Batching CSIDH Group Actions using AVX-512

Hao Cheng, Georgios Fotiadis, Johann Großschädl, Peter Y. A. Ryan and Peter B. Rønne

DCS and SnT, University of Luxembourg, Esch-sur-Alzette, Luxembourg
{hao.cheng,georgios.fotiadis,johann.groszschaedl,peter.ryan,peter.roenne}@uni.lu

Abstract. Commutative Supersingular Isogeny Diffie-Hellman (or CSIDH for short) is a recently-proposed post-quantum key establishment scheme that belongs to the family of isogeny-based cryptosystems. The CSIDH protocol is based on the action of an ideal class group on a set of supersingular elliptic curves and comes with some very attractive features, e.g. the ability to serve as a "drop-in" replacement for the standard elliptic curve Diffie-Hellman protocol. Unfortunately, the execution time of CSIDH is prohibitively high for many real-world applications, mainly due to the enormous computational cost of the underlying group action. Consequently, there is a strong demand for optimizations that increase the efficiency of the class group action evaluation, which is not only important for CSIDH, but also for related cryptosystems like the signature schemes CSI-FiSh and SeaSign. In this paper, we explore how the AVX-512 vector extensions (incl. AVX-512F and AVX-512IFMA) can be utilized to optimize constant-time evaluation of the CSIDH-512 class group action with the goal of, respectively, maximizing throughput and minimizing latency. We introduce different approaches for batching group actions and computing them in SIMD fashion on modern Intel processors. In particular, we present a hybrid batching technique that, when combined with optimized (8×1)-way prime-field arithmetic, increases the throughput by a factor of 3.64 compared to a state-of-the-art (non-vectorized) x64 implementation. On the other hand, vectorization in a 2-way fashion aimed to reduce latency makes our AVX-512 implementation of the group action evaluation about 1.54 times faster than the state-of-the-art. To the best of our knowledge, this paper is the first to demonstrate the high potential of using vector instructions to increase the throughput (resp. decrease the latency) of constant-time CSIDH.

Keywords: Post-Quantum Cryptography · Isogeny-Based Cryptography · CSIDH · AVX-512IFMA · Software Optimization · Constant-Time Implementation

1 Introduction

Quantum computing exploits quantum-mechanical effects and phenomena, such as state superposition and entanglement, to efficiently solve certain computational problems, in particular optimization and search problems [KLM07]. However, quantum computing has also a destructive side since it is assumed that a quantum computer with a few thousand logical qubits would be capable to break essentially any public-key cryptosystem in use today [RNSL17]. The dawning era of quantum computing has spurred much research on Post-Quantum Cryptography (PQC), a sub-domain of cryptography concerned with the design, analysis and implementation of cryptosystems that are expected to resist attacks executed on both conventional and quantum computers [SL21]. Almost all of the to-date existing post-quantum key establishment and signature algorithms fall into one of five categories, which are lattice-based cryptography, multivariate cryptography, hash-based cryptography, code-based cryptography, and supersingular isogeny cryptography. These categories differ with respect to the hard mathematical problems their security is based

on, but also in terms of computational cost, key lengths, and the length of ciphertexts (resp. signatures) [SL21]. The security of isogeny-based cryptosystems rests upon the intractability of the problem of finding an explicit isogeny between two (supersingular) elliptic curves over a finite field that are known to be isogenous [DeF17]. While isogeny-based schemes are computation-intensive, their key sizes are among the smallest of the five categories and come even close to that of pre-quantum elliptic curve schemes.

Various isogeny-based cryptosystems have appeared in the literature in the past ten years. SIKE (short for Supersingular Isogeny Key Encapsulation) is a key encapsulation mechanism whose security relies upon the supersingular isogeny walk problem between two elliptic curves in the same isogeny class, which asks to find a path made of isogenies of small degree [Cos19]. A variant of SIKE is an alternative candidate in the third round of the PQC standardization project of the NIST [JAC+20]. CSIDH (an abbreviation of Commutative Supersingular Isogeny Diffie-Hellman) is an "ECDH-like" key-exchange scheme based on a commutative group action of an ideal class group [CLM+18]. Given an initial elliptic curve E, a secret key in CSIDH is an ideal class $\mathfrak{a}$ in a class group (represented by its list of exponents), and the corresponding public key can be obtained by computing the group action $E' = \mathfrak{a} \star E$. The security of CSIDH is based on the hard problem of finding an isogeny path from the isogenous curves E and E'. CSIDH has received a lot of attention in recent years since it comes with highly attractive features like efficient validation of public keys, making it suitable for non-interactive (i.e. static) key exchange protocols. In fact, CSIDH can serve as "drop-in" replacement for classical ECDH key exchange and does even comply with the requirements of "0-RTT" protocols such as QUIC. Furthermore, class group actions provide a rich foundation for the design of various other cryptosystems, e.g. signature schemes [BKV19, DG19]. However, the downside of CSIDH is that the computation of group actions is very costly, which makes CSIDH extremely slow, not only in relation to X25519 [Ber06] and other pre-quantum ECDH variants, but also when compared to SIKE. For example, while an Intel Skylake processor can execute a variable-base scalar multiplication on Curve25519 in less than 100 k cycles [NS20] and a SIKEp434 encapsulation or decapsulation in about 10 M cycles [JAC+20], the to-date best constant-time implementation of a CSIDH-512 group action evaluation and key validation requires close to 240 M clock cycles [HLKA20].

The lengthy computation time of CSIDH poses a major obstacle for its application in security protocols like TLS or HTTPS when taking into account that, for example, the web servers of large enterprises like Google or Facebook are confronted with thousands of HTTPS requests per second. In order to be able to cope with such extreme volumes of traffic, the server infrastructure of such enterprises often includes a so-called TLS termination proxy or TLS reverse proxy, which transparently translates HTTPS sessions to TCP sessions for back-end servers (e.g. web or database servers), see [JHH+11]. This offloading of the TLS termination to a dedicated proxy frees the web server from having to execute computation-intensive TLS handshakes that involve public-key operations to authenticate the server to the client and establish a shared secret key using e.g. X25519 key exchange [Ber06]. A TLS termination proxy equipped with a high-end 64-bit Intel processor clocked at 4 GHz is (in theory) able to perform 40,000 X25519 key exchanges per second per core since, as mentioned before, a variable-base scalar multiplication on Curve25519 costs below 100 k cycles[1]. Replacing X25519 by SIKEp434 would decrease the (theoretical) upper bound of the number of key exchanges per second on one core to around 400. Even worse, when X25519 gets replaced by CSIDH-512, the number of key exchanges per core would go down to a mere 17 per second, which is more than three orders of magnitude below the (theoretical) throughput of X25519. Therefore, it is little surprising that techniques to speed up CSIDH are eagerly sought.

[1] These 40,000 key exchanges per second are a *theoretical upper bound* for the throughput of a single processor core, which can only be reached under the assumption that the core executes nothing else than scalar multiplications (i.e. all other operations, such as the transfer of public keys, are ignored).

Contributions. The straightforward way of maximizing the throughput of CSIDH is to minimize the latency of the underlying group action. However, we demonstrate in this paper that the usual approach of maximizing throughput by minimizing latency leads to sub-optimal results on Intel processors that are equipped with recent vector (i.e. SIMD) extensions such as AVX-512. To be more concrete, we show that, when using AVX-512 instructions, minimizing the latency of one group action requires different optimization strategies than maximizing the throughput of several group actions that are executed in SIMD fashion. We explain how the "limb-slicing" method presented in [CGT+21] can be applied to compute eight independent CSIDH group actions in parallel using AVX-512 instructions, whereby each group action uses a 64-bit element of a 512-bit vector. Limb-slicing is somewhat related to the well-known "bit-slicing" technique used in symmetric cryptography since it increases throughput at the expense of latency. We discuss in detail the obstacles we had to overcome to efficiently batch group actions and execute them in a SIMD-parallel way. Further, we describe software optimization techniques that enable a highly-efficient (8×1)-way parallel execution of the prime-field arithmetic operations using AVX-512F and AVX-512IFMA instructions. We also present a latency-optimized implementation of the group action for AVX-512IFMA, which can be used to speed up client-side TLS processing (while our throughput-optimized implementation targets the server side[2] and can be used for TLS termination as described in [JHH+11]). Our results for CSIDH-512 show that batch processing and limb-slicing achieve a throughput gain by a factor of 3.64 compared to an optimized (but non-vectorized) x64 implementation. In light of the recent debate about the post-quantum security of CSIDH-512, we emphasize that our optimizations can also be applied to parameter sets with larger primes, and we expect similar improvements in performance over non-vectorized implementations.

2 Preliminaries

In this section, we give a brief overview of the CSIDH protocol and the CSIDH class group action of Castryck, Lange, Martindale, Panny, and Renes [CLM+18]. Further, we summarize the existing constant-time implementations of the CSIDH class group action. For a detailed analysis of the theory of elliptic curves that is relevant for isogeny-based cryptography, we refer the reader to the lecture notes of De Feo [DeF17].

2.1 The CSIDH Key Exchange Protocol

The CSIDH protocol works over a finite field $\mathbb{F}_p$, where p is a large prime of the special form $p = 4 \cdot \ell_1 \cdots \ell_n - 1$ and $\ell_1 < \ldots < \ell_n$ are small odd primes. In addition, it uses supersingular elliptic curves[3] E_A, defined over $\mathbb{F}_p$ and represented in Montgomery form $E_A : y^2 = x^3 + Ax^2 + x$, with $A^2 \neq 4$, where the $\mathbb{F}_p$-endomorphism ring[4] of such curves is isomorphic to an order in the imaginary quadratic field $\mathbb{Q}(\sqrt{-p})$. Specifically, the authors in [CLM+18] choose a supersingular Montgomery curve E_0 (i.e. $A = 0$) with $p \equiv 3 \pmod 4$, where in this case $\mathrm{End}_{\mathbb{F}_p}(E_0) \cong \mathbb{Z}[\sqrt{-p}]$. Further, we define $\mathcal{E}\ell\ell_{\mathbb{F}_p}(\mathbb{Z}[\sqrt{-p}])$ as the set of all supersingular elliptic curves with the same $\mathbb{F}_p$-endomorphism ring $\mathbb{Z}[\sqrt{-p}]$.

[2] A TLS server under heavy load may have to serve thousands of connections per second, which means it may have to compute eight or more key exchanges every few milliseconds. On the other hand, if the load is low, it makes more sense to use a latency-optimized implementation. But when the load increases and the server gets confronted with (at least) eight connections in a short period of time, switching from the latency-optimized to the throughput-optimized implementation will lead to better performance. To date, OpenSSL and other TLS stacks do not support the batching of public-key cryptosystems, but an integration of batch processing is possible as demonstrated by SSLShader (see [JHH+11] for details).

[3] An elliptic curve E defined over $\mathbb{F}_p$ is called *supersingular*, iff $\#E(\mathbb{F}_p) = p+1$, otherwise it is *ordinary*.

[4] For an elliptic curve E, the $\mathbb{F}_p$-endomorphism ring $\mathrm{End}_{\mathbb{F}_p}(E)$ contains all endomorphisms from E to itself, that are defined over $\mathbb{F}_p$.

The CSIDH Class Group Action. The ideal class group $\mathrm{Cl}(\mathbb{Z}[\sqrt{-p}])$ acts freely and transitively on $\mathcal{E}\ell\ell_{\mathbb{F}_p}(\mathbb{Z}[\sqrt{-p}])$, via isogenies[5] (Theorem 7 in [CLM+18]). Every principal ideal $(\ell_i) \subset \mathbb{Z}[\sqrt{-p}]$ splits as a product of prime ideals $(\ell_i) = \mathfrak{l}_i\bar{\mathfrak{l}}_i = \langle \ell_i, \pi - 1\rangle\langle \ell_i, \pi + 1\rangle$, where $\pi = \sqrt{-p}$ is the Frobenius endomorphism[6] and since (ℓ_i) is principal, we get $\bar{\mathfrak{l}}_i = \mathfrak{l}_i^{-1} \in \mathrm{Cl}(\mathbb{Z}[\sqrt{-p}])$. In CSIDH we are interested in computing the action of an ideal $\mathfrak{a} = \mathfrak{l}_1^{e_1} \cdots \mathfrak{l}_n^{e_n} \in \mathrm{Cl}(\mathbb{Z}[\sqrt{-p}])$, where $e_1, \ldots, e_n$ are small exponents, chosen uniformly from some interval $[-b, b]$. This is done by computing in sequence the action of the ideal $\mathfrak{l}_i$, if $e_i \geq 0$, or $\bar{\mathfrak{l}}_i$, if $e_i < 0$, exactly $|e_i|$ times for every $i \in \{1, \ldots, n\}$.

Algorithm 1: Computing the class group action for CSIDH [CLM+18]

Input: $A \in \mathbb{F}_p$ and a list of integers $(e_1, \ldots, e_n)$.
Output: $B \in \mathbb{F}_p$, such that $(\mathfrak{l}_1^{e_1} \cdots \mathfrak{l}_n^{e_n}) \star E_A = E_B$, where $E_B : y^2 = x^3 + Bx^2 + x$.

while some $e_i \neq 0$ **do**
  Sample a random $x \in \mathbb{F}_p$
  $s \leftarrow +1$ if $x^3 + Ax^2 + x$ is a square in $\mathbb{F}_p$, else $s \leftarrow -1$
  Let $S = \{i \mid e_i \neq 0, \mathrm{sign}(e_i) = \mathrm{sign}(s)\}$. **If** $S = \emptyset$ **then** start over with a new x.
  Let $P = (x : 1)$, $q \leftarrow \prod_{i \in S} \ell_i$ and compute $T \leftarrow [(p+1)/q]P$
  for each $i \in S$ **do**
    $R \leftarrow [q/\ell_i]T$ // R is the kernel generator
    if $R \neq \infty$ **then**
      Compute $\phi : E_A \rightarrow E_B = \mathfrak{l}_i \star E_A$ with $\ker(\phi) = \langle R \rangle$
      $A \leftarrow B$, $T \leftarrow \phi(T)$, $q \leftarrow q/\ell_i$, $e_i \leftarrow e_i - s$
return B

For the action of the ideal $\mathfrak{l}_i$ we choose a point $R \in E(\mathbb{F}_p)$ of order ℓ_i that lies in the kernel of $\pi - 1$ and compute the isogeny $\phi_{\mathfrak{l}_i} : E \rightarrow E/\langle R \rangle = \mathfrak{l}_i \star E$, with $\ker(\phi_{\mathfrak{l}_i}) = \langle R \rangle$ and $\deg(\phi_{\mathfrak{l}_i}) = \ell_i$. For the action of the ideal $\bar{\mathfrak{l}}_i$ we choose a random point $R \in E(\mathbb{F}_{p^2})$ (i.e. the quadratic twist of E), of order ℓ_i in the kernel of $\pi + 1$. Note that in this case, $R = (x, iy)$, where $x, y \in \mathbb{F}_p$ and $i = \sqrt{-1}$. Then we compute the isogeny $\phi_{\bar{\mathfrak{l}}_i} : E \rightarrow E/\langle R \rangle = \bar{\mathfrak{l}}_i \star E$, with $\ker(\phi_{\bar{\mathfrak{l}}_i}) = \langle R \rangle$ and $\deg(\phi_{\bar{\mathfrak{l}}_i}) = \ell_i$. Both isogenies are computed using the Vélu formulæ [Vél71], which require $O(\ell_i \log p^2)$ bit operations, hence they are efficiently computed for relatively small primes ℓ_i. Iterating each isogeny computation $|e_i|$ times, depending on the sign of e_i and composing the resulting isogenies in each step, yields the final codomain curve $\mathfrak{a} \star E = (\mathfrak{l}_1^{e_1} \cdots \mathfrak{l}_n^{e_n}) \star E$ (see Algorithm 1 [CLM+18]).

