superlinearly in the post-quantum security level. For comparison, if the known attacks are optimal, then the SIDH key size grows linearly in the post-quantum security level.

However, even in a post-quantum world, it is not at all clear how much weight to put on these asymptotics. It is not clear, for example, how large the keys will have to be before the subexponential attacks begin to outperform the exponential-time non-quantum attacks or an exponential-time Grover search. It is not clear when the superlinear growth in CSIDH key sizes will outweigh the factor 6 mentioned above. For applications that need non-interactive key exchange in a post-quantum world, the SIDH/SIKE family is not an option, and it is important to understand what influence these attacks have upon CSIDH key sizes. The asymptotic performance of these attacks is stated in [15], but it is challenging to understand the concrete performance of these attacks for specific CSIDH parameters.

**1.1. Contributions of this paper.** The most important bottleneck in the quantum attacks mentioned above is the cost of evaluating a group action, a series of isogenies, in superposition. Each quantum attack incurs this cost many times; see below. The goals of this paper are to analyze and optimize this cost. We focus on CSIDH because CSIDH is much faster than CRS.

Our main result has the following shape: the CSIDH group action can be carried out in $B$ nonlinear bit operations (counting ANDs and ORs, allowing free XORs and NOTs) with failure probability at most $\epsilon$. (All of our algorithms know when they have failed.) This implies a reversible computation of the CSIDH group action with failure probability at most $\epsilon$ using at most $2B$ Toffoli gates (allowing free NOTs and CNOTs). This in turn implies a quantum computation of the CSIDH group action with failure probability at most $\epsilon$ using at most $14B$

[3] When the goal is for pre-quantum attacks to take $2^\lambda$ operations (without regard to memory consumption), CRS, CSIDH, SIDH, and SIKE all choose primes $p \approx 2^{4\lambda}$. The CRS and CSIDH keys and ciphertexts use (approximately) $\log_2 p \approx 4\lambda$ bits, whereas the SIDH and SIKE keys and ciphertexts use $6\log_2 p \approx 24\lambda$ bits for 3 elements of $\mathbb{F}_{p^2}$. There are compressed variants of SIDH that reduce $6\log_2 p$ to $4\log_2 p \approx 16\lambda$ (see [1]) and to $3.5\log_2 p \approx 14\lambda$ (see [19] and [75]), at some cost in run time.

$T$-gates (allowing free Clifford gates). Appendix A reviews these cost metrics and their relationships.

We explain how to compute pairs $(B, \epsilon)$ for any given CSIDH parameters. For example, we show how to compute CSIDH-512 for uniform random exponent vectors in $\{-5, \ldots, 5\}^{74}$ using

- 1118827416420 $\approx 2^{40}$ nonlinear bit operations using the algorithm of Section 7, or
- 765325228976 $\approx 0.7 \cdot 2^{40}$ nonlinear bit operations using the algorithm of Section 8,

in both cases with failure probability below $2^{-32}$. CSIDH-512 is the smallest parameter set considered in [15]. For comparison, computing the same action with failure probability $2^{-32}$ using the Jao–LeGrow–Leonardi–Ruiz-Lopez algorithm [38], with the underlying modular multiplications computed by the same algorithm as in Roetteler–Naehrig–Svore–Lauter [63], would use approximately $2^{51}$ nonlinear bit operations.

We exploit a variety of algorithmic ideas, including several new ideas pushing beyond the previous state of the art in isogeny computation, with the goal of obtaining the best pairs $(B, \epsilon)$. We introduce a new constant-time variable-degree isogeny algorithm, a new application of the Elligator map, new ways to handle failures in isogeny computations, new combinations of the components of these computations, new speeds for integer multiplication, and more.

**1.2. Impact upon quantum attacks.** Kuperberg [46] introduced an algorithm using $\exp\big((\log N)^{1/2+o(1)}\big)$ queries and $\exp\big((\log N)^{1/2+o(1)}\big)$ operations on $\exp((\log N)^{1/2+o(1)})$ qubits to solve the order-$N$ dihedral hidden-subgroup problem. Regev [61] introduced an algorithm using only a polynomial number of qubits, although with a worse $o(1)$ for the number of queries and operations. A followup paper by Kuperberg [47] introduced further algorithmic options.

Childs, Jao, and Soukharev [17] pointed out that these algorithms could be used to attack CRS. They analyzed the asymptotic cost of a variant of Regev's algorithm in this context. This cost is dominated by queries, in part because the number of queries is large but also because the cost of each query is large. Each query evaluates the CRS group action using a superposition of group elements.

We emphasize that computing the exact attack costs for any particular set of CRS or CSIDH parameters is complicated and requires a lot of new work. The main questions are (1) the exact number of queries for various dihedral-hidden-subgroup algorithms, not just asymptotics; and (2) the exact cost of each query, again not just asymptotics.

The first question is outside the scope of our paper. Some of the simpler algorithms were simulated for small sizes in [46], [10], and [11], but Kuperberg commented in [46, page 5] that his "experiments with this simulator led to a false conjecture for [the] algorithm's precise query complexity".

Our paper addresses the second question for CSIDH: the concrete cost of quantum algorithms for evaluating the action of the class group, which means computing isogenies of elliptic curves in superposition.

**1.3. Comparison to previous claims regarding query cost.** Bonnetain and Schrottenloher claim in [11, online versions 4, 5, and 6] that CSIDH-512 can be broken in "only" $2^{71}$ quantum gates, where each query uses $2^{37}$ quantum gates ("Clifford+T" gates; see Appendix A.4).

We work in the same simplified model of counting operations, allowing any number of qubits to be stored for free. We further simplify by counting only $T$-gates. We gain considerable performance from optimizations not considered in [11]. We take the best possible distribution of input vectors, disregarding the $2^2$ overhead estimated in [11]. Our final gate counts for each query are nevertheless much higher than the $2^{37}$ claimed in [11]. Even assuming that [11] is correct regarding the number of queries, the cost of each query pushes the total attack cost above $2^{80}$.

The query-cost calculation in [11] is not given in enough detail for full reproducibility. However, some details are provided, and given these details we conclude that costly parts of the computation are overlooked in [11] in at least three ways. First, to estimate the number of quantum gates for multiplication in $\mathbb{F}_p$, [11] uses a count of nonlinear bit operations for multiplication in $\mathbb{F}_2[x]$, not noticing that all known methods for multiplication in $\mathbb{Z}$ (never mind reduction modulo $p$) involve many more nonlinear bit operations than multiplication in $\mathbb{F}_2[x]$. Second, at a higher level, the strategy for computing an $\ell$-isogeny requires first finding a point of order $\ell$, an important cost not noticed in [11]. Third, [11] counts the number of operations in a *branching* algorithm, not noticing the challenge of building a *non-branching* (constant-time) algorithm for the same task, as required for computations in superposition. Our analysis addresses all of these issues and more.

**1.4. Memory consumption.** We emphasize that our primary goal is to minimize the number of bit operations. This cost metric pays no attention to the fact that the resulting quantum algorithm for, e.g., CSIDH-512 uses a quantum computer with $2^{40}$ qubits.

Most of the quantum-algorithms literature pays much more attention to the number of qubits. This is why [17], for example, uses a Regev-type algorithm instead of Kuperberg's algorithm. Similarly, [15] takes Regev's algorithm "as a baseline" given "the larger memory requirement" for Kuperberg's algorithm.

An obvious reason to keep the number of qubits under control is the difficulty of scaling quantum computers up to a huge number of qubits. Post-quantum cryptography starts from the assumption that there will be enough scalability to build a quantum computer using thousands of logical qubits to run Shor's algorithm, but this does not imply that a quantum computer with millions of logical qubits will be only 1000 times as expensive, given limits on physical chip size and costs of splitting quantum computation across multiple chips.

On the other hand, [11] chooses Kuperberg's algorithm, and claims that the number of qubits used in Kuperberg's algorithm is not a problem:

> The algorithm we consider has a subexponential memory cost. More precisely, it needs exactly one qubit per query, plus the fixed overhead of the oracle, which can be neglected.

Concretely, for CSIDH-512, [11, online versions 1, 2, 3] claim $2^{29.5}$ qubits, and [11, online versions 4, 5, 6] claim $2^{31}$ qubits. However, no justification is provided for the claim that the number of qubits for the oracle "can be neglected". There is no analysis in [11] of the number of qubits used for the oracle.

We are not saying that our techniques *need* $2^{40}$ qubits. On the contrary: later we mention various ways in which the number of qubits can be reduced with only moderate costs in the number of operations. However, one cannot trivially extrapolate from the memory consumption of CSIDH software (a few kilobytes) to the number of qubits used in a quantum computation. The requirement of reversibility makes it more challenging and more expensive to reduce space, since intermediate results cannot simply be erased. See Appendix A.3.

Furthermore, even if enough qubits are available, simply counting qubit operations ignores critical bottlenecks in quantum computation. Fault-tolerant quantum computation corrects errors in every qubit at every time step, even if the qubit is merely being stored; see Appendix A.5. Communicating across many qubits imposes further costs; see Appendix A.6. It is thus safe to predict that the actual cost of a quantum CSIDH query will be much larger than indicated by our operation counts. Presumably the gap will be larger than the gap for, e.g., the AES attack in [28], which has far fewer idle qubits and much less communication overhead.

**1.5. Acknowledgments.** Thanks to Bo-Yin Yang for suggesting factoring the average over vectors of the generating function in Section 7.3. Thanks to Joost Renes for his comments.

## 2 Overview of the computation

We recall the definition of the CSIDH group action, focusing on the computational aspects of the concrete construction rather than discussing the general case of the underlying algebraic theory.

**Parameters.** The only parameter in CSIDH is a prime number $p$ of the form $p = 4 \cdot \ell_1 \cdots \ell_n - 1$, where $\ell_1 < \cdots < \ell_n$ are (small) odd primes and $n \geq 1$. Note that $p \equiv 3 \pmod 8$ and $p > 3$.

**Notation.** For each $A \in \mathbb{F}_p$ with $A^2 \neq 4$, define $E_A$ as the Montgomery curve $y^2 = x^3 + Ax^2 + x$ over $\mathbb{F}_p$. This curve $E_A$ is supersingular, meaning that $\#E_A(\mathbb{F}_p) \equiv 1 \pmod p$, if and only if it has trace zero, meaning that $\#E_A(\mathbb{F}_p) = p + 1$. Here $E_A(\mathbb{F}_p)$ means the group of points of $E_A$ with coordinates in $\mathbb{F}_p$, including the neutral element at $\infty$; and $\#E_A(\mathbb{F}_p)$ means the number of points.

Define $S_p$ as the set of $A$ such that $E_A$ is supersingular. For each $A \in S_p$ and each $i \in \{1, \ldots, n\}$, there is a unique $B \in S_p$ such that there is an $\ell_i$-isogeny from $E_A$ to $E_B$ whose kernel is $E_A(\mathbb{F}_p)[\ell_i]$, the set of points $Q \in E_A(\mathbb{F}_p)$ with $\ell_i Q = 0$. Define $\mathcal{L}_i(A) = B$. One can show that $\mathcal{L}_i$ is invertible: specifically, $\mathcal{L}_i^{-1}(A) = -\mathcal{L}_i(-A)$. Hence $\mathcal{L}_i^e$ is defined for each integer $e$.

**Inputs and output.** Given an element $A \in S_p$ and a list $(e_1, \ldots, e_n)$ of integers, the CSIDH group action computes $\mathcal{L}_1^{e_1}(\mathcal{L}_2^{e_2}(\cdots(\mathcal{L}_n^{e_n}(A))\cdots)) \in S_p$.

**2.1. Distribution of exponents.** The performance of our algorithms depends on the distribution of the exponent vectors $(e_1, \ldots, e_n)$, which in turn depends on the context.

Constructively, [15] proposes to sample each $e_i$ independently and uniformly from a small range $\{-C, \ldots, C\}$. For example, CSIDH-512 in [15] has $n = 74$ and uses the range $\{-5, \ldots, 5\}$, so there are $11^{74} \approx 2^{256}$ equally likely exponent vectors. We emphasize, however, that all known attacks actually use considerably larger exponent vectors. This means that the distribution of exponents $(e_1, \ldots, e_n)$ our quantum oracle has to process is *not* the same as the distribution used constructively.

The first step in the algorithms of Kuperberg and Regev, applied to a finite abelian group $G$, is to generate a uniform superposition over all elements of $G$. CRS and CSIDH define a map from vectors $(e_1, \ldots, e_n)$ to elements $\mathfrak{l}_1^{e_1} \cdots \mathfrak{l}_n^{e_n}$ of the ideal-class group $G$. This map has a high chance of being surjective but it is far from injective: its kernel is a lattice of rank $n$. Presumably taking, e.g., $17^{74}$ length-74 vectors with entries in the range $\{-8, \ldots, 8\}$ produces a close-to-uniform distribution of elements of the CSIDH-512 class group, but the literature does not indicate how Kuperberg's algorithm behaves when each group element is represented as many different strings.

