

Further Optimizations of CSIDH: A Systematic Approach to Efficient Strategies, Permutations, and Bound Vectors

Aaron Hutchinson¹, Jason LeGrow¹, Brian Koziel², and Reza Azarderakhsh²

¹ Department of Combinatorics and Optimization and Institute for Quantum Computing, University of Waterloo, {a5hutchinson,jlegrow}@uwaterloo.ca

² Department of Computer and Electrical Engineering and Computer Science, Florida Atlantic University, {bkoziel2017, razarderakhsh}@fau.edu

Abstract. CSIDH is a recent post-quantum key establishment protocol based on constructing isogenies between supersingular elliptic curves. Several recent works give constant-time implementations of CSIDH along with some optimizations of the ideal class group action evaluation algorithm, including the SIMBA technique of Meyer *et al.* and the “two-point method” of Onuki *et al.* A recent work of Cervantes-Vázquez *et al.* details a number of improvements to the works of Meyer *et al.* and Onuki *et al.* Several of these optimizations—in particular, the choice of ordering of the primes, the choice of SIMBA partition and strategies, and the choice of bound vector which defines the secret keyspace—have been made in an *ad hoc* fashion, and so while they yield performance improvements it has not been clear whether these choices could be improved upon, or how to do so. In this work we present a framework for improving these optimizations using (respectively) linear programming, dynamic programming, and convex programming techniques. Our framework is applicable to any CSIDH security level, to all currently-proposed paradigms for computing the class group action, and to any choice of model for the underlying curves. Using our framework we find improved parameter sets for the two major methods of computing the group action: in the case of the implementation of Meyer *et al.* we obtain a 12.77% speedup *without* applying the further optimizations proposed by Cervantes-Vázquez *et al.*, while for that of Cervantes-Vázquez *et al.* under the two-point method we obtain a speedup of 5.06%, giving the fastest constant-time implementation of CSIDH to date.

1 Introduction

Isogenies between elliptic curves have gained increasing attention in the cryptographic world over the last several years. It is widely believed that the problem of constructing an isogeny between two given elliptic curves is hard, even with the power of quantum computing, and so it is natural to base cryptographic protocols around this problem. The use of isogenies in cryptography was initially proposed by Couveignes in [6], and was independently rediscovered by Stolbunov

and Rostovtsev in [18]. Perhaps the most well-known algorithm in isogeny-based cryptography is SIKE, one of the submissions to the National Institute for Standards and Technology's Post-Quantum Standardization process which is based on the Supersingular Isogeny Diffie-Hellman algorithm [8].

In 2018, Castryck, Lange, Martindale, Panny, and Renes proposed a similar key exchange algorithm titled Commutative Supersingular Isogeny Diffie-Hellman (CSIDH) in [2]. CSIDH uses the action of the ideal class group on the set of isomorphism classes of supersingular elliptic curves defined over \mathbb{F}_p to produce a key exchange algorithm reminiscent of the Diffie-Hellman method. Specifically, fix a prime of the form $p = 4\ell_1 \cdots \ell_n - 1$, where the ℓ_i are distinct small odd primes; in practice $\ell_1, \dots, \ell_{n-1}$ are the first $n-1$ odd primes, and ℓ_n is chosen small while ensuring p is prime. Let \mathcal{O} denote the \mathbb{F}_p -endomorphism ring of the supersingular Montgomery curve $E_0 : y^2 = x^3 + x$ defined over \mathbb{F}_p . Then \mathcal{O} has the property that each of the principal ideals $\ell_i \mathcal{O}$ splits into the product of $\mathfrak{l}_i = (\ell_i, \pi - 1)$ and $\bar{\mathfrak{l}}_i = (\ell_i, \pi + 1)$, where π is the Frobenius endomorphism of E_0 . Since $\ell_i \mathcal{O}$ is principal the elements of the ideal class group represented by these ideals are inverses, and so $[\mathfrak{l}_i]^{-1} = [\bar{\mathfrak{l}}_i]$ in the ideal class group.

To begin the key exchange protocol, Alice and Bob both select private keys of the form (e_1^A, \dots, e_n^A) and (e_1^B, \dots, e_n^B) , respectively, where each e_i^A and e_i^B is an integer chosen from some fixed interval $[-b, b]$. Alice uses her key to compute a curve E_A , defined as applying the action of the ideal $[\mathfrak{l}_1]^{e_1^A} \cdots [\mathfrak{l}_n]^{e_n^A}$ on the initial curve E_0 ; Bob proceeds analogously, using his own key to compute a curve E_B :

$$E_A := [\mathfrak{l}_1]^{e_1^A} \cdots [\mathfrak{l}_n]^{e_n^A} * E_0, \quad E_B := [\mathfrak{l}_1]^{e_1^B} \cdots [\mathfrak{l}_n]^{e_n^B} * E_0, \quad (1)$$

where $*$ denotes the ideal class group action. Alice then sends E_A to Bob and Bob sends E_B to Alice. Each party then computes the action of the ideal corresponding to their own private key on the curve they received from the other person; in particular, Alice computes E_{BA} and Bob computes E_{AB} , defined by:

$$E_{BA} := [\mathfrak{l}_1]^{e_1^A} \cdots [\mathfrak{l}_n]^{e_n^A} * E_B, \quad E_{AB} := [\mathfrak{l}_1]^{e_1^B} \cdots [\mathfrak{l}_n]^{e_n^B} * E_A. \quad (2)$$

The two curves E_{BA} and E_{AB} are \mathbb{F}_p -isomorphic since they both correspond to the action of $[\mathfrak{l}_1]^{e_1^A+e_1^B} \cdots [\mathfrak{l}_n]^{e_n^A+e_n^B}$ on the curve E_0 , by the commutativity of the ideal class group. The shared key is the \mathbb{F}_p -isomorphism class of $E_{BA} \cong E_{AB}$.

The original method proposed in [2] for carrying out the actions in (1) and (2) is to first choose a random point $P \in E[\pi \pm 1]$, where E is the current curve and π denotes the Frobenius endomorphism. The point P will have some order $|P| = \ell_1^{c_1} \cdots \ell_n^{c_n}$, where $c_i \in \{0, 1\}$ (after multiplication by 4). The curve $\prod_{c_i=1} [\mathfrak{l}_i]^{c_i} * E_A$ can be computed by iteratively multiplying out all but one prime from P to yield a point Q , constructing the isogeny $\varphi : E \rightarrow E/\langle Q \rangle$ via Vélu's formulas, and updating $P \leftarrow \varphi(P)$ and $E \leftarrow E/\langle Q \rangle$. One then repeats this procedure with a fresh point P , skipping any primes ℓ_i for which the action of the target ideal $[\mathfrak{l}_i]^{e_i}$ has been completed. Since the work of [2], there has been much focus on making the evaluation of the group action more efficient.

Previous CSIDH Optimizations. CSIDH is a very new construction, but there have been many contributions toward optimizing it. We focus on works which optimize the structure of the group action evaluation itself, and put less emphasis on methods which improve curve arithmetic, isogeny computation, *etc.*

Meyer and Reith gave the first optimization [14] in 2018. After choosing a random point P the user has the freedom to choose the order in which the action of the $[\mathfrak{l}_i]$ are computed by selecting which primes ℓ_i to multiply out of $|P|$ first. The authors of [14] noticed that computing the action in descending order of primes results in a speedup over using an ascending order. They make other notable computational contributions as well, such as projectivizing the curve coefficients and deriving formulas for the codomain curves using twisted Edwards curves. See [14] for full details.

Meyer, Campos, and Reith gave a second optimization [13] in late 2018. First, they proposed to change the keyspace interval $[-b, b]$ so that each private key value e_i is selected from its own interval $[0, b_i]$ and the target security level is still achieved. Each private key value having the same sign is desirable since ideals $[\mathfrak{l}_i]$ and $[\mathfrak{l}_j]^{-1}$ cannot be computed using the same initial point P , *i.e.*, once the field of definition of P is determined only the ideals of the corresponding sign can be considered. Furthermore the values b_i can be selected to achieve a speedup, and the authors use heuristics to find well-performing values for these parameters. Additionally the authors propose to use ‘dummy’ isogenies so that the same number of isogenies are always constructed, independent of the private key used. Specifically, e_i many ‘real’ isogenies and $b_i - e_i$ many dummy isogenies would be constructed, where the dummy computations would construct an isogeny but not update the points and curve coefficients to their new values. In essence, the isogenies are constructed but not used on dummy iterations. To our knowledge this was the first constant-time implementation of CSIDH.

One of the most notable contributions that Meyer, Campos, and Reith make in [13] is SIMBA (Splitting Isogenies into Multiple BAtches). The SIMBA technique partitions the primes $\{\ell_1, \dots, \ell_n\}$ into disjoint sets and evaluates the required group action on each subset individually. See Section 2.4 for more details on the SIMBA technique. The authors of [13] use a simple method for determining the partition, but one might also ask how to find an optimal partition.

A third optimization and constant-time version of CSIDH was performed by Onuki, Aikawa, Yamazaki, and Takagi in [16]. Here the authors retain signed key values e_i chosen from some interval $[-b_i, b_i]$. They track two randomly chosen points $P^+ \in E[\pi - 1]$ and $P^- \in E[\pi + 1]$ through the algorithm. For each prime ℓ_i , the appropriate point is used to derive a kernel generator according to the sign of e_i by multiplying out all other primes as before. Both P^+ and P^- are then mapped through the isogeny to the next curve, and the point not used to derive the kernel generator is multiplied by ℓ_i . This allows both the $[\mathfrak{l}_i]$ and $[\mathfrak{l}_i]^{-1}$ to be considered on each iteration instead of being limited to only one.

There have been a few other improvements to CSIDH which optimize lower level aspects of the algorithm, and we only briefly note them here. In [15] the authors describe how to perform the CSIDH algorithm using Edwards curves

instead of Montgomery curves, giving an algorithm comparable in operation cost. The authors of [12] implement CSIDH in embedded devices while optimizing the field arithmetic and group operations. In [3], XZ -coordinates are used on twisted Edwards curves with optimized addition chains for scalar multiplications, and two flaws in the constant-time implementations of [13] and [16] are repaired resulting in a speedup. The implementation of [3] is the fastest to date.

CSIDH Group Action Algorithm. Here we look at the ideal class group action evaluation algorithm performed in CSIDH as originally described in [2]. This algorithm takes input integers (e_1, \dots, e_n) and Montgomery curve coefficient $A \in \mathbb{F}_p$ and outputs the coefficient of the curve $[l_1]^{e_1} \cdots [l_n]^{e_n} * E_A$. The evaluation is given in Algorithm 1 as it is written in [2].

Algorithm 1: CSIDH Group Action Evaluation

```

Input :  $A \in \mathbb{F}_p$  and a list of integers  $(e_1, \dots, e_m)$ .
Output:  $B$  such that  $[l_1^{e_1} \cdots l_m^{e_m}]E_A = E_B$  (where  $E_B : y^2 = x^3 + Bx^2 + x$ ).
1 while some  $e_i \neq 0$  do
2   Sample a random  $x \in \mathbb{F}_p$ .
3   Set  $s \leftarrow +1$  if  $r := x^3 + Ax^2 + x$  is a square in  $\mathbb{F}_p$ , else  $s \leftarrow -1$ .
4   Let  $I = \{i | e_i \neq 0, \text{sign}(e_i) = s\}$ . If  $I = \emptyset$ , then start over with a new  $x$ .
5   Let  $t \leftarrow \prod_{i \in I} \ell_i$  and compute  $Q \leftarrow [(p+1)/t]P$ , where  $P := (x, \sqrt{r})$ .
6   for each  $i \in I$  do
7     Compute  $R \leftarrow [t/\ell_i]Q$ . If  $R = \infty$ , then skip this  $i$ .
8     Compute an isogeny  $\varphi : E_A \rightarrow E_B : y^2 = x^3 + Bx^2 + x$  with  $\ker \varphi = \langle R \rangle$ .
9     Set  $A \leftarrow B, Q \leftarrow \varphi(Q), t \leftarrow t/\ell_i$ , and finally  $e_i \leftarrow e_i - s$ .
10  end
11 end
12 Return  $A$ 

```

A given iteration of the loop on line (6) of Algorithm 1 would use a point Q to compute $[u]Q$ for some integer u , and then build an isogeny φ using $[u]Q$ as the generator for $\ker \varphi$. The following iteration will compute $[u/\ell_i]\varphi(Q)$ from $\varphi(Q)$. Writing u/ℓ_i as v , the effect from these two iterations is to compute $[v\ell_i]Q$ and $[v]\varphi(Q)$ given only the point Q . The algorithm as written accomplishes this by evaluating $[v\ell_i]$, evaluating φ , and finally evaluating $[v]$. If the integer v is large (as is often the case), this method potentially requires more effort than, say, computing $[v]Q$, then $[\ell_i][v]Q$, then $\varphi([v]Q)$.

