Improved Classical Cryptanalysis of the Computational Supersingular Isogeny Problem

Craig Costello[1], Patrick Longa[1], Michael Naehrig[1], Joost Renes[2⋆], and Fernando Virdia[3⋆⋆]

[1] Microsoft Research, Redmond, WA, USA
{craigco,plonga,mnaehrig}@microsoft.com
[2] Digital Security Group, Radboud University, Nijmegen, The Netherlands
j.renes@cs.ru.nl
[3] Information Security Group, Royal Holloway, University of London, UK
fernando.virdia.2016@rhul.ac.uk

Abstract. Two recent papers have made significant advances towards a better understanding of the concrete hardness of the computational supersingular isogeny (CSSI) problem; this problem underlies the supersingular isogeny key encapsulation (SIKE) protocol, which is the only isogeny-based submission to the NIST post-quantum standardization effort. The first paper, by Adj, Cervantes-Vázquez, Chi-Domínguez, Menezes and Rodríguez-Henríquez, argues that the van Oorschot-Wiener (vOW) parallel collision finding algorithm is the best choice of classical algorithm for CSSI, and subsequently shows that the SIKE team were too conservative in their classical security estimates. The second paper, by Jaques and Schanck, gives an in-depth analysis into the best known quantum algorithms for CSSI, concluding that quantum algorithms do not achieve a significant advantage over the vOW algorithm and showing that the SIKE team were overly conservative in their quantum security analysis as well. Both papers agree that significantly smaller parameters could be used in the SIKE proposal to achieve NIST's security requirements.

The main contribution of this work is an implementation of the van Oorschot-Wiener algorithm. We present a number of novel improvements, both to practical instantiations of the generic vOW algorithm and to its instantiation in the context of SIKE, that culminate in an improved classical cryptanalysis of CSSI. Subsequently, we study a set of three SIKE parameterizations – one from the original proposal, SIKEp751, and two from the two papers above, SIKEp434 and SIKEp610 – that we endorse for inclusion in future versions of the SIKE proposal. We provide assembly-optimized performance benchmarks for these parameter sets, which show that the SIKE protocol can be computed in approximately 6.5, 15.6 and 26 milliseconds on a 3.4GHz Intel Skylake processor at NIST's levels 1, 3, and 5, respectively.

Keywords: Post-quantum cryptography, supersingular elliptic curves, isogenies, SIDH, SIKE, parallel collision search, van Oorschot-Wiener algorithm.

1 Introduction

In their call for proposals of post-quantum public key algorithms, the United States National Institute of Standards and Technology (NIST) specifies five target security levels [Nat16, p. 18]. Levels 1, 3, and 5 are satisfied by a cryptosystem if any attack that breaks the relevant security definition requires computational resources comparable to or greater than those required for respective key searches on AES128, AES192 and AES256; levels 2 and 4 are satisfied if the required computational resources are at least as much as those required for respective collision searches on SHA256 and SHA384.

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

⋆⋆ Partially supported by the EPSRC and the UK government as part of the Centre for Doctoral Training in Cyber Security at Royal Holloway, University of London (EP/P009301/1). Part of this work was done while Fernando was an intern at Microsoft Research.

The supersingular isogeny key encapsulation (SIKE) proposal [JAC+17] – the actively secure version of Jao and De Feo's SIDH key exchange [JDF11] – is one of 17 second round candidate public key encryption or key establishment proposals, and the only proposal whose security is based on the computational supersingular isogeny (CSSI) problem. Currently, the best known classical and quantum attacks on the CSSI problem are generic *claw finding attacks*: given two functions $f\colon A \to C$ and $g\colon B \to C$ with domains of equal size, the *claw finding problem* is to find a pair (a, b) such that $f(a) = g(b)$. The original security analysis by Jao and De Feo [JDF11, §5.2] estimates the complexity of the CSSI problem by assuming the optimal black-box *asymptotic* complexities for the claw finding problem: classically, it can be solved in $O(|A| + |B|)$ time using $O(|A|)$ space, while on a quantum computer Tani [Tan09] gives an algorithm that instead runs in $O(\sqrt[3]{|A||B|})$ time. Following Jao and De Feo, the SIKE team used these asymptotics to specify three parameterizations that are intended to meet the requirements for security levels 1, 3, and 5.

Shortly after the NIST submission deadline, a 2018 paper by Adj, Cervantes-Vázquez, Chi-Domínguez, Menezes and Rodríguez-Henríquez [ACVCD+19] made a significant step towards a better understanding of the concrete classical complexity of the CSSI problem. Their paper shows that, despite its higher running time, the classical van Oorschot-Wiener (vOW) golden collision finding algorithm [vOW99] has significantly lower space requirements than the meet-in-the-middle claw finding algorithm, and their analysis concludes that the vOW algorithm should instead be used to assess the security of SIDH/SIKE against (known) classical attacks. Indeed, the best classical AES key search algorithms only require a modest amount of storage, so a fair and correct analysis of cryptosystems (with respect to the security levels 1, 3 and 5) must take into account the available time/memory trade-offs. Consequently, Adj *et al.* fix a conservative upper bound on a storage capacity that will be prohibitively costly for the foreseeable future, i.e., 2^{80} *units* of storage, and then use this storage to analyze the classical runtime of vOW against the meet-in-the-middle algorithm. Their analysis ultimately shows that the classical security estimates used by the SIKE team were rather conservative, and significantly smaller parameters can be used to achieve the requisite level of classical security corresponding to NIST's levels 1, 3, and 5.

In a very recent and complementary paper, Jaques and Schank [JS19] give an in-depth analysis into the complexity of the best known *quantum* algorithms for the CSSI problem. Similar to the classical analysis of Adj *et al.*, they conclude that the quantum security estimates in the SIKE proposal were extremely conservative. In fact, Jaques and Schank's precise quantum complexity analysis of the CSSI problem [JS19, §6] uses models of computation that allow to directly compare quantum algorithms with classical algorithms and shows that the best known quantum algorithms do not achieve a significant advantage over the classical vOW algorithm. In certain attack scenarios, they even conclude that it is the classical security that is the limiting factor, and that the vOW algorithm in the classical model is the best known (classical or quantum!) algorithm to solve CSSI. Thus, the precise, real-world complexity of the vOW parallel collision search algorithm is paramount in the discussion of future parameters for SIDH/SIKE.

Contributions. We present an implementation of the van Oorschot-Wiener algorithm that we aim to be a step towards a real-world, large-scale cryptanalytic effort. Our work extends that of Adj *et al.* by introducing novel improvements to implementations of the generic vOW collision finding algorithm and exploiting several optimizations specific to the contexts of SIDH and SIKE. Our extensions and improvements to the vOW implementation and analysis in [ACVCD+19] include:

– *Faster collision checking.* One of the main steps in the vOW algorithm is to check whether a given collision is the *golden collision* or not (see §2). This check occurs often enough that, experimentally, our optimized version of generic vOW found that collision checking constitutes close to 20% of the entire vOW algorithm (and this aligns with van Oorschot and Wiener's analysis, which also states 20% [vOW99, §4.2]). We give a novel method of performing this check much more efficiently. This algorithm is based on a method of cycle finding due to Sedgewick, Szymanski and Yao [SSY82], and it temporarily uses a small amount of local storage (this amount can be input dynamically as a parameter) during the random walks to accelerate the checking of a collision, once a collision is detected – see §3.4.

- *Precomputation.* Generic collision finding algorithms like vOW are often implemented to target high-speed symmetric primitives. One difference between those applications, and that of isogenies, is that the computation of large-degree isogenies becomes the overwhelming bottle-neck of the random walk functions, and thus of the overall algorithm. Subsequently, speeding up the isogeny computations translates directly to a similar speedup of the entire collision finding process. We show how to exhaust any available local memory to achieve such speedups via the precomputation of parts of the full isogeny tree – see §3.3.
- *SIKE-specific optimizations.* Although the best algorithm for the general CSSI problem is generic (in the sense that there are no better known choices of algorithm that exploit the underlying structure of the CSSI problem), we take advantage of multiple optimizations that apply to the specific instantiations defined in the SIKE proposal [JAC+17]. Firstly, we show how to optimally exploit their choice of the starting curve as a subfield curve, by defining our deterministic walk on (conjugate) classes of j-invariants; this modified walk is analogous to the walk that exploits the negation map in the application of the Pollard rho algorithm to the ECDLP [WZ99] – see §3.1. Secondly, we show how to exploit the fact that the SIKE proposal does not randomize the isomorphism class of the output curve (this possibility was already pointed out by De Feo, Jao and Plût [DFJP14]), by using the subsequent knowledge of the final dual isogeny within the vOW algorithm – see §3.1. We quantify the precise security loss suffered by these choices, and present the alternative choices that could be made to avoid this loss in the SIKE proposal – see §5.2.
- *Compression of distinguished points.* The runtime of vOW crucially depends on the total number of distinguished points that can be stored and accessed globally. Our implementation makes a natural choice of the distinguishing property that allows distinguished points to be stored in compressed form. While this observation is rather straightforward (see [vOW99, §6]), its application in our scenario (especially for larger CSSI instances where the proportion of distinguished elements becomes smaller) gives appreciable speedups to the runtime of vOW and improves the runtime of our algorithms compared to those of Adj *et al.* [ACVCD+19].

Based on the above extensions to [ACVCD+19], and the aggressive optimization of vOW described within this paper, we perform a range of experiments to solve CSSI at varying degrees that further reinforce the security estimates claimed by Adj *et al.* and by Jaques and Schanck. Subsequently, the second high-level contribution of this work is to provide optimized implementations of these new curves and updated SIKE performance numbers. In summary, our experimental results and estimates show that the proposed parameterizations for NIST security category 1, category 3 and category 5 achieve factor-1.4, 1.7 and 2.0 speedups in comparison to the original SIKE parameterizations, on a modern x64 Intel platform. We refer to §5.3 for complete details. We will be releasing our source code for the vOW implementation and for the optimized SIKE instantiations and they will be linked to a future version of this article.

Finally, in Appendix B we further consider the case when the vOW algorithm is used to target k unrelated public keys simultaneously. Here we extend the theoretical analysis given by van Oorschot and Wiener to show that, when $k = 2$, $k = 3$, and $k = 4$, the modified algorithm will (on average) solve at least one of the CSSI problems faster than the time taken to solve any given CSSI problem on its own.

2 Preliminaries: van Oorschot-Wiener's Collision Search

Prior to 2018, the literature on SIDH (starting with Jao and De Feo's original paper [JDF11]) has consistently cited a meet-in-the-middle algorithm for the claw finding problem as the best known classical algorithm for solving the CSSI problem. A crucial observation made by Adj *et al.* [ACVCD+19] in 2018 is that, while the meet-in-the-middle claw finding algorithm has the lowest known classical runtime for solving CSSI, its storage requirements are so large (for CSSI instances of cryptographic size) that its application is not meaningful in any reasonable model of cryptanalytic computation. Thus, Adj *et al.* instead fix a conservative limit on the total amount of storage available (that which is still said to be "prohibitively costly for the foreseeable future" [ACVCD+19, §5]), and analyze the runtime of relevant algorithms subject to this storage capacity. Their conclusion is that van Oorschot and Wiener's parallel golden collision search algorithm [vOW99] is the best classical algorithm for the CSSI problem.

After defining the CSSI problem in §2.1, we describe the classical meet-in-the-middle claw finding algorithm in §2.2, which is both simpler than, and helps motivate, the description of the vOW parallel collision finding algorithm that follows in §2.3. The complexity analysis of the generic vOW algorithm is in §2.4, and we conclude the section in §2.5 with a summary of the results concerning quantum algorithms from [JS19], those which put forward attack scenarios for which it can be argued that vOW is the best known (classical or quantum) algorithm for attacking SIKE, e. g. when an attacker is restricted by limited time.

2.1 The CSSI Problem

Herein we will restrict to the popular scenario whereby an instance of SIDH/SIKE is parameterized by first fixing a prime $p = 2^{e_2}3^{e_3} - 1$ with $2^{e_2} \approx 3^{e_3}$ and $e_3 \gg 1$; all known implementations, including those in the SIKE submission, specify a prime of this form. We work with the set of isomorphism classes of supersingular elliptic curves in characteristic p. There are roughly $p/12$ such classes, and these are identified by their $\mathbb{F}_{p^2}$-rational j-invariants [Sil09, p. 146]. Since $p \equiv 3 \bmod 4$, we will fix $\mathbb{F}_{p^2} = \mathbb{F}_p(i)$ with $i^2 + 1 = 0$ throughout. Each supersingular j-invariant belongs to the same isogeny class [Mes86], which is precisely the set of (isomorphism classes of) elliptic curves, E, where $\#E(\mathbb{F}_{p^2}) = (p+1)^2$ [Sil09, Ex 5.15].

In this paper, isogenies are non-constant rational maps between two elliptic curves that are also group homomorphisms. We will be working only with *separable* isogenies, meaning that the degree of any given isogeny is equal to the number of elements in its kernel. Any subgroup $G \in E$ determines a unique (up to isomorphism) isogeny whose kernel is G; this isogeny can be computed using Vélu's formulas [Vél71].

For a prime $\ell \neq p$, there are precisely $\ell + 1$ isogenies of degree ℓ that emanate from a given supersingular curve. This induces a graph $\mathcal{G}_\ell$ – called a supersingular isogeny graph – whose nodes are the supersingular isomorphism classes and whose vertices are the degree-ℓ isogenies (up to isomorphism) between them. With the exception of nodes corresponding to j-invariants 0 and 1728, the graph $\mathcal{G}_\ell$ is a connected $(\ell+1)$-regular multigraph which satisfies the Ramanujan expansion property (see [DFJP14, §2.1]). Since every isogeny $\phi\colon E \to E'$ has a unique (up to isomorphism) *dual* isogeny $\hat{\phi}\colon E' \to E$, we can view $\mathcal{G}_\ell$ as an undirected graph (excluding $j = 0, 1728$). We return to the case where the node has j-invariant 1728 in §3.1.

For any n with $p \nmid n$, the set of n-torsion points, $E[n] = \{P \in E(\bar{\mathbb{F}}_p) : [n]P = 0_E\}$, is such that $E[n] \cong \mathbb{Z}_n \oplus \mathbb{Z}_n$ is two-dimensional. Let $(\ell, e) \in \{(2, e_2), (3, e_3)\}$. Following [DFJP14, Problem 5.2] (see also [ACVCD+19, §2.4]), we define a simplified version of the CSSI problem that underlies the SIDH and SIKE protocols within the above context as follows.[4]

Definition 1 (CSSI). *Given two supersingular elliptic curves E and E/G defined over $\mathbb{F}_{p^2}$ such that there exists a (unique up to isomorphism) isogeny $\phi : E \to E/G$ of degree ℓ^e with (cyclic) kernel $\ker \phi = G$, the computational supersingular isogeny (CSSI) problem is to compute ϕ or, equivalently, to determine a generator for G.*

2.2 The Meet-in-the-middle Claw Finding Algorithm

The most naive approach to solving CSSI is to perform a brute force search for G. Since the number of cyclic subgroups of order ℓ^e in $E(\mathbb{F}_{p^2})$ is $(\ell+1)\ell^{e-1}$, this takes $O(\ell^e)$ time. The claw

[4] As in [ACVCD+19, §2.4, Problem 1], we opt to present the simplified version of the problem that deviates from the original definition of the CSSI problem in [DFJP14, Problem 5.2] by omitting the auxiliary torsion points because the algorithms considered here are independent of the information given by these points.

finding algorithm uses the fact that we can view $\mathcal{G}_\ell$ as an undirected graph, so that we can instead *meet-in-the-middle*. Following [JDF11] (and assuming for simplicity that e is even), we can build two trees of curves: the first is the set of all isomorphism classes $\ell^{e/2}$-isogenous to that of E, the second is the set of all isomorphism classes $\ell^{e/2}$-isogenous to that of E/G. While there are $(\ell+1)\ell^{e/2-1}$ classes in each set, with overwhelmingly high probability there will be only one class that lies in both sets [JDF11, §5.1]. This class corresponds to the node on the isogeny graph lying in the middle of the path from E to E/G, and once it is found, the CSSI problem is solved by composing[5] the $\ell^{e/2}$-isogeny emanating from E with the dual of that emanating from E/G. Assuming that the $(\ell+1)\ell^{e/2-1}$ classes emanating from one of the sides can be computed and stored, then the time taken to solve CSSI in this way is now $O(\ell^{e/2})$.