In his original paper on CRS, Couveignes [20] suggested instead generating a unique vector representing each group element as follows. Compute a basis for the lattice mentioned above; on a quantum computer this can be done using Shor's algorithm [67] which runs in polynomial time, and on a conventional computer this can be done using Hafner and McCurley's algorithm [29] which runs in subexponential time. This basis reveals the group size $\#G$ and an easy-to-sample set $R$ of representatives for $G$, such as $\{(e_1, 0, \ldots, 0) : 0 \leq e_1 < \#G\}$ in the special case that $\mathfrak{l}_1$ generates $G$; for the general case see, e.g., [50, Section 4.1]. Reduce each representative to a short representative, using an algorithm that finds a close lattice vector. If this algorithm is deterministic (for example, if all randomness used in the algorithm is replaced by pseudorandomness generated from the input) then applying it to a uniform superposition over $R$ produces a uniform superposition over a set of short vectors uniquely representing $G$.

The same idea was mentioned in the Childs–Jao–Soukharev paper [17] on quantum attacks against CRS, and in the description of quantum attacks in the CSIDH paper. However, close-vector problems are not easy, even in dimensions as small as 74. Bonnetain and Schrottenloher [11] estimate that CSIDH-512 exponent vectors can be found whose 1-norm is 4 times larger than vectors used constructively. They rely on a very large precomputation, and they do not justify their assumption that the 1-norm, rather than the $\infty$-norm, measures the cost of a class-group action in superposition. Jao, LeGrow, Leonardi, and Ruiz-Lopez [38] present an algorithm that guarantees $(\log p)^{O(1)}$ bits in each exponent, i.e., in the $\infty$-norm, but this also requires a subexponential-time precomputation, and the exponents appear to be rather large.

Perhaps future research will improve the picture of how much precomputation time and per-vector computation time is required for algorithms that find vectors

of a specified size; or, alternatively, will show that Kuperberg-type algorithms can handle non-unique representatives of group elements. The best conceivable case for the attacker is the distribution used in CSIDH itself, and we choose this distribution as an illustration in analyzing the concrete cost of our algorithms.

**2.2. Verification of costs.** To ensure that we are correctly computing the number of bit operations in our group-action algorithms, we have built a bit-operation simulator, and implemented our algorithms inside the simulator. The simulator is available from `https://quantum.isogeny.org/software.html`.

The simulator has a very small core that implements—and counts the number of—NOT, XOR, AND, and OR operations. Higher-level algorithms, from basic integer arithmetic up through isogeny computation, are built on top of this core.

The core also encapsulates the values of bits so that higher-level algorithms do not accidentally inspect those values. There is an explicit mechanism to break the encapsulation so that output values can be checked against separate computations in the Sage computer-algebra system.

**2.3. Verification of failure probabilities.** Internally, each of our group-action algorithms moves the exponent vector $(e_1, \ldots, e_n)$ step by step towards 0. The algorithm fails if the vector does not reach 0 within the specified number of iterations. Analyzing the failure probability requires analyzing how the distribution of exponent vectors interacts with the distribution of curve points produced inside the algorithm; each $e_i$ step relies on finding a point of order $\ell_i$.

We mathematically calculate the failure probability in a model where each generated curve point has probability $1 - 1/\ell_i$ of having order divisible by $\ell_i$, and where these probabilities are all independent. The model would be exactly correct if each point were generated independently and uniformly at random. We actually generate points differently, so there is a risk of our failure-probability calculations being corrupted by inaccuracies in the model. To address this risk, we have carried out various point-generation experiments, suggesting that the model is reasonably accurate. Even if the model is inaccurate, one can compensate with a minor increase in costs. See Sections 4.3 and 5.2.

There is a more serious risk of errors in the failure-probability calculations that we carry out within the model. To reduce this risk, we have carried out $10^7$ simple trials of the following type for each algorithm: generate a random exponent vector, move it step by step towards 0 the same way the algorithm does (in the model), and see how many iterations are required. The observed distribution of the number of iterations is consistent with the distribution that we calculate mathematically. Of course, if there is a calculation error that somehow affects only very small probabilities, then this error will not be caught by only $10^7$ experiments.

**2.4. Structure of the computation.** We present our algorithms from bottom up, starting with scalar multiplication in Section 3, generation of curve points in Section 4, computation of $\mathcal{L}_i$ in Section 5, and computation of the entire CSIDH group action in Sections 6, 7, and 8. Lower-level subroutines for basic integer and modular arithmetic appear in Appendices B and C respectively.

Various sections and subsections mention ideas for saving time beyond what we have implemented in our bit-operation simulator. These ideas include low-level speedups such as avoiding exponentiations in inversions and Legendre-symbol computations (see Appendix C.4), and higher-level speedups such as using division polynomials (Section 9) and/or modular polynomials (Section 10) to eliminate failures for small primes. All of the specific bit-operation counts that we state, such as the $1118827416420 \approx 2^{40}$ nonlinear bit operations mentioned above, are fully implemented.

## 3 Scalar multiplication on an elliptic curve

This section analyzes the costs of scalar multiplication on the curves used in CSIDH, supersingular Montgomery curves $E_A : y^2 = x^3 + Ax^2 + x$ over $\mathbb{F}_p$.

For CSIDH-512, our simulator shows (after our detailed optimizations; see Appendices B and C) that a squaring **S** in $\mathbb{F}_p$ can be computed in 349596 nonlinear bit operations, and that a general multiplication **M** in $\mathbb{F}_p$ can be computed in 447902 nonlinear bit operations, while addition in $\mathbb{F}_p$ takes only 2044 nonlinear bit operations. We thus emphasize the number of **S** and **M** in scalar multiplication (and in higher-level operations), although in our simulator we have also taken various opportunities to eliminate unnecessary additions and subtractions.

**3.1. How curves are represented.** We consider two options for representing $E_A$. The **affine** option uses $A \in \mathbb{F}_p$ to represent $E_A$. The **projective** option uses $A_0, A_1 \in \mathbb{F}_p$, with $A_0 \neq 0$, to represent $E_A$ where $A = A_1/A_0$.

The formulas to produce a curve in Section 5 naturally produce $(A_0, A_1)$ in projective form. Dividing $A_1$ by $A_0$ to produce $A$ in affine form costs an inversion and a multiplication. Staying in projective form is an example of what Appendix C.5 calls "eliminating inversions", but this requires some extra computation when $A$ is used, as we explain below.

The definition of the class-group action requires producing the output $A$ in affine form at the end of the computation. It could also be beneficial to convert each intermediate $A$ to affine form, depending on the relative costs of the inversion and the extra computation.

**3.2. How points are represented.** As in [51, page 425, last paragraph] and [53, page 261], we avoid computing the $y$-coordinate of a point $(x, y)$ on $E_A$. This creates some ambiguity, since the points $(x, y)$ and $(x, -y)$ are both represented as $x \in \mathbb{F}_p$, but the ambiguity does not interfere with scalar multiplication.

We again distinguish between affine and projective representations. As in [5], we represent both $(0, 0)$ and the neutral element on $E_A$ as $x = 0$, and (except where otherwise noted) we allow $X/0$, including $0/0$, as a projective representation of $x = 0$. The projective representation thus uses $X, Z \in \mathbb{F}_p$ to represent $x = X/Z$ if $Z \neq 0$, or $x = 0$ if $Z = 0$. These definitions eliminate branches from the scalar-multiplication techniques that we use.

**3.3. Computing $nP$.** We use the Montgomery ladder to compute $nP$, given a $b$-bit exponent $n$ and a curve point $P$. The Montgomery ladder consists of $b$

"ladder steps" operating on variables $(X_2, Z_2, X_3, Z_3)$ initialized to $(1, 0, x_1, 1)$, where $x_1$ is the $x$-coordinate of $P$. Each ladder step works as follows:

- Conditionally swap $(X_2, Z_2)$ with $(X_3, Z_3)$, where the condition bit in iteration $i$ is bit $n_{b-1-i}$ of $n$. This means computing $X_2 \oplus X_3$, ANDing each bit with the condition bit, and XORing the result into both $X_2$ and $X_3$; and similarly for $Z_2$ and $Z_3$.
- Compute $Y = X_2 - Z_2$, $Y^2$, $T = X_2 + Z_2$, $T^2$, $X_4 = T^2Y^2$, $E = T^2 - Y^2$, and $Z_4 = E(Y^2 + ((A+2)/4)E)$. This is a **point doubling**: it uses $2\mathbf{S} + 3\mathbf{M}$ and a few additions (counting subtractions as additions). We divide $A+2$ by 4 modulo $p$ before the scalar multiplication, using two conditional additions of $p$ and two shifts.
- Compute $C = X_3 + Z_3$, $D = X_3 - Z_3$, $DT$, $CY$, $X_5 = (DT + CY)^2$, and $Z_5 = x_1(DT - CY)^2$. This is a **differential addition**: it also uses $2\mathbf{S} + 3\mathbf{M}$ and a few additions.
- Set $(X_2, Z_2, X_3, Z_3) \leftarrow (X_4, Z_4, X_5, Z_5)$.
- Conditionally swap $(X_2, Z_2)$ with $(X_3, Z_3)$, where the condition bit is again $n_{b-1-i}$. We merge this conditional swap with the conditional swap at the beginning of the next iteration by using $n_{b-i-i} \oplus n_{b-i-2}$ as condition bit.

Then $nP$ has projective representation $(X_2, Z_2)$ by [9, Theorem 4.5]. The overall cost is $4b\mathbf{S} + 6b\mathbf{M}$ plus a small overhead for additions and conditional swaps.

Representing the input point projectively as $X_1/Z_1$ means computing $X_5 = Z_1(DT + CY)^2$ and $Z_5 = X_1(DT - CY)^2$, and starting from $(1, 0, X_1, Z_1)$. This costs $b\mathbf{M}$ extra. Beware that [9, Theorem 4.5] requires $Z_1 \neq 0$.

Similarly, representing $A$ projectively as $A_1/A_0$ means computing $X_4 = T^2(4A_0Y^2)$ and $Z_4 = E(4A_0Y^2 + (A_1 + 2A_0)E)$, after multiplying $Y^2$ by $4A_0$. This also costs $b\mathbf{M}$ extra.

**Other techniques.** The initial $Z_2 = 0$ and $Z_3 = 1$ (for an affine input point) are small, and remain small after the first conditional swap, saving time in the next additions and subtractions. Our framework for tracking sizes of integers recognizes this automatically. The framework does not, however, recognize that half of the output of the last conditional swap is unused. We could use dead-value elimination and other standard peephole optimizations to save bit operations.

Montgomery [53, page 260] considered carrying out many scalar multiplications at once, using affine coordinates for intermediate points inside each scalar multiplication (e.g., $x_2 = X_2/Z_2$), and batching inversions across the scalar multiplications. This could be slightly less expensive than the Montgomery ladder for large $b$, depending on the $\mathbf{S}/\mathbf{M}$ ratio. Our computation of a CSIDH group action involves many scalar multiplications, but not in large enough batches to justify considering affine coordinates for intermediate points. Computing the group action for a batch of inputs might change the picture, but for simplicity we focus on the problem of computing the group action for one input.

A more recent possibility is scalar multiplication on a birationally equivalent Edwards curve. Sliding-window Edwards scalar multiplication is somewhat less expensive than the Montgomery ladder for large $b$; see generally [8] and [34].

On the other hand, for constant-time computations it is important to use fixed windows rather than sliding windows. Despite this difficulty, we estimate that small speedups are possible for $b = 512$.

**3.4. Computing $P, 2P, 3P, \ldots, kP$.** An important subroutine in isogeny computation (see Section 5) is to compute the sequence $P, 2P, 3P, \ldots, kP$ for a constant $k \geq 1$.

We compute $2P$ by a doubling, $3P$ by a differential addition, $4P$ by a doubling, $5P$ by a differential addition, $6P$ by a doubling, etc. In other words, each multiple of $P$ is computed by the Montgomery ladder as above, but these computations are merged across the multiples (and conditional swaps are eliminated). This takes $2(k-1)\mathbf{S} + 3(k-1)\mathbf{M}$ for affine $P$ and affine $A$. Projective $P$ adds $\lfloor (k-1)/2 \rfloor \mathbf{M}$, and projective $A$ adds $\lfloor k/2 \rfloor \mathbf{M}$.

We could instead compute $2P$ by a doubling, $3P$ by a differential addition, $4P$ by a differential addition, $5P$ by a differential addition, $6P$ by a differential addition, etc. This again takes $2(k-1)\mathbf{S} + 3(k-1)\mathbf{M}$ for affine $P$ and affine $A$, but projective $P$ and projective $A$ now have different effects: projective $P$ adds $(k-2)\mathbf{M}$ if $k \geq 2$, and projective $A$ adds $\mathbf{M}$ if $k \geq 2$. The choice here also has an impact on metrics beyond bit operations: doublings increase space requirements but allow more parallelism.