CSIDH. The public parameters are the prime $p = 4 \cdot \ell_1 \cdots \ell_n - 1$, the starting curve $E_0 : y^2 = x^3 + x$ and a positive integer b, such that $(2b+1) \geq \sqrt[n]{\#\mathrm{Cl}(\mathbb{Z}[\sqrt{-p}])}$ in order to maintain security. Alice's secret key is a list of exponents $sk_A = (e_1, \ldots, e_n) \in [-b, b]^n$, while her public key is derived from the action of the ideal $\mathfrak{l}_1^{e_1} \cdots \mathfrak{l}_n^{e_n}$ on the curve E_0, using Algorithm 1, i.e. $pk_A = E_A = (\mathfrak{l}_1^{e_1} \cdots \mathfrak{l}_n^{e_n}) \star E_0$, which is sent to Bob. In the same vein, Bob's secret key is $sk_B = (d_1, \ldots, d_n) \in [-b, b]^n$, and his public key $pk_B = E_B = (\mathfrak{l}_1^{d_1} \cdots \mathfrak{l}_n^{d_n}) \star E_0$ is sent to Alice. For the shared secret, Alice and Bob compute the codomain curves $k_A = (\mathfrak{l}_1^{e_1} \cdots \mathfrak{l}_n^{e_n}) \star E_B$ and $k_B = (\mathfrak{l}_1^{d_1} \cdots \mathfrak{l}_n^{d_n}) \star E_A$ respectively, using Algorithm 1. The two curves are $\mathbb{F}_p$-isomorphic, because they are derived from the action of the ideal $\mathfrak{l}_1^{e_1+d_1} \ldots \mathfrak{l}_n^{e_n+d_n}$ on the initial curve E_0, as a result of the commutativity property of the ideal class group $\mathrm{Cl}(\mathbb{Z}[\sqrt{-p}])$. Note that the public keys and the shared secret, are elliptic curves in Montgomery form, hence they are represented as a single coefficient in $\mathbb{F}_p$.

[5] Let ∞_E be the point at infinity on E. An *isogeny* between two elliptic curves E and E', over $\mathbb{F}_p$, is a non-constant rational map $\phi : E \rightarrow E'$ with finite kernel, that maps ∞_E to $\infty_{E'}$.

[6] The Frobenius endomorphism π maps a point $P = (x, y)$ on an elliptic curve E to (x^p, y^p)

CSIDH features an efficient public-key validation process, which corresponds to testing whether the public key is a supersingular Montgomery curve, and it is accomplished with a series of scalar multiplications [CLM+18]. Castryck, Lange, Martindale, Panny, and Renes presented a concrete instantiation for CSIDH. They choose a 511-bit prime $p = 4 \cdot \ell_1 \cdots \ell_{74} - 1$, where $\ell_1, \ldots, \ell_{73}$ are the first odd primes starting from $\ell_1 = 3$, and $\ell_{74} = 587$. The secret exponents $(e_1, \ldots, e_{74})$ are sampled from $[-5, 5]^{74}$ (hence $b = 5$), in which case $74 \log_2(2 \cdot 5 + 1) \approx 256$. This instantiation is referred to as CSIDH-512.

Security of CSIDH. The security of CSIDH is based on the Group Action Inverse Problem (GAIP). That is, given two supersingular elliptic curves E and E', defined over $\mathbb{F}_p$, with the same $\mathbb{F}_p$-endomorphism ring $\mathbb{Z}[\sqrt{-p}]$, to find an ideal $\mathfrak{a} = \mathfrak{l}_1^{e_1} \cdots \mathfrak{l}_n^{e_n}$ such that $\mathfrak{a} \star E = E'$. The best known classical attack for solving GAIP is the meet-in-the-middle attack with fully exponential complexity $O(\sqrt{N})$, where $N = \#\mathrm{Cl}(\mathbb{Z}[\sqrt{-p}]) \approx \sqrt{p}$. In the quantum setting, Childs, Jao, and Soukharev [CJS14] have shown that solving the GAIP problem can be reduced to the abelian hidden-shift problem, for which the subexponential quantum algorithms of Regev [Reg04] and Kuperberg [Kup05] can be applied, where the latter has complexity $O(\sqrt{\log p})$ and the quantum space complexity $O(\log p)$.

Based on the above, Castryck, Lange, Martindale, Panny, and Renes [CLM+18] conjectured that CSIDH-512 would achieve NIST's post-quantum security level 1 based on the asymptotic complexity of Kuperberg's algorithm [Kup05, Kup13]. However, the concrete security of CSIDH-512 is under debate since the works of Peikert [Pei20], Bonnetain and Schrottenloher [BS20], and more recently Chávez-Saab, Chi-Domínguez, Jaques, and Rodríguez-Henríquez [CCJR20], estimate that the prime p should be significantly larger in order to meet NIST's security level 1. In particular, [CCJR20] suggests that p should be updated to 4096 bits.

2.2 Optimization Techniques for Constant-Time CSIDH

In practice, we require a constant-time implementation of Algorithm 1 to resist against side-channel attacks. Given a secret exponent list $(e_1, \ldots, e_n)$, Algorithm 1 computes $|e_1| + \ldots + |e_n|$ isogenies, and thus its execution time fully depends on the secret key. Meyer, Campos, and Reith [MCR19] presented three leakage scenarios that appear when implementing Algorithm 1. Timing leakage occurs since different secret keys lead to different execution times of evaluating the class group action. Power analysis leaks information on the sign distribution of the secret key since unbalanced, in terms of the sign, secret exponents lead to scalar multiplications with larger scalars. Cache timing attacks are also possible and leak information based on branch conditions or array indices. The authors in [CLM+18] argued that a constant-time implementation can be obtained when adding certain "dummy" operations, which will not be considered nor affect the final output of the group action. The first constant-time implementations of Algorithm 1 are due to Bernstein, Lange, Martindale, Panny [BLMP19] and Jalali, Azarderakhsh, Kermani, Jao [JAKJ19], which add a large amount of dummy operations and have a probability of failure in the class group action computation.

Meyer, Campos, and Reith. A constant-time implementation of the CSIDH class group action with significant optimizations is presented in [MCR19], and it is known as the "MCR-style". The algorithm uses only positive secret exponents e_i, each sampled from its own space $[0, b_i]$ where all b_i are chosen such that security is maintained. This mitigates power attacks, while the different intervals allow to reduce the number of large degree isogenies. Meyer et al. use dummy isogenies so that the same number of isogenies is computed in each class group action. For each i their algorithm computes e_i "real" and $b_i - e_i$ "dummy" ℓ_i-isogenies. Further, they use the Elligator 2 map [BHKL13] for sampling

points on the curve. As observed in [MR18], they compute the class group action in descending order in terms of the primes ℓ_i, which results in a speedup over the ascending order. The most significant optimization is the SIMBA-m-μ technique, where the idea is to partition the indices $\{1, \ldots, n\}$ into m disjoint subsets and evaluate the group action on each subset individually, which results in smaller scalars in step 7 of Algorithm 1. However, this should be done for a specific number of rounds μ, and the subsets should be merged back after this threshold. The authors argue that finding optimal values for m and μ, as well as for the upper bounds b_i is a hard task. They present various choices, based on the CSIDH-512 instance, where the best example is SIMBA-5-11.

Onuki, Aikawa, Yamazaki, and Takagi. In [OAYT19] the authors present a constant-time implementation of Algorithm 1, known as the "OAYT-style", that improves on the MCR-style by 29.03%. In their algorithm each e_i is also sampled from its own space, but in contrast to the MCR-style, each e_i is allowed to be negative as well. The algorithm mitigates timing attacks by keeping track of two points $P_0 \in E[\pi - 1]$ and $P_1 \in E[\pi + 1]$ and picking the appropriate point, depending on the sign of e_i, in order to create the kernel generator. Both points P_0 and P_1 are mapped through the isogeny, and the point not used to derive the kernel is multiplied by ℓ_i. The algorithm also uses the Elligator 2 map for generating points on the curve and dummy isogenies as in the MCR-style.

Cervantes-Vázquez, Chenu, Chi-Domínguez, De Feo, Rodríguez-Henríquez, and Smith. The work in [CCC+19] provides significant improvements on both the MCR- and OAYT-style algorithms. The authors present efficient formulas in the twisted Edwards model for performing isogeny computations, isogeny evaluations and curve operations (point addition/doubling). They also use differential addition chains which provide a 25% improvement compared to the classical Montgomery ladder, for computing scalar multiplications. Besides the optimizations for the MCR- and OAYT-style algorithms, the authors present a constant-time implementation that excludes the dummy operations, known as "dummy-free-style". Although this is less efficient compared to the MCR- and OAYT-style, it is resistant against fault-injection attacks, i.e. stronger attackers who can interfere in computations and determine whether a modification happened on a "real" or a "dummy" isogeny. Their optimized OAYT-style with SIMBA-3-8 is 39% faster than the MCR-style presented in [MCR19], and their dummy-free-style group action is two times slower compared to their OAYT-style implementation.

Optimal Strategies. In [HLKA20], Hutchinson, LeGrow, Koziel, and Azarderakhsh further studied problems of choosing the optimal bounds b_i for sampling secret exponents, the optimal ordering of primes ℓ_i, and the optimal partition m for SIMBA technique. Such selections are referred to as *optimal strategies*. Their optimal strategies offer 5.06% improvement for the OAYT-style implementation in [CCC+19]. In [CR20], Chi-Domínguez and Rodríguez-Henríquez generalized the computational strategies that are used in the SIKE implementation [JAC+20] for the case of CSIDH. Their new algorithms do not rely on the SIMBA approach and provide an improvement of 12.09%, 3.36%, and 10.58% compared to the MCR-, OAYT-, and dummy-free-style implementations of [CCC+19], respectively. The OAYT-style algorithm of [HLKA20], the MCR- and dummy-free-style algorithms of [CR20] are to date the most efficient constant-time implementations of CSIDH in the literature. These algorithms are further optimized by Adj, Chi-Domínguez, and Rodríguez-Henríquez [ACR20] when applying certain tricks that reduce the computational cost of new Vélu formulæ of [BDLS20].

2.3 AVX-512

AVX-512 is the latest incarnation of Intel Advanced Vector eXtension (AVX), which augments the execution environment of x64 by 32 registers of a length of 512 bits and various new instructions. AVX-512 contains multiple extensions, but a specific processor may support some but not all of them. From another perspective, all processors equipped with AVX-512 support the core extension named AVX-512 Foundation (AVX-512F).

AVX-512IFMA. Among the extensions of AVX-512, AVX-512IFMA (Integer Fused Multiply-Add), or simply IFMA, is very attractive for the public key cryptosystems whose underlying arithmetic is the large integer arithmetic, including isogeny-based cryptosystems. IFMA was first supported with Intel Cannon Lake and continued to be equipped in its successors such as Ice Lake and Tiger Lake processors. Intel described IFMA in [Int18] as "two new instructions for big number multiplication for acceleration of RSA vectorized SW and other Crypto algorithms (Public key) performance". Specifically, IFMA introduced two novel instructions `vpmadd52luq` and `vpmadd52huq`. An IFMA instruction first multiplies packed unsigned 52-bit integers in each 64-bit lane of two registers to produce a 104-bit intermediate product. It then adds either the low or the high 52-bit from the product with corresponding packed unsigned 64-bit integer (from the third register), and stores the final results in a destination register. Compared to `vpmuludq` or `vpmuldq` instruction in AVX-512F, IFMA not only offers wider multiplier (52-bit in IFMA vs 32-bit in AVX-512F) but also fuses the multiplication and the addition in a single instruction.

Target Platform. In this work, we target the Intel Ice Lake processor which supports IFMA extension. The specific processor we used in our experiments is Intel Core i3-1005G1 CPU clocked at 1.2 GHz.

Relevant Instructions. Table 1 lists the most relevant AVX-512 instructions used in this work, along with their latency and throughput data[7] on the Ice Lake processor which we obtained from Intel official document [Int20]; the throughput data is shown in Clock Per Instructions (CPI) ratio [Int18]; the instruction mnemonic is used to describe algorithms in this paper.

Table 1: The latency (in clock cycles) and throughput (CPI) of relevant AVX-512 instructions on Intel Ice Lake processor.

Type	Mnemonic	Instruction	Latency	CPI
Arithmetic	`ADD/SUB`	`vpaddq/vpsubq`	1	0.5
	`MUL`	`vpmuludq`	–	1
Logic	`SHL/SHR`	`vpsllq/vpsrlq`	1	1
	`AND`	`vpandd`	1	0.5
Permutation	`PERM`	`vpermq`	3	1
	`BCAST`	`vpbroadcastq`	3	1
	`ZERO`	`vpxorq`†	1	0.5
IFMA	`MACLO`	`vpmadd52luq`	4	1
	`MACHI`	`vpmadd52huq`	4	1

† XOR a `ZMM` register with itself to set it to zero.

[7] Here we consider the case that these instructions work on 512-bit `ZMM` registers not 128-bit `XMM` or 256-bit `YMM` registers. The instruction CPI of the latter case is lower since Port 1 can handle AVX-512 instructions working on `XMM` or `YMM` but on `ZMM` registers (see [Int18, Figure 2-1]).

3 Methods for Batching CSIDH Group Actions

Recall from Section 2.2 that the to-date fastest constant-time CSIDH implementations are the two OAYT-style variants of [HLKA20] and [CR20]. According to our measurements of their group action evaluation (see Table 3), the former is 1% faster than the latter. The optimization techniques used in both variants improve the OAYT-style implementation of [CCC+19] by 5%, and they are all considered in terms of x64 implementation. When the same optimization techniques are applied to AVX-512 software, the resulting effects may be different. In this work we focus on batching the OAYT-style implementation of [CCC+19]. However, we argue that the optimization techniques of [HLKA20, CR20] as well as [ACR20] can also be applied in our batched implementation.

The OAYT-style class group action algorithm of [CCC+19] is described in Algorithm 2, which was originally presented in [OAYT19]. Algorithm 2 takes advantage of the SIMBA-m-μ technique [MCR19], in which the set of indices $S = \{1, \ldots, n\}$ is partitioned into m subsets $S_1, \ldots, S_m$, where $S_j = \{j, m + j, 2m + j, \ldots\}$ for each $j \in \{1, \ldots m\}$. For the CSIDH-512 instantiation, the best choices according to [OAYT19] are $m = 3$ and $\mu = 8$. In this case, Algorithm 2 computes 404 "real" and "dummy" isogenies (i.e. the variable $t_{\max} = \sum_{i=1}^{n} b_i = 404$, see Appendix A for the bound vector $\boldsymbol{b}$ and the ordering of the small primes $\boldsymbol{\ell}$). Following [CCC+19], we define the constant-time equality test and constant-time conditional swap functions `isequal` and `cswap`, respectively, as:

$$\texttt{isequal}(x, y) = \begin{cases} 1, & \text{if} \quad x = y \\ 0, & \text{if} \quad x \neq y \end{cases}, \qquad \texttt{cswap}(x, y, a) = \begin{cases} x \leftrightarrow y, & \text{if} \quad a = 1 \\ x \nleftrightarrow y, & \text{if} \quad a = 0 \end{cases}$$

Further, we consider a constant-time function $\texttt{sign}(x)$, which returns 0 if $x < 0$ while returns 1 if $x \geq 0$. More details on Algorithm 2 are given in [OAYT19, CCC+19].

3.1 Obstacles to Batching CSIDH Group Actions

We conceive our batched software where eight CSIDH group action instances in the fashion of Algorithm 2 are to be computed simultaneously by AVX-512 instructions. Besides, each instance is computed in 64-bit lane, and instances are independent of each other. Since AVX-512 is in the paradigm of *Single Instruction Multiple Data*, from another perspective, this requires that the same instruction must be executed at the same time in eight instances. In other words, each of the eight instances in the execution of our batched software must process *the same instruction sequence* or, we say, *the same operation sequence* at a higher layer. This is a crucial requirement, in addition to having a constant-time implementation which is already accomplished by Algorithm 2. The sequence of operations in Algorithm 2, relies on specific conditional statements, which are (indirectly) affected by the value of the random curve points generated internally at line 7 to 8.