It was not until the recent work of Adj *et al.* [ACVCD+19] that the classical complexity of this claw finding algorithm in the context of CSSI analysis was scrutinized. Given that $\ell^{e/2} \approx p^{1/4}$, and that the smallest prime p used to instantiate SIDH/SIKE prior to [ACVCD+19] was larger than 2^{500}, Adj *et al.* argue that the $O(p^{1/4})$ storage required to solve the problem as described above is infeasible. Instead, they fix 2^{80} as an upper bound on the number of *units* that can be stored, and analyze the runtime of the claw finding algorithm subject to this storage capacity. A CSSI attacker can now only afford to store a small fraction of the $O(\ell^{e/2})$ nodes emanating from one side at any given time, trying all of the nodes from the other side, and repeating this process until the CSSI problem is solved. The analysis of Adj *et al.* therefore concludes that the meet-in-the-middle algorithm is, for CSSI instances of cryptographic relevance, more costly than the vOW algorithm described in the sequel.

2.3 Solving CSSI with van Oorschot-Wiener

Let $S = \{0,1\} \times \{0,\ldots,(\ell+1)\ell^{e/2-1}-1\}$, $E_0 = E$ and $E_1 = E/G$. That is, each element $(i,y) \in S$ represents a kernel subgroup on the elliptic curve E_i. For example, for $\ell = 2$ the proposal of Adj *et al.* [ACVCD+19, §4.4] is to define a correspondence between $(i,y) = (i,(b,k)) \in \{0,1\} \times (\{0,1,2\} \times \{0,\ldots,2^{e/2-1}-1\})$ and the cyclic subgroup

$$\langle R \rangle = \begin{cases} \langle P_i + [b2^{e/2-1}+k]Q_i \rangle & \text{if } b = 0,1\,, \\ \langle [2k]P_i + Q_i \rangle & \text{if } b = 2\,, \end{cases} \quad \text{where} \quad \langle P_i, Q_i \rangle = E_i[2^{e/2-1}]\,.$$

The function that maps $(i,y) \mapsto R$ is denoted h. Let $f : S \to S$ be the function that, on input of (i,y), computes the isogeny of degree $\ell^{e/2}$ with kernel subgroup determined by y emanating from E_i, evaluates the j-invariant of the image curve, and maps this j-invariant back to S using a function g. In order to make f behave like a (pseudo-)random function on S, the function g is chosen to be (pseudo-) random from $\mathbb{F}_{p^2}$ to S.

A collision for f is a pair $x, x' \in S$ with $f(x) = f(x')$ and $x \neq x'$. If f is modeled as a random function, the expected number of collisions (over the set of random functions) is around $|S|/2$ [vOW99, §4.2]. For SIDH we will rely on the function h described above, while for SIKE we define the function in §3.2 (in both cases for $\ell = 2$). Note that necessarily there exists one special collision, namely the one between the two subgroups (one on E and one on E/G) that map to the same j-invariant and solve the CSSI problem. Since this is the only useful collision to solve the problem, we follow convention [vOW99, ACVCD+19] and refer to this collision as the *golden collision*. For the remainder of this section we abstract away from the setting of isogenies, since it is not necessary to understand the van Oorschot-Wiener algorithm. That is, we view f as a truly random function on the set S for which we want to find a single golden collision.

The vOW algorithm requires a proportion θ of the points in $|S|$ to be *distinguished points*. The function that computes whether or not a point is distinguished can be any efficiently computable function of the x_i, so long as it ensures that close to $\theta \cdot |S|$ of the $|S|$ points will indeed be deemed distinguished. The algorithm searches for collisions of f by performing many iterative walks in

[5] Appendix F shows in detail how to actually compute a point of order ℓ^e that generates the kernel of the isogeny composition.

parallel as follows. Each walk selects a starting point $x_0 \in S$ at random, and produces a trail of points $x_i = f(x_{i-1})$ for $i = 1, 2, \ldots$ until a *distinguished point* x_d is reached. The triple (x_0, x_d, d) is then added to a single common list and the processor chooses a new starting point at random to produce a new trail.[6]

Let w denote the number of triples of the form (x_0, x_d, d) that can be stored in the list. To simplify memory access, van Oorschot and Wiener suggest making the memory address for a given triple a function of its distinguished point. For optimized parameterizations geared towards real-world CSSI instantiations, we will have $w \ll \theta \cdot |S|$, i. e. we will not be able to store enough triples to account for all of the distinguished points. This gives rise to three scenarios when we store a given triple in memory. The first is that the memory at the given address was empty, in which case we write the triple there and continue; the second is that the triple occupying the memory corresponded to a different distinguished point, in which case we overwrite it with the new triple and continue; the third scenario is that the two triples contain the same distinguished points, in which case we have a collision and we must now check whether or not this is the golden collision. Let these two triples be (x_0, x_d, d) and $(x'_0, x'_{d'}, d')$ with $x_d = x'_{d'}$, and without loss of generality assume $d' > d$. To check the collision, we walk x'_0 forward by iterating $(x'_0, d') \leftarrow (f(x'_0), d' - 1)$ until $d' = d$, so that both walks are the same number of steps from the distinguished point. We then step both walks forward in unison iterating $(x_0, x'_0) \leftarrow (f(x_0), f(x'_0))$ until we find the first point of where the walks collide, i. e. until we find $x_0 \neq x'_0$ such that $f(x_0) = f(x'_0)$. If this is the golden collision, we are done. Otherwise, we replace the old triple with the new triple and continue. Note that the expected value of d, i. e. the expected length of the trails, is geometrically distributed with mean $1/\theta$.

Van Oorschot and Wiener note that there are two undesirable occurences that can arise during their algorithm. The first is that a trail collides with the starting point of another trail, which is called a *Robin Hood*. In practice, they note that θ is small enough that this occurence is rare. If this happens, we replace the triple in memory by the triple that was found last. The second potential pitfall is that a walk can enter into a cycle that does not contain a distinguished point. In [vOW99] the suggested workaround is to set a maximum trail length (e. g. $20/\theta$), and to abandon trails beyond this point.

Perhaps the most subtle aspect of the algorithm is that we are essentially forced to restart the above process many times, for many different instantiations of the random function f. As is explained in [vOW99, §4.2], there are roughly $|S|/2$ collisions that exist for f, and on average we will have to find this many collisions before we encounter the golden collision. However, not all collisions are equally likely to occur, and for any given f, the golden collision may have a very low probability of detection. For example, it could be that one or both of the two points that constitute the golden collision have very few trails leading into them, or in the extreme case, none at all; this would mean we would have to be extremely lucky to find the collision, i. e. by choosing the two points randomly as starting points. Thus, van Oorschot and Wiener explain that the best average runtime is achieved by trying a function f until a requisite number of distinguished points have been found (how many will be discussed in the next subsection), and then restarting with a new function until the golden collision is found. Henceforth, we will use f_n with $n \in \mathbb{Z}$ instead of f, where the subscript will be used to index the different function versions used in one golden collision search.

2.4 Complexity Analysis of van Oorschot-Wiener

Van Oorschot and Wiener give a complexity analysis for finding a golden collision [vOW99, §4.2]. However, they note that their complexity analysis is "flawed", giving multiple reasons as to why a precise closed formula for the runtime is difficult to achieve. Instead, after obtaining a general form for the runtime formula, they choose to determine several of the constants experimentally.

[6] In our scenario, where many collisions will be encountered before the golden collision is found, starting new trails (rather than continuing on from distinguished points) avoids the potential of falling into a cycle and repeatedly detecting the same collisions [vOW99, p.6, Footnote 5].

We reproduce the *flawed* analysis from van Oorschot and Wiener, since we will refer back to this analysis throughout. Recall that w is the number of triples (x_0, x_d, d) that can be stored in memory. At any time when the memory is full, the average number of points on trails leading to those w distinguished points is w/θ.

Writing $N = |S|$ and given any element of S, (uniformly) randomly generated as output of the random function f_n, the probability of it being on the pre-existing trails is therefore $w/(N\theta)$. Thus, on average we compute $N\theta/w$ points per collision. Checking a collision using the method described above requires $2/\theta$ steps on average, which gives the total average cost per collision as $N\theta/w + 2/\theta$. Taking $\theta = \sqrt{2w/N}$ minimizes this cost to $\sqrt{8N/w}$ steps to find a collision. As $N/2$ collisions are (on average) required to find the golden collision, we require (on average) $\sqrt{2N^3/w}$ function iterations to solve the CSSI problem.

Let m be the number of processors run in parallel and t the time taken to evaluate the function f_n. Since the algorithm parallelizes perfectly [vOW99, §3] (in theory), the total runtime T required to find the golden collision is

$$T = \frac{2.5}{m}\sqrt{N^3/w} \cdot t\,, \tag{1}$$

where 2.5 is one of the constants determined experimentally in [vOW99]. Some adjustments need to be made to the parameters because the phase where the memory is being filled with distinguished points is not accurately captured in the analysis. To describe the true performance of the algorithm, the fraction of distinguished points is set to $\theta = \alpha\sqrt{w/N}$ and the optimal constant α is determined experimentally. The heuristic analysis by van Oorschot and Wiener suggests $\alpha = 2.25$, which is verified by the analysis of Adj *et al.* in the case of SIDH, but we will elaborate on this constant more throughout the paper (e. g. Table 4).

The number w of distinguished points that can be stored has a crucial influence on the runtime of the vOW algorithm as can be seen from the above formula. It is therefore important to store distinguished points as compactly as possible. If the criterion for a point to be distinguished is the number of leading or trailing zeroes in its bit representation, these zeroes do not have to be stored allowing for a shorter bitlength for x_d in the triple (x_0, x_d, d). Given a distinguished point rate of θ, the number of zeroes would then be $\lfloor -\log\theta \rfloor$. The counter d must be large enough to store the number of steps in the longest trail, for example d must have $\lceil \log(20/\theta) \rceil$ bits. A distinguished point can thus be stored with about $2\log N + \log 20$ bits as most of the counter can be stored in the space of the omitted zero bits.

The total runtime involves the assumption that f_n behaves like an average random function. The average behavior can be achieved by using a number of different function versions f_n as explained above. To decide how long a function for a given n should be run before moving on, van Oorschot and Wiener introduce the constant β. The function version needs to be changed and distinguished points in memory discarded after $\beta \cdot w$ distinguished points have been produced. This constant is determined heuristically, analogously to the determination of α. For that purpose, a single n is fixed and run until $\beta \cdot w$ distinguished points are produced. In the meantime, the number of function iterations (i) and distinct collisions (c) are counted. The number of function versions can then be approximated as $n/(2c)$, while the expected run-time can be estimated as $in/(2c)$. It is concluded that the latter is minimal for $\beta = 10$.

We note that this experiment is extremely useful. Namely, it provides a very close estimate on the run-time without having to run the full algorithm. For that reason, we run the same experiment to estimate the impact of improved collision checking (see Fig. 3 in §3.4).

2.5 Quantum Algorithms for CSSI

The quantum claw finding algorithm by Tani [Tan09] has been referenced as being the best known quantum algorithm to solve the CSSI problem in the original security analysis by Jao and De Feo [JDF11, §5.2], as well as in the SIKE submission [JAC+17]. It relies on a generalization of Grover's search algorithm by Szegedy [Sze04] and uses quantum walks on Johnson graphs to solve the claw finding problem with a query complexity of $O(\sqrt[3]{\ell^e})$.

In their recent paper, Jaques and Schanck [JS19] provide an in-depth analysis of quantum algorithms for claw finding applied to the CSSI problem. In particular, they analyse the complexity of implementing and querying quantum memory, which is needed in Tani's algorithm and which previously had not been taken into account in the quantum security estimates for SIDH and SIKE. Jaques and Schanck introduce models of computation that allow direct comparison between classical and quantum algorithms. They model quantum computers via qubit arrays that are memory peripherals for classical controllers and provide a realistic parallel RAM model of computation and the associated cost metrics. Along with Tani's algorithm, they also consider a direct application of Grover search to claw finding. Their analysis shows that, due to previously neglected costs for quantum data structures, the quantum security estimates for the SIDH and SIKE parameters can be increased. It further shows that in some attack scenarios, classical security is the limiting factor for achieving a specified security level. While quantum algorithms promise to be more efficient for attackers with limited memory, classical vOW outperforms quantum algorithms for attackers with limited time. With respect to Tani's query-optimal algorithm that has been previously used for quantum security estimates for SIDH and SIKE, Jaques and Schanck [JS19, §6.2] state that "Our conclusion is that an adversary with enough quantum memory to run Tani's algorithm with the query-optimal parameters could break SIKE faster by using the classical control hardware to run van Oorschot-Wiener."

3 Parallel Collision Search for Supersingular Isogenies

In this section we describe multiple optimizations that we employ when specializing the van Oorschot-Wiener algorithm to SIKE. We elaborate on specific improvements related to design choices of SIKE in §3.1, while we explain decisions related to the vOW algorithm in §3.2. Finally, we show how to make use of local memory for precomputation in §3.3 and to improve collision locating in §3.4.

3.1 Solving SIKE Instances

Although the problem underlying SIKE is closely related to the original SIDH problem, there are slight differences due to design decisions. In this section we elaborate on those and their impact on the van Oorschot-Wiener algorithm. That is, we show how to reduce the search space from size $3 \cdot 2^{e_2-1}$ (resp. $4 \cdot 3^{e_3-1}$) to 2^{e_2-4} (resp. 3^{e_3-1}).

As usual, let $\{\ell, m\} = \{2, 3\}$ and let $\phi : E \to E_A$ be an isogeny of degree ℓ^{e_ℓ} for which the goal is to retrieve the (cyclic) kernel $\ker \phi$. We opt to represent curves in their Montgomery form [Mon87] $E_A : y^2 = x^3 + Ax^2 + x$ with Montgomery constant $A \in \mathbb{F}_{p^2}$. Being in Montgomery form allows for the use of very efficient arithmetic and for that reason, it has been the choice in the SIKE proposal. Further note that for SIKE, if $\{U, V\}$ is a basis for $E[m^{e_m}]$, then the points $\phi(U), \phi(V)$ are given as well. However, as we do not use these points on E_A and assume the simplified version of the CSSI problem as presented in Definition 1, we can just think of a challenge as given by the curve E_A.

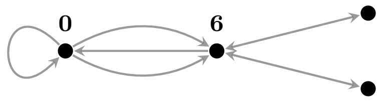
Since isogenies of degree ℓ^{e_ℓ} are determined by cyclic subgroups of size ℓ^{e_ℓ}, an easy counting argument shows that there are exactly $(\ell + 1)\ell^{e_\ell-1}$ of them. This forms the basis for the general algorithm specified for SIDH by Adj *et al.* [ACVCD$^+$19], essentially defining a random function on the set of cyclic subgroups.

Moving to SIKE, we observe that an important public parameter of the SIKE specification is the choice of the starting curve E_0. Since $p = 2^{e_2} \cdot 3^{e_3} - 1$ is congruent to 3 modulo 4 for $e_2 > 1$, the curve $y^2 = x^3 + x$ is supersingular for any choice of (large) e_2 and e_3.