A similar observation holds on a larger scale. For simplicity suppose line (4) of Algorithm 1 computes $I = \{1, \dots, n\}$. The goal of the loop on line (6) is to use the initial point Q defined on line (5) to successively compute the points

$$\begin{aligned}
(1.) & \quad [\ell_1 \cdots \ell_{n-1}]Q \\
(2.) & \quad [\ell_1 \cdots \ell_{n-2}]\varphi_1(Q) \\
(3.) & \quad [\ell_1 \cdots \ell_{n-3}]\varphi_2\varphi_1(Q) \\
& \vdots \qquad \qquad \qquad \vdots
\end{aligned}$$

$$\begin{array}{ll}
(n-1.) & [\ell_1]\varphi_{n-2} \cdots \varphi_1(Q) \\
(n.) & \varphi_{n-1}\varphi_{n-2} \cdots \varphi_1(Q)
\end{array}$$

while also constructing the isogenies φ_i as needed. These n points can be computed from Q in a wide variety of different ways, and is entirely reminiscent of the problem of efficiently constructing an isogeny of degree ℓ^n detailed by De Feo, Jao, and Plût in [8]. In fact, if one takes all primes ℓ_i above to be some common prime ℓ , the problem of efficiently computing the n points defined above reduces to precisely the problem solved in [8], which makes use of “optimal strategies”.

We point out that the user has the freedom to iterate through the set I in any fashion desired due to the ideal class group being abelian. If a different order of iteration is chosen, the corresponding points (as well as the curves themselves) computed by the algorithm will differ since the sequence of points $\{[\ell_1 \cdots \ell_{i-1}]\varphi_{n-i} \cdots \varphi_1(Q)\}$ depends on the ordering. Changing the ordering changes the computations involved, and so the computations for some orderings may require less effort than others. As far as we are aware, all previous implementations of CSIDH at the time of this writing use heuristics to select a well-performing permutation of the primes ℓ_i , and a systematic method of determining an efficient permutation remains a relatively untouched problem.

Contributions. The contributions of this work are as follows:

- We detail a general framework for analyzing and optimizing the CSIDH group-action evaluation algorithm. This framework applies to any CSIDH parameter set and can be tailored to further optimize any other CSIDH implementation to date, such as those of [2,13,14,16,3]. Specifically, we use our framework to optimize parameters used in any CSIDH instantiation:
 - We generalize the concept of the *measure* of a *strategy*, originally defined in [8]. Any strategy on n leaves provides a method for carrying out the CSIDH algorithm. We analyze these strategies and are able to find globally optimal strategies when fixing the permutation parameter. A dynamic programming approach similar to that of [8] will easily find these optimal strategies for practical CSIDH parameters.
 - We frame the problem of finding an optimal permutation of the primes ℓ_i —for a fixed strategy—as a linear program; that is, an optimization problem in which the objective function and constraints are affine functions of the the permutation variables. This allows us to use linear programming techniques (*e.g.*, the simplex method) to find a corresponding optimal permutation. This technique extends in a straightforward fashion to SIMBA, and can be used to find not only an optimal permutation of primes for each batch, but also an optimal distribution of primes to the SIMBA substrategies of a fixed SIMBA strategy.
 - We derive a mathematical program to produce a bound vector which approximately optimizes the running time for the class group action algorithms used in CSIDH. We approximate the solution by relaxing to a convex program and applying an iterative rounding technique.

- We further generalize the SIMBA technique of [13] to allow for different SIMBA strategies on each round of the algorithm, and eliminate each prime ℓ_i from all strategies after the b_i^{th} round.
- We used our optimization techniques to find parameter sets consisting of efficient SIMBA strategies, permutations, and bound vectors for two previous constant-time implementations of CSIDH-512: that of Meyer *et al.* in [13], and Cervantes-Vázquez *et al.* in [3]. Our optimized implementations achieve a speedup of 12.77% over the original code of [13] (without the optimizations proposed by [3]), and a speedup of 5.06% over the original code of [3] using the two-point method. To the best of our knowledge this gives the fastest constant-time implementation of CSIDH to date.

This paper is organized as follows. Section 2 details the framework which we use to optimize CSIDH, and discusses strategies, measures, permutations, the two-point method [16], and SIMBA [3]. Section 3 develops theoretical methods for finding efficient parameters for computing the ideal class group action for CSIDH, including strategies, permutations, and bound vectors. In Section 4 we report the results of our implementation of our best parameter sets.

2 Preliminaries

2.1 General Framework for Optimization

Strategies. The idea of a *strategy* has been explored in [8], but we use an alternative definition to better suit our needs. For a positive integer n we let $T_n = (V, E)$ be the directed graph defined as follows. The vertices V of T_n are all points in the plane with integer coordinates which lie inside or on the boundary of the region bounded by the lines $x = 0$, $y = 0$, and $y = -x + n - 1$. The edges E of T_n consist of all line segments of unit length which connect two vertices in V . It follows that every edge is either horizontal or vertical. We turn T_n into a directed graph by orienting all horizontal edges to the right and all vertical edges upward.

Definition 1. A *strategy* (in T_n) is a subgraph of T_n such that:

1. The vertex $(0, 0)$ and all vertices on the line $y = -x + n - 1$ are in S ,
2. For each vertex v on the line $y = -x + n - 1$, there is a (not necessarily unique) path from $(0, 0)$ to v in S .

We write $|S| = n$ to mean S is a strategy in T_n .

To define our version of canonical strategy, we define a binary operator $\#$ called *join* on the set of all strategies. For strategies S_1 and S_2 , with $|S_1| = n_1$ and $|S_2| = n_2$, we define $S_1 \# S_2$ to be the strategy in $T_{n_1+n_2}$ constructed as follows:

1. $S_1 \# S_2$ contains the (unique) path connecting $(0, 0)$ to $(n_2, 0)$,
2. $S_1 \# S_2$ contains the (unique) path connecting $(0, 0)$ to $(0, n_1)$,
3. $S_1 \# S_2$ contains S_1 as a subgraph, shifted to the right n_2 units,

4. $S_1 \# S_2$ contains S_2 as a subgraph, shifted up n_1 units.

The join operator is both nonassociative and noncommutative. We say a strategy S in T_n is *canonical* if S can be expressed as $n - 1$ many applications of the join operator on the strategy T_1 ; *i.e.*, S is some parenthesization of $\underbrace{T_1 \# T_1 \# \cdots \# T_1}_n$.

Each canonical strategy has a unique such expression, and so it follows that the number of canonical strategies in T_n is the number of parenthesizations of a binary operator on n terms. This is exactly the n^{th} Catalan number. An easy induction shows that every vertex in a canonical strategy has indegree at most 1 and outdegree at most 2, and a vertex has outdegree 0 precisely when it lies on the line $y = -x + n - 1$. This allows one to associate a binary tree structure to each canonical strategy S , and we therefore say that $(0, 0)$ is the *root* of S , and the vertices on the line $y = -x + n - 1$ are the *leaves* of S .

Suppose we merge together all but the outermost join operation to write a canonical strategy as $S = S_1 \# S_2$ for some canonical strategies S_1 and S_2 ; we define $S^L := S_1$ to be the *left substrategy* of S , and $S^R := S_2$ to be the *right substrategy* of S . We emphasize that visually S^L lies to the right of the origin, and S^R lies above the origin. By definition of $\#$, we always have $|S_1 \# S_2| = |S_1| + |S_2|$.

In the context of CSIDH, we interpret the horizontal edges of a strategy as individual point multiplications and the vertical edges as isogeny evaluations, which motivates the following definitions. The n^{th} *multiplication-based* strategy MB_n is defined recursively as $MB_1 = T_1$ and $MB_n = T_1 \# MB_{n-1}$. The n^{th} *isogeny-based* strategy IB_n is defined recursively as $IB_1 = T_1$ and $IB_n = IB_{n-1} \# T_1$. As far as we are aware, every implementation of CSIDH uses (various sizes of) a multiplication-based strategy to perform the ideal class group action evaluation.

Our definition of strategy is entirely equivalent to that of a *full strategy* as defined in [8], and our canonical strategies are equivalent to those of [8]; we simply view the problem on a rectangular lattice as opposed to an equilateral triangular lattice, and the root of our strategies always corresponds to the origin.

Encoding Strategies. It will be convenient in our analysis and for algorithmic purposes to have a systematic method of writing down the edges which are present in a given strategy S . To do this we use two $\{0, 1\}$ -valued $(n-1) \times (n-1)$ sized matrices $H(S)$ and $V(S)$ (or simply H and V when S is clear), which respectively encode the horizontal and vertical edges of S . Specifically, $H_{ij} = 1$ if and only if the line segment connecting $(j-1, n-1-i)$ to $(j, n-1-i)$ is present in the strategy S , and $H_{ij} = 0$ otherwise. Similarly $V_{ij} = 1$ if and only if the line segment connecting $(j-1, n-i-1)$ to $(j-1, n-i)$ is present in S , and $V_{ij} = 0$ otherwise. Both H and V are lower triangular matrices since T_n is bounded by the line $y = -x + n - 1$. $H(T_n)$ and $V(T_n)$ are both lower triangular matrices in which every entry on and below the main diagonal is a 1. See Figure 2 in Appendix D for an example of a canonical strategy and its encoding matrices.

Measures. We now generalize the idea of *measure* from [8] to account for differing weights for differing edges, which we need to analyze CSIDH strategies.

Definition 2. A *measure* on T_n is a tuple $M = (\{p_i\}_{i=1}^n, f, g)$, where:

- $\{p_i\}_{i=1}^n$ is a sequence of positive real numbers,
- $f, g : \mathbb{R}^+ \rightarrow \mathbb{R}^+$ are some weight functions.

We assign weights to the edges of T_n using the measure M as follows. For $1 \leq i \leq n-1$ we assign the weight $f(p_i)$ to any horizontal edge which connects a vertex on the line $x = i-1$ to a vertex on the line $x = i$. For $1 \leq i \leq n-1$, we assign the weight $g(p_{n-i+1})$ to any vertical edge which connects a vertex on the line $y = i-1$ to a vertex on the line $y = i$. Any strategy in T_n inherits the weights from T_n .

Taking $\{p_i\}$ to be a constant sequence yields the original notion of measure defined in [8] when interpreted under our definition of T_n . Though the assignment of weights to vertical edges may seem unnatural, it is motivated by CSIDH, where the cost of the i^{th} isogeny evaluation depends on the degree of the isogeny, which in turn depends on the $(n-i+1)$ -th prime used. In this case, $f(p_i)$ represents the cost of multiplying a point by p_i , whereas $g(p_{n-i+1})$ represents the cost of evaluating an isogeny of degree p_{n-i+1} at a point.

Throughout this paper, differing measures will all use common weight functions f and g . We will often identify a measure M with its sequence $\{p_i\}_{i=1}^n$ and omit mention of the functions f and g .

Definition 3. The *cost* of a subgraph S of T_n for a given measure M is the sum of the weights of all edges in S . We write $(S)_M$ for the cost of S relative to M , or (S) when M is clear.

Equation (3) below gives a formula for the cost of a subgraph.