Specifically, a closer look at Algorithm 2 reveals that the sequence of operations (and instructions) that are carried out depends on whether the kernel generator R at line 13 is infinity or not. If $R \neq \infty$, the algorithm computes either a "real" or a"dummy" isogeny (depending on whether e_i is non-zero or not) in the "if"-branch, whereas it performs a scalar multiplication in the "else"-branch if $R = \infty$. In the scenario of eight parallel class group actions that we are considering, this is *problematic*, and especially, it is very likely that at some iterations the point R will be infinity at least in one of the parallel instances. In particular, the probability for a point of order ℓ_i to be infinity is $1/\ell_i$, which is considerably high when ℓ_i is small (e.g. $3, 5, 7, \ldots$ in CSIDH). This will cause a *mismatch* between the simultaneous instances and will affect other variables as well, such as the update of b'_i and the isogeny counter t_{isog} at line 21, leading to different instruction sequences for the different instances.

Clearly, in order to obtain a constant-time batched software where eight running instances follow the same instruction sequence, we need to mitigate the mismatch caused

Algorithm 2: The OAYT-style algorithm with SIMBA-m-μ for computing the CSIDH group action.

Input: $A \in \mathbb{F}_p$, bound $\boldsymbol{b} = (b_1, \ldots, b_n)$, exponents $\boldsymbol{e} = (e_1, \ldots, e_n)$ where $e_i \in [-b_i, b_i]$.
Output: $B \in \mathbb{F}_p$, such that $(\mathfrak{l}_1^{e_1} \cdots \mathfrak{l}_n^{e_n}) \star E_A = E_B$, where $E_B : y^2 = x^3 + Bx^2 + x$.

1 $(e'_1, \ldots, e'_n) \leftarrow (e_1, \ldots, e_n)$, $(b'_1, \ldots, b'_n) \leftarrow (b_1, \ldots, b_n)$
2 $E_B \leftarrow E_A$, $t_{\max} \leftarrow \sum_{i=1}^{n} b_i$, $t_{\text{isog}} \leftarrow 0$, $r \leftarrow 0$, $j \leftarrow 0$
3 **while** $t_{\text{isog}} < t_{\max}$ **do**
4 $j \leftarrow (j \bmod m) + 1$ // Subset index
5 **if** $r = \mu \cdot m$ **then**
6 $S_1 \leftarrow \{i \mid b_i \neq 0\}$, $j \leftarrow 1$, $m \leftarrow 1$ // Merge subsets
7 $u \leftarrow \texttt{random}(\{2, \ldots, (p-1)/2\})$, $q \leftarrow \prod_{i \in S_j} \ell_i$
8 $(P_0, P_1) \leftarrow \texttt{Elligator}(E_B, u)$ // $P_0 \in E[\pi - 1]$ and $P_1 \in E[\pi + 1]$
9 $(T_0, T_1) \leftarrow ([(p+1)/q]P_0, [(p+1)/q]P_1)$
10 **for each** $i \in S_j$ **do**
11 $s \leftarrow \texttt{sign}(e'_i)$
12 $\texttt{cswap}(T_0, T_1, 1 - s)$
13 $R \leftarrow [q/\ell_i]T_0$ // R is the kernel generator
14 **if** $R \neq \infty$ **then**
15 $w \leftarrow 1 - \texttt{isequal}(e'_i, 0)$
16 Compute isogeny $\phi : E_B \to E_C = \mathfrak{l}_i \star E_B$ s.t. $\ker(\phi) = \langle R \rangle$
17 $T_1 \leftarrow [\ell_i]T_1$
18 $(T_2, T_3) \leftarrow (\phi(T_0), \phi(T_1))$
19 $T_0 \leftarrow [\ell_i]T_0$
20 $\texttt{cswap}(T_0, T_2, w)$, $\texttt{cswap}(T_1, T_3, w)$, $\texttt{cswap}(B, C, w)$
21 $e'_i \leftarrow e'_i + (-1)^s \cdot w$, $b'_i \leftarrow b'_i - 1$, $t_{\text{isog}} \leftarrow t_{\text{isog}} + 1$
22 **else**
23 $T_1 \leftarrow [\ell_i]T_1$
24 $\texttt{cswap}(T_0, T_1, 1 - s)$, $q \leftarrow q/\ell_i$
25 **if** $b'_i = 0$ **then**
26 Remove i from S_j
27 $r \leftarrow r + 1$
28 **return** B

by the infinity check at line 14. We explore three methodologies that achieve a batching-friendly CSIDH group action and deal with this specific if-else statement. In our first method, the idea is to rewrite Algorithm 2 so that this if-else clause is no longer needed. This requires the computation of additional dummy isogenies in the case where the kernel point is infinity, and hence we refer to this method as *extra-dummy* (Section 3.2). In the second method, we still keep this if-else statement but we force all eight instances to always agree on the same branch. That is, if at least one kernel point in the eight instances is the point at infinity, then all instances will enter the "else"-branch at line 22. We refer to this methodology as *extra-infinity* method (Section 3.3). The third idea is based on the combination of the extra-dummy and extra-infinity methods, therefore we call it the *combined* method (Section 3.4).

Hybrid Mode. Notably, all of our three methods are constructed in a *hybrid mode* which significantly improves the performance of the batched CSIDH implementation. In the context of this paper, hybrid mode means that the entire batched software is composed of two different types of class group action implementations, namely the *batched component* and the *unbatched component*. The batched component is an *incomplete* implementation that performs eight class group action evaluations simultaneously. The unbatched component

is a latency-optimized implementation accelerating a single class group action evaluation (such as the implementation of Algorithm 2, presented in [CCC+19]). The key idea in the three methods is to first take advantage of the batched component to compute the main bulk of the CSIDH group action (including almost all isogeny computations) for all instances, and then use eight times in sequence the unbatched component to handle the remaining computations needed in each instance.

The three methods are based on the OAYT-style implementation of Algorithm 2 with SIMBA-m-μ. However, in Section 3.5, we show that all three methods can also be applied to batch the dummy-free-style algorithm of [CCC+19], which is considered to be secure against fault-injection attacks.

3.2 Extra-Dummy Method

Our first batching method initially aims at making Algorithm 2 independent of all inputs as well as *all randomness*. In brief, we remove the if-else clause (line 14 and line 22) that checks whether R is infinity, at a cost of *extra* dummy isogeny computations. This idea was first presented in [BLMP19] and also adopted in [JAKJ19] for a constant-time CSIDH implementation on 64-bit ARM processors. For both implementations there exists a probability of failure in computing the correct codomain curve, and a large number of dummy isogeny computations are required to make this probability negligible (e.g. 2^{-32}).

In detail, according to [CCC+19], given a point $P = (Y : T)$ represented in twisted Edwards YT-coordinates[8], we define a constant-time function for checking whether the point P is infinity as:

$$\texttt{isinfinity}(P) = \begin{cases} 1, & \text{if} \quad Y = T \Leftrightarrow P = \infty \\ 0, & \text{if} \quad Y \neq T \Leftrightarrow P \neq \infty \end{cases}$$

This time we compute a "real" isogeny from the kernel point R, if $R \neq \infty$ and $e'_i \neq 0$; whereas we compute a "dummy" isogeny if either $R = \infty$ or $e'_i = 0$. Similarly, these dummy isogenies will not be considered in the final result, but they will cause the counter t_{isog} to increment. Consequently, there is a possibility that for some indices i, the number of the computed "dummy" isogenies exceeds the value $b_i - |e_i|$ in which case we lose "real" isogenies that should be computed. This implies that although the algorithm will terminate, the resulting codomain curve will be incorrect since it will not correspond to the secret exponents $(e_1, \ldots, e_n)$. This probability of failure can be reduced by fixing the number of dummy isogenies to be computed, as done in [BLMP19, JAKJ19]. In other words, except for the dummy isogenies originally needed by Algorithm 2 to make the group action independent of the secret exponents (we call them *initial dummy isogenies*), we import extra dummy isogenies to make the group action independent of the randomness (we call them *extra dummy isogenies*). The modified group action has now the same operation sequence in each execution, which meets the requirement for batching. However, according to our calculation, it requires to import more than $\sum_{i=1}^{n} b_i$ extra dummy isogenies to make the failure probability below 2^{-32}. Hence, this idea needs to compute more than two times the number of isogenies needed in Algorithm 2, which significantly reduces the efficiency of the algorithm, while the probability of failure still exists.

Based on the above discussion, we are looking for a way to drastically reduce the number of extra dummy isogenies and eliminate the probability of failure, but meanwhile retain this batching-friendly fashion of group action. This can be done using the hybrid mode. As introduced in the previous subsection, the hybrid mode consists of the batched component and the unbatched component. In the batched component, we still compute $\sum_{i=1}^{n} b_i$ ("real" and "dummy") isogenies, where in this case, dummy isogenies appear

[8] A point P on a projective twisted Edwards curve in YT-coordinate representation is expressed as $P = (Y : T)$, where Y/T is the affine y-coordinate of the point P.

Algorithm 3: The batched component of our extra-dummy method for OAYT-style group action evaluation.

Input: $A \in \mathbb{F}_p$, bound $\boldsymbol{b} = (b_1, \ldots, b_n)$, exponents $\boldsymbol{e} = (e_1, \ldots, e_n)$ where $e_i \in [-b_i, b_i]$.
Output: $\hat{B} \in \mathbb{F}_p$, the lists $(\hat{b}_1, \ldots, \hat{b}_n)$ and $(e'_1, \ldots, e'_n)$.

1 $(e'_1, \ldots, e'_n) \leftarrow (e_1, \ldots, e_n)$, $(b'_1, \ldots, b'_n) \leftarrow (b_1, \ldots, b_n)$, $(\hat{b}_1, \ldots, \hat{b}_n) \leftarrow (b_1, \ldots, b_n)$
2 $E_{\hat{B}} \leftarrow E_A$, $t_{\max} \leftarrow \sum_{i=1}^{n} b_i$, $t_{\text{isog}} \leftarrow 0$, $r \leftarrow 0$, $j \leftarrow 0$
3 **while** $t_{\text{isog}} < t_{\max}$ **do**
4 $j \leftarrow (j \bmod m) + 1$ // Subset index
5 **if** $r = \mu \cdot m$ **then**
6 $S_1 \leftarrow \{i \mid b_i \neq 0\}$, $j \leftarrow 1$, $m \leftarrow 1$ // Merge subsets
7 $u \leftarrow \texttt{random}(\{2, \ldots, (p-1)/2\})$, $q \leftarrow \prod_{i \in S_j} \ell_i$
8 $(P_0, P_1) \leftarrow \texttt{Elligator}(E_{\hat{B}}, u)$ // $P_0 \in E[\pi - 1]$ and $P_1 \in E[\pi + 1]$
9 $(T_0, T_1) \leftarrow ([(p+1)/q]P_0, [(p+1)/q]P_1)$
10 **for each** $i \in S_j$ **do**
11 $s \leftarrow \texttt{sign}(e'_i)$
12 $\texttt{cswap}(T_0, T_1, 1 - s)$
13 $R \leftarrow [q/\ell_i]T_0$ // R is the kernel generator
14 $f \leftarrow 1 - \texttt{isinfinity}(R)$
15 $w \leftarrow 1 - \texttt{isequal}(e'_i, 0)$
16 Compute isogeny $\phi : E_{\hat{B}} \to E_C = \mathfrak{l}_i \star E_{\hat{B}}$ with $\ker(\phi) = \langle R \rangle$
17 $T_1 \leftarrow [\ell_i]T_1$
18 $(T_2, T_3) \leftarrow (\phi(T_0), \phi(T_1))$
19 $T_0 \leftarrow [\ell_i]T_0$
20 $\texttt{cswap}(T_0, T_2, f\&w)$, $\texttt{cswap}(T_1, T_3, f\&w)$, $\texttt{cswap}(\hat{B}, C, f\&w)$
21 $e'_i \leftarrow e'_i + (-1)^s \cdot (f\&w)$, $b'_i \leftarrow b'_i - 1$, $t_{\text{isog}} \leftarrow t_{\text{isog}} + 1$
22 $\hat{b}_i \leftarrow \hat{b}_i - f$
23 $\texttt{cswap}(T_0, T_1, 1 - s)$, $q \leftarrow q/\ell_i$
24 **if** $b'_i = 0$ **then**
25 Remove i from S_j
26 $r \leftarrow r + 1$
27 **return** $\hat{B}$, $(\hat{b}_1, \ldots, \hat{b}_n)$, $(e'_1, \ldots, e'_n)$

whenever a secret exponent is zero, or the kernel point is infinity. In addition, we keep track of all the dummy isogenies that occurred from infinity kernel points. After the batched component terminates, we take advantage of the unbatched component to compute "compensatory" isogenies based on the previously recorded infinity cases. Since this method adds extra dummy isogenies for each instance that occurred from the infinity cases, we refer to it as the extra-dummy method.

Algorithm 3 explains the batched component of our extra-dummy method. We first apply this batched component for computing eight group action instances in parallel. Hence, in our batched software, the input is composed of a secret exponent list and a starting curve:

$$\langle (e_1^{(1)}, \ldots, e_n^{(1)}),\ (e_1^{(2)}, \ldots, e_n^{(2)}),\ \ldots,\ (e_1^{(8)}, \ldots, e_n^{(8)}) \rangle \text{ and } \langle A^{(1)},\ A^{(2)},\ \ldots,\ A^{(8)} \rangle.$$

In Algorithm 3, apart from the bound list $(b'_1, \ldots, b'_n)$, we also create an additional bound list for each instance to record the infinity cases:

$$\langle (\hat{b}_1^{(1)}, \ldots, \hat{b}_n^{(1)}),\ (\hat{b}_1^{(2)}, \ldots, \hat{b}_n^{(2)}),\ \ldots,\ (\hat{b}_1^{(8)}, \ldots, \hat{b}_n^{(8)}) \rangle,$$

which only decreases when an isogeny is computed from a non-infinity kernel point (line 22 in Algorithm 3). At the beginning of the batched component, each list $(\hat{b}_1^{(k)}, \ldots, \hat{b}_n^{(k)})$ is initialized to $\boldsymbol{b}$ (same as $(b'_1, \ldots, b'_n)$).

When the batched component terminates, it outputs for each instance, a curve coefficient $\hat{B}^{(k)}$, the list $(\hat{b}_1^{(k)}, \ldots, \hat{b}_n^{(k)})$, and the list of exponents $(e'^{(k)}_1, \ldots, e'^{(k)}_n)$, where most of the $e'^{(k)}_i$ (as well as current $\hat{b}_i^{(k)}$) are 0, for $i \in \{1, \ldots, n\}$ and $k \in \{1, \ldots, 8\}$. As a result, there are only a few "real" (and "dummy") remaining isogenies that need to be computed for each instance, based on $(e'^{(k)}_1, \ldots, e'^{(k)}_n)$ and $(\hat{b}_1^{(k)}, \ldots, \hat{b}_n^{(k)})$. Our experiments indicate that for each instance there are often around 10 ("real" and "dummy") isogenies remaining to be computed, i.e. current $\sum_{i=1}^{n} \hat{b}_i^{(k)} \approx 10$. We compute the remaining isogenies by executing the unbatched component for each instance in sequence:

$$B^{(k)} \leftarrow \texttt{unbatched}(\hat{B}^{(k)}, (\hat{b}_1^{(k)}, \ldots, \hat{b}_n^{(k)}), (e'^{(k)}_1, \ldots, e'^{(k)}_n))$$

for $k \in \{1, \ldots, 8\}$. Following the concrete CSIDH-512 parameters of [CCC+19], for each instance, there are exactly 404 isogeny computations ("real" and "dummy") in the batched component while a few isogeny computations corresponding to non-zero $\hat{b}_i^{(k)}$ in the unbatched component. Thus, for each instance, the extra-dummy method computes only a few more extra isogenies, compared to the conventional OAYT-style implementation (Algorithm 2). Moreover, since the unbatched component has no failure probability, we conclude that the extra-dummy method has no failure probability either.