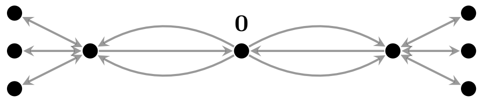
The initial step. Any point R of order ℓ^{e_ℓ} on E_0 satisfies $R = [s]P + [r]Q$ for $r, s \in \mathbb{Z}_{\ell^{e_\ell}}$. It follows by the order of R that one of s or r does not vanish modulo ℓ. However, the SIKE specification [JAC$^+$17, §1.3.8] assumes s to be invertible and simply sets $s = 1$. Firstly, this choice simplifies implementation by making the secret key a sequence of random bits that is easy to sample. Secondly, in the case of $\ell = 2$ an appropriate choice of P, Q allows the avoidance of exceptional cases in the isogeny arithmetic [Ren18, Lemma 2]. The main consequence is that the key space has size[7] ℓ^{e_ℓ} as opposed to $(\ell+1)\ell^{e_\ell - 1}$.

Finally, we note that although nodes in the isogeny graph generally have in-degree $\ell+1$, this is not true for vertices adjacent or equal to $j = 0$ or $j = 1728$. In particular, the curve $E_0 : y^2 = x^3 + x$ has j-invariant $j = 1728$ which in the case of $\ell = 2$ has in-degree 2, while its (only) adjacent node has in-degree 4. This is shown in Fig. 1a. In the case of $\ell = 3$ the curve has in-degree 2, while its adjacent nodes have in-degree 5. This is shown in Fig. 1b. This illustrates that although the number of distinct kernels is ℓ^{e_ℓ}, the number of distinct walks (say, as a sequence of j-invariants) in the isogeny graph is only 2^{e_2-1} (resp. $2 \cdot 3^{e_3-1}$) for $\ell = 2$ (resp. $\ell = 3$). We align the two (without loss of precision) by starting our walks from the curve $E_6 : y^2 = x^3 + 6x^2 + x$ in the case of $\ell = 2$. If $\ell = 3$, we can define the kernel on a curve in the class of the left or right adjacent node to $j = 1728$ (the choice indicated by a single bit).

The underlying reason for this happening is that E_0 has a non-trivial automorphism group containing the distortion map ψ that maps $(x, y) \mapsto (-x, iy)$ (with inverse $-\psi$). For any kernel $\langle R \rangle$ of size ℓ^{e_ℓ} we have $E_0/\langle R \rangle \cong E_0/\langle \psi(R) \rangle$ while $\langle R \rangle \neq \langle \psi(R) \rangle$, essentially collapsing the two kernels into a single walk in the graph. For example, in Fig. 1 we see that the walk of size 1 from node 0 to node 6 can be represented by two kernels (i. e. $\langle (i, 0) \rangle$ and $\langle \psi(i, 0) \rangle$). Note also that the loop on node 0 in the 2-isogeny graph has kernel $(0, 0) = [2^{e_2-1}]Q$, which can never appear in the computation of the kernel generated by $R = P + [r]Q$.



(a) The 2-isogeny graph



(b) The 3-isogeny graph

Fig. 1: Isogeny graphs starting from curves $y^2 = x^3 + Ax^2 + x$ where nodes are labeled by their A-coefficient

The final step. We observe that the elliptic curves are in Montgomery form, while isogenies of degree 2^{e_2} are computed as a sequence of 4-isogenies. As already noted in the original SIDH proposal [DFJP14, §4.3.2], the choice of arithmetic in SIKE leads to the kernel of the final isogeny being mapped to one of $(1, \pm\sqrt{A+2})$. These points define the same kernel subgroup (as they are inverse to one another). Consequently, the (class of the) curve $E_A/\langle (1, \pm\sqrt{A+2}) \rangle$ is isogenous to E_0 by an isogeny of degree 2^{e_2-2}, and isogenous to the Montgomery curve E_6 by an isogeny of degree 2^{e_2-3}. Therefore, replacing E_A by $E_A/\langle (1, \pm\sqrt{A+2}) \rangle$ reduces the number of distinct walks to 2^{e_2-3} in the case of $\ell = 2$.

For $\ell = 3$ the representative E_A of its isomorphism class can be obtained as the co-domain curve of a 3-isogeny starting from any of its adjacent nodes. As far as we know, this does not leak any information about the final 3-isogeny.

The Frobenius endomorphism. Recall that every isomorphism class can be represented by an elliptic curve E defined over $\mathbb{F}_{p^2}$, and that it has an associated Frobenius map $\pi : E \to E^{(p)}$ mapping $(x, y) \mapsto (x^p, y^p)$. Given any kernel $\langle R \rangle \subset E$, the fact that we are in characteristic p gives rise to the identity

$$j(E/\langle R \rangle)^p = j(E^{(p)}/\langle \pi(R) \rangle) \,.$$

[7] Technically, the specification actually makes the assumption that r is taken modulo the largest power of 2 less than or equal to ℓ^{e_ℓ}. This only slightly impacts our statements in the case $\ell = 3$ and we shall ignore it in our discussion.

As a result, it suffices to search for a path to a curve with j-invariant equal to $j(E_A)$ or $j(E_A)^p$. In other words, we define an equivalence relation on the set of j-invariants by $j_0 \sim j_1$ if and only if $j_1 \in \{j_0, j_0^p\}$. Finding a path to E_A reduces to finding a path to any representative of the class $[j(E_A)]$. In Fig. 2 we show how the classes propagate through the 2-isogeny graph starting at E_6. A very similar structure appears in the 3-isogeny graph. Note that we assume that the degree of our walk is approximately $\sqrt{p}$, making it unlikely for endomorphisms of that degree to exist. As such, the leaves of trees such as Fig. 2 will most probably all be distinct.

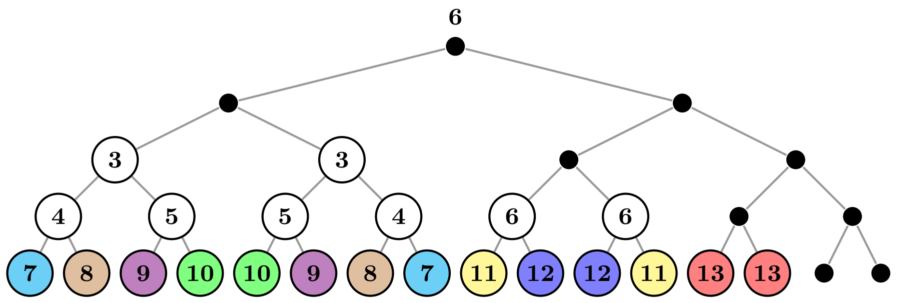


Fig. 2: Part of the 2-isogeny graph for any large $p = 2^{e_2} \cdot 3^{e_3} - 1$ starting at $E_6 : y^2 = x^3 + 6x^2 + x$. The black dots represent curves defined over $\mathbb{F}_p$, while j-invariants in the same equivalence class are denoted by equal numbers. All edges represent a 2-isogeny and its dual. In particular, there are exactly $2^3 + 1 = 9$ classes at distance 4 from E_6.

Although the number of classes is approximately half the number of j-invariants, it is perhaps not obvious how to translate this into a computational advantange. First assume that $\ell = 2$, and that the optimizations specified above are taken into consideration. That is, we start on the curve E_6 and look for an isogeny of degree 2^{e_2-3} to the curve E_A. As usual, kernels are of the form $P + [r]Q$ for some basis $\{P, Q\}$. Note that there is no reason to choose P and Q exactly as (multiples of) those in the SIKE specification, so we expand on a particularly simple choice here.

Recall first that $\#E_6(\mathbb{F}_p) = 2^{e_2} \cdot 3^{e_3}$ [Sil09, Exercise V.5.10]. Since the $\mathbb{F}_p$-rational endomorphism ring of E is isomorphic to one of $\mathbb{Z}[\pi]$ or $\mathbb{Z}[(1+\pi)/2]$ [DG16, Proposition 2.4], a result by Lenstra [Len96, Theorem 1(a)] tells us that

$$E_6(\mathbb{F}_p) \cong \begin{cases} \mathbb{Z}_{3^{e_3}} \times \mathbb{Z}_{2^{e_2}} & \text{if } \mathrm{End}_{\mathbb{F}_p}(E) \cong \mathbb{Z}[\pi]\,, \\ \mathbb{Z}_{3^{e_3}} \times \mathbb{Z}_{2^{e_2-1}} \times \mathbb{Z}_2 & \text{if } \mathrm{End}_{\mathbb{F}_p}(E) \cong \mathbb{Z}[\frac{1+\pi}{2}]\,. \end{cases}$$

Consequently, there exists an $\mathbb{F}_p$-rational point of order 2^{e_2-3} and we can choose Q to be this element. Moreover, $p \equiv 7 \mod 8$ implies that $\sqrt{2} \in \mathbb{F}_p$, and therefore that $E_6[2] \subset E_6(\mathbb{F}_p)$. In other words, π acts trivially on points of order 2. Since π fixes Q and has eigenvalues ± 1, for any other element P such that $\langle P, Q \rangle = E_6[2^{e_2-3}]$, the action of Frobenius is given by

$$\pi|_{\langle P,Q \rangle} = \begin{pmatrix} -1 & 0 \\ \mu & 1 \end{pmatrix}\,, \quad \text{for some } \mu \in \mathbb{Z}_{2^{e_2-3}}\,.$$

Note that $[2^{e_2-2}]P$ has order 2 and therefore is fixed under π. As a result, μ is even. Replacing P by $P - \frac{\mu}{2}Q$ leads to a basis $\{P, Q\}$ such that $\pi(P) = -P$ and $\pi(Q) = Q$. Note that the value of μ can be easily found (e. g. by using the Pohlig-Hellman algorithm [Sha71]) since the group is extremely smooth.

Given such a basis $\{P, Q\}$, the conjugate of the j-invariant determined by $\langle R = P + [r]Q \rangle$ is given by the isogeny with kernel $\langle -\pi(R) = P + [2^{e_2-3} - r]Q \rangle$. As a result, every class $\{j, j^p\}$ can uniquely be represented by $r \in \{0, 1, \ldots, 2^{e_2-4}\}$. If we start the algorithm by separately testing $r = 2^{e_2-4}$, the remainder can be reduced to searching for kernels $\langle P + [r]Q \rangle$ where $r \in \{0, 1, \ldots, 2^{e_2-4} - 1\}$. This reduces the search space to size 2^{e_2-4}.

By a completely analogous (and even simpler) argument we can fix a basis of $E[3^{e_3-1}]$ on any of the two adjacent nodes of E_0 in the 3-isogeny graph such that the action of π on this basis is described by a diagonal matrix with eigenvalues ± 1. Similar to the case of $\ell = 2$, this allows a reduction of the search space from $2 \cdot 3^{e_3-1}$ to (approximately) 3^{e_3-1}.

3.2 Applying van Oorschot-Wiener to SIKE

In this section we fix $\ell = 2$ and describe in detail how to implement the van Oorschot-Wiener algorithm (with parameters defined as in §§2.3–2.4). In particular, we point out a subtle mistake in the algorithm (appearing already in the original paper [vOW99] and also used in the work of Adj *et al.* [ACVCD$^+$19]) and show how to overcome this. That is, we show how to define distinguishedness to achieve the average runtime for a *fixed instance*. Consequently, we can use precomputation to analyze the behavior of van Oorschot-Wiener applied to SIKE at a much larger scale.

Again, we assume to be given a challenge curve E_A that is isogenous of degree 2^{e_2-3} to E_6 and aim to find the isogeny. We write $e = e_2/2$ and let $S = \{0, 1, \dots, 2^{e-1} - 1\}$. Fix points $P, Q \in E_6$ and $U, V \in E_A$ such that $E_6[2^{e-1}] = \langle P, Q \rangle$ and $E_A[2^{e-2}] = \langle U, V \rangle$, where $\pi(P) = -P$ and $\pi(Q) = Q$.

The step function. We begin by describing the function family f_n. As we will be walking on classes (of size 1 or 2) inside $\mathbb{F}_{p^2}$, we begin by defining a canonical representative of the class. Since the conjugate of $j = a + b \cdot i \in \mathbb{F}_{p^2}$ is simply $\bar{j} = a - b \cdot i$, we can say that j is *even* whenever $\text{lsb}(b) = 0$. Using >> to denote the rightshift operator, we then define the function h from the set S to the set of supersingular j-invariants by

$$h : r \mapsto \begin{cases} j & \text{if } \text{lsb}(j) = 0 \\ \bar{j} & \text{if } \text{lsb}(j) = 1 \end{cases}, \text{ for } j = \begin{cases} j(E_6/\langle P + [r >> 1]Q \rangle) & \text{if } \text{lsb}(r) = 0 \\ j(E_A/\langle U + [r >> 1]V \rangle) & \text{if } \text{lsb}(r) = 1 \end{cases}.$$

In other words, the least significant bit of r determines whether we compute an isogeny starting from E_6 or E_A, while we always ensure to end up on an even j-invariant. Finally, we define $f_n : S \to S$ by $f_n(r) = g_n(h(r))$, where g_n is a hash function indexed by n that maps $h(r)$ back into S. More concretely, we let g_n be the extended output function (XOF) based on the Merkle-Damgård construction [Mer79] around the AES-NI instruction set (see §4.1), with the initialization vector determined by n.

We note that the Frobenius map $\pi : (x, y) \mapsto (x^p, y^p)$ is an endomorphism on E_6, but not (necessarily) on E_A. Given an element $r \in \{0, 1, \dots 2^{e-2} - 1\}$, the kernels of the form $P + [r]Q$ determine isogenies of degree 2^{e-1} starting from E_6, yet it follows from §3.1 that they correspond to exactly 2^{e-2} (distinct) equivalence classes of j-invariants. The kernels of the form $U + [r]V$ determine isogenies of degree 2^{e-2} from E_A, all of which lead to distinct and non-conjugate j-invariants. Thus h maps bijectively into a set of size $2^{e-1} - 1$, with only a single collision determined by the isogeny from E_6 to E_A.

Distinguished points and memory. Assume the memory w to have size a (positive) power of 2. This is not technically necessary, but simplifies both the arguments and the implementation. Elements of S are represented by exactly $e - 1$ bits and we assume that $\log w \ll e - 1$.

Adj *et al.* [ACVCD$^+$19, §4.4] determined the memory position of a triple (r_0, r_d, d) using the $\log w$ least significant bits of MD5$(3, r_d)$. Moreover, the value r_d is distinguished if and only if MD5$(2, r_d) \leq 2^{32}\theta \bmod 2^{32}$ (viewing the output of MD5 as an integer). Although the algorithm will run, it has several complications.

1. Calling a hash function at every step to check for distinguishedness causes overhead in the algorithm. Similarly, requiring a hash function computation for every read and write operation to memory causes unnecessary overhead.

2. The algorithm (typically) requires the use of several functions f_n for distinct n. Since the memory location of elements is independent of n, distinguished points generated by distinct f_n will collide in memory but not lead to a collision in either of the functions. To counteract this, one could keep track of n in memory. As this is costly, the approach of Adj *et al.* is to zero out the memory when the maximum number of distinguished points for a given n is reached. This can get expensive as well, especially in the case of large distributed memory.
3. The distinguishedness property is independent of n. Although the runtime of the algorithm is estimated to be $2.5\sqrt{|S|^3/w}$ by van Oorschot and Wiener [vOW99, §4.2], this is only true if one takes the average over all collisions. However, in the setting of SIKE (and in many settings where one wants to find a specific collision), the value of the collision is fixed. That is, if the golden collision of the function f is determined by values $r, s \in S$ such that $f(r) = f(s)$, then the golden collision of f_n (for all n) is also determined by r and s. The runtime will be above average if one or both of r and s are distinguished. This is explained by the fact that the algorithm will sample a new value every time it reaches r or s, only computing $f_n(r)$ or $f_n(s)$ whenever they are sampled as initial values. Since distinguishedness is independent of n, this behavior will propagate throughout all the f_n.