Permutations. In our original problem of optimizing CSIDH, we have the freedom to choose the order in which the primes ℓ_i are used. Choosing a different order will result in a permuted measure M , and so we need to take into account all possible permutations of M in our analysis.

Definition 4. Let $\text{Sym}(n)$ denote the symmetric group on $\{1, 2, \dots, n\}$. We let $\sigma \in \text{Sym}(n)$ act on a measure $M = \{p_i\}_{i=1}^n$ by defining $\sigma \cdot M = \{p_{\sigma(i)}\}_{i=1}^n$.

The cost of a strategy S under the permuted measure $\sigma \cdot M$ is

$$(S)_{\sigma M} = \sum_{i=1}^{n-1} f(p_{\sigma(i)}) \sum_{j=1}^{n-1} H_{j,i} + \sum_{i=1}^{n-1} g(p_{\sigma(i+1)}) \sum_{j=1}^{n-1} V_{i,j}. \quad (3)$$

Our goal is to find an algorithm which determines a pair (S, σ) for a given measure M such that $(S)_{\sigma M}$ is minimal among all such pairs. This would yield an optimal method for evaluating the ideal class group action for CSIDH.

2.2 Mitigating Leakage Under Arbitrary Strategies

As first pointed out by Meyer *et al.* in [13] one may use dummy isogenies in CSIDH so that the number of isogenies constructed during the group action evaluation is independent of the private key. One issue that arises from using dummy isogenies is that additional multiplications are required on iterations that construct a dummy isogeny. This is because a real isogeny evaluation within the algorithm reduces the order of the point by a factor of the degree ℓ of the isogeny. If the isogeny is dummy, then the value of the point won't be updated and the factor ℓ will remain. In this situation we should instead multiply the point by ℓ to remove this factor.

Since strategies different from the multiplication-based strategy may require multiple isogeny evaluations on a given iteration, instead of multiplying all the points by ℓ we can simply multiply the initial randomly chosen point by any primes which will correspond to a dummy isogeny construction before the evaluation of the strategy begins. In this way we remove the 'bad' factors at the start by means of a single scalar multiplication per prime. This can be done in a secure fashion by using two copies of the point, multiplying one of them by each prime (not just the primes for dummies) while conditionally swapping the two points depending on the private key value for the current prime.

2.3 Two-Point Method and Parallelization

In [16], Onuki *et al.* find improved performance by tracking two points through each strategy: one from $E[\pi - 1]$ and one from $E[\pi + 1]$. When reaching an isogeny construction, the appropriate point is used depending on the sign of the private key in the corresponding position.

In the multiplication-based strategy, having two points results in a negligible cost increase since only one of the two points needs to be multiplied to derive the kernel generator of the isogeny (though both points are still evaluated under the isogeny). When using other strategies this luxury is not an option since the path from the root to the leaf under consideration may pass through internal branch vertices, and so both points should be multiplied through nearly the entire strategy; the exception is horizontal paths within the strategy that end at a leaf and contain no branch vertices, in which case one can only multiply through whichever point is needed at the leaf node. In a non-parallel computation model, this would result in highly increased cost since it uses roughly double the number of point multiplications.

As a potential remedy, one might parallelize the operations on the two points together, allowing strategies different from the multiplication-based strategy to feasibly be used. We theorize that the parallelization results of Hutchinson and Karabina in [11] apply in this case, but we do not pursue this avenue here.

2.4 Splitting Isogenies into Multiple Batches (SIMBA)

In [13], Meyer *et al.* propose to partition the set of primes $\{\ell_1, \dots, \ell_{74}\}$ into m many disjoint subsets to evaluate the group action on each subset individually.

The output curve from evaluating the action on one subset is fed as the input curve to the next, and a new initial point P is chosen for each iteration of each subset. They focus exclusively on positive private key values so that P is always chosen from $E[\pi - 1]$, and it's more likely that $|P|$ contains larger prime factors than smaller ones. Consequently, after a given number of rounds on a fixed key it's more likely that lower degree isogenies will still need to be constructed than higher ones. Meyer *et al.* therefore find it beneficial to merge the primes back into one set after μ many iterations and run CSIDH as originally proposed (but still using dummy isogenies) to construct the remaining isogenies. They call this technique Splitting Isogenies into Multiple Batches, or SIMBA- m - μ .

Within our framework, SIMBA can be summarized as: partition the primes $\{\ell_1, \dots, \ell_n\}$ into m subsets, associate some strategy with each subset, and evaluate each strategy using the primes from each subset. Fresh points are randomly chosen for each strategy and must be multiplied by every prime not in the current subset, as well as by 4, prior to beginning the operations within the strategy.

We can generalize this further. First, there is no reason that the same strategy and permutation must be used for each of the subsets, so we are free to choose optimal parameters on each of them. Second, it's not required that the same partitioning be used each round. That is, once the strategies for each of the subsets have been evaluated once, we could optionally repartition the primes and use a different collection of strategies. This is quite advantageous since if any value b_i in the private key bound vector \mathbf{b} is small in comparison to the rest of the vector, the prime ℓ_i can simply be removed from the partitioning after b_i number of rounds since all degree ℓ_i isogenies (both real and dummy) have likely been constructed by that point. This also eliminates the need of merging the batches after μ rounds since each batch is on a ‘minimal’ set of primes to begin with. Overall this has the effect of eliminating a significant number of redundant operations, although it yields a much more complex algorithm.

This motivates the following definition. Recall that we identify a measure M with its sequence $\{p_i\}$, leaving the weight functions f and g implicit.

Definition 5. For a collection of numbers $M = \{p_1, \dots, p_n\}$, a **SIMBA strategy** S is a collection of pairs $(S_1, M_1), \dots, (S_m, M_m)$ such that

1. S_i is a strategy (under Definition 1) for $i = 1, \dots, m$,
2. M_i is a measure for S_i for $i = 1, \dots, m$,
3. M is the disjoint union of M_1, \dots, M_m .

The S_i are referred to as the **SIMBA substrategies**, and M_i the **SIMBA submeasures**, of S . We say $(|S_1|, \dots, |S_m|)$ is the **SIMBA partition** of S .

We can also encode SIMBA strategies as matrices; see Appendix A for details.

2.5 General Algorithm

Once a strategy and permutation have been chosen, the method for evaluating them is fairly intuitive and at a high level closely mimics the procedure for evaluating a strategy for SIDH [8]. See Appendix D for an example, and Appendix C for the complete algorithm description.

3 Optimization Methods

For much of this section we work over an arbitrary set of primes $M = \{p_1, \dots, p_n\}$, and all strategies, permutations, and measures will reference these primes. These primes can be thought of as some subset of the odd primes used in CSIDH. Sections 3.1, 3.2, and 3.3 respectively tackle optimizing the strategy, permutation, and bound vector variables. Finally, in Section 3.4, we discuss how the three optimization algorithms come together to produce a full parameter set for CSIDH.

3.1 Optimizing the Strategies

Let M be a measure. In this section we fix an arbitrary permutation σ and describe a method for determining an optimal canonical strategy for the permuted measure σM . That is, we optimize $(S)_{\sigma M}$ over S for fixed σ and M . For this section by replacing M with σM we may assume that σ is the identity permutation. This reduces the problem to finding an optimal strategy for a measure M . This is done nearly identically to the method described in [8] for constant measures.

Theorem 1. *Fix a measure $M = \{p_i\}_{i=1}^n$. Suppose S is a canonical strategy for which $(S)_M$ is minimal over all canonical strategies for M . If $k = |S^L|$, then S^L and S^R are canonical strategies for which $(S^L)_{M^L}$ and $(S^R)_{M^R}$ are minimal over all canonical strategies for M^L and M^R , respectively, where $M^L := \{p_i\}_{i=n-k+1}^n$ and $M^R := \{p_i\}_{i=1}^{n-k}$.*

Theorem 1 is a generalization of [8, Lemma 4.5]. The proof is essentially the same, with the appropriate generalizations made—it appears in Appendix B.

Definition 6. *For a measure $M = \{p_i\}_{i=1}^n$ with $n > 1$, for $1 \leq k \leq n - 1$ we define the k -th **left** and **right submeasures** of M as*

$$M_k^L = \{p_i\}_{i=n-k+1}^n \quad M_k^R = \{p_i\}_{i=1}^{n-k}.$$

Let $C(M)$ be the cost of an optimal strategy under the measure $M = \{p_i\}_{i=1}^n$. As a consequence of Theorem 1, $C(M)$ can be computed recursively as

$$C(M) = \min_{k=1, \dots, n-1} \left\{ C(M_k^L) + C(M_k^R) + \sum_{i=1}^{n-k} f(p_i) + \sum_{i=n-k+1}^n g(p_i) \right\}. \quad (4)$$

Just as in the case of finding an optimal strategy for SIDH in [8], the above equality again suggests a dynamic programming approach for finding an optimal strategy in our generalized setting. That is, we compute $C(\{p_i\}_{i=1}^n)$ by using a sliding window submeasure which increases in size: we iterate $k = 1, \dots, n$ and $j = 1, \dots, n - k + 1$ and compute $C(\{p_i\}_{i=j}^{j+k-1})$ using equation (4) with the length-one measure initial values $C(p_i) = 0$ for all i . Here, k represents the window size and j represents the window position. This gives an $\tilde{O}(n^2)$ algorithm computing the cost of the best strategy, and an optimal strategy can

be constructed by keeping track of an index at which the minimum occurs at each step. Alternatively, one may construct the matrices H and V for the optimal strategy recursively as defined in Section 2.1.

In the two-point setting of [16], a similar result holds by doubling most of the above summations. The discussion in Section 2.3 suggests that every edge in the strategy should have double weight, except those which lie on a horizontal path ending in a leaf and containing no branch node. This occurs precisely when the left substrategy is T_1 . Thus for the two-point scenario we have the recursion

$$C(M) = \min \left(\left\{ C(M_1^R) + \sum_{i=1}^{n-1} f(p_i) + 2g(p_n) \right\} \cup \left\{ C(M_k^L) + C(M_k^R) + \sum_{i=1}^{n-k} 2f(p_i) + \sum_{i=n-k+1}^n 2g(p_i) : k = 2, \dots, n-1 \right\} \right). \quad (5)$$

3.2 Optimizing the Permutations

We now fix a full strategy S and measure M , and show how to use mathematical programming to find a permutation σ which minimizes $(S)_{\sigma M}$. Write $M = (\{p_i\}_{i=1}^n, f, g)$, and define vectors $\mu = [f(p_i)]_{i=1}^n$ and $\nu = [g(p_i)]_{i=1}^n$.

Let H and V be the matrices that encode the edges of S . If the primes are permuted according to σ , then by Equation (3) we have

$$(S)_{\sigma M} = \sum_{i=1}^{n-1} \sum_{j=1}^{n-1} H_{i,j} \mu_{\sigma(j)} + \sum_{i=1}^{n-1} \sum_{j=1}^{n-1} V_{i,j} \nu_{\sigma(i+1)}.$$

In order to simplify this expression and write it in a form that is amenable to standard optimization techniques, we will use the permutation matrix representation of $\text{Sym}(n)$. For any $\sigma \in \text{Sym}(n)$, let $\rho(\sigma) \in \{0,1\}^{n \times n}$ be defined by $\rho(\sigma) = \sum_{i=1}^n e_i e_{\sigma(i)}^T$ where $\{e_i\}_{i=1}^n$ are the standard basis vectors. Letting $T_L = [I_{n-1} | \mathbf{0}]$, $T_R = [\mathbf{0} | I_{n-1}]$, and $\Sigma = \rho(\sigma)$ with I_{n-1} an identity matrix of size $n-1$, we can write $(S)_{\sigma M} = \langle T_L^T H^T \mathbf{1} \mu^T + T_R^T V^T \mathbf{1} \nu^T, \Sigma \rangle_F$ where $\langle \cdot, \cdot \rangle_F$ is the Frobenius inner product. Then the problem of finding the optimal permutation for a given strategy and measure is to minimize the above quantity subject to Σ being a permutation matrix; more succinctly:

$$\begin{aligned} & \text{Minimize} && \langle T_L^T H^T \mathbf{1} \mu^T + T_R^T V^T \mathbf{1} \nu^T, \Sigma \rangle_F \\ & \text{Subject to} && \Sigma \mathbf{1} = \mathbf{1} \\ & && \mathbf{1}^T \Sigma = \mathbf{1}^T \\ & && \Sigma \geq 0 \\ & && \Sigma \in \mathbb{Z}^{n \times n} \end{aligned} \quad (6)$$

Problem (6) is an integer linear program. Relaxing the integrality constraint, we obtain a linear program whose feasible region is B_n —the Birkhoff polytope in \mathbb{R}^{n^2} . The vertices of B_n are precisely the $n \times n$ permutation matrices; then, by the

Fundamental Theorem of Linear Programming, there is an optimal solution to the relaxed problem which is feasible (and hence optimal) for (6). Such a solution can be found easily using standard techniques (e.g. the simplex method).