## 4 Generating points on an elliptic curve

This section analyzes the cost of several methods to generate a random point on a supersingular Montgomery curve $E_A : y^2 = x^3 + Ax^2 + x$, given $A \in \mathbb{F}_p$. As in Section 2, $p$ is a standard prime congruent to 3 modulo 8.

Sometimes one instead wants to generate a point on the twist of the curve. The **twist** is the curve $-y^2 = x^3 + Ax^2 + x$ over $\mathbb{F}_p$; note that $-1$ is a non-square in $\mathbb{F}_p$. This curve is isomorphic to $E_{-A}$ by the map $(x, y) \mapsto (-x, y)$. Beware that there are several slightly different concepts of "twist" in the literature; the definition here is the most useful definition for CSIDH, as explained in [15].

**4.1. Random point on curve or twist.** The conventional approach is as follows: generate a uniform random $x \in \mathbb{F}_p$; compute $x^3 + Ax^2 + x$; compute $y = (x^3 + Ax^2 + x)^{(p+1)/4}$; and check that $y^2 = x^3 + Ax^2 + x$.

One always has $y^4 = (x^3 + Ax^2 + x)^{p+1} = (x^3 + Ax^2 + x)^2$ so $\pm y^2 = x^3 + Ax^2 + x$. About half the time, $y^2$ will match $x^3 + Ax^2 + x$; i.e., $(x, y)$ will be a point on the curve. Otherwise $(x, y)$ will be a point on the twist.

Since we work purely with $x$-coordinates (see Section 3.2), we skip the computation of $y$. However, we still need to know whether we have a curve point or a twist point, so we compute the Legendre symbol of $x^3 + Ax^2 + x$ as explained in Appendix C.4.

The easiest distribution of outputs to mathematically analyze is the uniform distribution over the following $p + 1$ pairs:

- $(x, +1)$ where $x$ represents a curve point;
- $(x, -1)$ where $x$ represents a twist point.

One can generate outputs from this distribution as follows: generate a uniform random $u \in \mathbb{F}_p \cup \{\infty\}$; set $x$ to $u$ if $u \in \mathbb{F}_p$ or to 0 if $u = \infty$; compute the Legendre symbol of $x^3 + Ax^2 + x$; and replace symbol 0 with $+1$ if $u = 0$ or $-1$ if $u = \infty$.

For computations, it is slightly simpler to drop the two pairs with $x = 0$: generate a uniform random $x \in \mathbb{F}_p^*$ and compute the Legendre symbol of the value $x^3 + Ax^2 + x$. This generates a uniform distribution over the remaining $p - 1$ pairs.

**4.2. Random point on curve.** What if twist points are useless and the goal is to produce a point specifically on the curve (or vice versa)? One approach is to generate, e.g., 100 random curve-or-twist points as in Section 4.1, and select the first point on the curve. This fails with probability $1/2^{100}$. If a computation involves generating $2^{10}$ points in this way then the overall failure probability is $1 - (1 - 1/2^{100})^{2^{10}} \approx 1/2^{90}$. One can tune the number of generated points according to the required failure probability.

We save time by applying "Elligator" [7], specifically the Elligator 2 map. Elligator 2 is defined for all the curves $E_A$ that we use, *except* the curve $E_0$, which we discuss below. For each of these curves $E_A$, Elligator 2 is a fast injective map from $\{2, 3, \ldots, (p-1)/2\}$ to the set $E_A(\mathbb{F}_p)$ of curve points. This produces only about half of the curve points; see Section 5.2 for analysis of the impact of this nonuniformity upon our higher-level algorithms.

Here are the details of Elligator 2, specialized to these curves, further simplified to avoid computing $y$, and adapted to allow twists as an option:

- Input $A \in \mathbb{F}_p$ with $A^2 \neq 4$ and $A \neq 0$.
- Input $s \in \{1, -1\}$. This procedure generates a point on $E_A$ if $s = 1$, or on the twist of $E_A$ if $s = -1$.
- Input $u \in \{2, 3, \ldots, (p-1)/2\}$.
- Compute $v = A/(u^2 - 1)$.
- Compute $e$, the Legendre symbol of $v^3 + Av^2 + v$.
- Compute $x$ as $v$ if $e = s$, otherwise $-v - A$.

To see that this works, note first that $v$ is defined since $u^2 \neq 1$, and is nonzero since $A \neq 0$. One can also show that $A^2 - 4$ is nonsquare for all of the CSIDH curves, so $v^3 + Av^2 + v \neq 0$, so $e$ is 1 or $-1$. If $e = s$ then $x = v$ so $x^3 + Ax^2 + x$ is a square for $s = 1$ and a nonsquare for $s = -1$. Otherwise $e = -s$ and $x = -v - A$ so $x^3 + Ax^2 + x = -u^2(v^3 + Av^2 + v)$, which is a square for $s = 1$ and a nonsquare for $s = -1$. This uses that $v$ and $-v - A$ satisfy $(-v - A)^2 + A(-v - A) + 1 = v^2 + Av + 1$ and $-v - A = -u^2 v$.

The $(p-3)/2$ different choices of $u$ produce $(p-3)/2$ different curve points, but we could produce any particular $x$ output twice since we suppress $y$.

**The case $A = 0$.** One way to extend Elligator 2 to $E_0$ is to set $v = u$ when $A = 0$ instead of $v = A/(u^2 - 1)$. The point of the construction of $v$ is that $x^3 + Ax^2 + x$ for $x = -v - A$ is a non-square times $v^3 + Av^2 + v$, i.e., that $(-v - A)/v$ is a non-square; this is automatic for $A = 0$, since $-1$ is a non-square.

We actually handle $E_0$ in a different way: we precompute a particular base point on $E_0$ whose order is divisible by $(p+1)/4$, and we always return this point if $A = 0$. This makes our higher-level algorithms slightly more effective (but we disregard this improvement in analyzing the success probability of our algorithms), since this point guarantees a successful isogeny computation starting from $E_0$; see Section 5. The same guarantee removes any need to generate other points on $E_0$, and is also useful to start walks in Section 10.

**4.3. Derandomization.** Rather than generating random points, we generate a deterministic sequence of points by taking $u = 2$ for the first point, $u = 3$ for the next point, etc. We precompute the inverses of $1 - 2^2$, $1 - 3^2$, etc., saving bit operations.

An alternative, saving the same number of bit operations, is to precompute inverses of $1 - u^2$ for various random choices of $u$, embedding the inverses into the algorithm. This guarantees that the failure probability of the outer algorithm for any particular input $A$, as the choices of $u$ vary, is the same as the failure probability of an algorithm that randomly chooses $u$ upon demand for each $A$.

We are heuristically assuming that failures are not noticeably correlated across choices of $A$. To replace this heuristic with a proof, one can generate the $u$ sequence randomly for each input. This randomness, in turn, is indistinguishable from the output of a cipher, under the assumption that the cipher is secure. In this setting one cannot precompute the reciprocals of $1 - u^2$, but one can still batch the inversions.

## 5 Computing an $\ell$-isogenous curve

This section analyzes the cost of computing a single isogeny in CSIDH. There are two inputs: $A$, specifying a supersingular Montgomery curve $E_A$ over $\mathbb{F}_p$; and $i$, specifying one of the odd prime factors $\ell_i$ of $(p+1)/4 = \ell_1 \cdots \ell_n$. The output is $B = \mathcal{L}_i(A)$. We abbreviate $\ell_i$ as $\ell$ and $\mathcal{L}_i$ as $\mathcal{L}$.

Recall that $B$ is characterized by the following property: there is an $\ell$-isogeny from $E_A$ to $E_B$ whose kernel is $E_A(\mathbb{F}_p)[\ell]$. Beyond analyzing the costs of computing $B = \mathcal{L}(A)$, we analyze the costs of applying the $\ell$-isogeny to a point on $E_A$, obtaining a point on $E_B$. See Section 5.4.

The basic reason that CSIDH is much faster than CRS is that the CSIDH construction allows (variants of) Vélu's formulas [72, 18, 62] to use points in $E_A(\mathbb{F}_p)$, rather than points defined over larger extension fields. This section focuses on computing $B$ via these formulas. The cost of these formulas is approximately linear in $\ell$, assuming that a point of order $\ell$ is known. There are two important caveats here:

- Finding a point of order $\ell$ is not easy to do in constant time. See Section 5.1. We follow the obvious approach, namely taking an appropriate multiple of a random point; but this is expensive—recall from Section 3 that a 500-bit Montgomery ladder costs $2000\mathbf{S} + 3000\mathbf{M}$ when $A$ and the input point are affine—and has failure probability approximately $1/\ell$.

- In some of our higher-level algorithms, $i$ is a *variable*. Then $\ell = \ell_i$ is also a variable, and Vélu's formulas are variable-time formulas, while we need constant-time computations. Generic branch elimination produces a constant-time computation taking time approximately linear in $\ell_1 + \ell_2 + \cdots + \ell_n$, which is quite slow. However, we show how to do much better, reducing $\ell_1 + \ell_2 + \cdots + \ell_n$ to $\max\{\ell_1, \ell_2, \ldots, \ell_n\}$, by exploiting the internal structure of Vélu's formulas. See Section 5.3.

There are other ways to compute isogenies, as explored in [42, 23]:

- The "Kohel" strategy: Compute a univariate polynomial whose roots are the $x$-coordinates of the points in $E_A(\mathbb{F}_p)[\ell]$. Use Kohel's algorithm [45, Section 2.4], which computes an isogeny given this polynomial. This strategy is (for CSIDH) asymptotically slower than Vélu's formulas, but could nevertheless be faster when $\ell$ is very small. Furthermore, this strategy is deterministic and always works.
- The "modular" strategy: Compute the possible $j$-invariants of $E_B$ by factoring modular polynomials. Determine the correct choice of $B$ by computing the corresponding isogeny kernels or, on subsequent steps, simply by not walking back.

We analyze the Kohel strategy in Section 9, and the modular strategy in Section 10.

**5.1. Finding a point of order $\ell$.** We now focus on the problem of finding a point of order $\ell$ in $E_A(\mathbb{F}_p)$. By assumption $(p+1)/4$ is a product of distinct odd primes $\ell_1, \ldots, \ell_n$; $\ell = \ell_i$ is one of those primes; and $\#E_A(\mathbb{F}_p) = p+1$. One can show that $E_A(\mathbb{F}_p)$ has a point of order 4 and is thus cyclic:

$$E_A(\mathbb{F}_p) \cong \mathbb{Z}/(p+1) \cong \mathbb{Z}/4 \times \mathbb{Z}/\ell_1 \times \cdots \times \mathbb{Z}/\ell_n\,.$$

We *try* to find a point $Q$ of order $\ell$ in $E_A(\mathbb{F}_p)$ as follows:

- Pick a random point $P \in E_A(\mathbb{F}_p)$, as explained in Section 4.
- Compute a "cofactor" $(p+1)/\ell$. To handle the case $\ell = \ell_i$ for variable $i$, we first use bit operations to compute the list $\ell'_1, \ldots, \ell'_n$, where $\ell'_j = \ell_j$ for $j \neq i$ and $\ell'_i = 1$; we then use a product tree to compute $\ell'_1 \cdots \ell'_n$. (Computing $(p+1)/\ell$ by a general division algorithm could be faster, but the product tree is simpler and has negligible cost in context.)
- Compute $Q = ((p+1)/\ell)P$ as explained in Section 3.

If $P$ is a uniform random element of $E_A(\mathbb{F}_p)$ then $Q$ is a uniform random element of $E_A(\mathbb{F}_p)[\ell] \cong \mathbb{Z}/\ell$. The order of $Q$ is thus the desired $\ell$ with probability $1 - 1/\ell$. Otherwise $Q$ is $\infty$, the neutral element on the curve, which is represented by $x = 0$. Checking for $x = 0$ is a reliable way to detect this case: the only other point represented by $x = 0$ is $(0, 0)$, which is outside $E_A(\mathbb{F}_p)[\ell]$.

**Different concepts of constant time.** Beware that there are two different notions of "constant time" for cryptographic algorithms. One notion is that the

time for each operation is independent of *secrets*. This notion allows the CSIDH user to generate a uniform random element of $E_A(\mathbb{F}_p)[\ell]$ and try again if the point is $\infty$, guaranteeing success with an average of $\ell/(\ell-1)$ tries. The time varies, but the variation is independent of the secret $A$.

A stricter notion is that the time for each operation is independent of *all* inputs. The time depends on parameters, such as $p$ in CSIDH, but does not depend on random choices. We emphasize that a quantum circuit operating on many inputs in superposition is, by definition, using this stricter notion. We thus choose the sequence of operations carried out by the circuit, and analyze the probability that this sequence fails.

**Amplifying the success probability.** Having each 3-isogeny fail with probability $1/3$, each 5-isogeny fail with probability $1/5$, etc. creates a correctness challenge for higher-level algorithms that compute many isogenies.