We give a solution to all of these problems. First, we note that elements of our set are simply uniform bit strings of length $e-1$. Since the value r_d of the triple will always be the output of the (random) step function, we simply let the $\log w$ least significant bits determine the memory location. More precisely, we store the triple (r_0, r_d, d) in the memory location indexed by $(r_d + n) \bmod w$. Notice that we choose the location to be dependent on n. Therefore, even if two values collide in memory, the values that are stored will be distinct and we notice this immediately. In that case, the stored value can simply be overwritten without checking for a collision. Of course, it could happen that a value is written to memory by both f_n and f_{n+w} and none in between. But for reasonable values of n and w this will (almost) never happen, and only incurs the (relatively small) cost of checking for a collision when it does.

Secondly, we define the distinguishedness property. Since this should be independent of the memory location, we use the value of $r_d \gg \log w$. As usual, using all the remaining $e - 1 - \log w$ independent bits of r_d, we define an integer bound by $B = \theta \cdot 2^{e-1-\log w}$. We then define r_d to be distinguished if and only if

$$(r_d \gg \log w) + n \cdot B \leq B \mod 2^{e-1-\log w} .$$

In that case, every value of S is distinguished for approximately one in every B functions f_n. Although we do not prove that this reduces every instance to the average case, this holds true heuristically.

We observe that in this case the most significant bits $r_d \gg \log w$ of a distinguished element r_d are not always zero. This would be preferable since it reduces the memory requirement, not needing to store the top bits that are zero [vOW99, §6]. Instead we can simply write the value $(r_d \gg \log w) + n \cdot B \mod 2^{e-1-\log w}$ to memory, which by definition is less than (or equal to) B. Adding and subtracting $n \cdot B$ modulo $2^{e-1-\log w}$ when writing to and reading from memory has negligible overhead.

Remark 1. The problems we address appear for SIDH, while the above description solves them for SIKE. An analogous solution works for SIDH, but one should be careful that the values of S are *not* uniform bit strings. That is, they are elements $(i, b, k) \in \{1, 2\} \times \{0, 1, 2\} \times \{0, \ldots, 2^{e_2/2} - 1\}$ [ACVCD+19, §4.4] which are represented as $(3 + e_2/2)$-bit strings where the least significant bit determines i and two next lower order bits determine b. Instead, we define the memory location by the value $((r_d \gg 3) + n) \mod w$ and the distinguishedness property by

$$(r_d \gg (\log w + 3)) + n \cdot B \leq B \mod 2^{e-1-\log w} , \quad B = \theta \cdot 2^{e-4-\log w} .$$

In this case one should be even more careful not to lose too much precision for θ, but again the assumption that $e - 1 \gg \log w$ should alleviate this. In all of our instances this is not a concern.

Precomputing the step function and experiments. The main upside to the above modifications is that every instance will have a guaranteed average runtime of (approximately) $2.5\sqrt{|S|^3/w}$. As such, we do not have to worry about running into an unlucky instance.

However, there is a second useful consequence. That is, to analyze the behavior of our modifications it is sufficient to analyze a single instance. Now observe that any function f_n is of the form $f_n = g_n \circ h$, where h is fixed across the different n and by far the most expensive part of the evaluation of f_n. For testing any instance for which $h(S)$ fits into our memory, we can therefore simply precompute all of $h(r)$ for all $r \in S$ and store them in a table indexed by r. The evaluation of the step function $f_n(r)$ then simply looks up $h(r)$ in the table, followed by an application of g_n (which is comparitively fast). This improves the speed of our benchmarks significantly, while not affecting any outcomes with respect to a precise analysis of van Oorschot-Wiener.

We summarize the results so far in Table 1, comparing the results of our implementation to the expected theoretical outcome as well as the results of Adj *et al.* [ACVCD+19]. Note that our results are close to optimal, and showcase the expected speedup of a factor $\sqrt{6^3} \approx 15\times$ in the number of steps when moving from SIDH to SIKE. Moreover, we note that our implementation of SIDH also obtains a factor $2\times$ speedup compared to [ACVCD+19]. It is not clear why this happens.

Table 1: The average number of function versions n and evaluations of f_n used for finding an isogeny of degree 2^{e_2}. The expected value (Exp.) for the number of function versions resp. steps is reported as $0.45 \cdot |S|/w$ resp. $\log(2.5 \cdot \sqrt{|S|^3/w})$, for set size $|S| = 3 \cdot 2^{e_2/2}$ resp. $|S| = 2^{e_2/2-1}$ for SIDH resp. SIKE. The numbers are averaged over 1000 iterations and use 20 cores.

		Function versions					Steps				
		Exp.		[ACVCD+19]	**This**		Exp.		[ACVCD+19]	**This**	
e_2	$\log w$	SIDH	SIKE	SIDH	SIDH	SIKE	SIDH	SIKE	SIDH	SIDH	SIKE
32	9	173	29	319	177	28	23.20	19.32	24.38	23.29	19.58
36	10	346	58	838	342	54	25.70	21.82	27.25	25.74	21.89
40	11	691	115	1015	677	103	28.20	24.32	29.01	28.33	24.40
44	13	691	115	942	704	107	30.20	26.32	30.91	30.37	26.42
48	13	2765	461	–	–	434	33.20	29.32	–	–	29.38
52	15	2765	461	–	–	422	35.20	31.32	–	–	31.34
56	17	2765	461	–	–	424	37.20	33.32	–	–	33.38

3.3 Partial Isogeny Precomputation

Computationally, the most expensive part in the van Oorschot-Wiener step function is the (repeated application of the) evaluation of the isogeny for a given kernel subgroup $\langle R \rangle$ of order $\ell^{e_\ell/2}$. In order to alleviate this burden, one can precompute the isogeny tree partially. For example, one can compute all possible isogenies of a fixed degree $\varDelta$ from the starting curve and store a table of the corresponding image curves together with some torsion points that help to complete the isogeny walks from these intermediate curves. The extreme case, when the full isogeny tree of depth e_ℓ is precomputed, corresponds to the meet-in-the-middle algorithm as described by Adj *et al.* [ACVCD+19]. Precomputing isogenies of smaller degree presents a trade-off between memory and computation time for the vOW step function. We elaborate on how to do this in detail. As this applies to the general case of SIDH treated by Adj *et al.*, we discuss that first. We then specialize to $\ell = 2$ and finally consider SIKE instances.

Let E be a supersingular curve with torsion points $P, Q \in E$ such that $\langle P, Q\rangle = E[\ell^d]$, for some $d > 0$ (typically $d \approx e_\ell/2$ in the instantiations of van Oorschot-Wiener). Let $R = [s]P + [r]Q$ be a point of order ℓ^d, and $\phi : E \to E/\langle R\rangle$ an isogeny of degree ℓ^d with kernel $\langle R\rangle$. Recall that ℓ does not divide both r and s. We split the isogeny ϕ into two isogenies in the usual way, with the first having degree ℓ^{Δ} for some $0 < \Delta < d$ as follows.

Write $s = s_0 + s_1\ell^{\Delta}$ and $r = r_0 + r_1\ell^{\Delta}$ for $s_0, r_0 \in \mathbb{Z}_{\ell^{\Delta}}$ and $s_1, r_1 \in \mathbb{Z}_{\ell^{d-\Delta}}$. Then $R = [s_0]P + [r_0]Q + [\ell^{\Delta}]([s_1]P + [r_1]Q)$, while the point $R_\Delta = [\ell^{d-\Delta}]R = [s_0]([\ell^{d-\Delta}]P) + [r_0]([\ell^{d-\Delta}]Q)$ generates the kernel of the isogeny $\phi_\Delta : E \to E/\langle R_\Delta\rangle$ of degree ℓ^{Δ}. The point $\phi_\Delta(R)$ on $E/\langle R_\Delta\rangle$ has order $\ell^{d-\Delta}$ and determines an isogeny $\psi_\Delta : E/\langle R_\Delta\rangle \to E/\langle R\rangle$ of degree $\ell^{\Delta-d}$ such that $\phi = \psi_\Delta \circ \phi_\Delta$. Crucially, the first pair of partial scalars ($s_0 = s \mod \ell^{\Delta}, r_0 = r \mod \ell^{\Delta}$) determines ϕ_Δ and the points $\phi_\Delta([s_0]P + [r_0]Q)$, $\phi_\Delta([\ell^{\Delta}]P)$ and $\phi_\Delta([\ell^{\Delta}]Q)$ on $E/\langle R_\Delta\rangle$. Given this curve and these points, the second pair of partial scalars ($s_1 = \lfloor s/\ell^{\Delta}\rfloor, r_1 = \lfloor r/\ell^{\Delta}\rfloor$) determines $\ker\psi_\Delta = (\phi_\Delta([s_0]P + [r_0]Q)) + [s_1]\phi_\Delta([\ell^{\Delta}]P) + [r_1]\phi_\Delta([\ell^{\Delta}]Q)$ and allows to complete the isogeny ϕ. Therefore, precomputation consists of computing a table with entries

$$\left[E/\langle R_\Delta\rangle, \phi_\Delta([s_0]P + [r_0]Q), \phi_\Delta([\ell^{\Delta}]P), \phi_\Delta([\ell^{\Delta}]Q)\right],$$

for all $(s_0, r_0) \in \mathbb{Z}^2_{\ell^{\Delta}}$ such that ℓ does not divide both s_0 and r_0. Such a table entry can then be used to compute any full degree isogeny of degree ℓ^d with kernel point $R = [s]P + [r]Q$ such that $s \equiv s_0 \mod \ell^{\Delta}$ and $r \equiv r_0 \mod \ell^{\Delta}$ and any (s_1, r_1).

However, we show that it suffices to store only two points on $E/\langle R_\Delta\rangle$. If $\ell \nmid s$, then we can assume that $s = 1$ and $R = P + [r]Q$ for $r \in \mathbb{Z}_{\ell^d}$. In this case we have $R_\Delta = [\ell^{d-\Delta}]P + [r_0 \cdot \ell^{d-\Delta}]Q$ and the precomputed table only needs to contain entries of the form

$$\left[E/\langle R_\Delta\rangle, P_\Delta = \phi_\Delta(P + [r_0]Q), Q_\Delta = \phi_\Delta([\ell^{\Delta}]Q)\right] \tag{2}$$

for all $r_0 \in \mathbb{Z}_{\ell^{\Delta}}$. The kernel of ψ_Δ (for completing ϕ) can be computed as $\phi_\Delta(R) = P_\Delta + [r_1]Q_\Delta$ for any r with $r \equiv r_0 \mod \ell^{\Delta}$. If instead $\ell \mid s$, then $\ell \nmid r$ and $R = [\ell t]P + Q$ for some $t \in \mathbb{Z}_{\ell^{d-1}}$ such that $s = \ell t$. In that case table entries are of the form

$$\left[E/\langle R_\Delta\rangle, P_\Delta = \phi_\Delta([\ell^{\Delta}]P), Q_\Delta = \phi_\Delta([\ell t_0]P + Q)\right]$$

for all $t_0 \in \mathbb{Z}_{\ell^{\Delta-1}}$, while $\ker\psi_\Delta = [t_1]P_\Delta + Q_\Delta$. Altogether, the table contains $\ell^{\Delta} + \ell^{\Delta-1} = (\ell+1)\cdot\ell^{\Delta-1}$ entries and reduces the cost of any isogeny of degree ℓ^d from $d\log d$ to $(d-\Delta)\log(d-\Delta)$ [DFJP14, §4.2.2].

Now we move on to SIKE and fix $\ell = 2$. That is, we assume $s = 1$ and every table entry to be of the form (2). Recall that the function h takes as input a value $r \in \mathbb{Z}_{\ell^{e-1}}$ (where $e = e_2/2$) and computes an isogeny with kernel $\langle P + [r >> 1]Q\rangle$ on E_6 if $\mathrm{lsb}(r) = 0$, and an isogeny with kernel $\langle U + [r >> 1]V\rangle$ on E_A otherwise. The latter reflects the case above with $d = e - 2$ perfectly, leading to a precomputed table of size 2^{Δ} from E_A while reducing the cost of the isogeny from $(e-2)\log(e-2)$ to $(e-2-\Delta)\log(e-2-\Delta)$. The case of the curve E_6 is slightly different due to the presence of the Frobenius endomorphism. Although there are 2^{e-2} distinct equivalence classes of j-invariants, the degree of the corresponding isogenies is 2^{e-1}. As such, we compute a table of size 2^{Δ} comprising of the equivalence classes of j-invariants at depth $\Delta + 1$ away from E_6.[8] The cost of the isogenies reduces from $(e-1)\log(e-1)$ to $(e-2-\Delta)\log(e-2-\Delta)$. As a result, the degree of isogenies used throughout the whole implementation is the fixed number $e - 2 - \Delta$. In particular, choosing Δ such that $e - 2 - \Delta \equiv 0 \bmod 2$ allows the use of the 4-isogenies used in SIKE.

Computing an isogeny tree. To obtain the lookup table one computes image curves and torsion points for all isogenies of degree 2^{Δ} (resp. $2^{\Delta+1}$) and stores them indexed by their respective

[8] This slightly changes how an element $r_0 + r_1 2^{\Delta} \in \mathbb{Z}_{2^{e-2}}$, for $r_0 \in \mathbb{Z}_{2^{\Delta}}$ and $r_1 \in \mathbb{Z}_{2^{e-2-\Delta}}$, corresponds to an isogeny. Instead of kernel $\langle P + [r_0 + r_1 2^{\Delta}]Q\rangle$, it now gives rise to the kernel $\langle P + [r_0 + r_1 2^{\Delta+1}]Q\rangle$. This has no impact on the algorithm.

kernel representation. Adj *et al.* [ACVCD+19, Section 3.2] describe a depth-first-search approach to compute the required curves as the leaves of a full 2-isogeny tree of depth $e_2/2$ for the meet-in-the-middle algorithm (c. f. [ACVCD+19, Fig. 1]). This method is much more efficient than the naive way of computing full isogenies of degree $2^{e_2/2}$ for all possible kernel points. Obviously, this algorithm can be applied for partial trees to compute isogenies of degree 2^{Δ} (resp. $2^{\Delta+1}$) and an analogous version can utilize a 4-isogeny tree.

Using memory for precomputation. Storing a table of curve and point data to precompute parts of the isogenies obviously requires memory. Depending on the specific setup and communication properties of the network of parallel processors that runs a specific instance of the van Oorschot-Wiener algorithm, this memory could instead be used for the main memory that stores distinguished points. In other words, the memory to store precomputed tables might take away a certain amount from the distinguished point storage space.

Assume that due to latency and communication constraints, each of the m parallel processors needs its own table of size $\tau(\Delta)$ for precomputation, and for simplicity that every processor carries out the same depth precomputation. For example, for the SIDH case of Adj *et al.* [ACVCD+19] we would assume each processor to have precomputed a table of size $\tau(\Delta) = 2 \cdot (2^{\Delta} + 2^{\Delta-1}) = 3 \cdot 2^{\Delta}$. For SIKE we assume each processor to have precomputed a table of size $\tau(\Delta) = 2 \cdot 2^{\Delta} = 2^{\Delta+1}$.

Remark 2. In an actual distributed implementation, the situation might be different and favor precomputation more. For example, it is reasonable to assume that several processors in a multi-core machine are able to share a precomputed table, reducing the constant m that we have used in the above inequality. Furthermore, depending on the design of the main memory, each machine in a distributed attack setup may have memory available that cannot contribute to the main memory and might as well be used to store a table for a limited amount of precomputation. In such situations, using memory for lookup tables might not have any negative effect on the overall runtime of the van Oorschot-Wiener algorithm. In Example 1 we show that speed-ups for cryptographic parameters can be obtained with very small tables, making this scenario more realistic.