For SIMBA and the two-point method, (6) must be modified to account for changes to the cost model; these modifications are described in Appendix E.1.

3.3 Optimizing the Bound Vector

We now leave behind the setting of full generality and return to CSIDH, where we consider the primes $M = \{\ell_1, \dots, \ell_n\}$. Castryck *et al.* in [2] propose to select the values of the private key (e_1, \dots, e_n) from some common interval $[-b, b]$. Meyer *et al.* in [13] instead consider sampling each value e_i from its own interval $[0, b_i]$, where the vector $\mathbf{b} = (b_1, \dots, b_n)$ is to be chosen so that a speedup is gained while still maintaining a target security level. In [13] the authors state that trying to find optimal values of b_i leads to a large integer optimization problem which is not likely to be solvable exactly. They give some vectors \mathbf{b} that they found heuristically, but gave no details on the method used to find the provided values. We give details on our optimization problem now.

To write a mathematical program for the optimal exponent bound vector \mathbf{b} , we must determine the relationship between \mathbf{b} and the cost of computing the (real and dummy) isogenies for the group action, using a given strategy, as well as the constraints that must be enforced on \mathbf{b} in order to ensure security.

The requirement to maintain security in the case of non-negative exponents (*à la* [13]) is that ideals of the form $\mathfrak{l}_1^{e_1} \cdots \mathfrak{l}_n^{e_n}$ for $0 \leq e_i \leq b_i$ cover the class group nearly uniformly. An analysis was performed in [16] when selecting e_i from the intervals $[-b_i, b_i]$, which can be easily adapted to the case $[0, b_i]$. Under this adaptation, the requirement for the vector \mathbf{b} when selecting each e_i from the interval $[0, b_i]$ is that $\prod(b_i + 1)$ is at least the size of the class group. By the heuristics in [4] the size of the class group is approximately \sqrt{p} (recall that $p = 4\ell_1 \cdots \ell_n - 1$), and so we need $\prod(b_i + 1) \geq \sqrt{p}$ as a constraint in the optimization problem. Then, sufficient security can be guaranteed by enforcing

$$\prod_{i=1}^n (b_i + 1) \geq \sqrt{p} \iff \sum_{i=1}^n \log_2(b_i + 1) \geq \frac{1}{2} \log_2 p =: \lambda. \quad (7)$$

This reformulated constraint is convex, which is computationally convenient.

In the case of exponents which are not restricted to be non-negative (*à la* [2,16]) the argument of [16] applies without modification, and we arrive at a similarly-reformulated convex constraint as (7) where b_i is replaced with $2b_i$.

All that remains is to determine the cost of computing the isogenies when executing a given strategy. As before, let $\mu_{\sigma(i)}$ and $\iota_{\sigma(i)}$ denote the cost of evaluating multiplication-by- $\ell_{\sigma(i)}$ maps and evaluating $\ell_{\sigma(i)}$ -isogenies, respectively. As well, let $\kappa_{\sigma(i)}$ be the combined cost of computing the kernel points from a given generator and computing the codomain curve of an $\ell_{\sigma(i)}$ -isogeny.

We must consider two cases: rounds in which $\ell_{\sigma(i)}$ is ‘active’ (that is, there are still $\ell_{\sigma(i)}$ -isogenies to be computed), and rounds in which $\ell_{\sigma(i)}$ is ‘inactive’ (that is, there are no more $\ell_{\sigma(i)}$ -isogenies to compute).

$\ell_{\sigma(i)}$ is active. In this case, we must:

1. Compute one $\ell_{\sigma(i)}$ -isogeny kernel and codomain curve, incurring cost $\kappa_{\sigma(i)}$.
2. Evaluate $[\ell_{\sigma(i)}]$ for each 1 in i^{th} column of H , if $i \leq n-1$, at cost $(\mathbf{1}^T H)_i \mu_{\sigma(i)}$
3. Evaluate an $\ell_{\sigma(i)}$ -isogeny for each 1 in $(i-1)^{\text{th}}$ row of V , if $i \geq 2$, at cost $(V\mathbf{1})_{i-1} \iota_{\sigma(i)}$.

$\ell_{\sigma(i)}$ is inactive. In this case, we must evaluate $[\ell_{\sigma(i)}]$ once, at cost $\mu_{\sigma(i)}$.

Let c_i denote the cost associated with prime ℓ_i in an active round, and d_i denote the cost associated with prime ℓ_i in an inactive round. In the event that the starting point in every round is of full order (so that an isogeny of each order can be computed in each round), there are b_i active rounds for ℓ_i and $\max_j \{b_j\} - b_i$ inactive rounds for ℓ_i . Thus the total cost associated with ℓ_i is

$$c_i \cdot b_i + d_i \cdot (\max_j \{b_j\} - b_i) = (c_i - d_i) \cdot b_i + \max_j \{b_j\} d_i$$

so that the total cost across all i is $\langle \mathbf{c} - \mathbf{d}, \mathbf{b} \rangle + \max_j \{b_j\} \mathbf{1}^T \mathbf{d}$, where

$$\mathbf{c} = \Sigma^{-1} ((\mathbf{1}^T H T_L)^T \circ (\Sigma \boldsymbol{\mu}) + (T_R^T V \mathbf{1}) \circ (\Sigma \boldsymbol{\iota}) + \Sigma \boldsymbol{\kappa}) \text{ and } \mathbf{d} = \boldsymbol{\mu}$$

where \circ is the Hadamard product.

So far we have accounted only for the cost of the first $\max_j \{b_j\}$ strategy executions. If each execution always lets us evaluate isogenies of each active degree ℓ_i this would be sufficient; however, we are not guaranteed that our initial points P will be of full order, so it is possible that there will be some active primes for which we cannot construct the required isogenies. When this happens, we must perform additional rounds of computation. To account for this additional cost, we estimate the number of additional rounds required and their cost.

The point P_0 allows us to compute the required $\ell_{\sigma(i)}$ -isogeny if and only if:

1. $P_0 \in E[\pi - 1]$ (in case $b_{\sigma(i)} > 0$), or $P_0 \in E[\pi + 1]$ (in case $b_{\sigma(i)} < 0$); and,
2. $\ell_{\sigma(i)}$ divides the order of P_0 .

If we choose $\mathbf{b} \geq \mathbf{0}$ (as proposed in [13]), or use the two-point technique of [16], at the beginning of each strategy round these conditions are satisfied with probability $\frac{\ell_{\sigma(i)} - 1}{\ell_{\sigma(i)}}$, since for each i we have $E[\ell_i, \pi \pm 1] \cong \mathbb{Z}/\ell_i \mathbb{Z}$. For large $\ell_{\sigma(i)}$ the success probability is relatively high, and so we expect most of the isogenies will be computed during the $\max_j \{b_j\}$ rounds. Though we can in principle compute the expected cost of each additional round for a given bound vector \mathbf{b} , this cost is not a convex function of \mathbf{b} , and its inclusion in the mathematical program would make it difficult to solve. Instead, acknowledging that few isogenies need to be computed, and that these isogenies will likely correspond to small primes for which isogeny evaluations are cheap, we approximate the expected cost of an additional round by $\mathbf{1}^T \boldsymbol{\mu}$. Despite being inexact, this approximation works well enough in practice to yield a runtime improvement.

It remains to determine the expected number of required additional rounds. The expected total number of rounds required to complete the required $\ell_{\sigma(i)}$

isogenies is $\frac{\ell_{\sigma(i)}}{\ell_{\sigma(i)}-1} b_{\sigma(i)}$, and $b_{\sigma(i)}$ rounds which include the prime $\ell_{\sigma(i)}$ are completed. Thus the number of additional rounds required for $\ell_{\sigma(i)}$ is expected to be $\frac{b_{\sigma(i)}}{\ell_{\sigma(i)}-1}$. The maximum of this quantity over all i is then the number of additional rounds expected to be required to finish the algorithm.

From the above, given a pair (H, V) of strategy matrices and a permutation matrix Σ , we use the following program to estimate the optimal bound vector when using SIMBA with only one torsion point:

$$\begin{aligned} \text{Minimize } & \langle \mathbf{c} - \mathbf{d}, \mathbf{b} \rangle + \max_j \{b_j\} \mathbf{1}^T \mathbf{d} + \max_j \left\{ \frac{b_j}{\ell_j - 1} \right\} \mathbf{1}^T \boldsymbol{\mu} \\ \text{Subject to } & \sum_{i=1}^n \log_2(b_i + 1) \geq \lambda \\ & \mathbf{b} \geq 0 \\ & \mathbf{b} \in \mathbb{Z}^n \end{aligned} \quad (8)$$

Problem (8) is a convex mixed-integer nonlinear program (convex MINLP) which, for small enough instances, can be solved exactly. We solve Problem (8) for the CSIDH-512 parameter set and our optimal (Permutation, Strategy) pair using Couenne [1] running on the NEOS server [7,9,10].

For larger parameter sets, it may not be feasible to solve the MINLP exactly. To approximate the solution in this regime, we propose the following scheme method. Begin by relaxing to a continuous convex program by removing the constraint $\mathbf{b} \in \mathbb{Z}^n$ and solving. Let (CP_0) denote the relaxed problem and $\hat{\mathbf{b}}^{(0)}$ its solution. Construct a new program (CP_1) by adding the constraint $b_{i_0} = \lceil \hat{b}_{i_0}^{(0)} \rceil$, where i_0 is the index of the entry of $\hat{\mathbf{b}}^{(0)}$ which is closest to integer. Then for $1 \leq k \leq n-1$, we repeat this process: solve (CP_k) and fix the entry of \mathbf{b} which is nearest to an integer in $\hat{\mathbf{b}}^{(k)}$. In (CP_n) , all but one variable is fixed; solve the problem and round the only unfixed variable up to ensure sufficient security.

In our numerical experiments, this approximate bound vector performs very well, with average running time within 0.3% of the exactly optimal bound vector.

When using two torsion points in each strategy, the process is essentially the same, except that the coefficient vectors change slightly (because we sometimes have to perform two computations—one for each torsion point—rather than one) and that the mathematical program uses a different bound to ensure security. This is explained precisely in Appendix E.2.

3.4 The Complete Optimization Methodology

So far, we have defined the optimization methodology only piecewise; here we present the full optimization ‘pipeline,’ starting from a measure $M = (\{\ell_i\}_{i=1}^n, f, g)$ and ending with a complete parameter set: a bound vector, and a collection of SIMBA strategies and permutations to use for each round. We present the routine we used for plain SIMBA (*à la* [13]) here; details of the method used for the two-point technique (with SIMBA) appear in Appendix E.3.

1. We first search for a SIMBA strategy $S = (S_1, S_2, \dots, S_m)$ and corresponding permutation Σ . In particular, we apply Algorithm 2 on measure $M =$

$(\{\ell_i\}_{i=1}^n, f, g)$. We chose $T = 1000, m_{\min} = 1, m_{\max} = 5$. In initial searches, we did not bound the sizes of the SIMBA substrategies; going forward, we chose to bound the size of each SIMBA substrategy by

$$\max \left\{ 2, \left\lfloor \frac{n}{m+2} \right\rfloor \right\} \leq |S_j| \leq \left\lceil \frac{n}{m} \right\rceil + 15 \quad \forall 1 \leq j \leq m.$$

(where m is the number of SIMBA substrategies), because initial searches suggested that this range was most promising. This S will be the SIMBA strategy that is used in the first round of computing the class group action.