3.3 Extra-Infinity Method

We assume that eight different instances of Algorithm 2 are computed in parallel. In brief, the idea in our second method is that for each iteration of the inner loop (line 10 to line 26 of Algorithm 2), if the kernel generator is infinity in at least one of the eight instances, then we force all instances to execute the "else"-branch at line 22. In particular, we define the variable *inf* as:

$$inf = \texttt{isinfinity}(R^{(1)}) \mid \texttt{isinfinity}(R^{(2)}) \mid \ldots \mid \texttt{isinfinity}(R^{(8)}),$$

where $R^{(k)}$ denotes the kernel generator in the k^{th} simultaneous instance. If $inf = 0$, then $R^{(k)} \neq \infty$ for all $k \in \{1, \ldots, 8\}$ and all eight instances will enter the "if"-branch at line 14 in Algorithm 2, in order to compute a "real" or a "dummy" ℓ_i-isogeny. On the other hand, if $inf = 1$, then $R^{(k)} = \infty$ for at least one $k \in \{1, \ldots, 8\}$ and all instances will proceed to the "else"-branch. In this case, we need to execute the scalar multiplication $T_0^{(k)} = [\ell_i]T_0^{(k)}$, in addition to $T_1^{(k)} = [\ell_i]T_1^{(k)}$. This is not needed in Algorithm 2, because the ℓ_i-torsion part of the point T_0 has already vanished (since $R = \infty$). In our approach, when $inf = 1$, we are forcing all instances to proceed as if all $R^{(k)}$ were infinity, however the ℓ_i-torsion parts of some points $T_0^{(k)}$ have not vanished.

However, when $inf = 1$, the above idea imports some extra infinity-related computations, which in principal are not needed by every instance. In addition to the two scalar multiplications for $T_0^{(k)}$ and $T_1^{(k)}$ in the "else"-branch, these infinity-related computations may include more point generation operations (using the Elligator map at line 8), which will result in more scalar multiplications for the full order points $P_0^{(k)}, P_1^{(k)}$ (line 9) and the kernel generator $R^{(k)}$ (line 13). For this reason, we refer to this method as the extra-infinity method. Note also that the probability of $inf = 1$ is $1 - (1 - 1/\ell_i)^8$, which is considerably higher when ℓ_i is small (e.g. 3, 5, and 7). As a result, an increased number of $inf = 1$ cases (and hence, an increased number of infinity-related computations) is expected, which affects the efficiency of the extra-infinity method.

We mitigate this problem by considering again the hybrid mode. More precisely, we divide the primes $\boldsymbol{\ell} = (\ell_1, \ldots, \ell_n)$ into two subsets, $\boldsymbol{\ell}_{\text{batch}}$ for the batched component and $\boldsymbol{\ell}_{\text{unbatch}}$ for the unbatched component. $\boldsymbol{\ell}_{\text{unbatch}}$ contains only the smaller primes, specifically 3, 5, 7, 11, 13, 17 and 19 in our implementation, whereas $\boldsymbol{\ell}_{\text{batch}}$ includes the

remaining primes in $\boldsymbol{\ell}$. In the same way, the bound list $\boldsymbol{b} = (b_1, \ldots, b_n)$ and the secret exponent list $\boldsymbol{e^{(k)}} = (e_1^{(k)}, \ldots, e_n^{(k)})$ of each instance are split in two subsets, i.e. $\{\boldsymbol{b}_{\text{batch}}, \boldsymbol{b}_{\text{unbatch}}\}$ and $\{\boldsymbol{e}_{\text{batch}}^{(k)}, \boldsymbol{e}_{\text{unbatch}}^{(k)}\}$ for $k \in \{1, \ldots, 8\}$.

In the extra-infinity method, we first execute the batched component for eight parallel group action instances, to compute the isogenies for the larger primes with the corresponding subsets. The batched component outputs the resulting curve $\hat{B}^{(k)}$ for each instance:

$$\langle \hat{B}^{(1)}, \hat{B}^{(2)}, \ldots, \hat{B}^{(8)} \rangle \leftarrow \texttt{batched}(\langle A^{(1)}, A^{(2)}, \ldots, A^{(8)} \rangle, \boldsymbol{b}_{\text{batch}}, \langle \boldsymbol{e}_{\text{batch}}^{(1)}, \boldsymbol{e}_{\text{batch}}^{(2)}, \ldots, \boldsymbol{e}_{\text{batch}}^{(8)} \rangle)$$

Then we execute the unbatched component (such as Algorithm 2) sequentially in order to obtain the correct codomain curve for each instance:

$$B^{(k)} \leftarrow \texttt{unbatched}(\hat{B}^{(k)}, \boldsymbol{b}_{\text{unbatch}}, \boldsymbol{e}_{\text{unbatch}}^{(k)})$$

for $k \in \{1, \ldots, 8\}$. The number of total isogenies ("real" and "dummy") that are computed in the batched component for each instance is the sum of all b_i in $\boldsymbol{b}_{\text{batch}}$, which is 358 when considering the CSIDH-512 parameters of [CCC+19]. In the unbatched component, the number of total isogenies ("real" and "dummy") is 46, i.e. the sum of all b_i in $\boldsymbol{b}_{\text{unbatch}}$.

3.4 The Combination of Extra-Dummy and Extra-Infinity Methods

Before we introduce the combined method, we give a few more details on the extra-dummy and extra-infinity methods. We consider an example where in an iteration of the inner loop, n_{inf} of the eight kernel points $R^{(k)} = \infty$, in the batched component of both methods. The extra-dummy method will complete the computations of this iteration (from line 14 to 25 in Algorithm 3), and later it will compute n_{inf} "compensatory" isogenies with the unbatched component. On the other hand, the extra-infinity method will enter its "else"-branch to compute two scalar multiplications, for all eight instances, and it may later perform the other infinity-related computations, which are in theory needed by n_{inf} instances. Based on the operations that are carried out in each method, we observe that the extra-dummy method handles the infinity cases more efficiently than the extra-infinity method when only few $R^{(k)} = \infty$, hence when n_{inf} is small. On the other hand, the extra-infinity method seems to be more efficient in handling the infinity cases, when most of the eight $R^{(k)} = \infty$ (i.e. when n_{inf} is close to 8).

Based on the above observation, our idea is to combine the two approaches, aiming at obtaining a more efficient method. In order to do this, we set the variable n_{inf} as:

$$n_{\text{inf}} = \texttt{isinfinity}(R^{(1)}) + \texttt{isinfinity}(R^{(2)}) + \ldots + \texttt{isinfinity}(R^{(8)}),$$

so that $n_{\text{inf}} \in [0, 8]$. Taking Algorithm 3 to describe this combined method, after the computation at line 13 (where the kernel generator $R^{(k)}$ is computed), we add an if-else statement to check if the variable n_{inf} is within a predefined threshold n_{thld}. If $n_{\text{inf}} \leq n_{\text{thld}}$ which means there are few $R^{(k)} = \infty$, we do the same computations as in the extra-dummy method (line 14 to 22 of Algorithm 3). On the other hand, if $n_{\text{inf}} > n_{\text{thld}}$ which means there are more $R^{(k)} = \infty$, we proceed to the "else"-branch of the extra-infinity method and perform the two scalar multiplications $T_1^{(k)} = [\ell_i]T_1^{(k)}$ and $T_0^{(k)} = [\ell_i]T_0^{(k)}$. After this if-else statement, the operations at line 23 to line 25 will be performed. Additionally, the unbatched component of the combined method is the same as the one in the extra-dummy method. From our experiments, when the threshold $n_{\text{thld}} = 3$, this combined method provides the best performance, and particularly, it is slightly faster than the extra-dummy method and quite faster than the extra-infinity method.

3.5 Batching Dummy-Free-Style Group Actions

The methods that we have considered in Sections 3.2, 3.3, and 3.4 for batching CSIDH group actions are all based on the OAYT-style algorithm of [OAYT19] and require the computation of dummy isogenies. More precisely, recall that Algorithm 2 computes $|e_i|$ "real" and $b_i - |e_i|$ "dummy" isogenies. Such implementations are vulnerable to fault injection attacks. As observed in [CCC+19], an attacker can modify the codomain curve or the images of the points T_0, T_1 under the isogeny ϕ in Algorithm 2 (fault injections), and if the result is correct, he knows that a dummy isogeny is computed and thus $e_i = 0$. This is also true in Algorithm 3. If the same modification produces the correct result, then the attacker knows that either $e_i = 0$, or the current kernel generator $R = \infty$.

In [CCC+19], Cervantes-Vázquez, Chenu, Chi-Domínguez, De Feo, Rodríguez-Henríquez, and Smith presented a constant-time evaluation of the CSIDH class group action, without the need of dummy isogeny computations [CCC+19, Algorithm 5]. In their dummy-free approach, the secret exponents are chosen such that $e_i \in [-b_i, b_i]$ and $e_i \equiv b_i \pmod 2$. This choice allows the algorithm to compute the required $|e_i|$ isogenies, while for the remaining $b_i - |e_i|$, it alternates between the actions of the ideals $\mathfrak{l}_i$ and $\mathfrak{l}_i^{-1}$ and hence these $b_i - |e_i|$ isogenies cancel out (see [CCC+19, Section 5] for details). The dummy-free approach is based on the SIMBA-m-μ technique, where the implementation of [CCC+19] uses $m = 5$ and $\mu = 11$ (see Appendix A for the selection of parameters).

We argue that the three methods that we have introduced in Sections 3.2, 3.3, and 3.4 can also be used to batch the dummy-free-style algorithm of [CCC+19]. In particular, for the extra-dummy and the combined methods, we are still using dummy isogenies for the case where the kernel generator $R = \infty$, however, these dummy isogenies do not reveal any information about the secret exponent, since they depend only on the random kernel generator. For the extra-infinity method, we follow the same strategy as in Section 3.3, with the only difference being that the small prime list in the unbatched component is $\boldsymbol{\ell}_{\text{unbatch}} = (3, 5, 7, 11, 13, 17, 19, 23, 29)$. For the combined method in the dummy-free-style, the optimal threshold to achieve the best performance is $n_{\text{thld}} = 5$.

4 (8×1)-Way Prime-Field Arithmetic

In the batched components of all batching methods, we use the same curve and isogeny arithmetic implementation that we developed, based on [CCC+19], with minor optimizations to better fit the batched software. At a lower layer, we developed all the needed (8×1)-way[9] prime-field operations from scratch, using respectively AVX-512F and IFMA, by taking advantage of "limb-slicing" technique [CGT+21]. This section only studies our IFMA vectorized implementation of prime-field operations. Compared to the IFMA version, the AVX-512F implementation has two fundamental differences; (i) it uses `vpmuludq` instead of IFMA instructions to perform vector multiplication; (ii) the field element is represented in radix-2^{29} (with 18 limbs) due to the 32-bit multiplier.

4.1 Radix-2^{52} Limb Vector Set

IFMA naturally provides a reduced radix representation, namely radix-2^{52}, for large integers. Fortunately, radix-2^{52} is well-suited for CSIDH-512. Specifically, there are ten limbs for a 511-bit field element under radix-2^{52}. When considering a smaller radix, such as radix-2^{51}, the representation of an element will require at least eleven limbs, which leads to a higher consumption than radix-2^{52}. Formally, a field element f represented in

[9] The $(n \times m)$-way implementation performs n field operations in parallel, where each field operation is executed in a m-way parallel fashion and, thus, uses m elements of a vector.

radix-2^{52} is shown as:

$$f = f_0 + 2^{52} f_1 + 2^{104} f_2 + 2^{156} f_3 + 2^{208} f_4 + 2^{260} f_5 + 2^{312} f_6 + 2^{364} f_7 + 2^{416} f_8 + 2^{468} f_9,$$

where $0 \leq f_i < 2^{52}$ for $0 \leq i \leq 9$. This representation allows field elements to be up to 520-bit during computations.

The main data structure of our parallel software is a (8×1)-way *limb vector set*, which is composed of eight radix-2^{52} elements. Given eight integers $a, b, c, d, e, f, g, h \in \mathbb{F}_p$, a (8×1)-way limb vector set $\boldsymbol{V}$ is defined as:

$$\boldsymbol{V} = \langle a, b, c, d, e, f, g, h \rangle = \begin{Bmatrix} [a_0, b_0, c_0, d_0, e_0, f_0, g_0, h_0] \\ [a_1, b_1, c_1, d_1, e_1, f_1, g_1, h_1] \\ \vdots \\ [a_9, b_9, c_9, d_9, e_9, f_9, g_9, h_9] \end{Bmatrix} = (V_0, V_1, \ldots, V_9), \quad (1)$$

where each $V_i = [a_i, b_i, c_i, d_i, e_i, f_i, g_i, h_i]$ is called a limb vector. All the inputs and outputs of our (8×1)-way field operations are limb vector sets of which each limb is precisely 52 bits long. In terms of our field operations, we saved the final subtraction in Montgomery reduction, and our addition and subtraction perform reduction modulo $2p$ instead of p. This means all the integers inputted to or outputted from our field operations are in the range $[0, 2p-1]$. We use $\boldsymbol{P} = \langle p, p, p, p, p, p, p, p \rangle$ to denote an (8×1)-way limb vector set of prime p, and $\boldsymbol{Q} = 2 \times \boldsymbol{P} = \langle 2p, 2p, 2p, 2p, 2p, 2p, 2p, 2p \rangle$.

4.2 Field Addition and Subtraction

Field addition $\boldsymbol{Z} \leftarrow \boldsymbol{X} + \boldsymbol{Y} \bmod \boldsymbol{Q}$ is performed in three steps. At first, we add $\boldsymbol{X}$ with $\boldsymbol{Y}$ and store their sum in $\boldsymbol{Z}$. We then subtract $\boldsymbol{Q}$ from $\boldsymbol{Z}$, so there might be negative results in some lanes of $\boldsymbol{Z}$. Finally, we create a 512-bit mask vector where the 64-bit element is set to all-1 if the corresponding lane's integer in $\boldsymbol{Z}$ is negative, or to all-0 if non-negative. Through the bitwise AND between this mask vector and $\boldsymbol{Q}$, we add $2p$ to the negative integers in $\boldsymbol{Z}$ whereas add 0 to the non-negative integers. There are only two steps in the field subtraction $\boldsymbol{Z} \leftarrow \boldsymbol{X} - \boldsymbol{Y} \bmod \boldsymbol{Q}$, which first subtracts $\boldsymbol{Y}$ from $\boldsymbol{X}$ and then carries out a same final step of field addition.

4.3 Field Multiplication

Field multiplication has a significant impact on the performance of any isogeny-based cryptosystem and deserves special care. The field multiplication used in CSIDH is Montgomery multiplication [Mon85] which consists of two phases, namely integer multiplication and Montgomery reduction. There exist some different variants of Montgomery multiplication, often termed with their implementation fashion, such as Separated Operand Scanning (SOS) [KAK96], Finely Integrated Product Scanning (FIPS) [KAK96], Karatsuba-Comba-Montgomery (KCM) [GAST05] and etc. The number of instructions (including addition, multiplication, load/store) and memory consumption required for different variants are different. Taking these two factors into account, implementers can choose a proper variant when they develop software especially on resource-constrained devices like AVR or ARM microcontrollers. For the cost comparison of different variants we refer to [KAK96, Table 1] and [GAST05, Table 4]. However, things become more complicated when developing on a processor with more computing power. Considering our case, Ice Lake processor is equipped with ten execution ports (and various execution units), so the processor can execute several instructions simultaneously. Excluding the number of needed instructions and instruction latency/throughput, we are supposed to take instruction-level parallelism into account. The memory consumption receives less attention in this case since an Ice Lake machine usually possesses a GB-level memory.

Table 2: Information of our (8×1)-way implementation of field multiplication and squaring.