However, for now we continue with the above worst case assumption that m tables need to be stored. As shown in Section 2.4, each distinguished point is represented with roughly e_2 bits (i. e. about $\frac{1}{2}\log p$ bits) since $\log|S| = e_2/2 - 1$. This takes into account that the $\lfloor -\log\theta \rfloor$ leading zeros in a distinguished point are omitted in memory. Every entry in the precomputed table can be represented by three $\mathbb{F}_{p^2}$ elements (i. e. about $6\log p$ bits). Therefore, each such table element uses memory that could store about 12 distinguished points instead. For precomputation depth Δ, the table entries thus use space for $12 \cdot \tau(\Delta)$ distinguished points. This means that the main memory for the van Oorschot-Wiener algorithm is reduced from w to $w - 12 \cdot \tau(\Delta) \cdot m$ points, assuming that each of the m processors stores its own table. Thus, the number of iterations of the step function by the algorithm increases by a factor $1/\sqrt{1 - 12 \cdot \tau(\Delta) \cdot m/w}$. Note that this is well-defined since $12 \cdot \tau(\Delta) \cdot m$ cannot exceed the maximum available memory w.

While taking away memory increases the expected number of function iterations, precomputation reduces the cost of the step function by a factor $\sigma(\Delta)$. We have $\sigma(\Delta) = (e - \Delta)\log(e - \Delta)/(e \log e)$ for SIDH (given e_2 is even), while

$$\sigma(\Delta) = \frac{1}{2}\left(\frac{(e-2-\Delta)\log(e-2-\Delta)}{(e-2)\log(e-2)} + \frac{(e-2-\Delta)\log(e-2-\Delta)}{(e-1)\log(e-1)}\right)$$

in the case of SIKE (separating the two equally likely cases where we start from E_6 resp. E_A). The total runtime of the van Oorschot-Wiener algorithm decreases if

$$\frac{\sigma(\Delta)}{\sqrt{1 - 12 \cdot \tau(\Delta) \cdot m/w}} < 1\,.$$

Example 1. Let $p = 2^{216} \cdot 3^{137} - 1$ and $(e, m, w) = (108, 2^{64}, 2^{80})$, following the setup of [ACVCD+19, Remark 6]. For both SIKE resp. SIDH the (near) optimal pre-computation depth is $\Delta = 6$, for which each processor pre-computes a local table of $12 \cdot \tau(\Delta)$ distinguished elements that requires only about 41 resp. 62 kilobytes of memory (2.34% resp. 3.52% of the full memory w). In both cases, this leads to reduction of the cost of the step function by a factor $\sigma(\Delta) \approx 0.93$. For SIKE, we reduce the runtime of the full algorithm by a factor approximately 0.94. For SIDH, this factor is about 0.95.

However, a more realistic example assumes that many processors can share the precomputation table. In our setup, a machine of 40 cores can share a single table. In that case, the optimal depth is found at $\Delta = 12$. For SIKE, we use a table of about 2.7 megabytes (approximately 3.75% of the total memory w). This reduces the cost of the algorithm by a factor 0.88. For SIDH we obtain a table size of 4.0 megabytes (5.63% of the total memory). The runtime of the algorithm is reduced by a factor 0.89.

In Table 2 we demonstrate the effect of precomputation on the SIKE step function.

Table 2: Effect of precomputation on the running time of the SIKE step function. Numbers represent the cumulative running time in seconds of 1000000 calls to the step function, for the corresponding modulus and precomputation depth Δ. All experiments were run on Atomkohle.

	Δ														
e_2	0	2	4	6	8	10	12	14	16	18	20	22	24	26	28
32	20.51	17.96	15.47	13.09	10.91	8.84	7.17	4.92	—	—	—	—	—	—	—
36	23.50	20.46	17.91	15.45	13.08	10.85	8.82	7.18	4.84	—	—	—	—	—	—
40	26.79	23.60	20.97	18.45	15.96	13.60	11.42	9.35	7.62	5.00	—	—	—	—	—
44	29.37	26.34	23.58	21.01	18.44	15.96	13.60	11.38	9.32	7.70	4.89	—	—	—	—
48	32.48	29.57	26.88	24.21	21.33	18.80	16.25	13.83	11.57	9.41	7.70	4.87	—	—	—
52	36.38	32.93	29.92	27.13	24.15	21.53	18.85	16.36	13.93	11.64	9.48	7.76	4.87	—	—
56	40.05	35.48	33.29	29.67	26.80	25.60	21.46	18.94	16.43	14.60	11.83	9.73	8.03	4.89	—
60	41.56	38.54	35.72	32.73	29.91	27.09	24.38	21.69	19.17	16.68	14.26	12.03	9.95	8.26	4.96

3.4 Fast Collision Checking

As discussed in Remark 2, in a real-world implementation processors are likely to have local memory available that cannot contribute to the main memory (that which is used for storing the list of triples containing distinguished points). In what follows, we describe another way to make use of such memory to give significant speedups to the overall runtime of van Oorschot-Wiener. Analogous to the previous subsection, even under the assumption we are consuming memory that could otherwise be used to store distinguished points, we argue that dedicating a moderate amount of storage to this faster collision checking will give speedups to the overall runtime.

Recall from §2.3 that a single walk in the van Oorschot-Wiener algorithm starts at a point $x_0 \in S$ and produces a trail of points $x_i = f(x_{i-1})$ for $i = 1, 2, \ldots,$ until it reaches a distinguished point x_d. Assume that this triple (x_0, x_d, d) is then passed to main memory and a collision with a previously stored triple, say (y_0, y_e, e), is encountered. If it is not a mere memory collision, this means that $x_d = y_e$. Our task is now to check if we have found the golden collision, and this amounts to locating the indices $i < d$ and $j < e$ for which $x_i \neq y_j$ and $f(x_i) = f(y_j)$, i.e., locating the first point where the two paths coincide. Van Oorschot and Wiener note that, since d and e have expected value $1/\theta$, retracing the two paths from their starting points to the colliding point requires $2/\theta$ total steps on average [vOW99, p. 9]. Our goal is to reduce the number of function iterations used for retracing and thus lower the overall runtime.

Saving intermediate values. Suppose that we still store distinguished point triples in main memory as usual, but that there is enough local memory to store an additional $n-1$ points intermittently (more on what we mean by intermittently in a moment) during our walk from x_0 to x_d. Furthermore, assume that the checking of collisions takes place on a processor that has access to the local memory, now storing $n+1$ points. Relabeling x_0 as x_{d_0} and x_d as x_{d_n}, this means that when we detect that a collision has occured, we will now be able to use the points $(x_{d_0}, x_{d_1}, \ldots, x_{d_n})$, where $0 = d_0 < d_1 < \cdots < d_n$, together with the points (y_0, y_e), to find the first point of collision more efficiently.

We start by copying y_0 to y' and walking it forward as $y' \leftarrow f(y')$ until it is the same distance away from the distinguished points as the closest of the saved points, say x_{d_j}. We check whether $y' = x_{d_j}$. If not, we set $y_0 \leftarrow y'$ and step y' forward $d_{j+1} - d_j$ steps and compare again until it collides with one of the $n+1$ saved points, say x_{d_k}. Note that equality checks are only done when the walking point is the same distance away from the distinguished point as one of the x_{d_i} and not at every step as in the original collision checking function. Once we detect the minimal index k such that $y' = x_{d_k}$, we know that the collision must take place between the points $x_{d_{k-1}}$ and x_{d_k}. At this point, we can call the original collision checking function without saving intermediate points on the two triples $(x_{d_{k-1}}, x_{d_k}, d_k - d_{k-1})$ and $(y_0, y', d_k - d_{k-1})$. Note that if the collision occurs at x_{d_0}, we have a Robin Hood and return `false`.

Let us take a look at what we have gained. First of all the trail with the stored points is not retraced at all, only in the final call to the original collision checking, but this happens on a single subinterval of length $d_k - d_{k-1}$, which in general is much smaller than stepping from x_{d_0} to the collision. The trail starting at y_0 is fully retraced to the collision, where additional steps are taken that cover the colliding interval. The savings are larger when intervals are shorter and thus when more intermediate points are saved. This approach is implemented in our software.

Fig. 3 shows the reduction in the number of function steps for checking and locating collisions when running the van Oorschot-Wiener algorithm on an AES-based function with a set of size 2^{30} and memory of size 2^{15}. With $\alpha = 2.25$, the average walk length is $1/\theta \approx 80$. There is an immediate gain for even allowing a small number of intermediate points to be saved, however the additional gains become smaller when increasing this number. The reason for this is that when the number of points that can be stored approaches the average length of the trails, almost every point is stored and adding more memory does not significantly increase the number of points that are actually saved, nor influence the intermediate interval lengths.

Remark 3. There is potential for further improvement by allowing storage for $2n-2$ points. We proceed as above, store $n-1$ points while walking the trail, but also store $n-1$ intermediate points when retracing the trail from y_0 during collision checking. When the collision is encountered, we set $(y_0, y_e) \leftarrow (x_{d_{k-1}}, x_{d_k})$. We can overwrite the $n-1$ elements in $(x_{d_0}, \ldots, x_{d_{k-2}}, x_{d_{k+1}}, x_{d_n})$ with the $n-1$ points left intermittently along the walk from (the original) y_0 to x_{d_k}, and free up the $n-1$ duplicate points so that we can repeat this procedure. We recursively continue in this way until we have $y_e = f(y_0)$, at which point we can check whether the collision is golden. This is made precise in Algorithm 1 (see Appendix A), which is written recursively. Note that the choice of splitting the space for $2n-2$ points in half eases the exposition, but might be suboptimal. Finding the optimal allocation of memory to the different trails should be determined for a large scale cryptanalytic effort based on how much memory is available.

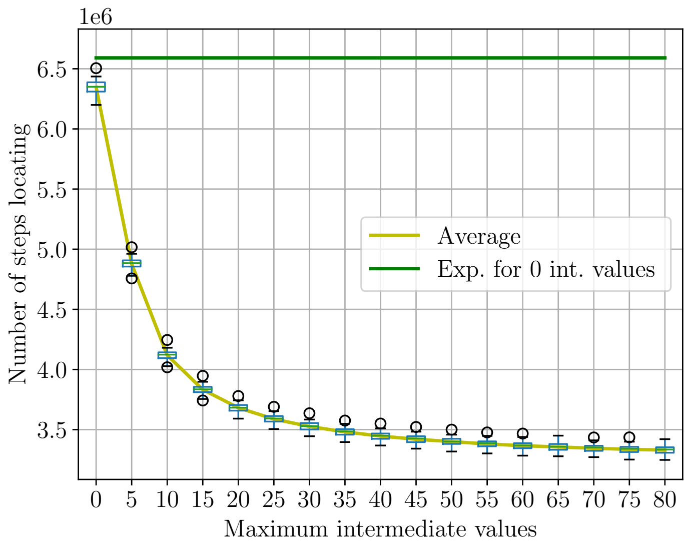


Fig. 3: Number of steps used for locating a collision as a function of maximum amount of intermediate values allowed for the AES-based random function with $\log |S| = 30$, $\log w = 15$. Averaged over 64 function versions, using 28 cores and run on Atomkohle.

How to save points intermittently. It remains to describe how we choose to store the $n-1$ intermediate points along our walks. Given the expected trail length of $1/\theta$, a reasonable approach would be to store points at regular intervals of length $1/(n\theta)$. However, we could end up walking much further than $1/\theta$, meaning there would be a larger distance between the final intermediate point and the distinguished point; or, our trail could be much shorter than $1/\theta$, meaning there would be unused memory that could have decreased the average gap between intermediate points. The ideal scenario is that all of the $(n-1)$ additional points have been used, and are as close to being equally spaced as possible, when the distinguished point is reached. Since we do not know in advance how long our trails will be, the best approach will involve overwriting previously placed points in such a way that the distances between points grows as the length of the trail does.

We modify an algorithm used by Sedgewick, Szymanski and Yao [SSY82] in their target application of finding cycles in random walks. In the first n steps of the trail, we exhaust the allocated memory by storing a point at every step, so that $(d_0, d_1, \ldots, d_{n-1}, d_n) = (0, 1, \ldots, n-1, n)$, and the points are all at distance 1 from one another. At any stage of the procedure, define $\delta = \min_{j>0}\{d_j - d_{j-1}\}$. From hereon, every δ steps, we simply look for the smallest value of j where $d_j - d_{j-1} = \delta$, remove the point x_{d_j} from the list, and add the current point to the list. At some point, the last point that is δ away from another point will be deleted and replaced by a point that is twice as far away; by definition, the δ is simultaneously doubled, and all of the points in the list will again be δ away from each other. This process is applied at lines 18 and 20 of Algorithm 1 (see Appendix A).

4 Implementation

Our implementation of the van Oorschot-Wiener algorithm is in C, relying on Microsoft's SIDH library [Mic19] for the underlying curve arithmetic. We have modified their code to support smaller primes, and added non-constant time operations if beneficial (e. g. finite field inversions). For parallel computations we use the gcc implementation of OpenMP 4.5 [Ope15]. For simplifying batch experiments we wrote a SWIG [Bea96] interface. The experiments are run across the different machines reported in Table 3.

Table 3: A summary of the machines used to run experiments.

Name	CPUs	Base freq.	Physical cores	GCC v.
Atomkohle	$2 \times$ Intel(R) Xeon(R) E5-2690 v4	2.60GHz	2×14	6.3.0
Solardiesel	$2 \times$ Intel(R) Xeon(R) Gold 6138	2.00GHz	2×20	6.3.0

The software contains three step functions to run experiments. The first is a generic fast random function, and the other two are those arising from random walks in the 2-isogeny graph as determined by the SIDH (see §2.3) and SIKE (see §3.2) specifications. This allows the use of a fast random function to verify that our implementation matches the expected asymptotic values (confirming the original vOW analysis [vOW99]) and linear speed-up on larger sets (see Appendix C), while also displaying our improvements in the SIDH and SIKE settings (e. g. as shown in Table 1).

4.1 Selecting a XOF and PRNG

One goal of the implementation is to verify the runtime against the asymptotic theoretical values, using a fast random function. Adj *et al.* [ACVCD+19] choose to use an MD5-based random function for this purpose. We have instead opted for a custom XOF based on the Merkle-Damgård construction [Mer79] around the AES-NI instruction set. This provides much better performance on modern hardware, while guaranteeing cryptographic properties of the function. Regarding our PRNG, we use AES-CTR mode based on the AES-NI instructions. We discuss in Appendix E alternative options we considered. In Table 4 we reproduce [vOW99, Table 1] which computes the $O(\cdot)$ constant in front of the expected number of steps for the optimal choice of θ and is used to determine the constant α, to demonstrate the validity of our pseudo-random step function.

Table 4: Reproduction of [vOW99, Table 1], using the AES-based XOF on Solardiesel. The table reports the number of function steps required to find the golden collision divided by $\sqrt{(|S|^3/w)}$. The experiments are averaged over 1000 function versions and run with 20 cores.

	$\log w$							
$\log \lvert S\rvert$	2	4	6	8	10	12	14	16
20	3.90	2.87	2.62	2.52	2.48	2.45	2.40	2.28
24	3.99	2.89	2.60	2.51	2.48	2.48	2.47	2.45
28	3.95	2.92	2.59	2.51	2.49	2.48	2.48	2.47
32	4.07	2.90	2.61	2.51	2.49	2.48	2.48	2.48
36	4.22	2.94	2.60	2.52	2.49	2.48	2.48	2.48

4.2 Towards a Distributed Implementation

Although not within the scope of this work, our software aims to make distributing experiments over the internet a straightforward extension. In contrast to other parallel cryptanalytic algorithms (e. g. Pollard rho [Pol75]), the van Oorschot-Wiener algorithm presents some issues regarding

[8] While online sources report 20 physical cores on each Xeon(R) E5-2673 v4, `/proc/cpuinfo` reports otherwise. Notably, Intel does not provide a webpage for this specific revision.

synchronization of the random function being used across machines in any specific window of time. Indeed, this makes the addition of "hot-pluggin" machines to the computational pool and keeping them up to date much harder than in other contexts, e.g., for a large-scale ECDLP effort [BBB+09]. In this section we discuss some of the issues and possible solutions.