2. Using the strategy and permutation obtained in step 1., we approximately solve the program (8) using the iterative rounding technique described in Section 3.3 to obtain a bound vector \mathbf{b} .
3. For $2 \leq k \leq \max_j \{b_j\}$, let $M_k^{(\mathbf{b})} = (\{\ell_i\}_{i: b_i \geq k}, f, g)$. To obtain a permutation and SIMBA strategy for the k^{th} round, we run Algorithm 2 on the measure $M_k^{(\mathbf{b})}$. We used $T = 100, m_{\min} = 1, m_{\max} = 5$. As in Step 1., for each number m of substrategies, we bound the size of each SIMBA substrategy by

$$\max \left\{ 2, \left\lfloor \frac{n}{m+2} \right\rfloor \right\} \leq |S_j| \leq \left\lceil \frac{n}{m} \right\rceil + 15 \quad \forall 1 \leq j \leq m.$$

Algorithm 2: Our stochastic search algorithm for an optimal strategy and permutation.

Input : A measure M of size n . Natural numbers T, m_{\min}, m_{\max} . An initial permutation σ^* .

Output: A permutation σ and SIMBA strategy S

- 1 Choose $m^* \leftarrow \{m_{\min}, m_{\min} + 1, \dots, m_{\max}\}$ uniformly at random
- 2 Choose $P^* = (n_1, n_2, \dots, n_{m^*})$, a partition of n , uniformly at random
- 3 Set $S^* = (S_1^*, S_2^*, \dots, S_{m^*}^*)$ to be the optimal SIMBA strategy with SIMBA substrategies of size (n_1, n_2, \dots, n_m) for the measure $\sigma^* M$
- 4 Set $C^* = (S^*)_{\sigma^* M}$
- 5 **for** i from 1 to T **do**
- 6 Set $(\sigma, C) \leftarrow (\sigma^*, C^*)$
- 7 Choose $m \leftarrow \{m_{\min}, m_{\min} + 1, \dots, m_{\max}\}$ uniformly at random
- 8 **do**
- 9 Set $C' \leftarrow C$
- 10 Choose $P = (n_1, n_2, \dots, n_m)$, a partition of n , uniformly at random
- 11 Set $S = (S_1, S_2, \dots, S_m)$ to be the optimal SIMBA strategy with SIMBA substrategies of size (n_1, n_2, \dots, n_m) for the measure σM
- 12 Set σ to be the optimal permutation for S and M
- 13 Set $C \leftarrow (S)_{\sigma M}$
- 14 **while** $C < C'$
- 15 **if** $C < C^*$ **then**
- 16 Set $(\sigma^*, m^*, P^*, S^*, C^*) \leftarrow (\sigma, m, P, S, C)$
- 17 **end**
- 18 **end**
- 19 **Return** (P^*, σ^*, S^*)

Operation	\mathbf{M}	\mathbf{S}	\mathbf{a}	
			Montgomery	Edwards
LADDER	$8t - 4$	$4t - 2$	$8t - 6$	$8t - 6$
EVAL	$2\ell - 2$	2	$\ell + 1$	$\ell + 3$
KER	$2\ell - 6$	$\ell - 3$	$4\ell - 12$	$3\ell - 11$
CODOM	$\ell + 2t^* - 1$	$2t + 6$	6	2

Table 1. Costs for various operations. \mathbf{M} , \mathbf{S} , and \mathbf{a} respectively represent multiplications, squarings, and additions in \mathbb{F}_p . Here ℓ is an odd prime, $t = \lceil \log_2(\ell) \rceil$, and t^* is the Hamming weight of ℓ . For the purposes of the model, we estimate $t^* \approx \frac{1}{2} \lceil \log_2 \ell \rceil$.

4 Implementation

In terms of formulating a cost model, there are essentially two scenarios: using Montgomery curves with the formulas of [14], or using twisted Edwards curves with the formulas of [3]. The costs for various operations are summarized in Table 1. We use \mathbf{M} to denote \mathbb{F}_p multiplications, \mathbf{S} to denote \mathbb{F}_p squarings, and \mathbf{a} to denote \mathbb{F}_p additions/subtractions. In the table, ℓ is interpreted as an odd prime. LADDER refers to computing $[\ell]P$ for a given point P using the Montgomery ladder. The operation KER denotes the cost of computing the kernel points $P, [2]P, \dots, [\frac{\ell-1}{2}]P$ of an isogeny φ from a given generator P of order ℓ . In the Montgomery setting, the KER table entry includes the cost of the computing the points $[i]P$, as well as the $\ell - 1$ additions required for computing the sums and differences of these coordinates described in Algorithm 4 of [5]. CODOM considers constructing the codomain of a degree ℓ isogeny φ given its kernel points $\langle P \rangle$. EVAL computes $\varphi(Q)$ for a given point Q , assuming the kernel points are already computed. We point out that each operation requires the same number of multiplications and squarings independent of the setting (e.g., Montgomery or Edwards), but the number of additions and subtractions vary.

In the context of a measure $M = (\{\ell_i\}, f, g)$ on a strategy for CSIDH, $f(\ell_i)$ represents the cost of performing the operation $(\ell_i, P) \mapsto [\ell_i]P$, while $g(\ell_i)$ represents the cost to evaluate an isogeny of degree ℓ_i at some point (assuming the kernel points have been computed already). In practice, we therefore take f as the sum over the LADDER row of Table 1 and g as the sum over the EVAL row, including only one of the ‘Montgomery’ or ‘Edwards’ columns according to the appropriate context. We set $\mathbf{S} = 0.8\mathbf{M}$ and $\mathbf{a} = 0\mathbf{M}$.

Implementation Details. We applied our results to two settings:

1. The work of Meyer, Campos, and Reith in [13] (based on previous work of Meyer and Reith in [14]). Here, Montgomery curves are used with points represented in XZ -coordinates. To compute the codomain curve of an isogeny, a conversion to a Twisted-Edwards model is used. This method uses non-negative private key values, and so only one point is traced through a strategy at a time. We refer to this as the ‘MCR method’.
2. The work of Cervantes-Vázquez, Chenu, Chi-Domínguez, De Feo, Rodríguez-Henríquez, and Smith in [3]. Here, twisted Edwards curves are used exclu-

sively with points represented using YZ -coordinates. The authors apply formulas for the Edwards setting to both the MCR method and the two-point technique of [16], along with a projectivized Elligator map and optimized addition chains for scalar multiplication. We call this the “CCCDRS” method.

In each setting we used the optimization techniques of Section 3.4 to find full CSIDH parameter sets at the 128-bit security level, where we take the primes ℓ_i suggested by [2] for CSIDH-512. It should be pointed out that Peikert in [17] suggests that the parameters given by [2] for CSIDH-512 may not actually provide 128 bits of security, but we consider this parameter set in order to directly compare with previous optimizations of CSIDH; all of the results in this work are compatible with any collection of distinct odd primes used for CSIDH. We implemented Algorithm 2 in a combination of Octave and Matlab in order to construct SIMBA strategies, permutations, and bound vectors for each of the implementations described below.

Table 2 summarizes our results for each of the implementations we consider. The values of the table reflect the median over 1024 iterations of a single group action evaluation, including validation of supersingularity of the output curve. All of the tests were executed on a i7-7500k clocked at 2.70 GHz running on a single core only. All tests were performed using optimized field arithmetic.

The first row of Table 2 gives the original implementation of CSIDH [2]. This implementation is *not* constant-time and is included only for reference.

For the MCR method we used the publicly-available code of [13], modified to fit our optimized parameter set (which includes an optimized SIMBA strategy and corresponding permutation for each round, and a bound vector). Our implementation of the MCR method used a custom Sage script which takes a strategy and permutation as input and outputs C code which efficiently executes them—in particular, merging consecutive point multiplications (horizontal paths in the strategy for which no internal leaf in the path is a branch). This Sage script allowed us to test a wide variety of strategies without having to write custom code for each one. The implementation did not use the optimizations suggested by [3]. Compared with [13], our results yielded a 12.77% speedup.

For the CCCDRS implementations we only considered the two-point version, and we did not find any SIMBA substrategies that outperform the multiplication-based strategy. Consequently our C code generation script for this implementation only produces code for custom SIMBA substrategy *sizes*, permutations, and bound vectors. We used an Octave script to produce C header files that can be used as drop-in replacements for corresponding header files in the implementation of [3] to implement our custom parameters.

To demonstrate how optimizing each parameter using our techniques affects the efficiency of the implementations, we provide benchmarks for three CCCDRS method implementations. The first we denote as CCCDRS-1, in which we use the bound vector of [3] and a single SIMBA strategy S and corresponding permutations for the full measure $M = \{\ell_i\}$ found using Algorithm 2; here, the same strategy S is used in each round. For CCCDRS-1, we achieve a speedup of only 1.09% over the original implementation of [3]. Our second implementation

is denoted CCCDRS-2, in which we modify CCCDRS-1 to use optimized permutations and a SIMBA strategy on the submeasure $M_i^{(b)}$ in the i^{th} round, for $1 \leq i \leq 7 = \max_j \{b_j\}$, rather than a SIMBA strategy and permutation on the full measure M . For CCCDRS-2 we attained a speedup of 3.77% over [3]. Finally we have CCCDRS-3, where we use a bound vector obtained by the technique of Section 3.3 on top of the optimizations of CCCDRS-2. CCCDRS-3 applies all of the optimizations of Section 3, and with it we achieved a speedup of 5.06%. All of our code and the final parameter sets used for these tests can be found here:

<https://github.com/AaronHutchinson/CSIDH>

Implementation	M	S	a	Latency (Mcycles)	Speedup (%)
CSIDH [2]	463 287	136 654	416 891	165.1	-
MCR [13]	1 036 675	425 377	1 020 712	407.87	-
This work (MCR)	905 200	312 483	859 759	355.77	12.77
CCCDRS [3] (Two pt.)	664 936	224 081	750 992	251.21	-
This work (CCCDRS-1)	659 816	223 793	745 710	248.47	1.09
This work (CCCDRS-2)	637 352	218 635	724 958	241.74	3.77
This work (CCCDRS-3)	632 444	209 310	704 576	238.51	5.06

Table 2. Field operation counts and latency for seven implementations of CSIDH-512.

5 Conclusions

We developed systematic techniques for optimizing three parameters used in the CSIDH group action evaluation algorithm: the strategy, permutation of the primes ℓ_i , and bound vector from which private key values are chosen. Prior works in this area have used *ad hoc* methods to determine these parameters, and as far as we are aware this work is the first step in the direction of determining an optimal parameter set. Our implementation results show that significant speedups can be achieved when using our techniques to find efficient parameter sets. In light of recent cryptanalysis (in particular, [17]), new CSIDH parameter sets will have to be derived to meet NIST security levels. The optimization methods presented here can be used to contribute to these parameter sets (in the form of the bound vector) and to efficient class group action evaluation algorithms.

Acknowledgements The authors would like to thank the reviewers for their helpful comments. This work is supported in parts by NSF CNS-1801341, NSF GRFP-1939266, and NIST-60NANB17D184.

References

1. P. Belotti, J. Lee, L. Liberti, F. Margot, and A. Wächter. Branching and bounds tightening techniques for non-convex MINLP. *Optimization Methods and Software*, 24(4-5):597–634, 2009.
2. Wouter Castryck, Tanja Lange, Chloe Martindale, Lorenz Panny, and Joost Renes. CSIDH: An Efficient Post-Quantum Commutative Group Action. In Thomas Peyrin and Steven Galbraith, editors, *Advances in Cryptology – ASIACRYPT 2018*, pages 395–427, Cham, 2018. Springer International Publishing.