Field Operation	ISA	Integer Mul/Sqr	Reduction	Structure	Latency[†]
Multiplication	IFMA	Product-Scan.	Operand-Scan.	Interleaved	436
Squaring	IFMA	Product-Scan.	Operand-Scan.	Interleaved	344
Multiplication	AVX-512F	Karatsuba	Product-Scan.	Separated	848
Squaring	AVX-512F	Product-Scan.	Product-Scan.	Interleaved	723

[†] Latency (in clock cycles) is the execution time of eight parallel Montgomery multiplication/squaring instances, and it was measured on Ice Lake i3-1005G1 processor with turbo boost disabled.

Currently, most of the related AVX-512 implementations are designed for accelerating 1-, 2- or 4-way Montgomery multiplication. However, these optimization techniques are not ideally applicable to our 8-way software. We discuss these implementations in more detail in Section 5.1. Due to the "limb-slicing" pattern, our 8-way Montgomery multiplication essentially "duplicates" 1-way implementation to eight lanes by AVX-512 instructions. To the best of our knowledge, there are only two AVX-512 implementations of this type in the literature. Takahashi proposed both AVX-512F and IFMA implementation of 8-way Montgomery multiplication in [Tak20], but this software works on 62-bit and 52-bit operands, respectively, and not in the case of large integers. Buhrow, Gilbert, and Haider in [BGH21] presented a Block Product Scanning (BPS) variant of Montgomery multiplication, which is based on radix-2^{32} representation. An 8-way 512-bit BPS variant implemented with AVX-512F takes 189 clock cycles for each instance, which translates to 1512 clock cycles for a whole 8-way implementation.

In order to find an optimal field multiplication for our software, the best way is to develop the corresponding 8-way AVX-512 implementation of various Montgomery multiplication variants and select the fastest one among them. From an algorithmic viewpoint, all the variants differ in three aspects: (i) different methods to implement integer multiplication, e.g. operand-scanning, product-scanning or the advanced technique such as Karatsuba algorithm [KO63]; (ii) different methods to implement Montgomery reduction, e.g. operand-scanning or product-scanning; (iii) whether Montgomery reduction is *separated* from or *interleaved* with integer multiplication (and how it is interleaved in the latter case). For our IFMA version, we conducted experiments in which we developed a dozen of implementation candidates of 8-way field multiplication based on various combinations from above three aspects. Notably, our 8-way implementation candidates are not straightforwardly "duplicating" the ordinary 1-way x64 implementation of different variants (or we say combinations). We rather concentrated on improving instruction-level parallelism in each implementation candidate. In order to achieve this purpose, we tried to improve the ports utilization by optimizing dependency chains in the code. From our benchmarking results on Ice Lake processor, the implementation candidate with the lowest latency is shown in Algorithm 11 at Appendix C, which possesses a similar structure as Coarsely Integrated Hybrid Scanning (CIHS) [KAK96], and it serves as field multiplication in our 8-way IFMA software. Our field multiplication uses product-scanning for integer multiplication and utilizes operand-scanning to handle Montgomery reduction, and reduction is interleaved with the second outer loop of product-scanning integer multiplication (line 7 to 15 in Algorithm 11). As for our AVX-512F version, we carried out a similar procedure to evaluate also a dozen of AVX-512F implementation candidates. The optimal field multiplication in AVX-512F version switches to Karatsuba algorithm for integer multiplication since there are 18 limbs of each element, and uses a product-scanning Montgomery reduction that is separated from integer multiplication.

The information and latency of field multiplication in both versions are shown in Table 2, which indicates that our Karatsuba-based AVX-512F implementation outperforms the BPS variant in [BGH21]. We herein emphasize on the importance of using an optimal

field multiplication in such parallel AVX-512 software of an isogeny-based cryptosystem. In our experiments for IFMA version, the 8-way Separated Product Scanning (SPS) variant [LG14] or 8-way original FIPS variant [KAK96] (i.e. the one that has not been optimized for improving instruction-level parallelism) costs more than 700 clock cycles, i.e. taking 60% more CPU-cycles than Algorithm 11. From our experiments, using such unsuitable field multiplication and squaring implementation will finally result in up to 30% more CPU-cycles for CSIDH group action evaluation compared to the one using optimal variants.

4.4 Field Squaring

Most of the existing CSIDH implementations, e.g. [MCR19, OAYT19, CCC+19, CR20, HLKA20], take advantage of a same x64 assembly implementation of field operations originally from [CLM+18]. In this assembly implementation, a field squaring just invokes a field multiplication in which two operands are same. In other words, field squaring possesses the same latency as field multiplication. In this work, we developed a dedicated Montgomery squaring instead of directly using field multiplication. Specifically, compared to field multiplication, our field squaring utilizes a dedicated integer squaring instead of integer multiplication.

In essence, integer squaring is a special instance of multiplication, where all partial products in the form of $f_i \cdot f_j$ with $i \neq j$ appear twice due to $f_i \cdot f_j = f_j \cdot f_i$. A classic technique for optimizing squaring is to just compute these partial products once and double them, thereby saving numerous multiplication instructions. We develop our integer squaring by this classic technique, and again we developed many squaring candidates to obtain an optimal implementation. The information of our field squaring implementation is also listed in Table 2 where it proves a dedicated field squaring saves at least 15% CPU-cycles than a field multiplication. The algorithmic description of our IFMA 8-way Montgomery squaring is shown in Algorithm 12 at Appendix C.

5 Low-Latency Implementation

In our hybrid mode which is introduced in Section 3.1, the low-latency implementation of a single group action evaluation serves as the unbatched component and is needed by each instance. More importantly, this low-latency implementation can also be used in more applications e.g. accelerating the CSIDH key exchange protocol on the client side.

In this section we describe our (2×4)-way IFMA implementation, which is developed for accelerating a single group action evaluation and used as the unbatched component in our IFMA throughput-optimized software. In the case of AVX-512F, our experiments showed that the (2×4)-way AVX-512F implementation is slower than the x64 implementation of [CCC+19]. Hence, for our AVX-512F throughput-optimized software, we use the [CCC+19] implementation as the unbatched component.

5.1 (2×4)-Way Field Operations

Neither the structure nor the radix of the (2×4)-way limb vector set is the same compared to the (8×1)-way set. To be specific, we take advantage of (2×4)-way interleaved vectors combined with radix-2^{43} this time. The (2×4)-way limb vector set $\boldsymbol{V} = \langle a, b \rangle$ is defined as follows:

$$\boldsymbol{V} = \langle a, b \rangle = \left\{ \begin{matrix} [a_0, a_3, a_6, a_9\ , b_0, b_3, b_6, b_9\ ] \\ [a_1, a_4, a_7, a_{10}, b_1, b_4, b_7, b_{10}] \\ [a_2, a_5, a_8, a_{11}, b_2, b_5, b_8, b_{11}] \end{matrix} \right\} = (V_0, V_1, V_2). \tag{2}$$

Each limb vector $V_i = [a_i, a_{i+3}, a_{i+6}, a_{i+9}, b_i, b_{i+3}, b_{i+6}, b_{i+9}]$ contains four limbs from each integer a and b, and limbs are arranged in an interleaved pattern. The reason for using

radix-2^{43} instead of radix-2^{52} is now easily inferred from the Equation (2). It is because there will still remain three limb vectors if using radix-2^{52} (ten limbs for each integer), whereas radix-2^{43} offers more headroom in each limb (a_i or b_i) and is thus friendly for delaying the carry propagation. Similar to the (8×1)-way implementation, our (2×4)-way implementation also saves a final subtraction in Montgomery reduction and performs modulo $2p$ instead of p reduction in field addition and subtraction.

Mixed Addition and Subtraction. In curve and isogeny arithmetic, we can generally perform a pair of field addition and subtraction simultaneously, but not two additions or two subtractions due to sequential dependency. Therefore, it makes more sense to develop a parallel and mixed operation of addition and subtraction. We denote this mixed operation as "$\pm$". Formally, it works as $\langle r, s\rangle \leftarrow \langle a, b\rangle \pm \langle c, d\rangle$ where $r = a + c \bmod 2p$ and $s = b - d \bmod 2p$. In essence, this mixed operation executes the similar steps described in Section 4.2. At first, we construct two (2×4)-way limb vector sets $\langle c, 0\rangle$ and $\langle 2p, d\rangle$. We add $\langle a, b\rangle$ with $\langle c, 0\rangle$, and then subtract $\langle 2p, d\rangle$ from the sum to reach $\langle a + c - 2p, b - d\rangle$. The final step is similar to Section 4.2 in order to ensure the results of this mixed operation are in $[0, 2p - 1]$ by a mask vector.

Multiplication. As we mentioned in Section 4.3, some work has already been done for accelerating 1-, 2- or 4-way Montgomery multiplication or squaring with AVX-512. Several papers have been published which focus on using IFMA to accelerate 1-way large integer arithmetic such as integer multiplication [GK16, KG19] and Montgomery squaring [DG18].

Algorithm 4: (2×4)-way Montgomery multiplication using IFMA

Input: Operands $\boldsymbol{X} = \langle a, b\rangle$ and $\boldsymbol{Y} = \langle c, d\rangle$, prime $\boldsymbol{P} = \langle p, p\rangle$, $w = -p^{-1} \bmod 2^{43}$.
Output: Product $\boldsymbol{Z} = \langle e, f\rangle$ where $e = a \cdot c \cdot 2^{-516} \bmod 2p$ and $f = b \cdot d \cdot 2^{-516} \bmod 2p$.

$L_i \leftarrow$ `ZERO`, $H_i \leftarrow$ `ZERO` for $i \in \{0, 1, \ldots, 14\}$
$W \leftarrow$ `BCAST`(w), $M \leftarrow$ `BCAST`$(2^{43} - 1)$
for i **from** 0 **to** 11 **by** 1 **do**
  $T \leftarrow$ `PERM`$(Y_{i \bmod 3}, 0\text{x}55 \cdot \lfloor i/3 \rfloor)$
  for j **from** 0 **to** 2 **by** 1 **do**
    $L_{i+j} \leftarrow$ `MACLO`(L_{i+j}, T, X_j)
    $H_{i+j} \leftarrow$ `MACHI`(H_{i+j}, T, X_j)
  $U \leftarrow$ `AND`(`MACLO`(`ZERO`, W, `PERM`$(L_i, 0\text{x}00)), M)$
  for j **from** 0 **to** 2 **by** 1 **do**
    $L_{i+j} \leftarrow$ `MACLO`(L_{i+j}, U, P_j)
    $H_{i+j} \leftarrow$ `MACHI`(H_{i+j}, U, P_j)
  $L_{i+3} \leftarrow$ `ADD`$(L_{i+3}, 0\text{x}77, L_{i+3},$ `PERM`$(L_i, 0\text{x}39))$
  $L_{i+1} \leftarrow$ `ADD`$(L_{i+1}, 0\text{x}11, L_{i+1},$ `SHR`$(L_i, 43))$
  $L_{i+1} \leftarrow$ `ADD`$(L_{i+1},$ `SHL`$(H_i, 9))$
$L_{13} \leftarrow$ `ADD`$(L_{13},$ `SHL`$(H_{12}, 9))$
$L_{14} \leftarrow$ `ADD`$(L_{14},$ `SHL`$(H_{13}, 9))$
$Z_0 \leftarrow L_{12}$, $Z_1 \leftarrow L_{13}$, $Z_2 \leftarrow L_{14}$
for i **from** 0 **to** 1 **by** 1 **do**
  $C \leftarrow$ `SHR`$(Z_i, 43)$
  $Z_i \leftarrow$ `AND`(Z_i, M)
  $Z_{i+1} \leftarrow$ `ADD`(Z_{i+1}, C)
$C \leftarrow$ `SHR`$(Z_2, 43)$
$Z_2 \leftarrow$ `AND`(Z_2, M)
$Z_0 \leftarrow$ `ADD`$(Z_0, 0\text{x}\text{EE}, Z_0,$ `PERM`$(C, 0\text{x}93))$
return $\boldsymbol{Z} = (Z_0, Z_1, Z_2)$

Edamatsu and Takahashi in [ET20] presented an IFMA implementation of single large integer multiplication, which takes advantage of Karatsuba algorithm. Apart from the work on 1-way implementation acceleration, Orisaka, Aranha, and López presented a well-designed and fast (4×2)-way Montgomery multiplication for SIDH in [OAL18] by AVX-512F, and their approach is working on the 4-way interleaved vectors. We designed our (2×4)-way IFMA Montgomery multiplication based on the approach of [OAL18] with several modifications: (i) we use IFMA instructions to replace `vpmuldq` and save `vpaddq`; (ii) we apply our (2×4)-way limb vector set; (iii) we implement integer multiplication and reduction in interleaved fashion instead of separated one which is originally-used, because the interleaved fashion is measured to be faster than the separated one from our experiments. Our (2×4)-way field multiplication is described in Algorithm 4. Vector sets $\boldsymbol{L}$ and $\boldsymbol{H}$ respectively accumulate the partial products produced by `vpmadd52lo` and `vpmadd52hi`. Notably, excluding the computation at line 14, there is no dependency between $\boldsymbol{L}$ and $\boldsymbol{H}$ in the main loop (line 3 to 14), which benefits the efficient utilization of ports.

Squaring. Orisaka et al. did not present a dedicated integer squaring in [OAL18] but planned it as a future work. We herein propose a fast integer squaring by using the classic optimization technique that we described in Section 4.4. Our integer squaring can be slightly modified to fit any (2×4)- or (4×2)-way AVX-512 Montgomery squaring that uses interleaved vectors, e.g. the integer squaring needed in [OAL18]. Our method is described in Algorithm 5, which saves 24 IFMA instructions compared to an integer multiplication (corresponding to line 5 to 7 in Algorithm 4) which requires 72 IFMA instructions in total. We keep the output of Algorithm 5 in two sets $\boldsymbol{L}$ and $\boldsymbol{H}$, since our Montgomery reduction is designed to directly work on them. Our complete Montgomery squaring just replaces the integer multiplication part of Algorithm 4 by Algorithm 5.

Algorithm 5: (2×4)-way integer squaring using IFMA

Input: Operand $\boldsymbol{X} = \langle a, b \rangle$.
Output: $\boldsymbol{L} = \langle lo(e), lo(f) \rangle$ and $\boldsymbol{H} = \langle hi(e), hi(f) \rangle$, where $e = a^2$ and $f = b^2$.

1 $L_i \leftarrow$ `ZERO`, $H_i \leftarrow$ `ZERO` for $i \in \{0, 1, \ldots, 14\}$
2 **for** i **from** 0 **to** 11 **do**
3 $\quad k \leftarrow i \bmod 3$
4 $\quad T \leftarrow \texttt{PERM}(X_k, 0\text{x}55 \cdot \lfloor i/3 \rfloor)$
5 $\quad L_{i+k} \leftarrow \texttt{MACLO}(L_{i+k}, T, X_k)$
6 $\quad H_{i+k} \leftarrow \texttt{MACHI}(H_{i+k}, T, X_k)$
7 $\quad D \leftarrow \texttt{ADD}(T, T)$ // Skip this addition when $k = 2$
8 $\quad$ **for** j **from** $k + 1$ **to** 2 **by** 1 **do** // Skip this loop when $k = 2$
9 $\quad\quad L_{i+j} \leftarrow \texttt{MACLO}(L_{i+j}, D, X_j)$
10 $\quad\quad H_{i+j} \leftarrow \texttt{MACHI}(H_{i+j}, D, X_j)$
11 **return** $\boldsymbol{L} = (L_0, L_1, \ldots, L_{14})$, $\boldsymbol{H} = (H_0, H_1, \ldots, H_{14})$

5.2 Curve and Isogeny Arithmetic

Following [CCC+19], the curve arithmetic mainly includes y-coordinate point doubling, point addition and scalar multiplication (using addition chains) on twisted Edwards curve, whereas the isogeny operations contains y-coordinate isogeny computation and isogeny evaluation. Fortunately, all of the above five operations can be internally parallelized in 2-way, where the cost[10] switches from $i\boldsymbol{M} + j\boldsymbol{S}$ to $\frac{i}{2}\boldsymbol{M^2} + \frac{j}{2}\boldsymbol{S^2}$.