A simple workaround is to generate many points $Q_1, Q_2, \ldots, Q_N$, and use bit operations on the points to select the first point with $x \neq 0$. This fails if all of the points have $x = 0$. Independent uniform random points have overall failure probability $1/\ell^N$. One can make $1/\ell^N$ arbitrarily small by choosing $N$ large enough: for example, $1/3^N$ is below $1/2^{32}$ for $N \geq 21$, and is below $1/2^{256}$ for $N \geq 162$.

We return to the costs of generating so many points, and the costs of more sophisticated alternatives, when we analyze algorithms to compute the CSIDH group action.

**5.2. Nonuniform distribution of points.** We actually generate random points using Elligator (see Section 4.2), which generates only $(p-3)/2$ different curve points $P$. At most $(p+1)/\ell$ of these points produce $Q = \infty$, so the failure chance is at most $(2/\ell)(p+1)/(p-3) \approx 2/\ell$.

This bound cannot be simultaneously tight for $\ell = 3$, $\ell = 5$, and $\ell = 7$ (assuming $3 \cdot 5 \cdot 7$ divides $p+1$): if it were then the Elligator outputs would include all points having orders dividing $(p+1)/3$ or $(p+1)/5$ or $(p+1)/7$, but this accounts for more than 54% of all curve points, contradiction.

Points generated by Elligator actually appear to be much better distributed modulo each $\ell$, with failure chance almost exactly $1/\ell$. Experiments support this conjecture. Readers concerned with the gap between the provable $2/\ell$ and the heuristic $1/\ell$ may prefer to add or subtract a few Elligator 2 outputs, obtaining a distribution provably close to uniform (see [70]) at a moderate cost in performance. A more efficient approach is to accept a doubling of failure probability and use a small number of extra iterations to compensate.

We shall later see other methods of obtaining rational $\ell$-torsion points, e.g., by pushing points through $\ell'$-isogenies. This does not make a difference in the analysis of failure probabilities.

For comparison, generating a random point on the curve or twist (see Section 4.1) has failure probability above $1/2$ at finding a curve point of order $\ell$. See Section 6.2 for the impact of this difference upon higher-level algorithms.

**5.3. Computing an $\ell$-isogenous curve from a point of order $\ell$.** Once we have the $x$-coordinate of a point $Q$ of order $\ell$ in $E_A(\mathbb{F}_p)$, we compute the $x$-coordinates of the points $Q, 2Q, 3Q, \ldots, ((\ell-1)/2)Q$. We use this information to compute $B = \mathcal{L}(A)$, the coefficient determining the $\ell$-isogenous curve $E_B$.

Recall from Section 3.4 that computing $Q, 2Q, 3Q, \ldots, ((\ell-1)/2)Q$ costs $(\ell-3)\mathbf{S} + 1.5(\ell-3)\mathbf{M}$ for affine $Q$ and affine $A$, and just $1\mathbf{M}$ extra for affine $Q$ and projective $A$. The original CSIDH paper [15] took more time here, namely $(\ell-3)\mathbf{S} + 2(\ell-3)\mathbf{M}$, to handle projective $Q$ and projective $A$. We decide, based on comparing $\ell$ to the cost of an inversion, whether to spend an inversion converting $Q$ to affine coordinates.

Given the $x$-coordinates of $Q, 2Q, 3Q, \ldots, ((\ell-1)/2)Q$, the original CSIDH paper [15] took approximately $3\ell\mathbf{M}$ to compute $B$. Meyer and Reith [49] pointed out that CSIDH benefits from Edwards-coordinate isogeny formulas from Moody and Shumow [54]; we reuse this speedup. These formulas work as follows:

- Compute $a = A + 2$ and $d = A - 2$.
- Compute the Edwards $y$-coordinates of $Q, 2Q, 3Q, \ldots, ((\ell-1)/2)Q$. The Edwards $y$-coordinate is related to the Montgomery $x$-coordinate by $y = (x-1)/(x+1)$. We are given each $x$ projectively as $X/Z$, and compute $y$ projectively as $Y/T$ where $Y = X - Z$ and $T = X + Z$. Note that $Y$ and $T$ naturally occur as intermediate values in the Montgomery ladder.
- Compute the product of these $y$-coordinates: i.e., compute $\prod Y$ and $\prod T$. This uses a total of $(\ell-3)\mathbf{M}$.
- Compute $a' = a^\ell(\prod T)^8$ and $d' = d^\ell(\prod Y)^8$. Each $\ell$th power takes a logarithmic number of squarings and multiplications; see Appendix C.4.
- Compute, projectively, $B = 2(a'+d')/(a'-d')$. Subsequent computations decide whether to convert $B$ to affine form.

These formulas are almost three times faster than the formulas used in [15]. The total cost of computing $B$ from $Q$ is almost two times faster than in [15].

**Handling variable $\ell$.** We point out that the isogeny computations for $\ell = 3$, $\ell = 5$, $\ell = 7$, etc. have a Matryoshka-doll structure, allowing a constant-time computation to handle many different values of $\ell$ with essentially the same cost as a single computation for the largest value of $\ell$.

Concretely, the following procedure takes approximately $\ell_n\mathbf{S} + 2.5\ell_n\mathbf{M}$, and allows any $\ell \le \ell_n$. If the context places a smaller upper bound upon $\ell$ then one can replace $\ell_n$ with that upper bound, saving time; we return to this idea later.

Compute the Montgomery $x$-coordinates and the Edwards $y$-coordinates of $Q, 2Q, 3Q, \ldots, ((\ell_n-1)/2)Q$. Use bit operations to replace each Edwards $y$-coordinate with 1 after the first $(\ell-1)/2$ points. Compute the product of these modified $y$-coordinates; this is the desired product of the Edwards $y$-coordinates of the first $(\ell-1)/2$ points. Finish computing $B$ as above. Note that the exponentiation algorithm in Appendix C.4 allows variable $\ell$.

**5.4. Applying an $\ell$-isogeny to a point.** The following formulas define an $\ell$-isogeny from $E_A$ to $E_B$ with kernel $E_A(\mathbb{F}_p)[\ell]$. The $x$-coordinate of the image

of a point $P_1 \in E_A(\mathbb{F}_p)$ under this isogeny is

$$x(P_1) \prod_{j \in \{1,2,\ldots,(\ell-1)/2\}} \left( \frac{x(P_1)x(jQ) - 1}{x(P_1) - x(jQ)} \right)^2 .$$

Each $x(jQ)$ appearing here was computed above in projective form $X/Z$. The ratio $(x(P_1)x(jQ) - 1)/(x(P_1) - x(jQ))$ is $(x(P_1)X - Z)/(x(P_1)Z - X)$. This takes $2\mathbf{M}$ to compute projectively if $x(P_1)$ is affine, and thus $(\ell-1)\mathbf{M}$ across all $j$. Multiplying the numerators takes $((\ell-3)/2)\mathbf{M}$, multiplying the denominators takes $((\ell-3)/2)\mathbf{M}$, squaring both takes $2\mathbf{S}$, and multiplying by $x(P_1)$ takes $1\mathbf{M}$, for a total of $(2\ell-3)\mathbf{M} + 2\mathbf{S}$.

If $x(P_1)$ is instead given in projective form as $X_1/Z_1$ then computing $X_1X - Z_1Z$ and $X_1Z - Z_1X$ might seem to take $4\mathbf{M}$, but one can instead compute the sum and difference of $(X_1 - Z_1)(X + Z)$ and $(X_1 + Z_1)(X - Z)$, using just $2\mathbf{M}$. The only extra cost compared to the affine case is four extra additions. This speedup was pointed out by Montgomery [53] in the context of the Montgomery ladder. The initial CSIDH software accompanying [15] did not use this speedup but [49] mentioned the applicability to CSIDH.

In the opposite direction, if inversion is cheap enough to justify making $x(P_1)$ *and* every $x(jQ)$ affine, then $2\mathbf{M}$ drops to $1\mathbf{M}$, and the total cost drops to approximately $1.5\ell\mathbf{M}$.

As in Section 5.3, we allow $\ell$ to be a variable. The cost of variable $\ell$ is the cost of a single computation for the maximum allowed $\ell$, plus a minor cost for bit operations to select relevant inputs to the product.

## 6 Computing the action: basic algorithms

Jao, LeGrow, Leonardi, and Ruiz-Lopez [38] suggested a three-level quantum algorithm to compute $\mathcal{L}_1^{e_1} \cdots \mathcal{L}_n^{e_n}$. This section shows how to make the algorithm an order of magnitude faster for any particular failure probability.

**6.1. Baseline: reliably computing each $\mathcal{L}_i$.** The lowest level in [38] *reliably* computes $\mathcal{L}_i$ as follows. Generate $r$ uniform random points on the curve or twist, as in Section 4.1. Multiply each point by $(p+1)/\ell_i$, as in Section 5.1, hoping to obtain a point of order $\ell_i$ on the curve. Use Vélu's formulas to finish the computation, as in Section 5.3.

Each point has success probability $(1/2)(1 - 1/\ell_i)$, where $1/2$ is the probability of obtaining a curve point (rather than a twist point) and $1 - 1/\ell_i$ is the probability of obtaining a point of order $\ell_i$ (rather than order 1). The chance of all $r$ points failing is thus $(\ell_i + 1)^r/(2\ell_i)^r$, decreasing from $(2/3)^r$ for $\ell_i = 3$ down towards $(1/2)^r$ as $\ell_i$ grows . One chooses $r$ to obtain a failure probability as small as desired for the isogeny computation, and for the higher levels of the algorithm.

The lowest level optionally computes $\mathcal{L}_i^{-1}$ instead of $\mathcal{L}_i$. The approach in [38], following [15], is to use points on the twist instead of points on the curve; an alternative is to compute $\mathcal{L}_i^{-1}(A)$ as $-\mathcal{L}_i(-A)$.

The middle level of the algorithm computes $\mathcal{L}_i^e$, where $e$ is a variable whose absolute value is bounded by a constant $C$. This level calls the lowest level exactly $C$ times, performing a series of $C$ steps of $\mathcal{L}_i^{\pm 1}$, using bit operations on $e$ to decide whether to retain the results of each step. The $\pm 1$ is chosen as the sign of $e$, or as an irrelevant 1 if $e = 0$.

The highest level of the algorithm computes $\mathcal{L}_1^{e_1} \cdots \mathcal{L}_n^{e_n}$, where each $e_i$ is between $-C$ and $C$, by calling the middle level $n$ times, starting with $\mathcal{L}_1^{e_1}$ and ending with $\mathcal{L}_n^{e_n}$. Our definition of the action applied $\mathcal{L}_n^{e_n}$ first, but the $\mathcal{L}_i$ operators commute with each other, so the order does not matter.

**Importance of bounding each exponent.** We emphasize that this algorithm requires each exponent $e_i$ to be between $-C$ and $C$, i.e., requires the vector $(e_1, \ldots, e_n)$ to have $\infty$-norm at most $C$.

We use $C = 5$ for CSIDH-512 as an illustrative example, but all known attacks use larger vectors (see Section 2.1). $C$ is chosen in [38] so that every input, every vector in superposition, has $\infty$-norm at most $C$; smaller values of $C$ create a failure probability that needs to be analyzed.

We are not saying that the $\infty$-norm is the only important feature of the input vectors. On the contrary: our constant-time subroutine to handle variable-$\ell$ isogenies creates opportunities to share work between separate exponents. See Section 5.3 and Section 7.

**Concrete example.** For concreteness we consider uniform random input vectors $e \in \{-5, \ldots, 5\}^{74}$. The highest level calls the middle level $n = 74$ times, and the middle level calls the lowest level $C = 5$ times. Taking $r = 70$ guarantees failure probability at most $(2/3)^{70}$ at the lowest level, and thus failure probability at most $1 - (1 - (2/3)^{70})^{74 \cdot 5} \approx 0.750 \cdot 2^{-32}$ for the entire algorithm.

This type of analysis is used in [38] to select $r$. We point out that the failure probability of the algorithm is actually lower, and a more accurate analysis allows a smaller value of $r$. One can, for example, replace $(1 - (2/3)^r)^{74}$ with $\prod_i (1 - (\ell_i + 1)^r / (2\ell_i)^r)$, showing that $r = 59$ suffices for failure probability below $2^{-32}$. With more work one can account for the distribution of input vectors $e$, rather than taking the worst-case $e$ as in [38]. However, one cannot hope to do better than $r = 55$ here: there is a $10/11$ chance that at least one 3-isogeny is required, and taking $r \leq 54$ means that this 3-isogeny fails with probability at least $(2/3)^{54}$, for an overall failure chance at least $(10/11)(2/3)^{54} > 2^{-32}$.

With the choice $r = 70$ as in [38], there are $74 \cdot 5 \cdot 70 = 25900$ iterations, in total using more than 100 million multiplications in $\mathbb{F}_p$. In the rest of this section we will reduce the number of iterations by a factor 30, and in Section 7 we will reduce the number of iterations by another factor 3, with only moderate increases in the cost of each iteration.