The function versions used in van Oorschot-Wiener have a certain "shelf life", expressed in terms of the number of distinguished points to be found by using it (see §2.4). Although the algorithm parallelizes perfectly in theory, different CPUs may have different base frequencies and instruction sets, meaning that they may find distinguished points at different rates. A single-machine multi-threaded implementation of the algorithm may consider having a global counter of the distingushed points found at any point in time. This poses two problems: it implies a lot of (very cheap) indirect communication across cores, and requires care avoiding race-conditions on the counter. For example, one could address the latter issue by protecting the counter with a `#pragma omp critical` directive, which should not create parallelization issues if distinguished points are found rarely enough. A similar global value can then be used to express the current function version being used across cores.

While this works on small experiments, it would not scale towards a real cryptanalytic effort. For example, when running van Oorschot-Wiener across 2^{64} machines with $\theta \approx 2^{-16}$, at every step approximately 2^{48} distinguished points would be found, causing the `#pragma omp critical` directive to be a bottleneck [ACVCD+19, Remark 6]. Similarly, when running an instance over the internet, reading and writing at every step from a global value introduces difficult synchronization and latency issues.

Benchmarking. One solution to minimize the amount of core synchronization, is to assign to every participating CPU a certain portion of the total number of distinguished points to be mined for every function version, and have them synchronize information about the function version/state of the search less frequently. To achieve this, we include a benchmarking function call. This runs a fixed number of iterations of the algorithm, and measures how long each core takes (or a single core if the CPU is a simple, multithreaded one). This information can then be passed to the central database during a setup phase, so that it can decide how many points per function version to assign to each core. In our case this is not necessary, since our experiments were run on single machines with identical cores. Hence we simply assign an equal fraction of points to every core. We do not investigate how to efficiently run the setup phase on a remote database any further. Of course, benchmarking would have to be redone when adding or removing CPUs from the computational pool.

Core synchronization. We now consider when and how to update the function version across cores, giving three possible approaches. We refer to the first on as the *windowed* approach; cores work in isolation recovering their portion of distinguished points, and consequently update only their internal function version counter during a window of $\mathcal{W}$ versions. At the end of every $\mathcal{W}$'s function use, it updates its function version to a current "master" value. This could be remote (determined by the database measuring distinguished points received), or copied from a "master" thread in the local machine.

The second approach is called *stakhanov*; cores recover their assigned number of distinguished points, and then keep using the same random function and periodically checking (to either the remote database or locally if running on a single machine), whether the other cores are already done with that function version. When all cores finish, they update their respective function version counter and start with the new function version.

The final method is similar to stakhanov, but lets cores that finish their portion of points before others simply wait (or busy-wait) for others to finish, without doing extra work.

We have run experiments with all three methods on the same problem, and the stakhanov method clearly comes out on top in our setting. In Fig. 4 we present the result for SIKE with $e_2 = 32$ and $\log w = 9$. We decide to plot the *inverse* of the average wall time (in seconds), since that should show a linear improvement (as it does).

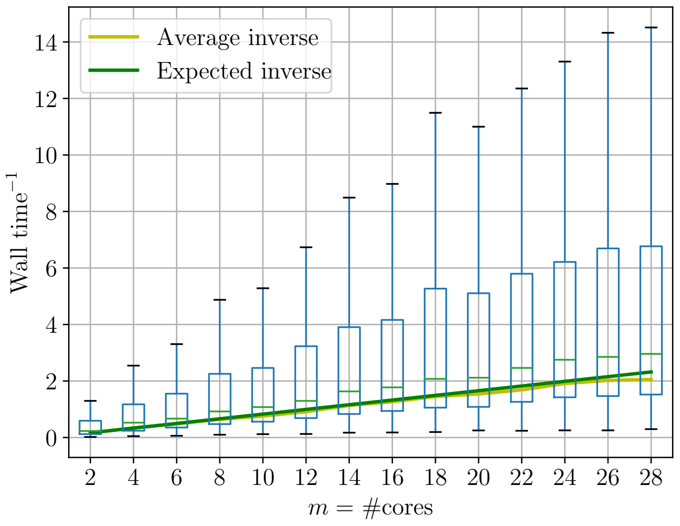


Fig. 4: Inverse wall time as a function of the number of cores used for the attack, averaged over 1000 function versions on Atomkohle. The experiment is run on SIKE with $e_2 = 32$, $\log w = 9$ with the *stakhanov* synchronization strategy, using no precomputation. Outliers are hidden to improve plot scale. The expected value was computed by picking the average value on the lowest number of cores, and scaling it by the appropriate number for the other number of cores. As expected, the inverse wall time grows linearly with the number of cores used.

Database versus peer-to-peer. A central issue when implementing vOW at scale is that of organizing the memory in a distributed system. One option is to abstract the memory to an external database. This would probably be partitioned into various smaller memory banks, connected each with multiple entry points, as to be able to handle a large amount of connections at every step (say, 2^{48}). Since the values being written into and read from the database are random and with random addresses, no caching would likely be useful. A barebones implementation could consist in a `mmap`'d amount of storage across different machines, with deamons running on as many ports as possible to handle the incoming connections. This way, cores running the attack require no memory for storing points, and can use all they have for example to precompute isogeny tables. A different strategy could be to identify the database partitions with the storage/memory available on the machines running the attack. It is not clear which method would require less total communication (for example, machines writing to their own partition would not need to communicate in the second, peer-to-peer-like configuration) and lead to lower wall times (machines in the peer-to-peer setting would have less memory to use for isogeny precomputation). A peer-to-peer setting would make hot-plugging machines nearly impossible.

5 Recommendations and Benchmarks for SIKE

In this section, we propose to use the parameters recommended by Adj *et al.* [ACVCD+19] and Jaques and Schanck [JS19] for SIKE targetting the NIST security categories 1 and 3, and recommend to move the original SIKE parameters for category 3 up to category 5. After providing concrete classical security estimates for the recommended parameter sets in §5.1, we propose refinements to the SIKE submission to deal with the issues arising in §3. Although the impact of the SIKE-specific optimizations to the van Oorschot-Wiener algorithm described in §3 is small, some of the countermeasures are simple and increase the security at essentially no cost. These modifications do not affect the recommendations for parameters made in [ACVCD+19] and [JS19]. We build on these works and provide assembly-optimized implementations, significantly improving the speed and key sizes corresponding to the three NIST security levels. We discuss the refinements in §5.2, and present benchmarks in §5.3.

5.1 Concrete Security of Proposed Parameters

The analysis of Adj *et al.* [ACVCD+19] of the complexity of van Oorschot-Wiener prompts the introduction of parameter sets that are significantly smaller (and thus faster and more compact) than those found in the SIKE proposal [JAC+17]. In particular, they put forward the prime $p434 = 2^{216}3^{137} - 1$ as being a possible choice of an underlying base field for an SIDH/SIKE parameterization that they say meets the NIST category 2 security requirements [ACVCD+19, §5.4]. Furthermore, they put forward the prime $p610 = 2^{305}3^{192} - 1$ that they say meets the NIST category 4 security requirements.

Note that the in-depth quantum security analysis by Jaques and Schanck [JS19] adds further confidence to the security of these parameters. They consider both Grover's search and Tani's algorithm – the algorithm previously cited as the relevant quantum attack against CSSI – and conclude that "an adversary with enough quantum memory to run Tani's algorithm with query-optimal parameters could break SIKE faster by using the classical control hardware to run van Oorschot-Wiener". Jaques and Schanck confirm that the number of classical gates to run quantum algorithms in their proposed model of computation against parameters with p434 and p610 exceed the bounds at the NIST security categories 1 and 3, respectively.

In Table 7 we use Equation (1) to count the number of x64 instructions required for an average run of vOW for each of the three parameter sets, i. e. those defined by the primes p434, p610 and p751. Following [ACVCD+19], we also fix $w = 2^{80}$ as a conservative limit on the number of units that can be stored. These instruction counts are intended to be lower bounds on the number of classical gates required to mount vOW, and we argue that these estimates are still conservative with respect to the true gate count. We use the instruction counts tallied in Table 5 and Table 6, which correspond to the number of instructions required for one call to the *half-size* isogeny computation that is needed in the vOW algorithm, for the 2- and 3-torsions respectively. Our analysis concludes that the number of classical gates required for (i) vOW on SIKEp434 is at least 2^{143}, (ii) vOW on SIKEp610 is at least 2^{210}, and (iii) vOW on SIKEp751 is at least 2^{262}. Note that the counts for (i) and (ii) closely agree with required classical gate counts given by Jaques and Schanck, who are also rather conservative in their costing of the related isogeny functions – see [JS19, §7.1].

Together with the analyses of Adj *et al.* and Jaques and Schanck, our analysis concludes that SIKEp434 meets the security requirements for NIST category 1, SIKEp610 meets the security requirements for NIST category 3. And, additionally, our analysis reveals that the prime $p751 = 2^{372}3^{239} - 1$ from the original SIKE proposal can be moved up from category 3 (where it was originally proposed) to underlie a SIKE implementation at category 5 – NIST's highest security level. While the estimate for the number of x64 instructions needed to mount vOW in the 2- and 3-torsions of SIKEp751 is estimated at 2^{262} and 2^{268} respectively, we expect in both cases that the number of classical gates required to run vOW in practice will exceed NIST's requisite 2^{272}. Moreover, we reiterate that the numbers in Table 7 correspond to our optimized vOW instantiation running on the SIKE specification as it stands; if the two recommendations for refinements described in the following subsection are followed, then we expect the number of gates required (in the 2-torsion case) to increase by a factor close to 2^3.

5.2 Refinements to the SIKE Specification

Recall from §3 that an adversary has several advantages due to design choices in the SIKE specification, i. e.

1. The presence of the distortion map on the node with $j = 1728$ leads to loops and double edges in the graph (Fig. 1). This reduces the entropy of the private and public keys.

Table 5: Costs for isogeny computation of degree $2^{\lfloor e_2/2 \rfloor - 2}$ (i. e. omitting single 2-isogenies when the exponent is odd) using an optimal strategy composed of quadrupling and 4-isogeny steps. DBL denotes a point doubling, 4-iso a 4-isogeny computation, **M** a multiplication, **S** a squaring, **add** an addition and **sub** a subtraction in $\mathbb{F}_{p^2}$. The symbol $i_{\mathbf{mul}}$ denotes the number of multiplication instructions, $i_{\mathbf{asl}}$ the number of addition, subtraction and logical instructions, $i_{\mathbf{mov}}$ the number of move instructions, and $\log(i_{\mathbf{sum}})$ is the logarithm of the total number of instructions. Cost for DBL is assumed to be $4\mathbf{M}+2\mathbf{S}+2\mathbf{add}+2\mathbf{sub}$ and for 4-iso it is $6\mathbf{M}+6\mathbf{S}+7\mathbf{add}+4\mathbf{sub}$.

	DBL	4-iso	**M**	**S**	**add**	**sub**	$i_{\mathbf{mul}}$	$i_{\mathbf{asl}}$	$i_{\mathbf{mov}}$	$\log(i_{\mathbf{sum}})$
SIKEp434	282	166	2124	1560	1726	1228	595476	2099108	1534760	22.01
SIKEp610	434	255	3266	2398	2653	1888	1638294	5433856	3553530	23.34
SIKEp751	548	334	4196	3100	3434	2432	3254832	9365124	9863656	24.42

Table 6: Costs for isogeny computation of degree $3^{\lfloor e_3/2 \rfloor}$ (i. e. omitting single 3-isogenies when the exponent is odd) using an optimal strategy composed of point tripling and 3-isogeny steps. TPL denotes a point tripling, 3-iso a 3-isogeny computation, **M** a multiplication , **S** a squaring, **add** an addition and **sub** a subtraction in $\mathbb{F}_{p^2}$. The symbol $i_{\mathbf{mul}}$ denotes the number of multiplication instructions, $i_{\mathbf{asl}}$ the number of addition, subtraction and logical instructions, $i_{\mathbf{mov}}$ the number of move instructions, and $\log(i_{\mathbf{sum}})$ is the logarithm of the total number of instructions. Cost for TPL is assumed to be $7\mathbf{M}+5\mathbf{S}+3\mathbf{add}+7\mathbf{sub}$ and for 3-iso it is $6\mathbf{M}+5\mathbf{S}+14\mathbf{add}+5\mathbf{sub}$.

	TPL	3-iso	**M**	**S**	**add**	**sub**	$i_{\mathbf{mul}}$	$i_{\mathbf{asl}}$	$i_{\mathbf{mov}}$	$\log(i_{\mathbf{sum}})$
SIKEp434	199	217	2695	2080	3635	2478	769445	2826741	2067722	22.43
SIKEp610	290	350	4130	3200	5770	3780	2112930	7266720	4861220	23.76
SIKEp751	395	429	5339	4120	7191	4910	4208868	12471749	13228173	24.83

Table 7: Average number of x64 instructions required to run the van Oorschot-Wiener attack on the 2- and 3-torsion of three SIKE parameterizations. All numbers are presented as the floor of their base-2 logarithms. The set sizes are $N = |S| = 2^{e_2/2-1}$ for the 2-torsion and $N = |S| = 3^{(e_3-1)/2}$ for the 3-torsion – see §3. The number of isogeny computations, #isog, is computed by setting $t = 1$ in Eq. (1), and the number of instructions required for each isogeny, $\lfloor \log(i_{\mathbf{sum}}) \rfloor$, are taken from Table 5 and Table 6. The total number of instructions, **vOW**, is taken as the product of the number of isogenies with the number of instructions required for each isogeny. This is intended to act as a lower bound on the number of gates required to solve the CSSI with the vOW algorithm.

	2-torsion				**3-torsion**			
	N	#isog	$i_{\mathbf{sum}}$	**vOW**	N	#isog	$i_{\mathbf{sum}}$	**vOW**
SIKEp434	107	121	22	**143**	107	122	22	**144**
SIKEp610	151	187	23	**210**	150	187	23	**210**
SIKEp751	185	238	24	**262**	188	244	24	**268**

2. The presence of the Frobenius endomorphism on the node with $j = 1728$ reduces the number of equivalence classes that are at a given distance from j.
3. For $\ell = 2$, the choice of arithmetic maps the kernel of the final 4-isogeny to a point with x-coordinate 1. As a result, the final step can be immediately recomputed from the public key.

The first problem is easiest to solve. By moving the starting node from E_0 to E_6 (with $j(E_6) = 287496$), the loop and double edge can be avoided in the case of $\ell = 2$. More concretely, setting up a torsion basis $\{P, Q\}$ of $E_6[2^e]$ such that $[2^{e-1}]Q = (0, 0)$ and choosing private keys $r \in \mathbb{Z}_{\ell^e}$ corresponding to kernels $\langle P + [r]Q \rangle$ implies this result. Note that the SIKE specification sets up Q to be a point of order 2^e defined over $\mathbb{F}_p$ [JAC+17, §1.3.3]. Such a point does not exist on E_6, as $E_6[2^e](\mathbb{F}_p) \cong \mathbb{Z}_{2^{e-1}} \times \mathbb{Z}_2$. As far as we can see, the only implication is that the description of Q becomes longer since it lies in $E_6(\mathbb{F}_{p^2}) \setminus E_6(\mathbb{F}_p)$.

It is not immediately obvious where the node of E_6 lies with respect to E_0 in the 3-isogeny graph, there is no reason to believe that it lies close. Therefore, we believe moving to E_6 also alleviates issues with double edges in the 3-isogeny graph as well.