[10] $\boldsymbol{M}$, $\boldsymbol{S}$, $\boldsymbol{A}$ denote a 1-way field multiplication, squaring, addition/subtraction operation; $\boldsymbol{M^2}$, $\boldsymbol{S^2}$, $\boldsymbol{A^2}$ denote a 2-way field multiplication, squaring, mixed addition and subtraction operation, respectively.

The Elligator 2 map was originally introduced in [BHKL13] for generating random points on Montgomery curves and was modified in [CCC+19] for the twisted Edwards case. The latter takes as input the values $A_0 = a$ and $A_1 = a - d$ where $a, d \in \mathbb{F}_p$ are the coefficients of the curve $E_{a,d}$ in twisted Edwards form, a random $u \in \{2, \ldots, (p-1)/2\}$ which is used to derive the random curve points. Then it outputs two points $P_0 \in E_{a,d}[\pi - 1]$ and $P_1 \in E_{a,d}[\pi + 1]$. The method of [CCC+19] requires $8\boldsymbol{M} + 3\boldsymbol{S} + 16\boldsymbol{A}$ plus one square test for the Legendre symbol, which we slightly improved by saving $2\boldsymbol{A}$. Our 2-way implementation of the Elligator 2 map is based on [CCC+19], and it is presented in Algorithm 6, with total cost $5\boldsymbol{M^2} + 1\boldsymbol{S^2} + 9\boldsymbol{A^2}$ plus the square test for the Legendre symbol.

Algorithm 6: Our 2-way implementation of Elligator 2 map

Input: The values $(A_0 : A_1) = (a : a - d)$, $u \leftarrow \texttt{random}(\{2, \ldots, (p-1)/2\})$ and Montgomery constant $R = 2^{516} \bmod p$.

Output: A pair of points $P_0 \in E_{a,d}[\pi - 1]$ and $P_1 \in E_{a,d}[\pi + 1]$.

1	$\alpha \leftarrow 0$	
2	$t_0 \leftarrow A_0 - A_1$	
3	$t_0 \leftarrow A_0 + t_0$	
4	$t_0 \leftarrow t_0 + t_0 = A'$	
5	$t_1 \leftarrow u \times R^2$	$s_1 \leftarrow t_0 \times A_1$
6	$t_2 \leftarrow t_1^2$	$s_2 \leftarrow t_0^2$
7	$t_3 \leftarrow t_2 + R$	$s_3 \leftarrow t_2 - R$
8	$t_4 \leftarrow A_1 \times s_3$	$s_4 \leftarrow s_1 \times s_3$
9	$t_5 \leftarrow t_4 \times t_4$	$s_5 \leftarrow s_2 \times t_2$
10	$t_5 \leftarrow s_5 + t_5$	$s_0 \leftarrow \alpha - t_0 = -A'$
11	$t_6 \leftarrow s_4 \times t_5$	$s_6 \leftarrow s_0 \times t_2$
12	$\texttt{cswap}(\alpha, t_1, \texttt{isequal}(t_6, 0))$	// $\alpha \leftarrow u$ if $t_6 = 0$; else $\alpha \leftarrow 0$
13	$t_3 \leftarrow \alpha \times t_3$	$s_3 \leftarrow \alpha \times t_4$
14	$t_5 \leftarrow t_0 + s_3$	$s_5 \leftarrow s_6 - t_3$
15	$t_3 \leftarrow t_3 + t_6$	
16	$m \leftarrow \texttt{issquare}(t_3)$	// $m \leftarrow 1$ if t_3 is a square in $\mathbb{F}_p$; else $m \leftarrow 0$
17	$Y_0 \leftarrow t_5 - t_4$	$T_0 \leftarrow t_5 + t_4$
18	$Y_1 \leftarrow s_5 - t_4$	$T_1 \leftarrow s_5 + t_4$
19	$\texttt{cswap}(Y_0, Y_1, 1 - m)$	$\texttt{cswap}(T_0, T_1, 1 - m)$
20	**return** $P_0 = (Y_0 : T_0)$ and $P_1 = (Y_1 : T_1)$	

Specifically, the value u will be used to derive the random curve points, and the Montgomery constant R is used to map the random value u to the Montgomery domain. The algorithm first generates the two points using XZ-coordinate representation, namely $P_0 = (X_0 : Z_0)$ and $P_1 = (X_1 : Z_1)$ on the birationally equivalent Montgomery curve, $C'Y^2Z^2 = C'X^3Z + A'X^2Z^2 + C'XZ^3$, where $A' = 2(a + d)$ and $C' = a - d$. More precisely, the two points are defined as:

$$P_0 = (A' + \alpha C'(u^2 - 1) : C'(u^2 - 1)) \quad \text{and} \quad P_1 = (-A'u^2 - \alpha C'(u^2 - 1) : C'(u^2 - 1)),$$

where $\alpha = 0$, if $A' \neq 0$; and $\alpha = u$, if $A' = 0$[11].

Then, the algorithm converts the two points in twisted Edwards form, using YT-coordinate representation, at lines 17 and 18. This is relatively cheap, since it requires only $2\boldsymbol{A^2}$ operations, namely

$$P_0 = (Y_0 : T_0) = (X_0 - Z_0 : X_0 + Z_0) \quad \text{and} \quad P_1 = (Y_1 : T_1) = (X_1 - Z_1 : X_1 + Z_1).$$

[11] Given a point $P = (X : Y : Z)$ on a Montgomery curve in projective form, the XZ-coordinate representation of P is $P = (X : Z)$, where $x = X/Z$ is the x coordinate of P in affine form.

At line 16, we use the constant time function `issquare` to check whether the value

$$t_3 = \alpha(u^2+1) + A'C'(u^2-1)((A'u)^2 + (C'(u^2-1))^2)$$

is a square in $\mathbb{F}_p$. If it is a square ($m = 1$), then the generated point lies on $E_{a,d}$, otherwise ($m = 0$), the point lies on the quadratic twist of $E_{a,d}$. At the final step (line 19), the two points are swapped according to m, so that the point $P_0 \in E_{a,d}[\pi - 1]$ and $P_1 \in E_{a,d}[\pi + 1]$.

In Appendix B we present our 2-way algorithms for point doubling and (differential) point addition, as well as the algorithms for the ℓ-isogeny computation and evaluation, where all algorithms use YT-coordinate representation on twisted Edwards curves. At the top layer, we respectively implemented an OAYT-style group action and a dummy-free-style group action according to [CCC+19].

6 Experimental Results

We downloaded the original CSIDH software [CLM+18], all the OAYT-style and dummy-free-style constant-time CSIDH software including [OAYT19], [CCC+19], [CR20] and [HLKA20]. All the source codes are publicly available. In particular, the source code of [CLM+18] is available at CSIDH website[12], while the authors of [CCC+19], [CR20] and [HLKA20] provided their source code links in their articles. In addition, although the authors of [OAYT19] did not give the link of their source code in the article, we found the source code repository of the implementation in [OAYT19] on GitHub[13].

In order to figure out the real improvement of our work, we benchmarked our software and the CSIDH group action evaluation of all the above implementations on an Intel Core i3-1005G1 Ice Lake CPU clocked at 1.2 GHz. All source codes were compiled with GCC version 9.3.0 and Turbo boost was disabled during the performance measurements. The results of the OAYT-style implementations are shown in Table 3, where the speedup ratio is defined by comparing the "CPU-cycles/#instances" between the baseline and the specific implementation, i.e. the normalized throughput. We use [CCC+19] as baseline, because in this way we know precisely how much our vector processing techniques improve the results (note that [CCC+19] also served as baseline in other papers, e.g. [CR20, HLKA20]).

Table 3: Benchmark of OAYT-style CSIDH-512 group action implementations on Ice Lake.

	Reference	ISA	Impl.	#Inst.	CPU-Cycles	Speedup†
	[CLM+18]‡	x64	1-way	1	133.7 M	1.52×
	[OAYT19]	x64	1-way	1	248.4 M	0.82×
	[CCC+19]	x64	1-way	1	203.6 M	1.00×
	[CR20]	x64	1-way	1	195.0 M	1.04×
	[HLKA20]	x64	1-way	1	194.7 M	1.05×
This work	Low-Latency	AVX-512F	(2×4)-way	1	232.2 M	0.88×
	Extra-Dummy	AVX-512F	(8×1)-way	8	858.0 M	1.90×
	Extra-Infinity	AVX-512F	(8×1)-way	8	1003.9 M	1.62×
	Combined	AVX-512F	(8×1)-way	8	850.1 M	1.92×
This work	Low-Latency	IFMA	(2×4)-way	1	132.1 M	1.54×
	Extra-Dummy	IFMA	(8×1)-way	8	454.1 M	3.59×
	Extra-Infinity	IFMA	(8×1)-way	8	550.5 M	2.96×
	Combined	IFMA	(8×1)-way	8	446.9 M	3.64×

† The speedup ratio is calculated with "CPU-cycles/#instances" and uses [CCC+19] as the baseline.
‡ This implementation is not constant-time.

[12] https://csidh.isogeny.org/software.html
[13] https://github.com/hiroshi-onuki/constant-time-csidh

As shown in Table 3, our 2-way low-latency IFMA implementation has roughly the same latency as the original non-constant-time implementation in [CLM+18], and it is about 1.5 times faster than the x64 implementation of [CCC+19]. Our (8×1)-way IFMA implementation, when applied with the combined batching method, takes 446.9 M clock cycles for eight parallel instances, which represents a 3.64 times higher throughput compared to the x64 implementation in [CCC+19]. An analysis of the execution times of our (8×1)-way software shows that all the IFMA implementations are nearly 1.9 times faster than the corresponding AVX-512F implementations, which confirms that the IFMA extension indeed significantly accelerates CSIDH compared to general AVX-512F.

Table 4: Benchmark of dummy-free-style CSIDH-512 group action implementations on Ice Lake.

	Reference	ISA	Impl.	#Inst.	CPU-cycles	Speedup†
	[CCC+19]	x64	1-way	1	433.3 M	1.00×
	[CR20]	x64	1-way	1	394.3 M	1.10×
This work	Low-Latency	AVX-512F	(2×4)-way	1	447.0 M	0.97×
	Extra-Dummy	AVX-512F	(8×1)-way	8	1811.0 M	1.91×
	Extra-Infinity	AVX-512F	(8×1)-way	8	2172.3 M	1.60×
	Combined	AVX-512F	(8×1)-way	8	1801.4 M	1.92×
This work	Low-Latency	IFMA	(2×4)-way	1	253.8 M	1.71×
	Extra-Dummy	IFMA	(8×1)-way	8	967.0 M	3.58×
	Extra-Infinity	IFMA	(8×1)-way	8	1220.5 M	2.84×
	Combined	IFMA	(8×1)-way	8	955.3 M	3.63×

† The speedup ratio is calculated with "CPU-cycles/#instances" and uses [CCC+19] as the baseline.

The benchmarking results of dummy-free-style implementations are summarized in Table 4. These results show that our proposed batching methods still work efficiently when applied to the dummy-free-style CSIDH group action and can yield an up to 3.63 times higher throughput compared to the x64 implementation in [CCC+19].

Performance Evaluation and Analysis. Though AVX-512 can work on eight 64-bit elements simultaneously with a single instruction, the theoretical maximum speedup factor of an AVX-512 implementation (compared to x64 implementation) of isogeny-based crypto is actually far from eight. The main reason is the multiplier. An x64 implementation executed on an Ice Lake CPU has to use a single multiplier sequentially, but this multiplier can execute a full $(64 \times 64 \to 128)$-bit multiplication, which is very beneficial for the field arithmetic. On the other hand, AVX-512F can execute eight $(64 \times 64 \to 64)$-bit multiplications (`vpmullq`) or eight $(32 \times 32 \to 64)$-bit multiplications (`vpmuludq/vpmuldq`) in parallel, whereby the latter is typically used in multi-precision arithmetic. An AVX-512IFMA instruction can perform eight multiplications on 52-bit operands, but the result is either the lower half or the upper half of the eight 104-bit products, i.e. two IFMA instructions are necessary. Taking the multiplication of 512-bit integers using the schoolbook method as example, an x64 implementation needs $8^2 = 64$ mul instructions for one instance, while AVX-512F needs at least $16^2 = 256$ vectorized mul instructions for eight instances (a radix-2^{29} representation would even need more instructions) and AVX-512IFMA requires $10^2 \cdot 2 = 200$ IFMA instructions for eight instances. Since the CPI of these mul instructions is same on Ice Lake CPU (see [Int20]), the approximate speed-up (compared to an x64 implementation) of AVX-512F and AVX-512IFMA is a factor of 2.0 and 2.56, respectively. This is also the case for the Montgomery reduction. As we mentioned before in Section 4.3, the field multiplication significantly affects the performance of CSIDH so that the theoretical maximum speedup factor of AVX-512 for CSIDH group action evaluation should be far from 8. Taking this analysis into account, our throughput-optimized AVX-512 implementations have the expected speed-ups.

As for the latency-optimized implementation, a 2-way IFMA latency-optimized implementation of SIKEp503 was presented by Kostic and Gueron in [KG19], which is 1.72 times faster than the x64 assembly implementation of SIKEp503. We can thus conclude that our 2-way IFMA low-latency implementations (which achieve speed-up factors of 1.54 and 1.71, respectively) also correspond to the expected acceleration. There are several reasons that make the 2-way latency-optimized implementation less efficient than the throughput-optimized implementation, including (i) the overheads caused by aligning and blending AVX-512 vectors in 2-way curve and isogeny operations; (ii) the fact that some point operations (e.g. y-coordinate doubling and Elligator 2) can not be parallelized in an ideal[14] 2-way fashion due to the dependencies of internal field operations; (iii) some computations in the field operations (e.g. the complete carry propagation) cannot be parallelized in an ideal (2×4)-way fashion due to sequential dependencies of instructions; (iv) the instruction-level parallelism of (2×4)-way is lower than (8×1)-way since four limbs are stored in one vector. For all these reasons and because of the 32-bit multiplier in AVX-512F, the 2-way AVX-512F implementation is actually slower than the x64 implementation, which is confirmed by our experimental results.

7 Conclusions

Vector engines like Intel's AVX have become steadily more powerful from one generation to the next, not only because of the addition of new functionality, but also through the extension of the supported vector lengths. The expectation of this trend to continue in the coming years makes AVX an important platform for the implementation of PQC, in particular for computation-intensive isogeny-based cryptosystems. Although CSIDH has a couple of highly-desirable and unique features, the massive computational cost of the underlying class group action hampers its deployment in security protocols like TLS. In this paper we demonstrated how the enormous parallel processing power of AVX-512 can be exploited to, respectively, maximize the throughput of eight instances and minimize the latency of one instance of CSIDH-512 group action evaluation; the former alleviates the burden of server-side TLS processing, while the latter is beneficial on the client side. Our latency-optimized implementation makes CSIDH-512 group action evaluation roughly 1.5 times faster compared to a state-of-the-art non-vectorized x64 implementation that can resist timing attacks. On the other hand, by developing efficient batching methods for the class group action and combining them with highly-optimized (8×1)-way parallel field arithmetic based on the "limb-slicing" technique, we were able to achieve a 3.6-fold gain in throughput compared to a state-of-the-art x64 implementation of the CSIDH-512 group action evaluation. In light of this significant improvement, we expect that our batching methods are also highly beneficial for optimizing CSIDH-based digital signature schemes, such as CSI-FiSh [BKV19] and SeaSign [DG19], in which multiple independent class group actions are computed in the key generation, signing and verification processes.

The correct parameterization of CSIDH (including the order of the underlying prime field) to achieve NIST's security level 1 is currently still a topic of debate. It was suggested that, for level-1 security, the prime p should be much longer than 512 bits, e.g. 4096 bits [CCJR20]. Our CSIDH software was developed in a modular and parameterized way so as to reduce the effort when adapting it for other parameter sets since the point arithmetic (e.g. point addition, point doubling, scalar multiplication) and also certain parts of the field arithmetic can be re-used.