**6.2. Fewer failures, and sharing failures.** We now introduce Algorithm 6.1, which improves upon the algorithm from [38] in three important ways. First, we use Elligator to target the curve (or the twist if desired); see Section 4.2. This reduces the failure probability of $r$ points from $(2/3)^r$ to, heuristically, $(1/3)^r$ for $\ell_i = 3$; from $(3/5)^r$ to $(1/5)^r$ for $\ell_i = 5$; from $(4/7)^r$ to $(1/7)^r$ for $\ell_i = 7$; etc.

**Algorithm 6.1:** Basic class-group action evaluation.

**Parameters:** Odd primes $\ell_1 < \cdots < \ell_n$, a prime $p = 4\ell_1 \cdots \ell_n - 1$, and positive integers $(r_1, \ldots, r_n)$.
**Input:** $A \in S_p$, integers $(e_1, \ldots, e_n)$.
**Output:** $\mathfrak{L}_1^{e_1} \cdots \mathfrak{L}_n^{e_n}(A)$ or "fail".

**for** $i \leftarrow 1$ **to** $n$ **do**
  **for** $j \leftarrow 1$ **to** $r_i$ **do**
    Let $s = \operatorname{sign}(e_i) \in \{-1, 0, +1\}$.
    Find a random point $P$ on $E_{sA}$ using Elligator.
    Compute $Q \leftarrow ((p+1)/\ell_i)P$.
    Compute $B$ with $E_B \cong E_{sA}/\langle Q \rangle$ if $Q \neq \infty$.
    Set $A \leftarrow sB$ if $Q \neq \infty$ and $s \neq 0$.
    Set $e_i \leftarrow e_i - s$ if $Q \neq \infty$.
Set $A \leftarrow$ "fail" if $(e_1, \ldots, e_n) \neq (0, \ldots, 0)$.
**Return** $A$.

Second, we allow a separate $r_i$ for each $\ell_i$. This lets us exploit the differences in failure probabilities as $\ell_i$ varies.

Third, we handle failures at the middle level instead of the lowest level. The strategy in [38] to compute $\mathfrak{L}_i^e$ with $-C \leq e \leq C$ is to perform $C$ iterations, where each iteration builds up many points on one curve and *reliably* moves to the next curve. We instead perform $r_i$ iterations, where each iteration *tries* to move from one curve to the next by generating just one point. For $C = 1$ this is the same, but for larger $C$ we obtain better tradeoffs between the number of points and the failure probability.

As a concrete example, generating 20 points on one curve with Elligator has failure probability $(1/3)^{20}$ for $\ell_i = 3$. A series of 5 such computations, overall generating 100 points, has failure probability $1 - (1 - (1/3)^{20})^5 \approx 2^{-29.37}$. If we instead perform just 50 iterations, where each iteration generates one point to move 1 step with probability 2/3, then the probability that we move fewer than 5 steps is just $3846601/3^{50} \approx 2^{-57.37}$; see Section 6.3. Our iterations are more expensive than in [38]—next to each Elligator computation, we always perform the steps for computing an $\ell_i$-isogeny, even if $Q = \infty$—but (for CSIDH-512 etc.) this is not a large effect: the cost of each iteration is dominated by scalar multiplication.

We emphasize that all of our algorithms take constant time. When we write "Compute $X \leftarrow Y$ if $c$" we mean that we always compute $Y$ and the bit $c$, and we then replace the $j$th bit $X_j$ of $X$ with the $j$th bit $Y_j$ of $Y$ for each $j$ if $c$ is set, by replacing $X_j$ with $X_j \oplus c(X_j \oplus Y_j)$. This is why Algorithm 6.1 always carries out the bit operations for computing an $\ell_i$-isogenous curve, as noted above, even if $Q = \infty$.

**Table 6.1.** Examples of choices of $r_i$ for Algorithm 6.1 for three levels of failure probability for uniform random CSIDH-512 vectors with entries in $\{-5,\dots,5\}$. Failure probabilities $\epsilon$ are rounded to three digits after the decimal point. The "total" column is $\sum r_i$, the total number of iterations. The "[38]" column is $74 \cdot 5 \cdot r$, the number of iterations in the algorithm of [38], with $r$ chosen as in [38] to have $1-(1-(2/3)^r)^{74\cdot 5}$ at most $2^{-1}$ or $2^{-32}$ or $2^{-256}$. Compare Table 6.2 for $\{-10,\dots,10\}$.

| $\epsilon$ \ $\ell_i$ | 3 | 5 | 7 | 11 | 13 | 17 | ... | 359 | 367 | 373 | 587 | total | [38] |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $0.499 \cdot 2^{-1}$ | 11 | 9 | 8 | 7 | 7 | 6 | ... | 5 | 5 | 5 | 5 | 406 | 5920 |
| $0.178 \cdot 2^{-32}$ | 36 | 25 | 21 | 18 | 17 | 16 | ... | 10 | 10 | 10 | 9 | 869 | 25900 |
| $0.249 \cdot 2^{-256}$ | 183 | 126 | 105 | 85 | 80 | 73 | ... | 37 | 37 | 37 | 34 | 3640 | 167610 |

**6.3. Analysis.** We consider the inner loop body of Algorithm 6.1 for a fixed $i$, hence write $\ell = \ell_i$, $e = e_i$, and $r = r_i$ for brevity.

Heuristically (see Section 5.2), we model each point $Q$ as independent and uniform random in a cyclic group of order $\ell$, so $Q$ has order 1 with probability $1/\ell$ and order $\ell$ with probability $1 - 1/\ell$. The number of points of order $\ell$ through $r$ iterations of the inner loop is binomially distributed with parameters $r$ and $1 - 1/\ell$. The probability that this number is $|e|$ or larger is $\text{prob}_{\ell,e,r} = \sum_{t=|e|}^{r} \binom{r}{t} (1-1/\ell)^t / \ell^{r-t}$. This is exactly the probability that Algorithm 6.1 successfully performs the $|e|$ desired iterations of $\mathcal{L}^{\text{sign}(e)}$.

Let $C$ be a nonnegative integer. The overall success probability of the algorithm for a particular input vector $(e_1,\dots,e_n) \in \{-C,\dots,C\}^n$ is

$$\prod_{i=1}^{n} \text{prob}_{\ell_i,e_i,r_i} \geq \prod_{i=1}^{n} \text{prob}_{\ell_i,C,r_i}.$$

Average over vectors to see that the success probability of the algorithm for a uniform random vector in $\{-C,\dots,C\}^n$ is $\prod_{i=1}^{n} \left( \sum_{-C \leq e \leq C} \text{prob}_{\ell_i,e,r_i} / (2C+1) \right)$.

**6.4. Examples of target failure probabilities.** The acceptable level of failure probability for our algorithm depends on the attack using the algorithm. For concreteness we consider three possibilities for CSIDH-512 failure probabilities, namely having the algorithm fail for a uniform random vector with probabilities at most $2^{-1}$, $2^{-32}$, and $2^{-256}$.

Our rationale for considering these probabilities is as follows. Probabilities around $2^{-1}$ are easy to test, and may be of interest beyond this paper for constructive scenarios where failing computations can simply be retried. If each computation needs to work correctly, and there are many computations, then failure probabilities need to be much smaller, say $2^{-32}$. Asking for every input in superposition to work correctly in one computation (for example, [38] asks for this) requires a much smaller failure probability, say $2^{-256}$. Performance results for these three cases also provide an adequate basis for estimating performance in other cases.

Table 6.1 presents three reasonable choices of $(r_1,\dots,r_n)$, one for each of the failure probabilities listed above, for the case of CSIDH-512 with uniform random

**Table 6.2.** Examples of choices of $r_i$ for Algorithm 6.1 for three levels of failure probability for uniform random CSIDH-512 vectors with entries in $\{-10,\ldots,10\}$. Failure probabilities $\epsilon$ are rounded to three digits after the decimal point. The "total" column is $\sum r_i$, the total number of iterations. The "[38]" column is $74 \cdot 10 \cdot r$, the number of iterations in the algorithm of [38], with $r$ chosen as in [38] to have $1-(1-(2/3)^r)^{74\cdot 10}$ at most $2^{-1}$ or $2^{-32}$ or $2^{-256}$. Compare Table 6.1 for $\{-5,\ldots,5\}$.

| $\epsilon$ \ $\ell_i$ | 3 | 5 | 7 | 11 | 13 | 17 | ... | 359 | 367 | 373 | 587 | total | [38] |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $0.521 \cdot 2^{-1}$ | 20 | 15 | 14 | 13 | 12 | 12 | ... | 10 | 10 | 10 | 10 | 786 | 13320 |
| $0.257 \cdot 2^{-32}$ | 48 | 34 | 30 | 25 | 24 | 22 | ... | 15 | 15 | 15 | 14 | 1296 | 52540 |
| $0.215 \cdot 2^{-256}$ | 201 | 139 | 116 | 96 | 90 | 82 | ... | 43 | 43 | 43 | 41 | 4185 | 335960 |

vectors with entries in $\{-5,\ldots,5\}$. For each target failure probability $\delta$ and each $i$, the table chooses the minimum $r_i$ such that $\sum_{-C\le e\le C} \mathrm{prob}_{\ell_i,e,r_i}/(2C+1)$ is at least $(1-\delta)^{1/n}$. The overall success probability is then at least $1-\delta$ as desired. The discontinuity of choices of $(r_1,\ldots,r_n)$ means that the actual failure probability $\epsilon$ is somewhat below $\delta$, as shown by the coefficients 0.499, 0.178, 0.249 in Table 6.1. We could move closer to the target failure probability by choosing successively $r_n, r_{n-1}, \ldots$, adjusting the probability $(1-\delta)^{1/n}$ at each step in light of the overshoot from previous steps. The values $r_i$ for $\epsilon \approx 0.499 \cdot 2^{-1}$ have been experimentally verified using a modified version of the CSIDH software. To illustrate the impact of larger vector entries, we also present similar data in Table 6.2 for uniform random vectors with entries in $\{-10,\ldots,10\}$.

The "total" column in Table 6.1 shows that this algorithm uses, e.g., 869 iterations for failure probability $0.178 \cdot 2^{-32}$ with vector entries in $\{-5,\ldots,5\}$. Each iteration consists mostly of a scalar multiplication, plus some extra cost for Elligator, Vélu's formulas, etc. Overall there are roughly 5 million field multiplications, accounting for roughly $2^{41}$ nonlinear bit operations, implying a quantum computation using roughly $2^{45}$ $T$-gates.

As noted in Section 1, using the algorithm of [38] on top of the modular-multiplication algorithm from [63] would use approximately $2^{51}$ nonlinear bit operations for the same distribution of input vectors. We save a factor 30 in the number of iterations compared to [38], and we save a similar factor in the number of bit operations for each modular multiplication compared to [63].

We do not analyze this algorithm in more detail: the algorithms we present below are faster.

## 7 Reducing the top nonzero exponent

Most of the iterations in Algorithm 6.1 are spent on exponents that are already 0. For example, consider the 869 iterations mentioned above for failure probability $0.178 \cdot 2^{-32}$ for uniform random CSIDH-512 vectors with entries in $\{-5,\ldots,5\}$.

**Algorithm 7.1:** Evaluating the class-group action by reducing the top nonzero exponent.

**Parameters:** Odd primes $\ell_1 < \cdots < \ell_n$ with $n \geq 1$, a prime $p = 4\ell_1 \cdots \ell_n - 1$, and a positive integer $r$.

**Input:** $A \in S_p$, integers $(e_1, \ldots, e_n)$.

**Output:** $\mathfrak{l}_1^{e_1} \cdots \mathfrak{l}_n^{e_n}(A)$ or "fail".

**for** $j \leftarrow 1$ **to** $r$ **do**

- Let $i = \max\{k : e_k \neq 0\}$, or $i = 1$ if each $e_k = 0$.
- Let $s = \operatorname{sign}(e_i) \in \{-1, 0, +1\}$.
- Find a random point $P$ on $E_{sA}$ using Elligator.
- Compute $Q \leftarrow ((p+1)/\ell_i)P$.
- Compute $B$ with $E_B \cong E_{sA}/\langle Q \rangle$ if $Q \neq \infty$, using the $\ell_i$-isogeny formulas from Section 5.3 with maximum degree $\ell_n$.
- Set $A \leftarrow sB$ if $Q \neq \infty$ and $s \neq 0$.
- Set $e_i \leftarrow e_i - s$ if $Q \neq \infty$.

Set $A \leftarrow$ "fail" if $(e_1, \ldots, e_n) \neq (0, \ldots, 0)$.

**Return** $A$.

Entry $e_i$ has absolute value $30/11$ on average, and needs $(30/11)\ell_i/(\ell_i - 1)$ iterations on average, for a total of $\sum_i (30/11)\ell_i/(\ell_i - 1) \approx 206.79$ useful iterations on average. This means that there are 662.21 useless iterations on average, many more than one would expect to be needed to guarantee this failure probability.