Secondly, although we have moved away from $j = 1728$ by the above, the curve E_6 still has a Frobenius endomorphism. Although one could move to a curve where the Frobenius map is not an endomorphism, this is a subtle issue (see Remark 4). Therefore, we suggest simply staying on E_6. Indeed, in that case it is not helpful to differentiate between j-invariants in the same equivalence class. As (almost) every equivalence class contains 2 representatives at a certain depth, one bit less randomness is needed to compute an isogeny of the same degree (see e. g. Fig. 2, where the final step could always move to the left node).

Remark 4. The curve $E_0 : y^2 = x^3 + x$ has a known endomorphism ring [Sil09, III.4.4], which is helpful in certain attack scenarios [Pet17]. Although one would prefer to start on a *random* node in the graph, there is no known way of randomly selecting one other than choosing a random walk in the isogeny graph. However, in that case the walk itself cannot be public and it is unclear how to verifiably achieve this.

Finally, we address the issue of the leakage of the final kernel on the Montgomery curve $E_A : y^2 = x^3 + Ax^2 + x$. For this purpose, we notice that for any $\bar{A} \in \mathbb{F}_{p^2}$ such that $j(E_{\bar{A}}) = j(E_A)$ we have

$$\bar{A} \in \left\{ \pm A, \pm \frac{3x_2 + A}{\sqrt{x_2^2 - 1}}, \pm \frac{3z_2 + A}{\sqrt{z_2^2 - 1}} \right\}, \tag{3}$$

where $x_2, z_2 \in \mathbb{F}_{p^2}$ are chosen such that $x^3 + Ax^2 + x = x(x - x_2)(x - z_2)$. That is, the isomorphism class contains exactly six Montgomery curves. One can show that each of the 6 distinct 4-isogenies emanating from $j(E_A)$ can be computed by selecting $\bar{A}$ as above and using a kernel point (of order 4) with x-coordinate 1. Therefore, randomly choosing $\bar{A}$ from any of the options in Eqn. (3) is equivalent to randomizing the kernel of the final isogeny.

Unfortunately, to the best of our knowledge, selecting $\bar{A}$ to be anything other than $\pm A$ requires the computation of an expensive square root. For this reason, we do not recommend fully randomizing A. However, we emphasize that the random selection of one of $\pm A$ leads to a single bit of randomization at essentially no computational effort, mapping the kernel of the dual to ± 1. Given that the isogeny computation requires exactly one bit less randomness due to the Frobenius equivalence classes, we can simply reuse this bit for randomizing. As a result, we only leak the kernel of the final 2-isogeny (with kernel $(0, 0)$) instead of the last 4-isogeny (see Fig. 5).

5.3 Performance Benchmarks

To assess the performance of the parameterizations proposed for SIKE, we wrote hand-optimized x64 assembly implementations of the field arithmetic for the primes p434 and p610, and integrated them into the SIDH library, version 3.0 [Mic19]. All our implementations are written in *constant time*, i. e. there are no secret memory accesses and no secret data branches. Hence, the software is protected against timing and cache attacks.

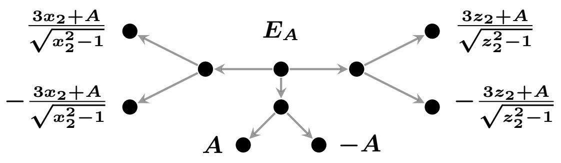

$\frac{3x_2+A}{\sqrt{x_2^2-1}}$
E_A
$\frac{3z_2+A}{\sqrt{z_2^2-1}}$
$-\frac{3x_2+A}{\sqrt{x_2^2-1}}$
$-\frac{3z_2+A}{\sqrt{z_2^2-1}}$
A
$-A$

Fig. 5: A sequence of 2-isogenies starting from the curve E_A. Each leaf node is labeled with the Montgomery coefficient $\bar{A}$ with $j(E_{\bar{A}}) = j(E_A)$ such that the isogeny from E_A to that node has kernel $(1, -)$. In particular, it is clear that selecting one of $\pm A$ reduces the leakage from 2 bits to 1 bit.

As an additional contribution, we sped up the implementations of the original parameters p503 and p751 by 11% resp. 16% using the faster tripling formula from [CH17], the Montgomery ladder approach by [FHOR18] and combining radix-r Montgomery reduction with the use of the `mulx` and `adx` instructions for the field multiplication, as done in [FHOR18]. The results on a 3.4GHz Intel Core i7-6700 processor from the Skylake microarchitecture family are displayed in Table 8. Following standard practice, TurboBoost was disabled during the tests. For compilation we used clang version 3.8.0 with the command `clang -O3`.

Our benchmarks show that the new SIKE parameters introduce roughly 1.4× and 1.5× speedups for level 1/2 and level 3/4, respectively (comparing the sum of the costs of the encapsulation and decapsulation operations). In addition, the public key sizes are reduced by approximately 14% resp. 19% for the levels above. The SIKE submission [JAC+17] does not report results for an optimized x64 assembly implementation of SIKEp964. In this case, we estimate a speedup of 2× when replacing this option by SIKEp751 for level 5, with a reduction of 22% in the public key size.

As a reference, our software achieves roughly the same performance as the implementation from [ACVCD+19] when running the SIDH protocol using p434 on the same Skylake CPU. In the case of p751, our software runs SIDH more than 1.2× faster, mainly due to a faster implementation of operations over $\mathbb{F}_{p^2}$.

Table 8: Performance benchmarks comparing the speed of SIKE Round 1 parameters and the proposed parameters in [ACVCD+19, JS19] and this work. The test results (public keys measured in bytes B, speed rounded to 10^5 cycles) were obtained on a 3.4GHz Intel Core i7-6700 (Skylake) processor, for the three SIKE operations: public key generation (Gen.), encapsulation (Enc.), and decapsulation (Dec.).

Target NIST security level	SIKE ([JAC+17] + **This**)					Proposed (**This**)				
	$\log p$	PK	Speed ($\times 10^6$ cc)			$\log p$	PK	Speed ($\times 10^6$ cc)		
			Gen.	Enc.	Dec.			Gen.	Enc.	Dec.
1	503	378 B	9.0	14.8	15.8	434	326 B	6.5	10.5	11.3
3	751	564 B	26.1	42.2	45.4	610	458 B	15.5	28.4	28.6
5	964	723 B	N/A	N/A	N/A	751	564 B	26.1	42.2	45.4

Acknowledgements. We thank Greg Zaverucha and Christian Konig for helpful discussions and their input to this paper, and Martin Albrecht for providing access to two of his machines for running our experiments.

References

[ACVCD$^+$19] Gora Adj, Daniel Cervantes-Vázquez, Jesús-Javier Chi-Domínguez, Alfred Menezes, and Francisco Rodríguez-Henríquez. On the Cost of Computing Isogenies Between Supersingular Elliptic Curves. In Carlos Cid and Michael J. Jacobson Jr., editors, *Selected Areas in Cryptography – SAC 2018*, pages 322–343, Cham, 2019. Springer International Publishing.

[AJK$^+$16] Reza Azarderakhsh, David Jao, Kassem Kalach, Brian Koziel, and Christopher Leonardi. Key compression for isogeny-based cryptosystems. In *Proceedings of the 3rd ACM International Workshop on ASIA Public-Key Cryptography*, AsiaPKC '16, pages 1–10, New York, NY, USA, 2016. ACM.

[BBB$^+$09] Daniel V. Bailey, Lejla Batina, Daniel J. Bernstein, Peter Birkner, Joppe W. Bos, Hsieh-Chung Chen, Chen-Mou Cheng, Gauthier van Damme, Giacomo de Meulenaer, Luis Julian Dominguez Perez, Junfeng Fan, Tim Gneysu, Frank Gurkaynak, Thorsten Kleinjung, Tanja Lange, Nele Mentens, Ruben Niederhagen, Christof Paar, Francesco Regazzoni, Peter Schwabe, Leif Uhsadel, Anthony Van Herrewege, and Bo-Yin Yang. Breaking ECC2K-130. Cryptology ePrint Archive, Report 2009/541, 2009. https://eprint.iacr.org/2009/541.

[Bea96] David M. Beazley. SWIG: An Easy to Use Tool for Integrating Scripting Languages with C and C++. In Mark Diekhans and Mark Roseman, editors, *Fourth Annual USENIX Tcl/Tk Workshop 1996, Monterey, California, USA, July 10-13, 1996*. USENIX Association, 1996.

[CH17] Craig Costello and Huseyin Hisil. A Simple and Compact Algorithm for SIDH with Arbitrary Degree Isogenies. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology – ASIACRYPT 2017*, pages 303–329, Cham, 2017. Springer International Publishing.

[CJL$^+$17] Craig Costello, David Jao, Patrick Longa, Michael Naehrig, Joost Renes, and David Urbanik. Efficient Compression of SIDH Public Keys. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology – EUROCRYPT 2017*, pages 679–706, Cham, 2017. Springer International Publishing.

[Col] Yann Collet. xxHash - Extremely fast non-cryptographic hash algorithm. https://cyan4973.github.io/xxHash/. Accessed: 2019-02-11.

[DFJP14] Luca De Feo, David Jao, and Jérôme Plût. Towards Quantum-Resistant Cryptosystems from Supersingular Elliptic Curve Isogenies. *Journal of Mathematical Cryptology*, 8(3):209–247, 2014.

[DG16] Christina Delfs and Steven D. Galbraith. Computing isogenies between supersingular elliptic curves over $\mathbb{F}_p$. *Des. Codes Cryptography*, 78(2):425–440, 2016. https://arxiv.org/abs/1310.7789.

[FHOR18] Armando Faz-Hernández, Julio López Hernandez, Eduardo Ochoa-Jiménez, and Francisco Rodríguez-Henríquez. A Faster Software Implementation of the Supersingular Isogeny Diffie-Hellman Key Exchange Protocol. *IEEE Trans. Computers*, 67(11):1622–1636, 2018.

[JAC$^+$17] D. Jao, R. Azarderakhsh, M. Campagna, C. Costello, L. De Feo, B. Hess, A. Jalali, B. Koziel, B. LaMacchia, P. Longa, M. Naehrig, J. Renes, V. Soukharev, and D. Urbanik. SIKE: Supersingular Isogeny Key Encapsulation. Manuscript available at sike.org/, 2017.

[JDF11] David Jao and Luca De Feo. Towards Quantum-Resistant Cryptosystems from Supersingular Elliptic Curve Isogenies. In Bo-Yin Yang, editor, *Post-Quantum Cryptography*, pages 19–34, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[JS19] Samuel Jaques and John M. Schanck. Quantum cryptanalysis in the RAM model: Claw-finding attacks on SIKE. Cryptology ePrint Archive, Report 2019/103, 2019. https://eprint.iacr.org/2019/103.

[KCP16] John Kelsey, Shu-jen Chang, and Ray Perlner. SHA-3 derived functions: cSHAKE, KMAC, TupleHash, and ParallelHash. Technical report, National Institute of Standards and Technology, 2016.

[Len96] Hendrik W. Lenstra, Jr. Complex Multiplication Structure of Elliptic Curves. *Journal of Number Theory*, 56(2):227 – 241, 1996.

[Mer79] Ralph C. Merkle. *Secrecy, authentication, and public key systems.* PhD thesis, Stanford University, 1979.

[Mes86] Jean-Francois Mestre. La méthode des graphes. Exemples et applications. In *Proceedings of the international conference on class numbers and fundamental units of algebraic number fields (Katata)*, pages 217–242, 1986.

[Mic19] Microsoft. SIDH Library v3.0. Available for download at https://github.com/Microsoft/PQCrypto-SIDH, 2015–2019.

[Mil04] Victor S. Miller. The Weil Pairing, and Its Efficient Calculation. *Journal of Cryptology*, 17(4):235–261, Sep 2004.

[Mon87] Peter L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of computation*, 48(177):243–264, 1987.

[mpp] Linux man-pages project. rand(3) - linux manual page. http://man7.org/linux/man-pages/man3/srand.3.html. Accessed: 2019-02-11.

[Nat16] National Institute of Standards and Technology. Post-quantum cryptography standardization, December 2016. https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Post-Quantum-Cryptography-Standardization.

[Ope15] OpenMP Architecture Review Board. OpenMP Application Program Interface Version 4.5, November 2015.

[Pat19] Kenneth G. Paterson. Personal communication, 2019.

[Pet17] Christophe Petit. Faster Algorithms for Isogeny Problems Using Torsion Point Images. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology – ASIACRYPT 2017*, pages 330–353, Cham, 2017. Springer International Publishing.

[Pol75] John M. Pollard. A monte carlo method for factorization. *BIT Numerical Mathematics*, 15(3):331–334, Sep 1975.

[Ren18] Joost Renes. Computing Isogenies Between Montgomery Curves Using the Action of $(0, 0)$. In Tanja Lange and Rainer Steinwandt, editors, *Post-Quantum Cryptography*, pages 229–247, Cham, 2018. Springer International Publishing.

[Sha71] Daniel Shanks. Class number, a theory of factorization, and genera. In *Proc. Symp. Pure Math*, volume 20, pages 415–440, 1971.

[Sil09] Joseph H Silverman. *The Arithmetic of Elliptic Curves*, volume 106. Springer Science & Business Media, 2009.

[SSY82] Robert Sedgewick, Thomas G Szymanski, and Andrew C Yao. The complexity of finding cycles in periodic functions. *SIAM Journal on Computing*, 11(2):376–390, 1982.

[Sze04] Mario Szegedy. Quantum Speed-Up of Markov Chain Based Algorithms. In *45th Symposium on Foundations of Computer Science (FOCS 2004), 17-19 October 2004, Rome, Italy, Proceedings*, pages 32–41, 2004.

[Tan09] Seiichiro Tani. Claw finding algorithms using quantum walk. *Theor. Comput. Sci.*, 410(50):5285–5297, 2009.

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

[vOW99] Paul C. van Oorschot and Michael J. Wiener. Parallel Collision Search with Cryptanalytic Applications. *Journal of Cryptology*, 12(1):1–28, 1999.

[WZ99] Michael J. Wiener and Robert J. Zuccherato. Faster Attacks on Elliptic Curve Cryptosystems. In Stafford Tavares and Henk Meijer, editors, *Selected Areas in Cryptography*, pages 190–200, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[ZSP$^+$18] Gustavo H. M. Zanon, Marcos A. Simplicio, Geovandro C. C. F. Pereira, Javad Doliskani, and Paulo S. L. M. Barreto. Faster Isogeny-Based Compressed Key Agreement. In Tanja Lange and Rainer Steinwandt, editors, *Post-Quantum Cryptography*, pages 248–268, Cham, 2018. Springer International Publishing.

A Recursive Algorithm for Fast Collision Checking

The pseudo-code presented in Algorithm 1 shows the recursive algorithm for fast collision checking with stored intermediate points as explained in Remark 3 in §3.4. We note that the function `IsGolden` called on line 6 simply checks whether the input points are different (ruling out a Robin Hood) and if so computes the j-invariants corresonding to the two kernel subgroups and returns `true` if they are equal, and `false` otherwise.