[14] We define an ideal 2-way parallelized fashion of point or isogeny operation has the cost of $\frac{i}{2}\boldsymbol{M^2} + \frac{j}{2}\boldsymbol{S^2} + \frac{k}{2}\boldsymbol{A^2}$ when the corresponding 1-way implementation has the cost of $i\boldsymbol{M} + j\boldsymbol{S} + k\boldsymbol{A}$.

Acknowledgements. The source code of the presented software is available online at https://gitlab.uni.lu/APSIA/AVX-CSIDH under GPLv3 license. We are grateful to the reviewers for their valuable comments and suggestions. We would also like to thank Tung Chou for his helpful feedback. Finally, we thank Chloe Martindale for the helpful discussions we had during the preparation of the paper. This work was supported by the Luxembourg National Research Fund (FNR) CORE project EquiVox (C19/IS/13643617/EquiVox/Ryan).

References

[ACR20] Gora Adj, Jesús-Javier Chi-Domínguez, and Francisco Rodríguez-Henríquez. On new Vélu's formulae and their applications to CSIDH and B-SIDH constant-time implementations. Cryptology ePrint Archive, Report 2020/1109, 2020. https://eprint.iacr.org/2020/1109.

[BDLS20] Daniel J. Bernstein, Luca De Feo, Antonin Leroux, and Benjamin Smith. Faster computation of isogenies of large prime degree. Cryptology ePrint Archive, Report 2020/341, 2020. https://eprint.iacr.org/2020/341.

[Ber06] Daniel J. Bernstein. Curve25519: New Diffie-Hellman speed records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *Public Key Cryptography — PKC 2006*, volume 3958 of *Lecture Notes in Computer Science*, pages 207–228. Springer Verlag, 2006.

[BGH21] Benjamin Buhrow, Barry Gilbert, and Clifton Haider. Parallel modular multiplication using 512-bit advanced vector instructions. In *Journal of Cryptographic Engineering*. Springer, 2021.

[BHKL13] Daniel J. Bernstein, Mike Hamburg, Anna Krasnova, and Tanja Lange. Elligator: Elliptic-curve points indistinguishable from uniform random strings. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, CCS'13, pages 967–980. Association for Computing Machinery, 2013.

[BKV19] Ward Beullens, Thorsten Kleinjung, and Frederik Vercauteren. Csi-fish: Efficient isogeny based signatures through class group computations. In Steven D. Galbraith and Shiho Moriai, editors, *Advances in Cryptology – ASIACRYPT 2019*, pages 227–247. Springer International Publishing, 2019.

[BLMP19] Daniel J. Bernstein, Tanja Lange, Chloe Martindale, and Lorenz Panny. Quantum circuits for the csidh: Optimizing quantum evaluation of isogenies. In Yuval Ishai and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2019*, pages 409–441. Springer International Publishing, 2019.

[BS20] Xavier Bonnetain and André Schrottenloher. Quantum security analysis of CSIDH. In Anne Canteaut and Yuval Ishai, editors, *Advances in Cryptology – EUROCRYPT 2020*, volume 12106 of *Lecture Notes in Computer Science*, pages 493–522. Springer, 2020.

[CCC+19] 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. Springer International Publishing, 2019.

[CCJR20] Jorge Chávez-Saab, Jesús-Javier Chi-Domínguez, Samuel Jaques, and Francisco Rodríguez-Henríquez. The SQALE of CSIDH: square-root vélu quantum-resistant isogeny action with low exponents. Cryptology ePrint Archive, Report 2020/1520, 2020. https://eprint.iacr.org/2020/1520.

[CGT+21] Hao Cheng, Johann Großschädl, Jiaqi Tian, Peter B. Rønne, and Peter Y. A. Ryan. High-throughput elliptic curve cryptography using avx2 vector instructions. In Orr Dunkelman, Michael J. Jacobson Jr., and Colin O'Flynn, editors, *Selected Areas in Cryptography – SAC 2020*, pages 698–719. Springer International Publishing, 2021.

[CJS14] Andrew M. Childs, David Jao, and Vladimir Soukharev. Constructing elliptic curve isogenies in quantum subexponential time. *J. Math. Cryptol.*, 8(1):1–29, 2014.

[CLM+18] 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. Springer International Publishing, 2018.

[Cos19] Craig Costello. Supersingular isogeny key exchange for beginners. Cryptology ePrint Archive, Report 2019/1321, 2019. https://eprint.iacr.org/2019/1321.

[CR20] Jesús-Javier Chi-Domínguez and Francisco Rodríguez-Henríquez. Optimal strategies for CSIDH. Cryptology ePrint Archive, Report 2020/417, 2020. https://eprint.iacr.org/2020/417.

[DeF17] Luca De Feo. Mathematics of Isogeny Based Cryptography. *CoRR*, abs/1711.04062, 2017.

[DG18] Nir Drucker and Shay Gueron. Fast modular squaring with avx512ifma. Cryptology ePrint Archive, Report 2018/335, 2018. https://eprint.iacr.org/2018/335.

[DG19] Luca De Feo and Steven D. Galbraith. SeaSign: Compact isogeny signatures from class group actions. In Yuval Ishai and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2019*, pages 759–789. Springer International Publishing, 2019.

[ET20] Takuya Edamatsu and Daisuke Takahashi. Accelerating large integer multiplication using intel avx-512ifma. In Sheng Wen, Albert Zomaya, and Laurence T. Yang, editors, *Algorithms and Architectures for Parallel Processing*, pages 60–74. Springer International Publishing, 2020.

[GAST05] Johann Großschädl, Roberto M. Avanzi, Erkay Savaş, and Stefan Tillich. Energy-efficient software implementation of long integer modular arithmetic. In Josyula R. Rao and Berk Sunar, editors, *Cryptographic Hardware and Embedded Systems — CHES 2005*, volume 3659 of *Lecture Notes in Computer Science*, pages 75–90. Springer Verlag, 2005.

[GK16] S. Gueron and V. Krasnov. Accelerating big integer arithmetic using intel ifma extensions. In *IEEE 23nd Symposium on Computer Arithmetic (ARITH)*, pages 32–38, 2016.

[HLKA20] Aaron Hutchinson, Jason LeGrow, Brian Koziel, and Reza Azarderakhsh. Further optimizations of csidh: A systematic approach to efficient strategies, permutations, and bound vectors. In Mauro Conti, Jianying Zhou, Emiliano Casalicchio, and Angelo Spognardi, editors, *Applied Cryptography and Network Security*, pages 481–501. Springer International Publishing, 2020.

[Int18] Intel Corporation. Intel 64 and ia-32 architectures optimization reference manual. Available online at https://software.intel.com/content/dam/develop/public/us/en/documents/64-ia-32-architectures-optimization-manual.pdf, 2018.

[Int20] Intel Corporation. 10th generation intel core processor based on ice lake microarchitecture instruction throughput and latency. Available online at https://software.intel.com/content/www/us/en/develop/download/10th-generation-intel-core-processor-instruction-throughput-and-latency-docs.html, 2020.

[JAC+20] David Jao, Reza Azarderakhsh, Matthew Campagna, Craig Costello, Luca De Feo, Basil Hess, Amir Jalali, Brian Koziel, Brian LaMacchia, Patrick Longa, Michael Naehrig, Joost Renes, Vladimir Soukharev, David Urbanik, Geovandro Pereira, Koray Karabina, and Aaron Hutchinson. Sike – supersingular isogeny key encapsulation. Available for download at https://csrc.nist.gov/CSRC/media/Projects/post-quantum-cryptography/documents/round-3/submissions/SIKE-Round3.zip, 2020.

[JAKJ19] Amir Jalali, Reza Azarderakhsh, Mehran Mozaffari Kermani, and David Jao. Towards optimized and constant-time csidh on embedded devices. In Ilia Polian and Marc Stöttinger, editors, *Constructive Side-Channel Analysis and Secure Design*, pages 215–231. Springer International Publishing, 2019.

[JHH+11] Keon Jang, Sangjin Han, Seungyeop Han, Sue B. Moon, and KyoungSoo Park. SSLShader: Cheap SSL acceleration with commodity processors. In David G. Andersen and Sylvia Ratnasamy, editors, *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2011)*. USENIX Association, 2011.

[KAK96] Çetin K. Koç, Tolga Acar, and Burton S. Kaliski. Analyzing and comparing Montgomery multiplication algorithms. *IEEE Micro*, 16(3):26–33, June 1996.

[KG19] Dusan Kostic and Shay Gueron. Using the new VPMADD instructions for the new post quantum key encapsulation mechanism SIKE. In Naofumi Takagi, Sylvie Boldo, and Martin Langhammer, editors, *26th IEEE Symposium on Computer Arithmetic, ARITH 2019, Kyoto, Japan, June 10-12, 2019*, pages 215–218. IEEE, 2019.

[KLM07] Phillip R. Kaye, Raymond Laflamme, and Michele Mosca. *An Introduction to Quantum Computing*. Oxford University Press, 2007.

[KO63] Anatoly A. Karatsuba and Yurii P. Ofman. Multiplication of multidigit numbers on automata. *Soviet Physics - Doklady*, 7(7):595–596, January 1963.

[Kup05] Greg Kuperberg. A subexponential-time quantum algorithm for the dihedral hidden subgroup problem. *SIAM J. Comput.*, 35(1):170–188, 2005.

[Kup13] Greg Kuperberg. Another subexponential-time quantum algorithm for the dihedral hidden subgroup problem. In Simone Severini and Fernando G. S. L. Brandão, editors, *8th Conference on the Theory of Quantum Computation,*

Communication and Cryptography, TQC 2013, May 21-23, 2013, Guelph, Canada, volume 22 of *LIPIcs*, pages 20–34. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2013.

[LG14] Zhe Liu and Johann Großschädl. New speed records for montgomery modular multiplication on 8-bit avr microcontrollers. In David Pointcheval and Damien Vergnaud, editors, *Progress in Cryptology – AFRICACRYPT 2014*, pages 215–234. Springer International Publishing, 2014.

[MCR19] 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. Springer International Publishing, 2019.

[Mon85] Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, April 1985.

[MR18] 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. Springer International Publishing, 2018.

[MS16] Dustin Moody and Daniel Shumow. Analogues of vélu's formulas for isogenies on alternate models of elliptic curves. *Math. Comput.*, 85(300):1929–1951, 2016.

[NS20] Kaushik Nath and Palash Sarkar. Efficient 4-way vectorizations of the Montgomery ladder. Cryptology ePrint Archive, Report 2020/378, 2020. https://eprint.iacr.org/2020/378.

[OAL18] Gabriell Orisaka, D. Aranha, and J. López. Finite field arithmetic using avx-512 for isogeny-based cryptography. In *XVIII Simpósio Brasileiro de Segurança da Informação e Sistemas Computacionais (SBSeg 2018)*, pages 49–56, 2018.

[OAYT19] Hiroshi Onuki, Yusuke Aikawa, Tsutomu Yamazaki, and Tsuyoshi Takagi. A faster constant-time algorithm of csidh keeping two points. Cryptology ePrint Archive, Report 2019/353, 2019. https://eprint.iacr.org/2019/353.

[Pei20] Chris Peikert. He gives c-sieves on the CSIDH. In Anne Canteaut and Yuval Ishai, editors, *Advances in Cryptology – EUROCRYPT 2020*, volume 12106 of *Lecture Notes in Computer Science*, pages 463–492. Springer, 2020.

[Reg04] Oded Regev. A subexponential time algorithm for the dihedral hidden subgroup problem with polynomial space. *arXiv preprint quant-ph/0406151*, 2004.

[RNSL17] Martin Roetteler, Michael Naehrig, Krysta M. Svore, and Kristin E. Lauter. Quantum resource estimates for computing elliptic curve discrete logarithms. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology – ASIACRYPT 2017*, volume 10625 of *Lecture Notes in Computer Science*, pages 241–270. Springer, 2017.

[SL21] Nigel Smart and Tanja Lange. Post-quantum cryptography: Current state and quantum mitigation. Technical report, European Union Agency for Cybersecurity (ENISA), 2021. Available for download at https://www.enisa.europa.eu/publications/post-quantum-cryptography-current-state-and-quantum-mitigation.

[Tak20] Daisuke Takahashi. Fast multiple montgomery multiplications using intel avx-512ifma instructions. In Osvaldo Gervasi, Beniamino Murgante, Sanjay Misra, Chiara Garau, Ivan Blečić, David Taniar, Bernady O. Apduhan, Ana Maria A. C. Rocha, Eufemia Tarantino, Carmelo Maria Torre, and Yeliz Karaca, editors, *Computational Science and Its Applications – ICCSA 2020*, pages 655–663. Springer International Publishing, 2020.

[Vél71] Jacques Vélu. Isogénies Entre Courbes Elliptiques. *Comptes Rendus de l'Académie des Sciences de Paris*, 273:238–241, 1971.

A Vectors of Small Primes and Exponent Bound

A.1 Instantiation of the OAYT-style algorithm

Based on [CCC+19], the ordering of the small primes $\boldsymbol{\ell} = (\ell_1, \ldots, \ell_{74})$ is:

$$\begin{aligned}\boldsymbol{\ell} = (&349, 347, 337, 331, 317, 313, 311, 307, 293, 283, 281, 277, 271,\\ &269, 263, 257, 251, 241, 239, 233, 229, 227, 223, 211, 199, 197,\\ &193, 191, 181, 179, 173, 167, 163, 157, 151, 149, 139, 137, 131,\\ &127, 113, 109, 107, 103, 101, 97, 89, 83, 79, 73, 71, 67,\\ &61, 59, 53, 47, 43, 41, 37, 31, 29, 23, 19, 17, 13,\\ &11, 7, 5, 3, 587, 373, 367, 359, 353)\end{aligned}$$

The secret key space or equivalently, the list of bounds for the secret exponents is:

$$\begin{aligned}\boldsymbol{b} = (&2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4,\\ &4, 4, 4, 4, 4, 4, 4, 4, 5, 5, 5, 5, 5, 5, 5, 6, 6, 6, 6, 6,\\ &7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 8, 9, 9, 9, 10, 10, 10,\\ &10, 9, 8, 8, 8, 7, 7, 7, 7, 7, 6, 5, 1, 2, 2, 2, 2)\end{aligned}$$

The OAYT-style implementation of [CCC+19] uses the SIMBA-3-8 technique.

A.2 Instantiation of the dummy-free-style algorithm

For the dummy-free algorithm, based on the instantiation of [CCC+19], the ordering of the small primes is the same as the OAYT-style, while the list of bounds for the secret exponents is:

$$\begin{aligned}\boldsymbol{b} = (&7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,\\ &8, 8, 8, 8, 8, 8, 8, 11, 11, 11, 11, 11, 11, 11, 11, 11,\\ &11, 11, 11, 11, 11, 11, 11, 11, 11, 13, 13, 13, 13, 13, 13, 13,\\ &13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13,\\ &13, 13, 13, 13, 13, 5, 7, 7, 7, 7)\end{aligned}$$

The dummy-free-style implementation of [CCC+19] uses the SIMBA-5-11 technique.

B 2-Way Curve and Isogeny Operations

Let $E_{a,d}$ be an elliptic curve over a finite field $\mathbb{F}_p$, represented in twisted Edwards form. In this section we present the 2-way implementation for y-coordinate doubling, (differential) addition, isogeny computation and the evaluation of an isogeny at a specific point on $E_{a,d}$. In all algorithms, any elliptic curve point is represented using projective YT-coordinates according to [CCC+19].