This section introduces a constant-time algorithm that achieves the same failure probability with far fewer iterations. For example, in the above scenario, just 294 iterations suffice to reduce the failure probability below $2^{-32}$. Each iteration becomes (for CSIDH-512) about 25% more expensive, but overall the algorithm uses far fewer bit operations.

**7.1. Iterations targeting variable $\ell$.** It is obvious how to avoid useless iterations for variable-time algorithms: when an exponent reaches 0, move on to the next exponent. In other words, always focus on reducing a nonzero exponent, if one exists.

What is new is doing this in constant time. This is where we exploit the Matryoshka-doll structure from Section 5.3, computing an isogeny for variable $\ell$ in constant time. We now pay for an $\ell_n$-isogeny in each iteration rather than an $\ell$-isogeny, but the iteration cost is still dominated by scalar multiplication. Concretely, for CSIDH-512, an average $\ell$-isogeny costs about 600 multiplications, and an $\ell_n$-isogeny costs about 2000 multiplications, but a scalar multiplication costs about 5000 multiplications.

We choose to reduce the top exponent that is not 0. "Top" here refers to position, not value: we reduce the nonzero $e_i$ where $i$ is maximized. See Algorithm 7.1.

**7.2. Upper bounds on the failure probability.** One can crudely estimate the failure probability of Algorithm 7.1 in terms of the 1-norm $E = |e_1| + \cdots + |e_n|$ as follows. Model each iteration as having failure probability $1/3$ instead of $1/\ell_i$; this produces a loose upper bound for the overall failure probability of the algorithm.

In this model, the chance of needing exactly $r$ iterations to find a point of order $\ell_i$ is the coefficient of $x^r$ in the power series

$$(2/3)x + (2/9)x^2 + (2/27)x^3 + \cdots = 2x/(3-x).$$

The chance of needing exactly $r$ iterations to find all $E$ points is the coefficient of $x^r$ in the $E$th power of that power series, namely $c_r = \binom{r-1}{E-1}2^E/3^r$ for $r \geq E$. See generally [74] for an introduction to the power-series view of combinatorics; there are many other ways to derive the formula $\binom{r-1}{E-1}2^E/3^r$, but we make critical use of power series for fast computations in Sections 7.3 and 8.3.

The failure probability of $r$ iterations of Algorithm 7.1 is at most the failure probability of $r$ iterations in this model, namely $f(r,E) = 1 - c_E - c_{E+1} - \cdots - c_r$. The failure probability of $r$ iterations for a uniform random vector with entries in $\{-C, \ldots, C\}$ is at most $\sum_{0 \leq E \leq nC} f(r,E)g[E]$. Here $g[E]$ is the probability that a vector has 1-norm $E$, which we compute as the coefficient of $x^E$ in the $n$th power of the polynomial $(1 + 2x + 2x^2 + \cdots + 2x^C)/(2C+1)$. For example, with $n = 74$ and $C = 5$, the failure probability in this model (rounded to 3 digits after the decimal point) is $0.999 \cdot 2^{-1}$ for $r = 302$; $0.965 \cdot 2^{-2}$ for $r = 319$; $0.844 \cdot 2^{-32}$ for $r = 461$; and $0.570 \cdot 2^{-256}$ for $r = 823$. As a double-check, we observe that a simple simulation of the model for $r = 319$ produces 241071 failures in 1000000 experiments, close to the predicted $0.965 \cdot 2^{-2} \cdot 1000000 \approx 241250$.

**7.3. Exact values of the failure probability.** The upper bounds from the model above are too pessimistic, except for $\ell_i = 3$. We instead compute the exact failure probabilities as follows.

The chance that $\mathcal{L}_1^{e_1} \cdots \mathcal{L}_n^{e_n}$ requires exactly $r$ iterations is the coefficient of $x^r$ in the power series

$$\left(\frac{(\ell_1 - 1)x}{\ell_1 - x}\right)^{|e_1|} \cdots \left(\frac{(\ell_n - 1)x}{\ell_n - x}\right)^{|e_n|}.$$

What we want is the average of this coefficient over all vectors $(e_1, \ldots, e_n) \in \{-C, \ldots, C\}^n$. This is the same as the coefficient of the average, and the average factors nicely as

$$\left(\sum_{-C \leq e_1 \leq C} \frac{1}{2C+1}\left(\frac{(\ell_1 - 1)x}{\ell_1 - x}\right)^{|e_1|}\right) \cdots \left(\sum_{-C \leq e_n \leq C} \frac{1}{2C+1}\left(\frac{(\ell_n - 1)x}{\ell_n - x}\right)^{|e_n|}\right).$$

We compute this product as a power series with rational coefficients: for example, we compute the coefficients of $x^0, \ldots, x^{499}$ if we are not interested in 500 or more iterations. We then add together the coefficients of $x^0, \ldots, x^r$ to find the exact success probability of $r$ iterations of Algorithm 7.1.

As an example we again take CSIDH-512 with $C = 5$. The failure probability (again rounded to 3 digits after the decimal point) is $0.960 \cdot 2^{-1}$ for $r = 207$; $0.998 \cdot 2^{-2}$ for $r = 216$; $0.984 \cdot 2^{-32}$ for $r = 294$; $0.521 \cdot 2^{-51}$ for $r = 319$; and $0.773 \cdot 2^{-256}$ for $r = 468$. We double-checked these averages against the results of Monte Carlo calculations for these values of $r$. Each Monte Carlo iteration sampled a uniform random 1-norm (weighted appropriately for the initial probability of each 1-norm), sampled a uniform random vector within that 1-norm, and computed the failure probability for that vector using the single-vector generating function.

**7.4. Analysis of the cost.** We have fully implemented Algorithm 7.1 in our bit-operation simulator. One iteration for CSIDH-512 uses $9208697761 \approx 2^{33}$ bit operations, including $3805535430 \approx 2^{32}$ nonlinear bit operations. More than 95% of the cost is explained as follows:

- Each iteration uses a Montgomery ladder with a 511-bit scalar. (We could save a bit here: the largest useful scalar is $(p+1)/3$, which is below $2^{510}$.) We use an affine input point and an affine $A$, so this costs $2044\mathbf{S} + 3066\mathbf{M}$.
- Each iteration uses the formulas from Section 5.3 with $\ell = 587$. This takes $602\mathbf{S} + 1472\mathbf{M}$: specifically, $584\mathbf{S} + 876\mathbf{M}$ for multiples of the point of order $\ell$ (again affine); $584\mathbf{M}$ for the product of Edwards $y$-coordinates; $18\mathbf{S} + 10\mathbf{M}$ for two $\ell$th powers; and $2\mathbf{M}$ to multiply by two 8th powers. (We merge the $6\mathbf{S}$ for the 8th powers into the squarings used for the $\ell$th powers.)
- Each iteration uses two inversions to obtain affine $Q$ and $A$, each $507\mathbf{S}+97\mathbf{M}$, and one Legendre-symbol computation, $506\mathbf{S} + 96\mathbf{M}$.

This accounts for $4166\mathbf{S}+4828\mathbf{M}$ per iteration, i.e., $4166 \cdot 349596 + 4828 \cdot 447902 = 3618887792 \approx 2^{32}$ nonlinear bit operations.

The cost of 294 iterations is simply $294 \cdot 3805535430 = 1118827416420 \approx 2^{40}$ nonlinear bit operations. This justifies the first $(B, \epsilon)$ claim in Section 1.

**7.5. Decreasing the maximum degrees.** Always performing isogeny computations capable of handling degrees up to $\ell_n$ is wasteful: With overwhelming probability, almost all of the 294 iterations required for a failure probability of less than $2^{-32}$ with the approach discussed so far actually compute isogenies of degree (much) less than $\ell_n$. For example, with $e$ uniformly random in $\{-5, \ldots, 5\}$, the probability that 10 iterations are not sufficient to eliminate all 587-isogenies is approximately $2^{-50}$. Therefore, using smaller upper bounds on the isogeny degrees for later iterations of the algorithm will not do much harm to the success probability while significantly improving the performance. We modify Algorithm 7.1 as follows:

- Instead of a single parameter $r$, we use a list $(r_1, \ldots, r_n)$ of non-negative integers, each $r_i$ denoting the number of times an isogeny computation capable of handling degrees up to $\ell_i$ is performed.
- The loop iterating from 1 through $r$ is replaced by an outer loop on $u$ from $n$ down to 1, and inside that an inner loop on $j$ from 1 up to $r_u$. The loop body is unchanged, except that the maximum degree for the isogeny formulas is now $\ell_u$ instead of $\ell_n$.

**Table 7.1.** Examples of choices of $r_i, \ldots, r_i$ for Algorithm 7.1 with reducing the maximal degree in Vélu's formulas for uniform random CSIDH-512 vectors with entries in $\{-5, \ldots, 5\}$. Failure probabilities $\epsilon$ are rounded to three digits after the decimal point.

| $\epsilon$ | $r_n \ldots r_1$ | $\sum r_i$ | avg. $\ell$ |
|---|---|---|---|
| $0.594 \cdot 2^{-1}$ | 5 3 4 5 3 5 5 4 3 5 4 3 4 4 3 4 3 4 3 3 3 4 3 3 3 4 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 2 4 2 3 3 3 3 2 3 3 3 2 3 3 2 3 2 3 2 3 2 2 3 2 2 2 1 1 1 0 0 | 218 | 205.0 |
| $0.970 \cdot 2^{-32}$ | 9 5 5 5 5 5 4 5 5 5 4 5 4 5 5 4 5 4 4 5 5 4 4 4 5 4 4 4 4 3 5 3 4 4 4 3 4 4 4 3 4 4 4 3 4 3 4 3 4 3 4 4 3 4 3 3 3 4 4 3 3 4 3 3 4 3 4 3 3 4 3 3 3 4 3 3 4 | 295 | 196.0 |
| $0.705 \cdot 2^{-256}$ | 34 8 6 6 5 6 6 5 5 6 5 6 5 5 5 5 6 5 5 5 5 6 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 4 6 5 5 5 5 5 4 6 5 5 5 5 5 5 5 6 5 5 6 6 6 6 7 7 11 16 38 | 469 | 182.7 |

For a given sequence $(r_1, \ldots, r_n)$, the probability of success can be computed as follows:

- For each $i \in \{1, \ldots, n\}$, compute the generating function
$$\phi_i(x) = \sum_{-C \le e_i \le C} \frac{1}{2C+1} \left( \frac{(\ell_i - 1)x}{\ell_i - x} \right)^{|e_i|}$$
of the number of $\ell_i$-isogeny steps that have to be performed.
- Since we are no longer only interested in the total number of isogeny steps to be computed, but also in their degrees, we cannot simply take the product of all $\phi_i$ as before. Instead, to account for the fact that failing to compute a $\ell_i$-isogeny before the maximal degree drops below $\ell_i$ implies a total failure, we iteratively compute the product of the $\phi_i$ from $k = n$ down to 1, but truncate the product after each step. Truncation after some power $x^t$ means eliminating all branches of the probabilistic process in which more than $t$ isogeny steps are needed for the computations so far. In our case we use $t = \sum_{j=i}^{n} r_j$ after multiplying by $\phi_i$, which removes all outcomes in which more isogeny steps of degree $\ge \ell_i$ would have needed to be computed.
- After all $\phi_i$ have been processed (including the final truncation), the probability of success is the sum of all coefficients of the remaining power series.

Note that we have only described a procedure to compute the success probabilty once $r_1, \ldots, r_n$ are known. It is unclear how to find the optimal values $r_i$ which minimize the cost of the resulting algorithm, while at the same time respecting a certain failure probability. We tried various reasonable-looking choices of strategies to choose the $r_i$ according to certain prescribed failure probabilities after each individual step. Experimentally, a good rule seems to be that the failure probability after processing $\phi_i$ should be bounded by $\epsilon \cdot 2^{2/i-2}$, where $\epsilon$ is the overall target failure probability. The results are shown in Table 7.1.

The average degree of the isogenies used constructively in CSIDH-512 is about 174.6, which is not much smaller than the average degree we achieve.

Since we still need to control the error probability, it does not appear that one can expect to get much closer to the constructive case.

Also note that the total number of isogeny steps for $\epsilon \approx 2^{-32}$ and $\epsilon \approx 2^{-256}$ is each only one more than the previous number $r$ of isogeny computations, hence one can expect significant savings using this strategy. Assuming that about 1/4 of the total time is spent on Vélu's formulas (which is close to the real proportion), we get a speedup of about 16% for $\epsilon \approx 2^{-32}$ and about 17% for $\epsilon \approx 2^{-256}$.

## 8 Pushing points through isogenies

Algorithms 6.1 and 7.1 spend most of their time on scalar multiplication. This section pushes points through isogenies to reduce the time spent on scalar multiplication, saving time overall.

The general idea of balancing isogeny computation with scalar multiplication was introduced in [22] in the SIDH context, and was reused in the variable-time CSIDH algorithms in [15]. This section adapts the idea to the context of constant-time CSIDH computation.