3. Daniel Cervantes-Vázquez, Mathilde Chenu, Jesús-Javier Chi-Domínguez, Luca De Feo, Francisco Rodríguez-Henríquez, and Benjamin Smith. Stronger and Faster Side-Channel Protections for CSIDH. In Peter Schwabe and Nicolas Thériault, editors, *Progress in Cryptology – LATINCRYPT 2019*, pages 173–193, Cham, 2019. Springer International Publishing.
4. H. Cohen and H. W. Lenstra. Heuristics on Class Groups of Number Fields. In Hendrik Jager, editor, *Number Theory Noordwijkerhout 1983*, pages 33–62, Berlin, Heidelberg, 1984. Springer Berlin Heidelberg.
5. Craig Costello and Huseyin Hisil. A Simple and Compact Algorithm for SIDH with Arbitrary Degree Isogenies. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology – ASIACRYPT 2017*, pages 303–329, Cham, 2017. Springer International Publishing.
6. Jean-Marc Couveignes. Hard Homogeneous Spaces. Cryptology ePrint Archive, Report 2006/291, 2006. <https://eprint.iacr.org/2006/291>.
7. Joseph Czyzyk, Michael P. Mesnier, and Jorge J. Moré. The neos server. *IEEE Journal on Computational Science and Engineering*, 5(3):68 – 75, 1998.
8. Luca De Feo, David Jao, and Jérôme Plût. Towards Quantum-Resistant Cryptosystems from Supersingular Elliptic Curve Isogenies. *J. Mathematical Cryptology*, 8(3):209–247, 2014.
9. Elizabeth D. Dolan. The neos server 4.0 administrative guide. Technical Memorandum ANL/MCS-TM-250, Mathematics and Computer Science Division, Argonne National Laboratory, 2001.
10. William Gropp and Jorge J. Moré. Optimization environments and the neos server. In Martin D. Buhman and Arieh Iserles, editors, *Approximation Theory and Optimization*, pages 167 – 182. Cambridge University Press, 1997.
11. Aaron Hutchinson and Koray Karabina. Constructing Canonical Strategies for Parallel Implementation of Isogeny Based Cryptography. In *19th International Conference on Cryptology in India, New Delhi, India, December 9–12, 2018, Proceedings*, pages 169–189. Springer, 12 2018.
12. Amir Jalali, Reza Azarderakhsh, Mehran Mozaffari Kermani, and David Jao. Towards Optimized and Constant-Time CSIDH on Embedded Devices. Cryptology ePrint Archive, Report 2019/297, 2019. <https://eprint.iacr.org/2019/297>.
13. Michael Meyer, Fabio Campos, and Steffen Reith. On Lions and Elligators: An Efficient Constant-Time Implementation of CSIDH. In Jintai Ding and Rainer Steinwandt, editors, *Post-Quantum Cryptography*, pages 307–325, Cham, 2019. Springer International Publishing.
14. Michael Meyer and Steffen Reith. A Faster Way to the CSIDH. In Debrup Chakraborty and Tetsu Iwata, editors, *Progress in Cryptology – INDOCRYPT 2018*, pages 137–152, Cham, 2018. Springer International Publishing.
15. Tomoki Moriya, Hiroshi Onuki, and Tsuyoshi Takagi. How to Construct CSIDH on Edwards Curves. Cryptology ePrint Archive, Report 2019/843, 2019. <https://eprint.iacr.org/2019/843>.
16. Hiroshi Onuki, Yusuke Aikawa, Tsutomu Yamazaki, and Tsuyoshi Takagi. (Short Paper) A Faster Constant-Time Algorithm of CSIDH Keeping Two Points. In Nuttapong Attrapadung and Takeshi Yagi, editors, *Advances in Information and Computer Security*, pages 23–33, Cham, 2019. Springer International Publishing.
17. Chris Peikert. He Gives C-Sieves on the CSIDH. Cryptology ePrint Archive, Report 2019/725, 2019. <https://eprint.iacr.org/2019/725>.
18. Alexander Rostovtsev and Anton Stolbunov. Public-Key Cryptosystem Based on Isogenies. Cryptology ePrint Archive, Report 2006/145, 2006. <https://eprint.iacr.org/2006/145>.

A Recursive Encoding of Strategies

When restricting to canonical strategies, the matrices H and V from Section 2.1 can alternatively be defined recursively. We define $H(T_1 \# T_1) = V(T_1 \# T_1) = [1]$ as an initial value, and compute the matrices of larger strategies as shown in Figure 1. These recursive definitions follow immediately from the definition of the join operator $\#$ given in Section 2.1. Note that for canonical strategies either one of the matrices H or V uniquely determines the other, and so S can be specified by giving only one of them. For computational purposes having both matrices is convenient, however.

$$\begin{aligned}
 H(S^L \# T_1) &= \begin{bmatrix} 0 & \mathbf{0} \\ \mathbf{e}_{n_1} & H(S^L) \end{bmatrix} & V(S^L \# T_1) &= \begin{bmatrix} 1 & \mathbf{0} \\ \mathbf{1} & V(S^L) \end{bmatrix} \\
 H(T_1 \# S^R) &= \begin{bmatrix} H(S^R) & \mathbf{0} \\ \mathbf{1} & 1 \end{bmatrix} & V(T_1 \# S^R) &= \begin{bmatrix} V(S^R) & \mathbf{0} \\ \mathbf{e}_1^T & 0 \end{bmatrix} \\
 H(S^L \# S^R) &= \begin{bmatrix} H(S^R) & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & 0 & \mathbf{0} \\ \mathbf{e}_{n_1-1} \mathbf{1}^T & \mathbf{e}_{n_1-1} & H(S^L) \end{bmatrix} & V(S^L \# S^R) &= \begin{bmatrix} V(S^R) & \mathbf{0} & \mathbf{0} \\ \mathbf{e}_1^T & 0 & \mathbf{0} \\ \mathbf{1} \mathbf{e}_1^T & \mathbf{0} & V(S^L) \end{bmatrix}
 \end{aligned}$$

Fig. 1. Recursive expressions for the encoding matrices H and V . Both S^L and S^R are canonical strategies with $|S^L| = n_1$ and $|S^R| = n_2$, with $n_1, n_2 > 1$. Every $H(S)$ and $V(S)$ matrix is square of dimension $|S| - 1$. We use $\mathbf{0}$ and $\mathbf{1}$ to represent (often nonsquare) matrices of the appropriate size with all entries 0 and 1, respectively; 0 and 1 represent individual matrix entries. As well, \mathbf{e}_i is the unit basis column vector of the appropriate size with a 1 in the i^{th} position and 0's elsewhere. We let A^T denote the transpose of a matrix A .

Encoding SIMBA Strategies. Just as for full strategies, it is desirable to encode SIMBA strategies as a pair of matrices (H, V) . If $S = (S_1, S_2)$ is a SIMBA strategy on two SIMBA substrategies, we define

$$H(S) = \begin{bmatrix} H(S_1) & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & 0 & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & H(S_2) \end{bmatrix} \quad \text{and} \quad V(S) = \begin{bmatrix} V(S_1) & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & 0 & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & V(S_2) \end{bmatrix}$$

where $H(S_i)$ and $V(S_i)$ are the encoding matrices of the S_i as defined in Section 2.1. Then, for a SIMBA strategy $S = (S_1, S_2, \dots, S_m)$ on $m \geq 3$ substrategies, we define

$$H(S) = \begin{bmatrix} H(S') & \mathbf{0} & \mathbf{0} \\ \hline \mathbf{0} & 0 & \mathbf{0} \\ \hline \mathbf{0} & \mathbf{0} & H(S_m) \end{bmatrix} \quad \text{and} \quad V(S) = \begin{bmatrix} V(S') & \mathbf{0} & \mathbf{0} \\ \hline \mathbf{0} & 0 & \mathbf{0} \\ \hline \mathbf{0} & \mathbf{0} & V(S_m) \end{bmatrix}$$

where $S' = (S_1, S_2, \dots, S_{m-1})$. This definition is used to make the optimization problems in Sections 3.2 and 3.3 compatible with SIMBA strategies.

B Proof of Theorem 1

Here we give a proof of Theorem 1, which we restate here for convenience.

Theorem 1. *Fix a measure $M = \{p_i\}_{i=1}^n$. Suppose S is a canonical strategy for which $(S)_M$ is minimal over all canonical strategies for M . If $k = |S^L|$, then S^L and S^R are canonical strategies for which $(S^L)_{M^L}$ and $(S^R)_{M^R}$ are minimal over all canonical strategies for M^L and M^R , respectively, where $M^L := \{p_i\}_{i=n-k+1}^n$ and $M^R := \{p_i\}_{i=1}^{n-k}$.*

Proof. If S is fixed, we first notice that the cost that S^L contributes to $(S)_M$ depends only on the last k entries of M . This can be seen through Equation (3) with σ taken as the identity. Alternatively, by the construction of $S = S^L \# S^R$ given in Section 2.1, S^L is contained within the lines $x = n - k, y = 0$, and $y = -x + n - 1$, and by the weight assignment given in Definition 2 each horizontal (resp. vertical) edge in this region is assigned some weight from the set $\{f(p_{n-k+1}), \dots, f(p_n)\}$ (resp. the set $\{g(p_{n-k+1}), \dots, g(p_n)\}$). By a similar argument, the cost that S^R contributes to $(S)_M$ depends only on the first $n - k$ entries of M . We can therefore view S^L as a strategy in T_k under the measure M^L , and S^R as a strategy in T_{n-k} under the measure M^R .

Let a be the (unique) path in T_n connecting the root $(0, 0)$ to the vertex $(n-k, 0)$ (seen as the root of S^L), and let b be the (unique) path in T_n connecting the root $(0, 0)$ to the vertex $(0, k)$ (seen as the root of S^R). Then the strategy S decomposes as a disjoint union $S = S^L \cup S^R \cup a \cup b$, and by the preceding paragraph the cost of S under M can therefore be written as

$$(S)_M = (S^L)_{M^L} + (S^R)_{M^R} + \sum_{i=1}^{n-k} f(p_i) + \sum_{i=n-k+1}^n g(p_i),$$

where the first and second summations represent the cost of the paths a and b , respectively, under M . We notice that these two summations depend only on M and k , and not on the substrategies S^L and S^R themselves. If either S^L or S^R is suboptimal under M^L or M^R , respectively, then we may get a lower cost strategy S by replacing the suboptimal strategy with an optimal one under its corresponding measure. This concludes the proof.

C Complete Description of the Group Action Algorithm

We give a general high level algorithm for evaluating the group action in CSIDH as follows. Let $\mathbf{b} = (b_1, \dots, b_n)$ be the bound vector used so that private key values e_i are chosen from the interval $[-b_i, b_i] \cap \mathbb{Z}$ (or $[0, b_i]$ in the case of a non-negative approach such as [13]), and let $r = \max_i b_i$. The algorithm runs through r many rounds, with the i -th round considering an active subset $M_i \subset \{\ell_1, \dots, \ell_n\}$ of primes. A SIMBA strategy $S_i = \{(S_{i,1}, M_{i,1}), \dots, (S_{i,m_i}, M_{i,m_i})\}$ is chosen (in advance) for each M_i , as well as a permutation $\sigma_{i,j}$ for each $M_{i,j}$. The i -th round iterates from $j = 1$ to m_i , evaluating each of the SIMBA sub-strategies $S_{i,j}$ under its corresponding permuted measure $\sigma_{i,j} M_{i,j}$ one time. Once all r rounds are complete a final phase is executed, in which any isogenies which remain to be constructed due to a failure (either real or dummy) are built by means of a multiplication-based strategy with a descending-prime permutation.

The subsets M_i are defined to consist of all primes ℓ_j for which $b_j - i$ is positive. This choice of M_i eliminates a great deal of redundancy since a prime ℓ_j is eliminated from all future strategies (save for the final phase) exactly upon finishing the b_j -th round. Using the results of Section 3 one can find well-performing parameter values for \mathbf{b} , the SIMBA strategy S , and the permutations $\sigma_{i,j}$. The final phase of the algorithm after the r rounds finish is entirely dependent upon the points randomly generated during execution, and so the strategy used for it cannot be optimized since it's not known in advance which primes will be present during this step and determining a strategy during run time is infeasible.