B.1 Point doubling

For a point $P = (Y_P : T_P)$ on the curve $E_{a,d}$, the point $R = [2]P$ is defined as:

$$Y_R = eY_P^2T_P^2 - (T_P^2 - Y_P^2)(eY_P^2 + a(T_P^2 - Y_P^2))$$
$$T_R = eY_P^2T_P^2 + (T_P^2 - Y_P^2)(eY_P^2 + a(T_P^2 - Y_P^2)),$$

where $e = a - d$ [CCC+19]. Algorithm 7 describes our 2-way doubling process using YT-coordinate arithmetic, with cost $2\boldsymbol{M^2} + 1\boldsymbol{S^2} + 3\boldsymbol{A^2}$.

Algorithm 7: 2-way implementation for YT-coordinate point doubling

Input: A point $P = (Y_P : T_P)$ on the curve $E_{a,d}$ and the values $(A_0 : A_1) = (a : a - d)$.
Output: The point $R = [2]P = (Y_R : T_R)$.

1	$t_0 \leftarrow Y_P^2$	$s_0 \leftarrow T_P^2$
2	$t_1 \leftarrow s_0 - t_0$	
3	$t_2 \leftarrow A_0 \times t_1$	$s_2 \leftarrow A_1 \times t_0$
4	$t_3 \leftarrow s_2 + t_2$	
5	$t_0 \leftarrow t_1 \times t_3$	$s_0 \leftarrow s_2 \times s_0$
6	$Y_R \leftarrow s_0 - t_0$	$T_R \leftarrow s_0 + t_0$
7	**return** $R = (Y_R : T_R)$	

B.2 Point addition

For point addition, we use the formulas that are presented in [CCC+19]. These formulas correspond to the differential addition using YT-coordinates on twisted Edwards curves. In particular, let $P = (Y_P : T_P)$ and $Q = (Y_Q : T_Q)$ be two points on the curve and let $PQ = P - Q = (Y_{P-Q} : T_{P-Q})$. The point $R = P + Q$ is derived from the coordinates of the points P, Q and PQ, using the formulas:

$$Y_R = (T_{P-Q} - Y_{P-Q})(Y_PT_Q + Y_QT_P)^2 - (T_{P-Q} + Y_{P-Q})(Y_PT_Q - Y_QT_P)^2$$
$$T_R = (T_{P-Q} - Y_{P-Q})(Y_PT_Q + Y_QT_P)^2 + (T_{P-Q} + Y_{P-Q})(Y_PT_Q - Y_QT_P)^2,$$

Algorithm 8 is the 2-way (differential) addition process using YT-coordinate arithmetic with cost $2\boldsymbol{M^2} + 1\boldsymbol{S^2} + 3\boldsymbol{A^2}$.

Algorithm 8: 2-way implementation for YT-coordinate (differential) addition

Input: Points $P = (Y_P : T_P)$, $Q = (Y_Q : T_Q)$ and $PQ = (Y_{P-Q} : T_{P-Q})$ on $E_{a,d}$.
Output: The point $R = P + Q = (Y_R : T_R)$.

1	$t_0 \leftarrow T_{P-Q} + Y_{P-Q}$	$s_0 \leftarrow T_{P-Q} - Y_{P-Q}$
2	$t_1 \leftarrow Y_P \times T_Q$	$s_1 \leftarrow Y_Q \times T_P$
3	$t_2 \leftarrow t_1 - s_1$	$s_2 \leftarrow t_1 + s_1$
4	$t_2 \leftarrow t_2^2$	$s_2 \leftarrow s_2^2$
5	$t_1 \leftarrow t_0 \times t_2$	$s_1 \leftarrow s_0 \times s_2$
6	$Y_R \leftarrow s_1 - t_1$	$T_R \leftarrow s_1 + t_1$
7	**return** $R = (Y_R : T_R)$	

B.3 ℓ-isogeny computation

Algorithm 9 describes the procedure for computing an isogeny of some odd degree ℓ, using YT-coordinate representation on twisted Edwards curves. The algorithm takes as input the

values $A_0 = a, A_1 = a - d$, where $E_{a,d}$ is an elliptic curve in twisted Edwards form, a point $P = (Y_P : T_P)$ and the degree of the isogeny $\ell = 2k + 1$. Then the algorithm computes the codomain curve $E_{a',d'}$ and the list of points $\{P_1 = (Y_1 : T_1), \ldots, P_k = (Y_k : T_k)\}$, where $P_i = [i]P$, for each $i \in \{1, \ldots, k\}$. Based on the work of Moody and Shumow [MS16], the coefficients of the codomain curve are defined as:

$$a' = a^\ell \left(\prod_{i=1}^{k} T_i \right)^8 \quad \text{and} \quad d' = d^\ell \left(\prod_{i=1}^{k} Y_i \right)^8 .$$

Algorithm 9 outputs the values $A'_0 = a'$ and $A'_1 = a' - d'$, as well as the list of points $\{P_1 = (Y_1 : T_1), \ldots, P_k = (Y_k : T_k)\}$.

Algorithm 9: 2-way ℓ-isogeny computation, with $\ell = 2k + 1$

Input: Point $P = (Y_P : T_P)$, $(A_0 : A_1) = (a : a - d)$, $\ell = 2k + 1$.
Output: Curve $(A'_0 : A'_1) = (a' : a' - d')$, list $\{P_1 = (Y_1 : T_1), \ldots, P_k = (Y_k : T_k)\}$.

$(\ell)_2 \leftarrow (b_n, \ldots, b_0)_2$ // binary representation of ℓ
$t_1 \leftarrow A_0 - A_1$
$t_0 \leftarrow A_0$, $Y_1 \leftarrow Y_P$, $Y_Q \leftarrow Y_P$ $\quad$ $s_0 \leftarrow t_1$, $T_1 \leftarrow T_P$, $T_Q \leftarrow T_P$
$P_2 \leftarrow [2]P = (Y_2 : T_2)$ // point doubling
for i **from** 3 **to** k **by** 1 **do**
  $Y_Q \leftarrow Y_Q \times Y_{i-1}$ $\quad$ $T_Q \leftarrow T_Q \times T_{i-1}$
  $P_i \leftarrow P_{i-1} + P = (Y_i : T_i)$ // point addition
end
$t_2 \leftarrow Y_Q \times Y_k$ $\quad$ $s_2 \leftarrow T_Q \times T_k$
$m \leftarrow \texttt{isequal}(\ell, 3)$
$\texttt{cswap}(Y_Q, t_2, 1 - m)$ $\quad$ $\texttt{cswap}(T_Q, s_2, 1 - m)$
for i **from** $n - 1$ **to** 0 **by** 1 **do**
  $t_0 \leftarrow t_0^2$ $\quad$ $s_0 \leftarrow s_0^2$
  if $b_i = 1$ **then**
    $t_0 \leftarrow t_0 \times A_0$ $\quad$ $s_0 \leftarrow s_0 \times t_1$
  end
end
for i **from** 0 **to** 2 **by** 1 **do**
  $Y_Q \leftarrow Y_Q^2$ $\quad$ $T_Q \leftarrow T_Q^2$
end
$A'_0 \leftarrow t_0 \times T_Q$ $\quad$ $A'_1 \leftarrow s_0 \times Y_Q$
$A'_1 \leftarrow A'_0 - A'_1$
return $(A'_0 : A'_1)$, $\{P_1 = (Y_1 : T_1), \ldots, P_k = (Y_k : T_k)\}$

B.4 ℓ-isogeny evaluation

Algorithm 10 computes the image $R = (Y_R : T_R)$ of a point $Q = (Y_Q : T_Q)$ under an isogeny of odd degree $\ell = 2k + 1$, that is computed with Algorithm 9. In particular, the algorithm takes as input the point Q and the list of points $\{P_1 = (Y_1 : T_1), \ldots, P_k = (Y_k : T_k)\}$ that was computed in Algorithm 9, where P is the kernel point and $P_i = [i]P$, for each $i \in \{1, \ldots, k\}$. Based on the formulas provided in [CCC+19], the image R of Q has coefficients:

$$Y_R = (T_Q + Y_Q) \left(\prod_{i=1}^{k} (Y_Q T_i + T_Q Y_i) \right)^2 - (T_Q - Y_Q) \left(\prod_{i=1}^{k} (Y_Q T_i - T_Q Y_i) \right)^2$$

$$T_R = (T_Q + Y_Q) \left(\prod_{i=1}^{k} (Y_Q T_i + T_Q Y_i) \right)^2 + (T_Q - Y_Q) \left(\prod_{i=1}^{k} (Y_Q T_i - T_Q Y_i) \right)^2$$

The total cost for our 2-way isogeny evaluation, that is described in Algorithm 10 is: $2k\boldsymbol{M^2} + 1\boldsymbol{S^2} + (k+2)\boldsymbol{A^2}$.

Algorithm 10: 2-way ℓ-isogeny evaluation, with $\ell = 2k+1$

Input: Point $Q = (Y_Q : T_Q)$ and list $\{P_1 = (Y_1 : T_1), \ldots, P_k = (Y_k : T_k)\}$.
Output: The point $R = \phi(Q) = (Y_R : T_R)$.

1 $t_0 \leftarrow Y_Q \times T_1$ | $s_0 \leftarrow T_Q \times Y_1$
2 $Y_R \leftarrow t_0 + s_0$ | $T_R \leftarrow t_0 - s_0$
3 **for** i **from** 2 **to** k **by** 1 **do**
4 $\quad t_0 \leftarrow Y_Q \times T_i$ | $s_0 \leftarrow T_Q \times Y_i$
5 $\quad t_1 \leftarrow t_0 + s_0$ | $s_1 \leftarrow t_0 - s_0$
6 $\quad Y_R \leftarrow Y_R \times t_1$ | $T_R \leftarrow T_R \times s_1$
7 **end**
8 $Y_R \leftarrow Y_R^2$ | $T_R \leftarrow T_R^2$
9 $t_0 \leftarrow T_Q + Y_Q$ | $s_0 \leftarrow T_Q - Y_Q$
10 $t_0 \leftarrow t_0 \times Y_R$ | $s_0 \leftarrow s_0 \times T_R$
11 $Y_R \leftarrow t_0 - s_0$ | $T_R \leftarrow t_0 + s_0$
12 **return** $R = (Y_R : T_R)$

C (8×1)-Way IFMA Field Multiplication and Squaring

Algorithm 11 has a similar structure as the Coarsely Integrated Hybrid Scanning (CIHS) method described in [KAK96]. The differences are that the first outer loop (line 3 to 6) of Algorithm 11 is product-scanning while the first outer loop in CIHS is operand-scanning, and Algorithm 11 takes more memory.

Algorithm 11: (8×1)-way Montgomery multiplication using IFMA

Input: Operands $\boldsymbol{X}$ and $\boldsymbol{Y}$, prime $\boldsymbol{P}$, $w = -p^{-1} \bmod 2^{52}$
Output: Product $\boldsymbol{Z} = \boldsymbol{X} \times \boldsymbol{Y} \times 2^{-520} \bmod \boldsymbol{Q}$

1 $Z_i \leftarrow$ `ZERO` for $i \in \{0, 1, \ldots, 19\}$
2 $W \leftarrow \texttt{BCAST}(w)$, $M \leftarrow \texttt{BCAST}(2^{52}-1)$
3 **for** i **from** 0 **to** 9 **by** 1 **do**
4 $\quad$ **for** j **from** 0 **to** i **by** 1 **do**
5 $\quad\quad Z_i \leftarrow \texttt{MACLO}(Z_i, X_j, Y_{i-j})$
6 $\quad\quad Z_{i+1} \leftarrow \texttt{MACHI}(Z_{i+1}, X_j, Y_{i-j})$
7 **for** i **from** 0 **to** 9 **by** 1 **do**
8 $\quad$ **for** j **from** $i+1$ **to** 9 **by** 1 **do** // Skip this loop when $i = 9$
9 $\quad\quad Z_{i+10} \leftarrow \texttt{MACLO}(Z_{i+10}, X_j, Y_{i-j+10})$
10 $\quad\quad Z_{i+11} \leftarrow \texttt{MACHI}(Z_{i+11}, X_j, Y_{i-j+10})$
11 $\quad T \leftarrow \texttt{MACLO}(\texttt{ZERO}, Z_i, W)$
12 $\quad$ **for** j **from** 0 **to** 9 **do**
13 $\quad\quad Z_{i+j} \leftarrow \texttt{MACLO}(Z_{i+j}, T, P_j)$
14 $\quad\quad Z_{i+j+1} \leftarrow \texttt{MACHI}(Z_{i+j+1}, T, P_j)$
15 $\quad Z_{i+1} \leftarrow \texttt{ADD}(Z_{i+1}, \texttt{SHR}(Z_i, 52))$
16 **for** i **from** 10 **to** 18 **by** 1 **do**
17 $\quad Z_{i+1} \leftarrow \texttt{ADD}(Z_{i+1}, \texttt{SHR}(Z_i, 52))$
18 $\quad Z_{i-10} \leftarrow \texttt{AND}(Z_i, M)$
19 $Z_9 \leftarrow Z_{19}$
20 **return** $\boldsymbol{Z} = (Z_0, Z_1, \ldots, Z_9)$

Algorithm 12: (8×1)-way Montgomery squaring using IFMA

Input: Operand $\boldsymbol{X}$, prime $\boldsymbol{P}$, $w = -p^{-1} \bmod 2^{52}$
Output: Product $\boldsymbol{Z} = \boldsymbol{X}^2 \times 2^{-520} \bmod \boldsymbol{Q}$

$Z_i \leftarrow \texttt{ZERO}$ for $i \in \{0, 1, \ldots, 19\}$
$W \leftarrow \texttt{BCAST}(w)$, $M \leftarrow \texttt{BCAST}(2^{52} - 1)$
for i **from** 1 **to** 9 **by** 1 **do**
    for j **from** 0 **to** $\lfloor (i-1)/2 \rfloor$ **by** 1 **do**
        $Z_i \leftarrow \texttt{MACLO}(Z_i, X_j, X_{i-j})$
        $Z_{i+1} \leftarrow \texttt{MACHI}(Z_{i+1}, X_j, X_{i-j})$
    $Z_i \leftarrow \texttt{ADD}(Z_i, Z_i)$
    if i *is odd* **then**
        $k \leftarrow (i-1)/2$
        $Z_{i-1} \leftarrow \texttt{MACLO}(Z_{i-1}, X_k, X_k)$
        $Z_i \leftarrow \texttt{MACHI}(Z_i, X_k, X_k)$
for i **from** 0 **to** 9 **by** 1 **do**
    for j **from** $\lfloor i/2 \rfloor + 6$ **to** 9 **by** 1 **do** // Skip this loop when $i = 8$ or 9
        $Z_{i+10} \leftarrow \texttt{MACLO}(Z_{i+10}, X_j, Y_{i-j+10})$
        $Z_{i+11} \leftarrow \texttt{MACHI}(Z_{i+11}, X_j, Y_{i-j+10})$
    $Z_{i+10} \leftarrow \texttt{ADD}(Z_{i+10}, Z_{i+10})$
    if i *is odd* **then**
        $k \leftarrow (i+9)/2$
        $Z_{i+9} \leftarrow \texttt{MACLO}(Z_{i+9}, X_k, X_k)$
        $Z_{i+10} \leftarrow \texttt{MACHI}(Z_{i+10}, X_k, X_k)$
    $T \leftarrow \texttt{MACLO}(\texttt{ZERO}, Z_i, W)$
    for j **from** 0 **to** 9 **do**
        $Z_{i+j} \leftarrow \texttt{MACLO}(Z_{i+j}, T, P_j)$
        $Z_{i+j+1} \leftarrow \texttt{MACHI}(Z_{i+j+1}, T, P_j)$
    $Z_{i+1} \leftarrow \texttt{ADD}(Z_{i+1}, \texttt{SHR}(Z_i, 52))$
for i **from** 10 **to** 18 **by** 1 **do**
    $Z_{i+1} \leftarrow \texttt{ADD}(Z_{i+1}, \texttt{SHR}(Z_i, 52))$
    $Z_{i-10} \leftarrow \texttt{AND}(Z_i, M)$
$Z_9 \leftarrow Z_{19}$
return $\boldsymbol{Z} = (Z_0, Z_1, \ldots, Z_9)$