**8.1. Why pushing points through isogenies saves time.** To illustrate the main idea, we begin by considering a sequence of just two isogenies with the same sign. Specifically, assume that, given distinct $\ell_1$ and $\ell_2$ dividing $p+1$, we want to compute $\mathcal{L}_1\mathcal{L}_2(A) = B$. Here are two different methods:

- **Method 1.** The method of Algorithm 6.1 uses Elligator to find $P_1 \in E_A(\mathbb{F}_p)$, computes $Q_1 \leftarrow [(p+1)/\ell_1]P_1$, computes $E_{A'} = E_A/\langle Q_1\rangle$, uses Elligator to find $P_2 \in E_{A'}(\mathbb{F}_p)$, computes $Q_2 \leftarrow [(p+1)/\ell_2]P_2$, and computes $E_B = E_{A'}/\langle Q_2\rangle$. Failure cases: if $Q_1 = \infty$ then this method computes $A' = A$, failing to compute $\mathcal{L}_1$; similarly, if $Q_2 = \infty$ then this method computes $B = A'$, failing to compute $\mathcal{L}_2$.
- **Method 2.** The method described in this section instead uses Elligator to find $P \in E_A(\mathbb{F}_p)$, computes $R \leftarrow [(p+1)/\ell_1\ell_2]P$, computes $Q \leftarrow [\ell_2]R$, computes $\varphi : E_A \to E_{A'} = E_A/\langle Q\rangle$ and $Q' = \varphi(R)$, and computes $E_B = E_{A'}/\langle Q'\rangle$. Failure cases: if $Q = \infty$ then this method computes $Q' = R$ (which has order dividing $\ell_2$) and $A' = A$, failing to compute $\mathcal{L}_1$; if $Q' = \infty$ then this method computes $B = A'$, failing to compute $\mathcal{L}_2$.

For concreteness, we compare the costs of these methods for CSIDH-512. The rest of this subsection uses approximations to the costs of lower-level operations to simplify the analysis. The main costs are as follows:

- For $p$ a 512-bit prime, Elligator costs approximately 600**M**.
- Given $P \in E(\mathbb{F}_p)$ and a positive integer $k$, the computation of $[k]P$ via the Montgomery ladder, as described in Section 3.3, costs approximately $10(\log_2 k)$**M**, i.e., approximately $(5120 - 10\log_2 \ell)$**M** if $k = (p+1)/\ell$.
- The computation of a degree-$\ell$ isogeny via the method described in Section 5.3 costs approximately $(3.5\ell + 2\log_2 \ell)$**M**.

- Given an $\ell$-isogeny $\varphi_\ell : E \to E'$ and $P \in E(\mathbb{F}_p)$, the computation of $\varphi_\ell(P)$ via the method described in Section 5.4 costs approximately $2\ell\mathbf{M}$.

Method 1 costs approximately

$$(1200 + 10240 + 3.5\ell_1 + 3.5\ell_2 - 8\log_2 \ell_1 - 8\log_2 \ell_2)\mathbf{M},$$

while Method 2 costs approximately

$$(600 + 5120 + 5.5\ell_1 + 3.5\ell_2 - 8\log_2 \ell_1 + 2\log_2 \ell_2)\mathbf{M}.$$

The savings of $(600 + 5120)\mathbf{M}$ clearly outweighs the loss of $(2\ell_1 + 10\log_2 \ell_2)\mathbf{M}$, since the largest value of $\ell_i$ is 587.

There are limits to the applicability of Method 2: it cannot combine two isogenies of opposite signs, it cannot combine two isogenies using the same prime, and it cannot save time in applying just one isogeny. We will analyze the overall magnitude of these effects in Section 8.3.

**8.2. Handling the general case, two isogenies at a time.** Algorithm 8.1 computes $\mathcal{L}_1^{e_1} \cdots \mathcal{L}_n^{e_n}(A)$ for any exponent vector $(e_1, \ldots, e_n)$. Each iteration of the algorithm tries to perform two isogenies: one for the top nonzero exponent (if the vector is nonzero), and one for the next exponent having the same sign (if the vector has another exponent of this sign). As in Section 7, "top" refers to position, not value.

The algorithm pushes the first point through the first isogeny, as in Section 8.1, to save the cost of generating a second point. Scalar multiplication, isogeny computation, and isogeny application use the constant-time subroutines described in Sections 3.3, 5.3, and 5.4 respectively. The cost of these algorithms depends on the bound $\ell_n$ for the prime for the top nonzero exponent and the bound $\ell_{n-1}$ for the prime for the next exponent. The two prime bounds have asymmetric effects upon costs; we exploit this by applying the isogeny for the top nonzero exponent *after* the isogeny for the next exponent.

Analyzing the correctness of Algorithm 8.1—assuming that there are enough iterations; see Section 8.3—requires considering three cases. The first case is that the exponent vector is 0. Then $i, i', s$ are initialized to $0, 0, 1$ respectively, and $i, i'$ stay 0 throughout the iteration, so $A$ does not change and the exponent vector does not change.

The second case is that the exponent vector is nonzero and the top nonzero exponent $e_i$ is the only exponent having sign $s$. Then $i'$ is 0 throughout the iteration, so the "first isogeny" portion of Algorithm 8.1 has no effect. The point $Q = R$ in the "second isogeny" portion is $cP$ where $c = (p+1)/\ell_i$, so $\ell_i Q = \infty$. If $Q = \infty$ then $i$ is set to 0 and the entire iteration has no effect, except for setting $A$ to $sA$ and then back to $s(sA) = A$. If $Q \neq \infty$ then $i$ stays nonzero and $A$ is replaced by $\mathcal{L}_i(A)$, so $A$ at the end of the iteration is $\mathcal{L}_i^s$ applied to $A$ at the beginning of the iteration, while $s$ is subtracted from $e_i$.

The third case is that the exponent vector is nonzero and that $e_{i'}$ is the next exponent having the same sign $s$ as the top nonzero exponent $e_i$. By construction $i' < i \leq n$ so $\ell_{i'} \leq \ell_{n-1}$. Now $R = cP$ where $c = (p+1)/(\ell_i \ell_{i'})$. The first isogeny

**Algorithm 8.1:** Evaluating the class-group action by reducing the top nonzero exponent and the next exponent with the same sign.

**Parameters:** Odd primes $\ell_1 < \cdots < \ell_n$ with $n \geq 1$, a prime $p = 4\ell_1 \cdots \ell_n - 1$, and a positive integer $r$.

**Input:** $A \in S_p$, integers $(e_1, \ldots, e_n)$.

**Output:** $\mathcal{L}_1^{e_1} \cdots \mathcal{L}_n^{e_n}(A)$ or "fail".

**for** $j \leftarrow 1$ **to** $r$ **do**

- Set $I \leftarrow \{k : 1 \leq k \leq n \text{ and } e_k \neq 0\}$.
- Set $i \leftarrow \max I$ and $s \leftarrow \text{sign}(e_i) \in \{-1, 1\}$, or $i \leftarrow 0$ and $s \leftarrow 1$ if $I = \{\}$.
- Set $I' \leftarrow \{k : 1 \leq k < i \text{ and } \text{sign}(e_k) = s\}$.
- Set $i' \leftarrow \max I'$, or $i' \leftarrow 0$ if $I' = \{\}$.
- **Twist.** Set $A \leftarrow sA$.
- **Isogeny preparation.** Find a random point $P$ on $E_A$ using Elligator.
- Compute $R \leftarrow cP$ where $c = 4 \prod_{1 \leq j \leq n, j \neq i, j \neq i'} \ell_j$.
- **First isogeny.** Compute $Q \leftarrow \ell_i R$, where $\ell_0$ means 1.
- [Now $\ell_{i'} Q = \infty$ if $i' \neq 0$.] Set $i' \leftarrow 0$ if $Q = \infty$.
- Compute $B$ with $E_B \cong E_A/\langle Q \rangle$ if $i' \neq 0$, using the $\ell_{i'}$-isogeny formulas from Section 5.3 with maximum degree $\ell_{n-1}$.
- Set $R$ to the image of $R$ in $E_B$ if $i' \neq 0$, using the $\ell_{i'}$-isogeny formulas from Section 5.4 with maximum degree $\ell_{n-1}$.
- Set $A \leftarrow B$ and $e_{i'} \leftarrow e_{i'} - s$ if $i' \neq 0$.
- **Second isogeny.** Set $Q \leftarrow R$.
- [Now $\ell_i Q = \infty$ if $i \neq 0$.] Set $i \leftarrow 0$ if $Q = \infty$.
- Compute $B$ with $E_B \cong E_A/\langle Q \rangle$ if $i \neq 0$, using the $\ell_i$-isogeny formulas from Section 5.3 with maximum degree $\ell_n$.
- Set $A \leftarrow B$ and $e_i \leftarrow e_i - s$ if $i \neq 0$.
- **Untwist.** Set $A \leftarrow sA$.

Set $A \leftarrow$ "fail" if $(e_1, \ldots, e_n) \neq (0, \ldots, 0)$.

**Return** $A$.

uses the point $Q = \ell_i R$, which is either $\infty$ or a point of order $\ell_{i'}$. If $Q$ is $\infty$ then $i'$ is set to 0; both $A$ and the vector are unchanged; the point $R$ must have order dividing $\ell_i$; and the second isogeny proceeds as above using this point. If $Q$ has order $\ell_{i'}$ then the first isogeny replaces $A$ with $\mathcal{L}_{i'}(A)$, while subtracting $s$ from $e_{i'}$ and replacing $R$ with a point of order dividing $\ell_i$ on the new curve (note that the $\ell_{i'}$-isogeny removes any $\ell_{i'}$ from orders of points); again the second isogeny proceeds as above.

**8.3. Analysis of the failure probability.** Consider a modified dual-isogeny algorithm in which the isogeny with a smaller prime is saved to handle later:

- Initialize an iteration counter to 0.

- Initialize an empty bank of positive isogenies.
- Initialize an empty bank of negative isogenies.
- For each $\ell$ in decreasing order:
  - While an $\ell$-isogeny needs to be done and the bank has an isogeny of the correct sign: Withdraw an isogeny from the bank, apply the isogeny, and adjust the exponent.
  - While an $\ell$-isogeny still needs to be done: Apply an isogeny, adjust the exponent, deposit an isogeny with the bank, and increase the iteration counter.

This uses more bit operations than Algorithm 8.1 (since the work here is not shared across two isogenies), but it has the same failure probability for the same number of iterations. We now focus on analyzing the distribution of the number of iterations used by this modified algorithm.

We use three variables to characterize the state of the modified algorithm before each $\ell$:

- $i \geq 0$ is the iteration counter;
- $j \geq 0$ is the number of positive isogenies in the bank;
- $k \geq 0$ is the number of negative isogenies in the bank.

The number of isogenies actually applied so far is $2i-(j+k) \geq i$. The distribution of states is captured by the three-variable formal power series $\sum_{i,j,k} s_{i,j,k} x^i y^j z^k$ where $s_{i,j,k}$ is the probability of state $(i,j,k)$. Note that there is no need to track which primes are paired with which; this is what makes the modified algorithm relatively easy to analyze.

If there are exactly $h$ positive $\ell$-isogenies to perform then the new state after those isogenies is $(i, j-h, k)$ if $h \leq j$, or $(i+h-j, h-j, k)$ if $h > j$. This can be viewed as a composition of two operations on the power series. First, multiply by $y^{-h}$. Second, replace any positive power of $y^{-1}$ with the same power of $xy$; i.e., replace $x^i y^j z^k$ for each $j < 0$ with $x^{i-j} y^{-j} z^k$.

We actually have a distribution of the number of $\ell$-isogenies to perform. Say there are $h$ isogenies with probability $q_h$. We multiply the original series by $\sum_{h \geq 0} q_h y^{-h}$, and then eliminate negative powers of $y$ as above. We similarly handle $h < 0$, exchanging the role of $(j, y)$ with the role of $(k, z)$.

As in the analyses earlier in the paper, we model each point $Q$ for an $\ell$-isogeny as having order 1 with probability $1/\ell$ and order $\ell$ with probability $1 - 1/\ell$, and we assume that the number of $\ell$-isogenies to perform is a uniform random integer $e \in \{-C, \ldots, C\}$. Then $q_h$ for $h \geq 0$ is the coefficient of $x^h$ in $\sum_{0 \leq e \leq C} (((\ell-1)x)/(\ell-x))^e/(2C+1)$; also, $q_{-h} = q_h$.

We reduce the time spent on these computations in three ways. First, we discard all states with $i > r$ if we are not interested in more than $r$ iterations. This leaves a cubic number of states for each $\ell$: every $i$ between 0 and $r$ inclusive, every $j$ between 0 and $i$ inclusive, and every $k$ between 0 and $i - j$ inclusive.