The group action evaluation algorithm is detailed in Algorithm 3, which calls Algorithms 4 and 5 as subroutines. The algorithms as written use the two point method of [16], but can be easily modified to use the non-negative key method of [14] by ignoring steps which involve the variables P_{-1} and $NegPts$. In this case some steps may be simplified, such as collapsing the loops (5) and (11) in Algorithm 4 down into one. As noted before, the conditional branching based on the private key values e_i seen in loops (8) in Algorithm 3 and (4) of Algorithm 5 can be performed securely by using constant time conditional swaps to dummy points. The branching on lines (15), (20) in Algorithm 4 and line (14) in Algorithm 5 can be handled similarly.

D Example—Evaluation of a Canonical Strategy

In Figure 2 we give a small example of a canonical strategy S in T_9 . Suppose that the primes involved in evaluating S are $L = \{p_1, p_2, \dots, p_9\}$, and are ordered as such after being permuted. In preparation to evaluate the strategy Algorithm 3 would choose random points $P_1 \in E_A[\pi - 1]$ and $P_{-1} \in E_A[\pi + 1]$ using Elligator, and these points would each be multiplied by all primes in the set $\{\ell_1, \dots, \ell_n\} \setminus \{p_1, \dots, p_9\}$ and by 4. The order of each of these points will then divide $p_1 p_2 \cdots p_9$.

Suppose that the private key values e_i corresponding to the primes p_i are

$$(e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9) = (1, -1, 0, 2, 0, -2, 0, 0, 1).$$

Algorithm 3: Our ideal class Group Action Algorithm

Parameters: $b = (b_1, \dots, b_n) \in \mathbb{Z}^n$. For $1 \leq i \leq \max_i b_i$, there are $m_i \in \mathbb{N}$ determining the number of SIMBA substrategies in the i th round. For each i and $1 \leq j \leq m_i$, there are strategies $S_{i,j}$ and permutations $\sigma_{i,j}$ on some set of primes $L_{i,j} \subset \{\ell_1, \dots, \ell_n\}$.

Input : $A \in \mathbb{F}_p$ and private key (e_1, \dots, e_n) , with $e_i \in [-b_i, b_i] \cap \mathbb{Z}$.

Output: $A' \in \mathbb{F}_p$ such that $[\ell_1^{e_1} \cdots \ell_n^{e_n}]E_A = E_{A'}$.

```

1  $t \leftarrow 0$ .
2  $c \leftarrow b$ .
3 for  $i = 1$  to  $\max b_i$  do
4   for  $j = 1$  to  $m_i$  do
5     Choose points  $P_1 \in E_A[\pi - 1]$  and  $P_{-1} \in E_A[\pi + 1]$  using Elligator.
6     Compute  $u \leftarrow 4 \cdot \prod_{\ell_k \notin L_{i,j}} \ell_k$ .
7     Update  $P_1 \leftarrow [u]P_1$  and  $P_{-1} \leftarrow [u]P_{-1}$ .
8     for  $\ell_k \in L_{i,j}$  do
9       If  $e_k = 0$ , then update  $P_1 \leftarrow [\ell_k]P_1$  and  $P_{-1} \leftarrow [\ell_k]P_{-1}$ .
10      If  $e_k \neq 0$ , then update  $P_{-sign(e_k)} \leftarrow [\ell_k]P_{-sign(e_k)}$ .
11    end
12     $A, e, c, t \leftarrow \text{EVAL}(S_{i,j}, \sigma_{i,j}, L_{i,j}; A, P_1, P_{-1}, e, c, t)$ .
13  end
14 end
15  $A \leftarrow \text{BUILDREMAINING}(A, e, c, t)$ 
16 Return  $A$ 

```

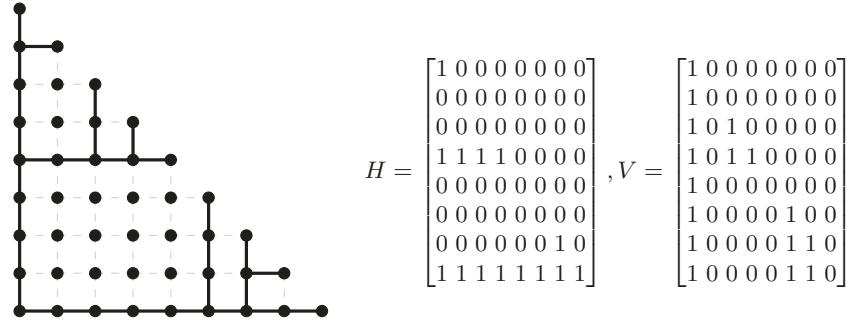


Fig. 2. A canonical strategy S (black lines) in T_9 (black and dashed grey lines), and the corresponding matrices $H(S)$ and $V(S)$. This strategy decomposes into copies of T_1 as $S = ((T_1 \# (T_1 \# T_1)) \# T_1) \# (((T_1 \# T_1) \# T_1) \# (T_1 \# T_1))$.

The innermost loop of Algorithm 3 would then multiply P_1 by $p_2p_3p_5p_6p_7p_8$ (so that its order divides $p_1p_4p_9$) and P_{-1} by $p_1p_3p_4p_5p_7p_8p_9$ (so that its order divides p_2p_6). We are then ready to begin evaluation of the strategy. For clarity, we will let Q_1 and Q_{-1} permanently denote these initial values of P_1 and P_{-1} , respectively, whereas the variables P_1 and P_{-1} will have their values updated as we progress through Algorithm 4.

Algorithm 4: EVAL: Strategy Evaluation

Parameters: A permutation σ represented as a subsequence of $\{1, 2, \dots, n\}$ of length m , a strategy S with m leaves represented by its vertical encoding matrix $V(S)$, defined on the primes $\{\ell_{\sigma(1)}, \dots, \ell_{\sigma(m)}\}$.

Input : $A \in \mathbb{F}_p$, representing a curve $E_A : y^2 = x^3 + Ax^2 + x$, points $P_1 \in E_A[\pi - 1]$ and $P_{-1} \in E_A[\pi + 1]$, current value e of private key, vector $c = (c_1, \dots, c_n)$ such that c_i many ℓ_i degree isogenies remain to be constructed, integer t denoting total number of isogenies already constructed.

Output: Updated values for A, e, c, t according to the isogenies constructed.

```

1 Initialize point arrays  $PosPts$  and  $NegPts$ , with  $PosPts[0] \leftarrow P_1$  and  $NegPts[0] \leftarrow P_{-1}$ .
2 for  $i = m - 1$  down to 1 do
3   Update  $P_1$  and  $P_{-1}$  to the last entries of  $PosPts$  and  $NegPts$ , respectively, and define  $k$  to be the index at which these entries appear.
4    $k' \leftarrow \max\{x : V(S)_{i,x} = 1\}$ .
5   for  $j = k + 1$  to  $k' - 1$  do
6     Update  $P_1 \leftarrow [\ell_{\sigma(j)}] P_1$  and  $P_{-1} \leftarrow [\ell_{\sigma(j)}] P_{-1}$ 
7     if  $V(S)_{i,j} = 1$  then
8       | Append  $P_1$  to  $PosPts$ , append  $P_{-1}$  to  $NegPts$ 
9     end
10   end
11   for  $j = k'$  to  $i$  do
12     | Update  $P_{sign(e_{\sigma(i)})} \leftarrow [\ell_{\sigma(i)}] P_{sign(e_{\sigma(j)})}$ .
13   end
14   if  $k = i$ , then delete the last entries of  $PosPts$  and  $NegPts$ .
15   if  $e_{\sigma(i)} \neq 0$  and  $P_{sign(e_{\sigma(i)})} \neq \infty$  then
16     | Construct isogeny  $\phi : E_A \rightarrow E_B$  of degree  $\ell_{\sigma(i)}$  with kernel  $\langle P_{sign(e_{\sigma(i)})} \rangle$ .
17     | Update all points in  $PosPts$  and  $NegPts$  with their images under  $\phi$ .
18     |  $A \leftarrow B$ ,  $c_{\sigma(i)} \leftarrow c_{\sigma(i)} - 1$ ,  $e_{\sigma(i)} \leftarrow e_{\sigma(i)} - sign(e_{\sigma(i)})$ ,  $t \leftarrow t + 1$ .
19   end
20   if  $e_{\sigma(i)} = 0$  then
21     | Construct dummy isogeny  $\phi : E_A \rightarrow E_B$  of degree  $\ell_{\sigma(i)}$  with kernel  $\langle P_{sign(e_{\sigma(i)})} \rangle$ .
22     |  $c_{\sigma(i)} \leftarrow c_{\sigma(i)} - 1$ ,  $t \leftarrow t + 1$ .
23   end
24 end
25 Return  $A, e, c, t$ 

```

The arrays in Algorithm 4 get initialized as $PosPts = [Q_1]$ and $NegPts = [Q_{-1}]$. The first branch node appears after 5 horizontal edges, and so both points are multiplied by $p_1 p_2 p_3 p_4 p_5$ and the results are stored into the arrays:

$$\begin{aligned}
PosPts &= [Q_1, [p_1 p_2 p_3 p_4 p_5] Q_1] \\
NegPts &= [Q_{-1}, [p_1 p_2 p_3 p_4 p_5] Q_{-1}].
\end{aligned}$$

Algorithm 5: BUILDREMAINING: Construct any remaining isogenies.

Input : $A \in \mathbb{F}_p$, current value e of private key, vector $c = (c_1, \dots, c_n)$ such that c_i many ℓ_i degree isogenies remain to be constructed, integer t denoting total number of isogenies already constructed

```

1 while  $t \neq \sum_{i=1}^n b_i$  do
2   Choose random points  $P_1 \in E_A[\pi - 1]$  and  $P_{-1} \in E_A[\pi + 1]$  using Elligator.
3   Update  $P_1 \leftarrow [4]P_1$  and  $P_{-1} \leftarrow [4]P_{-1}$ .
4   for  $k = 1$  to  $n$  do
5     if  $e_k = 0$ , then update  $P_1 \leftarrow [\ell_k]P_1$  and  $P_{-1} \leftarrow [\ell_k]P_{-1}$ .
6     if  $e_k \neq 0$ , then update  $P_{-sign(e_k)} \leftarrow [\ell_k]P_{-sign(e_k)}$ .
7   end
8   for  $k = n$  down to 1 do
9     if  $c_k \neq 0$  then
10    Assign  $Q \leftarrow P_{sign(e_k)}$ .
11    for  $i = 1$  to  $k - 1$  do
12      if  $c_i \neq 0$ , then update  $Q \leftarrow [\ell_i]Q$ .
13    end
14    if  $e_{\sigma(i)} \neq 0$  and  $P_{sign(e_k)} \neq \infty$  then
15      Construct isogeny  $\phi : E_A \rightarrow E_B$  of degree  $\ell_{\sigma(i)}$  with kernel  $\langle Q \rangle$ .
16      Update  $P_1 \leftarrow \phi(p_1)$  and  $P_{-1} \leftarrow \phi(P_{-1})$ 
17       $A \leftarrow B$ ,  $c_k \leftarrow c_k - 1$ ,  $e_{\sigma(i)} \leftarrow e_{\sigma(i)} - sign(e_{\sigma(i)})$ ,  $t \leftarrow t + 1$ .
18    end
19  end
20 end
21 end
22 Return  $A$ 

```

The final branch on this row of the strategy occurs after 1 additional horizontal edge, and so p_6 now gets multiplied and the results stored:

$$\begin{aligned} PosPts &= [Q_1, [p_1 p_2 p_3 p_4 p_5]Q_1, [p_1 p_2 p_3 p_4 p_5 p_6]Q_1] \\ NegPts &= [Q_{-1}, [p_1 p_2 p_3 p_4 p_5]Q_{-1}, [p_1 p_2 p_3 p_4 p_5 p_6]Q_{-1}]. \end{aligned}$$

This completes loop (5) of Algorithm 4. We have $e_9 = 1 > 0$, and so P_1 should be used to determine the kernel generator of the first isogeny. Loop (11) multiplies out primes p_7 and p_8 from P_1 so that it has value $[p_1 p_2 p_3 p_4 p_5 p_6 p_7 p_8]Q_1$. Provided that this point is nontrivial, it has order p_9 and will be used to construct the first isogeny φ_9 . All points in the arrays are evaluated under this isogeny (which also removes a factor of p_9 from their order):

$$\begin{aligned} PosPts &= [\varphi_9(Q_1), \varphi_9([p_1 p_2 p_3 p_4 p_5]Q_1), \varphi_9([p_1 p_2 p_3 p_4 p_5 p_6]Q_1)] \\ NegPts &= [\varphi_9(Q_{-1}), \varphi_9([p_1 p_2 p_3 p_4 p_5]Q_{-1}), \varphi_9([p_1 p_2 p_3 p_4 p_5 p_6]Q_{-1})]. \end{aligned}$$

The last points in the above arrays are the images of the points corresponding to the final branch node in row 8, column 6 of the strategy, and we use these

points to begin operations on row 7. There are no branch nodes on row 7 after column 6, and so the loop (5) is skipped. We have $e_8 = 0$, and so either point P_1 or P_{-1} can be multiplied by p_8 and then used to construct a dummy isogeny φ_8 . Dummy isogeny evaluations are performed on the arrays, completing row 7. The (dummy) images of the final array points correspond to the leaf node of row 6. One of the points $\varphi_9([p_1 p_2 p_3 p_4 p_5]Q_{\pm 1})$ would then immediately be used to construct the next isogeny (the loops (5) and (11) are skipped), but since $e_7 = 0$ we perform another dummy construction. Furthermore, since an isogeny evaluation landed us on a leaf node, we have no further use for the final entries of the arrays and they may be safely removed:

$$\begin{aligned} \text{Pos}Pts &= [\varphi_9(Q_1), \varphi_9([p_1 p_2 p_3 p_4 p_5]Q_1)] \\ \text{Neg}Pts &= [\varphi_9(Q_{-1}), \varphi_9([p_1 p_2 p_3 p_4 p_5]Q_{-1})]. \end{aligned}$$

The evaluation of the final points above again land at a leaf node and again loops (5) and (11) can be skipped. This time we have $e_6 = -2$, and so $\varphi_9([p_1 p_2 p_3 p_4 p_5]Q_{-1})$ is used as the kernel for isogeny φ_6 (recall that $p_7 p_8$ was removed from $|Q_{-1}|$ at the start). Again the previous isogeny evaluation arrived at a leaf node, and so the final entries of the arrays are removed:

$$\text{Pos}Pts = [\varphi_6 \varphi_9(Q_1)], \quad \text{Neg}Pts = [\varphi_6 \varphi_9(Q_{-1})].$$

On row 4 we proceed similar to row 8 and arrive at the arrays

$$\begin{aligned} \text{Pos}Pts &= [\varphi_6 \varphi_9(Q_1), \varphi_6 \varphi_9([p_1 p_2]Q_1), \varphi_6 \varphi_9([p_1 p_2 p_3]Q_1)] \\ \text{Neg}Pts &= [\varphi_6 \varphi_9(Q_{-1}), \varphi_6 \varphi_9([p_1 p_2]Q_{-1}), \varphi_6 \varphi_9([p_1 p_2 p_3]Q_{-1})]. \end{aligned}$$

We have $e_5 = 0$ and a dummy isogeny is constructed after one point is multiplied by p_4 . For row 3 we have $e_4 = 2$ and the point $\varphi_6 \varphi_9([p_1 p_2 p_3]Q_1)$ of order dividing p_4 is used for an isogeny φ_4 construction if possible. Since an evaluation landed on a leaf node, we remove the final array entries:

$$\begin{aligned} \text{Pos}Pts &= [\varphi_4 \varphi_6 \varphi_9(Q_1), \varphi_4 \varphi_6 \varphi_9([p_1 p_2]Q_1)] \\ \text{Neg}Pts &= [\varphi_4 \varphi_6 \varphi_9(Q_{-1}), \varphi_4 \varphi_6 \varphi_9([p_1 p_2]Q_{-1})]. \end{aligned}$$

We arrive in row 2 at a leaf node, and since $e_3 = 0$ we do a dummy construction while also removing the final array entries:

$$\text{Pos}Pts = [\varphi_4 \varphi_6 \varphi_9(Q_1)], \quad \text{Neg}Pts = [\varphi_4 \varphi_6 \varphi_9(Q_{-1})].$$

Since $e_2 = -1$, in row 1 we multiply $\varphi_4 \varphi_6 \varphi_9(Q_{-1})$ by p_1 to get a point of order dividing p_2 and construct the isogeny φ_2 . The final values of the arrays are then:

$$\text{Pos}Pts = [\varphi_2 \varphi_4 \varphi_6 \varphi_9(Q_1)], \quad \text{Neg}Pts = [\varphi_2 \varphi_4 \varphi_6 \varphi_9(Q_{-1})].$$

The value $e_1 = 1$ determines the final isogeny φ_1 , constructed using kernel generator $\varphi_2 \varphi_4 \varphi_6 \varphi_9(Q_1)$ of order p_1 .

Along the way, the nonzero key values e_i are decreased in magnitude any time a real isogeny is successfully constructed, and a counter t is incremented any time a real or dummy isogeny is constructed. This completes the execution of Algorithm 4 for this strategy, and Algorithm 3 would continue by executing all remaining strategies, and then running Algorithm 5 to construct any isogenies which failed to be constructed in any execution of Algorithm 4.

E Extensions to SIMBA and the Two-Point Method

E.1 Optimal Permutations for SIMBA and the Two-Point Method

An Extension to SIMBA Strategies. The arguments of Section 3.2 apply in a straightforward fashion to the case of SIMBA strategies; in this scenario, however, we have

$$(S)_{\sigma M} = \langle T_L^T H^T \mathbf{1} \boldsymbol{\mu}^T + T_R^T V \mathbf{1} \boldsymbol{\iota}^T, \Sigma \rangle_F + (m-1) \mathbf{1}^T \boldsymbol{\mu}$$

where m is the number of SIMBA substrategies of S . Notably the additional term $(m-1) \mathbf{1}^T \boldsymbol{\mu}$ is independent of the decision variables Σ , and so we can use Problem (6) without modification when optimizing the permutation for a given SIMBA strategy.

An Extension to the Two-Point Method. The arguments of Section 3.2 do not apply immediately to the two-point method, but can be extended to that setting. When optimizing permutations for the two-point method, we cannot use Program (6) without a minor modification. In this setting, all vertical edges require two isogeny evaluations and, for the horizontal edges, some require two point multiplications while some require only one. In particular, when considering the i^{th} row of H , let $k_i = \max_{1 \leq k \leq n-1} \{k : V_{i,k} = 1\}$. Then in order to be able to compute the isogeny evaluations specified by V , for each 1 among the first k entries of the i^{th} row of H , we must multiply both torsion points by the corresponding prime, while for the remaining 1s in that row, we only need to multiply one torsion point (the one which corresponds to the sign of e_i).

To construct the appropriate linear program for this setting, we define modified strategy matrices $\hat{H}(S)$ and $\hat{V}(S)$ by

$$\hat{H}(S)_{ij} = \begin{cases} H_{i,j} & \text{if } j \geq k_i \\ 2H_{i,j} & \text{if } j \leq k_i - 1 \end{cases} \quad \hat{V}(S) = 2V(S)$$

where k_i is as defined above. We can then use Problem (6) with the following modified objective function:

$$(S)_{\sigma M} = \langle T_L^T \hat{H}^T \mathbf{1} \boldsymbol{\mu}^T + T_R^T \hat{V} \mathbf{1} \boldsymbol{\iota}^T, \Sigma \rangle_F.$$

We note that when S is the multiplication-based strategy (or a SIMBA strategy all of whose SIMBA substrategies are multiplication-based), $\hat{H}(S) = H(S)$,

and so we can apply Problem (6) by instead modifying the cost model, using $2\boldsymbol{\iota}$ in place of $\boldsymbol{\iota}$; since the best SIMBA substrategies we have found for the two point method have all been multiplication-based, we employ this modification in our parameter-finding scripts.

E.2 Optimizing the Bound Vector in the Two-Point Method

When optimizing the bound vector for the two-point method, the techniques of Section 3.3 apply in a fairly straightforward fashion. In particular, defining

$$\mathbf{c} = \Sigma^{-1} \left((\mathbf{1}^T \hat{H} T_L)^T \circ (\Sigma \boldsymbol{\mu}) + (T_R^T \hat{V} \mathbf{1}) \circ (\Sigma \boldsymbol{\iota}) + \Sigma \boldsymbol{\kappa} \right) \text{ and } \mathbf{d} = 2\boldsymbol{\mu}$$

(where \hat{H} and \hat{V} are as defined in Section 3.2), the mathematical program is

$$\begin{aligned} \text{Minimize } & \langle \mathbf{c} - \mathbf{d}, \mathbf{b} \rangle + \max_j \{b_j\} \mathbf{1}^T \mathbf{d} + 2 \max_j \left\{ \frac{b_j}{\ell_j - 1} \right\} \mathbf{1}^T \boldsymbol{\mu} \\ \text{Subject to } & \sum_{i=1}^n \log_2(2b_i + 1) \geq \lambda \\ & \mathbf{b} \geq 0 \\ & \mathbf{b} \in \mathbb{Z}^n \end{aligned} . \quad (9)$$

As before, in the case that S is the multiplication-based strategy or a SIMBA strategy all of whose SIMBA substrategies are multiplication-based, we can instead alter the definition of \mathbf{c} to use a cost model with $2\boldsymbol{\iota}$ in place of $\boldsymbol{\iota}$. We apply the same iterative rounding technique to this program as to the program (8) in Section 3.3.

E.3 The Complete Optimization Methodology for SIMBA and the Two-Point Technique

The algorithm above applies essentially unchanged when using the two-point technique. In this setting the best SIMBA substrategies we found were consistently multiplication-based. We exhaustively searched for the optimal SIMBA decomposition and permutation when all SIMBA substrategies are multiplication-based. Our process was:

1. For each m between 1 and 5 and each partition $P = (n_1, n_2, \dots, n_m)$ of n with parts of size at least 2, compute the optimal permutation σ for the SIMBA strategy S whose substrategies are the multiplication-based strategies of size n_1, n_2, \dots, n_m . Choose the partition, permutation, and strategy with the lowest cost among these.
2. Using the strategy and permutation obtained in step 1., we approximately solve the program (9) using the iterative rounding technique described in Section 3.3 to obtain a bound vector \mathbf{b} .
3. For $2 \leq k \leq \max_j \{b_j\}$, let $M_k^{(\mathbf{b})} = (\{\ell_i\}_{i: b_i \geq k}, f, g)$. Applying the technique of step 1., find the optimal permutation and partition for each $M_k^{(\mathbf{b})}$.

Optimizing for Submeasures. Suppose M' is a proper submeasure of M . We note that the cost of evaluating a strategy S under M' is $(S)_{M'} + m\mathbf{1}^T \boldsymbol{\mu}_{M''}$, where M'' is the complement of M' in M and $\boldsymbol{\mu}_{M''}$ is the subvector of $\boldsymbol{\mu}$ corresponding to the indices present in M'' . This additional term accounts for the multiplications that are performed to remove the prime factors present in M'' from the order of the initial points of the strategy (see line (7) of Algorithm 3). We must account for this cost during algorithms which compare the cost of SIMBA strategies that consist of different numbers of SIMBA substrategies—in particular, when computing costs in Algorithm 2, we must modify lines (4) and (13) to include this term. When using the two-point method, the additional term is instead $2m\mathbf{1}^T \boldsymbol{\mu}_{M''}$, since the primes of M'' must be eliminated from the orders of both torsion points at the beginning of each SIMBA substrategy.