Second, we use fixed-precision arithmetic, rounding each probability to an integer multiple of (e.g.) $2^{-512}$. We round down to obtain lower bounds on success probabilities; we round up to obtain upper bounds on success probabilities; we

choose the scale $2^{-512}$ so that these bounds are as tight as desired. We could save more time by reducing the precision slightly at each step of the computation, and by using standard interval-arithmetic techniques to merge computations of lower and upper bounds.

Third, to multiply the series $\sum_{i,j,k} s_{i,j,k} x^i y^j z^k$ by $\sum_{h \geq 0} q_h y^{-h}$, we actually multiply $\sum_j s_{i,j,k} y^j$ by $\sum_{h \geq 0} q_h y^{-h}$ for each $(i,k)$ separately. We use Sage for these multiplications of univariate polynomials with integer coefficients. Sage, in turn, uses fast multiplication algorithms whose cost is essentially $bd$ for $d$ $b$-bit coefficients, so our total cost for $n$ primes is essentially $bnr^3$.

Concretely, we use under two hours on one core of a 3.5GHz Intel Xeon E3-1275 v3 to compute lower bounds on all the success probabilities for CSIDH-512 with $b = 512$ and $r = 349$, and under three hours[4] to compute upper bounds. Our convention of rounding failure probabilities to 3 digits makes the lower bounds and upper bounds identical, so presumably we could have used less precision.

We find, e.g., failure probability $0.943 \cdot 2^{-1}$ after 106 iterations, failure probability $0.855 \cdot 2^{-32}$ after 154 iterations, and failure probability $0.975 \cdot 2^{-257}$ after 307 iterations. Compared to the $207,294,468$ single-isogeny iterations required in Section 7.3, the number of iterations has decreased to 51.2%, 52.3%, 65.6% respectively.

**8.4. Analysis of the cost.** We have fully implemented Algorithm 8.1 in our bit-operation simulator. An iteration of Algorithm 8.1 uses $4969644344 \approx 2^{32}$ nonlinear bit operations, about 1.306 times more expensive than an iteration of Algorithm 7.1.

If the number of iterations were multiplied by exactly 0.5 then the total cost would be multiplied by 0.653. Given the actual number of iterations (see Section 8.3), the cost is actually multiplied by 0.669, 0.684, 0.857 respectively. In particular, we reach failure probability $0.855 \cdot 2^{-32}$ with $154 \cdot 4969644344 = 765325228976 \approx 0.7 \cdot 2^{40}$ nonlinear bit operations. This justifies the second $(B, \epsilon)$ claim in Section 1.

**8.5. Variants.** The idea of pushing points through isogenies can be combined with the idea of gradually reducing the maximum prime allowed in the Matryoshka-doll isogeny formulas. This is compatible with our techniques for analyzing failure probabilities.

A dual-isogeny iteration very late in the computation is likely to have a useless second isogeny. It should be slightly better to replace some of the last dual-isogeny iterations with single-isogeny iterations. This is also compatible with our techniques for analyzing failure probabilities.

There are many different possible pairings of primes: one can take any two distinct positions where the exponents have the same sign. Possibilities include reducing exponents from the bottom rather than the top; reducing the top nonzero exponent and the bottom exponent with the same sign; always pairing "high"

[4] It is unsurprising that lower bounds are faster: many coefficients $q_h$ round down to 0. We could save time in the upper bounds by checking for stretches of coefficients that round up to, e.g., $1/2^{512}$, and using additions to multiply by those stretches.

positions with "low" positions; always reducing the largest exponents in absolute value; always reducing $e_i$ where $|e_i|\ell_i/(\ell_i - 1)$ is largest. For some of these ideas it is not clear how to efficiently analyze failure probabilities.

This section has focused on reusing an Elligator computation and large scalar multiplication for (in most cases) two isogeny computations, dividing the scalar-multiplication cost by (nearly) 2, in exchange for some overhead. We could push a point through more isogenies, although each extra isogeny has further overhead with less and less benefit, and computing the failure probability becomes more expensive. For comparison, [15] reuses one point for every $\ell_i$ where $e_i$ has the same sign; the number of such $\ell_i$ is variable, and decreases as the computation continues. For small primes it might also save time to push multiple points through one isogeny, as in [22].

## 9 Computing $\ell$-isogenies using division polynomials

As the target failure probability decreases, the algorithms earlier in this paper spend more and more iterations handling the possibility of repeated failures for small primes $\ell$. This section presents and analyzes an alternative: a deterministic constant-time subroutine that uses division polynomials to *always* compute $\ell$-isogenies. See full version of paper online at https://ia.cr/2018/1059.

## 10 Computing $\ell$-isogenies using modular polynomials

Modular polynomials, like division polynomials, give a deterministic subroutine to compute $\ell$-isogenies. The advantage of modular polynomials over division polynomials is that modular polynomials are smaller for all $\ell \geq 5$. However, using modular polynomials requires solving two additional problems. See full version of paper online at https://ia.cr/2018/1059.

## References

[1] R. Azarderakhsh, D. Jao, K. Kalach, B. Koziel, and C. Leonardi. Key compression for isogeny-based cryptosystems. In *AsiaPKC@AsiaCCS*, pages 1–10. ACM, 2016.

[5] D. J. Bernstein. Curve25519: New Diffie-Hellman speed records. In *Public Key Cryptography*, volume 3958 of *LNCS*, pages 207–228. Springer, 2006.

[7] D. J. Bernstein, M. Hamburg, A. Krasnova, and T. Lange. Elligator: elliptic-curve points indistinguishable from uniform random strings. In *ACM Conference on Computer and Communications Security*, pages 967–980. ACM, 2013.

[8] D. J. Bernstein and T. Lange. Analysis and optimization of elliptic-curve single-scalar multiplication. In *Finite Fields and Applications 2007*, pages 1–19. AMS, 2008.

[9] D. J. Bernstein and T. Lange. Montgomery curves and the Montgomery ladder. In J. W. Bos and A. K. Lenstra, editors, *Topics in computational number theory inspired by Peter L. Montgomery*, pages 82–115. Cambridge, 2017.

[10] X. Bonnetain and M. Naya-Plasencia. Hidden shift quantum cryptanalysis and implications. In *ASIACRYPT*, volume 11274 of *LNCS*, pages 560–592. Springer, 2018.

[11] X. Bonnetain and A. Schrottenloher. Quantum security analysis of CSIDH and ordinary isogeny-based schemes, 2018. IACR Cryptology ePrint Archive 2018/537.
[15] W. Castryck, T. Lange, C. Martindale, L. Panny, and J. Renes. CSIDH: An efficient post-quantum commutative group action. In *ASIACRYPT*, volume 11274 of *LNCS*, pages 395–427. Springer, 2018.
[17] A. M. Childs, D. Jao, and V. Soukharev. Constructing elliptic curve isogenies in quantum subexponential time. *J. Mathematical Cryptology*, 8(1):1–29, 2014.
[18] C. Costello and H. Hişil. A simple and compact algorithm for SIDH with arbitrary degree isogenies. In *ASIACRYPT (2)*, volume 10625 of *LNCS*, pages 303–329. Springer, 2017.
[19] C. Costello, D. Jao, P. Longa, M. Naehrig, J. Renes, and D. Urbanik. Efficient compression of SIDH public keys. In *EUROCRYPT (1)*, volume 10210 of *LNCS*, pages 679–706, 2017.
[20] J.-M. Couveignes. Hard Homogeneous Spaces, 1997. IACR Cryptology ePrint Archive 2006/291.
[22] L. De Feo, D. Jao, and J. Plût. Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. *J. Mathematical Cryptology*, 8(3):209–247, 2014. IACR Cryptology ePrint Archive 2011/506.
[23] L. De Feo, J. Kieffer, and B. Smith. Towards practical key exchange from ordinary isogeny graphs. In *ASIACRYPT*, volume 11274 of *LNCS*, pages 365–394. Springer, 2018.
[28] M. Grassl, B. Langenberg, M. Roetteler, and R. Steinwandt. Applying Grover's algorithm to AES: quantum resource estimates. In *PQCrypto*, volume 9606 of *LNCS*, pages 29–43. Springer, 2016.
[29] J. L. Hafner and K. S. McCurley. A rigorous subexponential algorithm for computation of class groups. *J. Amer. Math. Soc.*, 2(4):837–850, 1989.
[34] H. Hişil. *Elliptic curves, group law, and efficient computation*. PhD thesis, Queensland University of Technology, 2010. https://eprints.qut.edu.au/33233/.
[36] D. Jao, R. Azarderakhsh, M. Campagna, C. Costello, L. De Feo, B. Hess, A. Jalali, B. Koziel, B. LaMacchia, P. Longa, M. Naehrig, J. Renes, V. Soukharev, and D. Urbanik. SIKE. Submission to [55]. http://sike.org.
[37] D. Jao and L. D. Feo. Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. In *PQCrypto*, volume 7071 of *LNCS*, pages 19–34. Springer, 2011.
[38] D. Jao, J. LeGrow, C. Leonardi, and L. Ruiz-Lopez. A subexponential-time, polynomial quantum space algorithm for inverting the CM group action. *J. Mathematical Cryptology*, 2018. To appear.
[42] J. Kieffer. *Étude et accélération du protocole d'échange de clés de Couveignes–Rostovtsev–Stolbunov*. Mémoire du Master 2, Université Paris VI, 2017. https://arxiv.org/abs/1804.10128.
[45] D. Kohel. *Endomorphism rings of elliptic curves over finite fields*. PhD thesis, University of California at Berkeley, 1996. http://iml.univ-mrs.fr/~kohel/pub/thesis.pdf.
[46] G. Kuperberg. A subexponential-time quantum algorithm for the dihedral hidden subgroup problem. *SIAM J. Comput.*, 35(1):170–188, 2005.
[47] G. Kuperberg. Another subexponential-time quantum algorithm for the dihedral hidden subgroup problem. In *TQC*, volume 22 of *LIPIcs*, pages 20–34. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2013.
[49] M. Meyer and S. Reith. A faster way to the CSIDH, 2018. IACR Cryptology ePrint Archive 2018/782.

[50] D. Micciancio. Improving lattice based cryptosystems using the Hermite normal form. In *CaLC*, volume 2146 of *LNCS*, pages 126–145. Springer, 2001.
[51] V. S. Miller. Use of elliptic curves in cryptography. In *CRYPTO*, volume 218 of *LNCS*, pages 417–426. Springer, 1985.
[53] P. L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48(177):243–264, 1987.
[54] D. Moody and D. Shumow. Analogues of Vélu's formulas for isogenies on alternate models of elliptic curves. *Math. Comput.*, 85(300):1929–1951, 2016.
[55] NIST. Post-quantum cryptography. https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Post-Quantum-Cryptography-Standardization.
[58] C. Petit. Faster algorithms for isogeny problems using torsion point images. In *ASIACRYPT (2)*, volume 10625 of *LNCS*, pages 330–353. Springer, 2017.
[61] O. Regev. A subexponential time algorithm for the dihedral hidden subgroup problem with polynomial space, 2004. https://arxiv.org/abs/quant-ph/0406151.
[62] J. Renes. Computing isogenies between Montgomery curves using the action of $(0, 0)$. In *PQCrypto*, volume 10786 of *LNCS*, pages 229–247. Springer, 2018.
[63] M. Roetteler, M. Naehrig, K. M. Svore, and K. E. Lauter. Quantum resource estimates for computing elliptic curve discrete logarithms. In *ASIACRYPT (2)*, volume 10625 of *LNCS*, pages 241–270. Springer, 2017.
[64] A. Rostovtsev and A. Stolbunov. Public-key cryptosystem based on isogenies, 2006. IACR Cryptology ePrint Archive 2006/145.
[67] P. W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM J. Comput.*, 26(5):1484–1509, 1997.
[68] A. Stolbunov. Constructing public-key cryptographic schemes based on class group action on a set of isogenous elliptic curves. *Adv. in Math. of Comm.*, 4(2):215–235, 2010.
[70] M. Tibouchi. Elligator squared: Uniform points on elliptic curves of prime order as uniform random strings. In *Financial Cryptography*, volume 8437 of *LNCS*, pages 139–156. Springer, 2014.
[72] J. Vélu. Isogénies entre courbes elliptiques. *Comptes Rendus de l'Académie des Sciences de Paris*, 273:238–241, 1971.
[74] H. S. Wilf. *generatingfunctionology*. Academic Press, 1994. https://www.math.upenn.edu/~wilf/DownldGF.html.
[75] G. Zanon, M. A. S. Jr., G. C. C. F. Pereira, J. Doliskani, and P. S. L. M. Barreto. Faster isogeny-based compressed key agreement. In *PQCrypto*, volume 10786 of *LNCS*, pages 248–268. Springer, 2018.

## A Cost metrics for quantum computation

See full version of paper online at https://ia.cr/2018/1059.

## B Basic integer arithmetic

See full version of paper online at https://ia.cr/2018/1059.

## C Modular arithmetic

See full version of paper online at https://ia.cr/2018/1059.