Algorithm 1: `FastCollisionCheck`

```
   Input: $\mathbf{x} = (x_{d_0}, x_{d_1}, \dots, x_{d_n})$, $(y_0, y_e)$ with $x_{d_n} = y_e$, $0 = d_0 < d_1 < \cdots < d_n$.
   Output: true, $x, y$, such that $f(x) = f(y)$ or false
 1 if $e = 1$ then
 2     while $d_n - d_{n-1} > 1$ do
 3         $x_{d_{n-1}} \leftarrow f(x_{d_{n-1}})$
 4         $d_{n-1} \leftarrow d_{n-1} + 1$
 5     end
 6     return IsGolden($x_{d_{n-1}}, y_0$)
 7 else
 8     while $e > d_n$ do
 9         $y_0 \leftarrow f(y_0)$ $e \leftarrow e - 1$
10     end
11     if $y_0 = x_{d_0}$ then
12         return false                                         // Robin Hood
13     end
14     $k = \min\{m \mid d_n - d_m \leq e\}$
15     Initialize $z \leftarrow y_0$
16     repeat
17         $x \leftarrow x_{d_k}$
18         Initialize fresh trail $\mathbf{z} = (z)$            // and/or overwrite existing
19         while $e > d_n - d_k$ do
20             $z \leftarrow f(z)$    (trail stored in $\mathbf{z}$)   // intermittent storage
21             $e \leftarrow e - 1$
22         end
23         $k \leftarrow k + 1$
24     until $z = x$
25     $(y_0, y_e) \leftarrow (x_{d_{k-2}}, x)$
26     $e \leftarrow d_{k-1} - d_{k-2}$
27     $\mathbf{x} \leftarrow \mathbf{z}$                        // overwrite
28     return FastCollisionCheck($\mathbf{x}, (y_0, y_e)$)
29 end
```

B Multi-target Attacks

In this section we focus on a specific type of multi-target attack: given k public keys, our goal is to break (i.e., solve the CSSI problem underlying) any one of them. We show that, on average, the expected vOW algorithm runtime is appreciably less for the cases of $k = 2$, $k = 3$ and $k = 4$. We then discuss the practical significance of these findings.

We assume that all public keys are generated in the same SIKE system, i.e., using the same starting curve, E. The set of curves in the public keys are then of the form E/G_1, E/G_2, ..., E/G_k, and the G_i are all subgroups of order ℓ^{e_ℓ} on E. We then explore two possibilities. The first is to simply combine the $k + 1$ curves into a run of vOW that walks uniformly between degree

$\ell^{e_\ell/2}$-subgroups on all of them; this is in §B.1. The second is to define the set S by duplicating the starting curve, the intuition here being that E is involved in k of the golden collisions that exist, while the other curves are each only involved in one; this is in §B.2.

B.1 Non-duplication of the Starting Curve

In this setting, the set S is extended to the set S' that contains elements such that their evaluations under the random function f map uniformly to subgroups belonging to curves in the set $\{E, E/G_1, E/G_2, \ldots, E/G_k\}$. The function f now maps S' into itself, $f : S' \to S'$, and we assume it is a random function. This means that any randomly selected pair of distinct elements from S' is a collision with probability $1/|S'|$. We clearly have k golden collisions. Write $N = |S|$ as usual and write $N' = |S'|$; then we have $N' = (k+1) \cdot N/2$.

Let T_k be the time taken to find the first golden collision during this attack. From §2.4, mimicking the *flawed* analysis in [vOW99], we can initially estimate $T_1 = (N/2) \cdot \sqrt{8N/w}$.

Now we have $N'/2$ total collisions (on average), and the number of collisions generated before finding the first golden collision is $N'/(2k)$, and thus

$$\begin{aligned} T_k &= \frac{N'}{2k} \cdot \sqrt{\frac{8N'}{w}} \\ &= \frac{(k+1) \cdot N}{4k} \cdot \sqrt{\frac{8(k+1) \cdot N}{2w}} \\ &= \left(\sqrt{\frac{(k+1)^3}{8k^2}} \right) \cdot T_1. \end{aligned}$$

Thus, for two public keys we have $T_2 \approx 0.9186 \cdot T_1$, for three we have $T_3 \approx 0.9428 \cdot T_1$, for four we have $T_4 \approx 0.9882 \cdot T_1$, but for $k \geq 5$ public keys, we have $T_k > T_1$.

B.2 Duplication of the Starting Curve

Recall from above that E is involved in k golden collisions, while the other k curves are each only involved in one. This bias prompted the extension of the vOW analysis to the scenario where we artificially duplicate E in our description of f to account for this; k duplicates of E increases the set size to $N' = kN$, but now we have k^2 golden collisions. We are essentially running k versions of the CSSI problem in parallel, i.e., using the same function at the same time.

A careful analysis reveals that T_k, the average time taken to find the first golden collision, is

$$\begin{aligned} T_k &= \left(\frac{k+1}{4k} \cdot |S| \right) \cdot \sqrt{\frac{8|S'|}{w}} \\ &= \left(\frac{(k+1)}{2\sqrt{k}} \right) \cdot T_1, \end{aligned}$$

so that $T_k > T_1$ for $k > 1$.

The intuitive reason here is that, while increasing k makes the number of golden collisions increase quadratically, the artificial duplication of E also necessarily makes the number of useless collisions (between copies of E) grow quadratically. Any two subgroups, G and H on E, that give a useless collision in memory, also find useless collisions with all of the copies of those subgroups in the copies of E.

B.3 Implications and Alternative Possibilities

The analysis in §B.1 reveals that combining two public keys into one run of vOW is worthwhile, if the adversary's goal is to break either one of them. The difference between the two analyses in §B.1 and §B.2 raises the question of whether there is some middle ground, e.g., for what values of k and

n is it advantageous to combine n public keys into one run of vOW by duplicating the starting curve $k < n$ times? Furthermore, it is unclear how increasing the number of collisions interacts with the need to change the function f regularly. Van Oorschot and Wiener's statement that "for a given function f, the golden collision may have a very low probability of detection" ultimately forces us to keep switching function versions, thereby rendering all of the prior distinguished points useless and essentially restarting. This, combined with the above analysis, raises the question of the interplay between the existence of multiple/many golden collisions and the success probability of any given function. So long as vOW remains the best attack against CSSI, we believe theoretical and experimental investigations in this direction to be worthwhile.

C Linear Speedup for Larger Experiments

Using our generic AES-based XOF we performed experiments up to $\log |S| = 52$, using the *stakhanov* sync strategy. We can see in Figure 6 that the speedup remains linear also for larger states and a higher number of cores, while the total number of steps across cores remains nearly constant, as shown in Figure 7.

Fig. 6: Box plot for wall time as a function of the number of cores used for the attack, averaged over 64 function versions on Atomkohle. AES-based random function with $\log |S| = 52$, $\log w = 13$. The expected value was computed by picking the average value on the lowest number of cores, and scaling it by the appropriate number for the other number of cores. Outliers are hidden to improve plot scale.

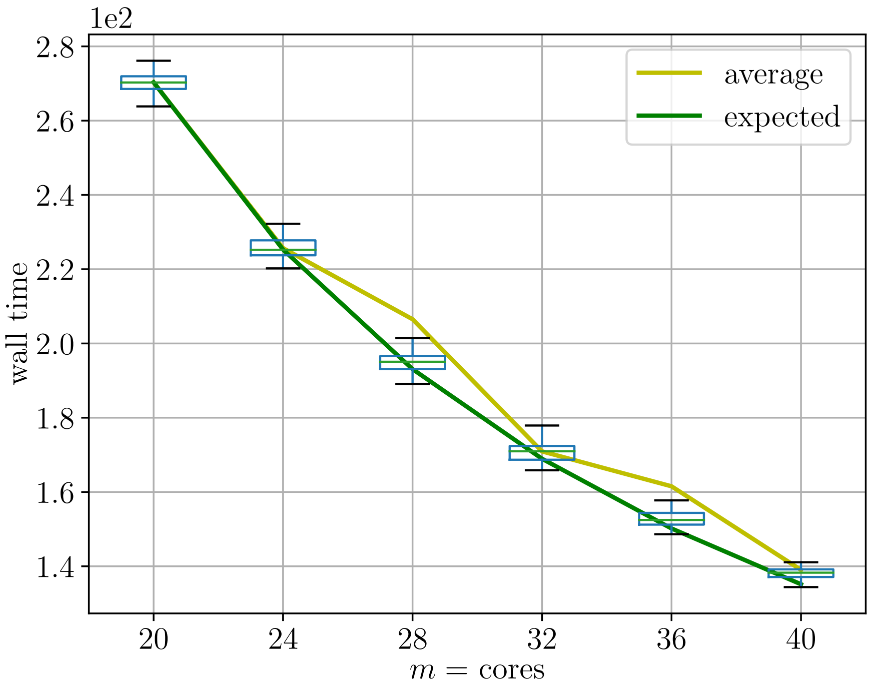
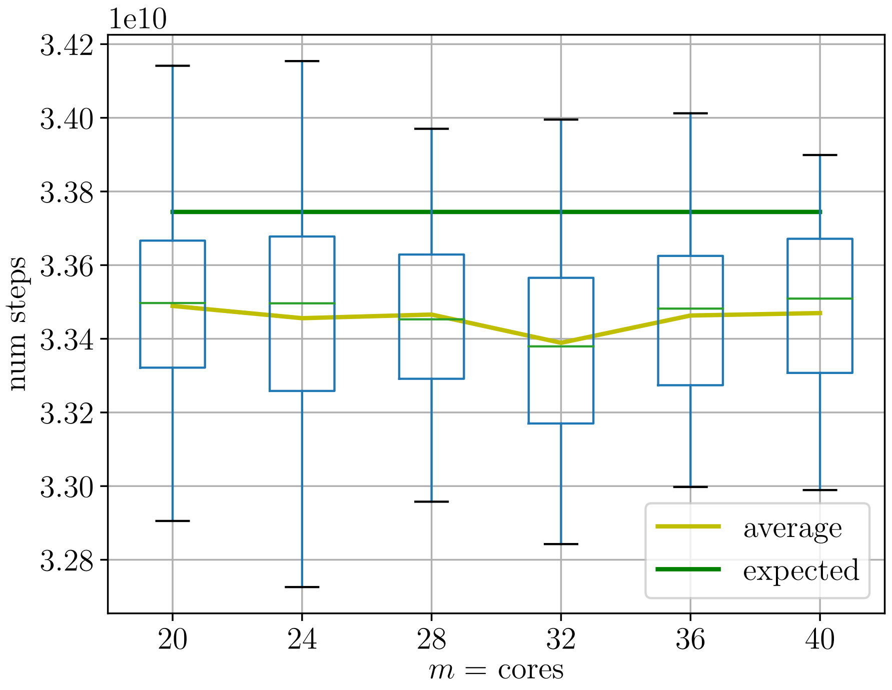


Fig. 7: Box plot for total number of step function calls made as a function of the number of cores used for the attack, averaged over 64 function versions on atomkohle. AES-based random function with $\log |S| = 52$, $\log w = 13$.



D Comparing Sync Strategies

In § 4.2, we described three different sync strategies for updating the version of f_n being used, and claimed that *stakhanov* was the best performing in our setting. Below, we provide plots for inverse wall time as a function of the number of cores being used to run vOW, showing the performance of the other two strategies, and how it indeed diverges from the expected value.

Fig. 8: Box plot for inverse wall time as a function of the number of cores used for the attack, averaged over 1000 function versions on Atomkohle. SIKE with $e_2 = 32$, $\log w = 9$ with windowed sync strategy for $\mathcal{W} = 10$, using no precomputation. Outliers are hidden to improve plot scale. The expected value was computed by picking the average value on the lowest number of cores, and scaling it by the appropriate number for the other number of cores.

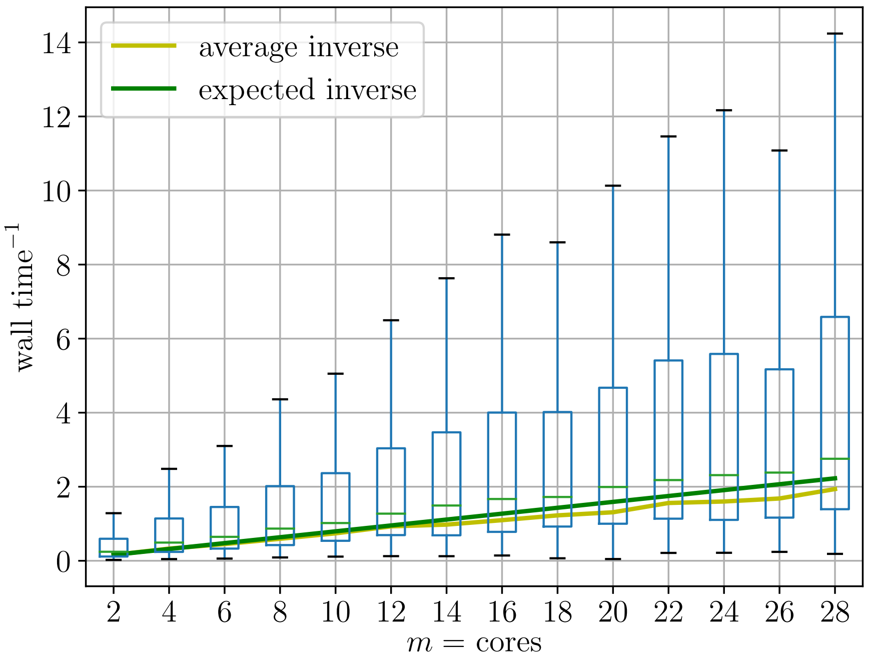
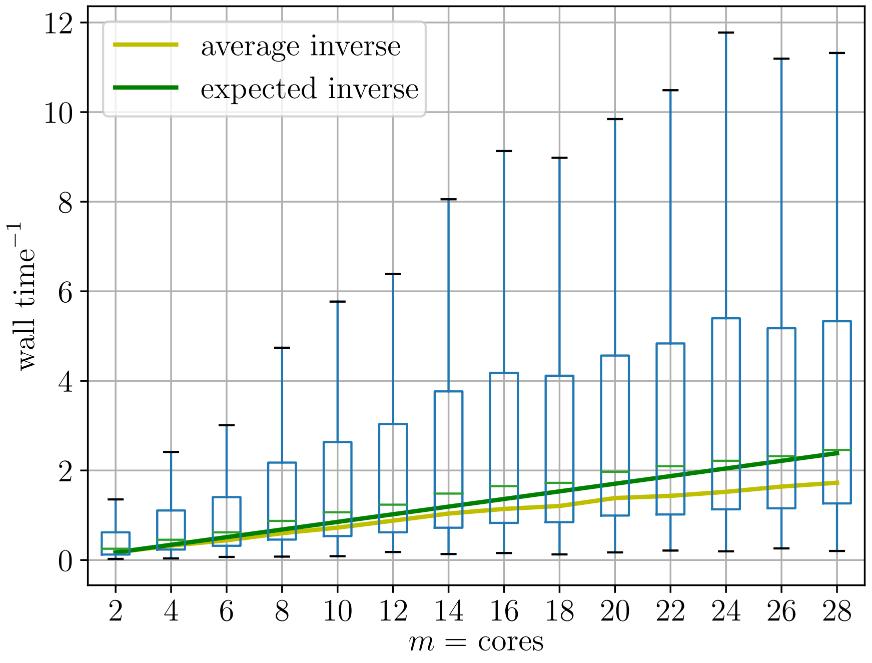


Fig. 9: Box plot for inverse wall time as a function of the number of cores used for the attack, averaged over 1000 function versions on Atomkohle. SIKE with $e_2 = 32$, $\log w = 9$ with the third proposed sync strategy, using no precomputation. Outliers are hidden to improve plot scale. The expected value was computed by picking the average value on the lowest number of cores, and scaling it by the appropriate number for the other number of cores.



E Non-cryptographic XOFs and PRNGs

One of our main concerns during development of the implementation was to be able to run fast examples using a generic random function, to check for asymptotic values being met. Adj *et al.* use an MD5-based random function for this purpose. Originally, we used an implementation of cSHAKE [KCP16] as XOF to construct the function, but it resulted in poor performance. We have hence moved to a custom XOF based on a Merkle-Damgard construction around the AES-NI instruction set.

We also considered using non-cryptographic hash functions, usually used as part of hash-tables implementations or for non-secure checksums. These provide random-looking output without formal guarantees regarding malleability (that should not be picked up by vOW), invertibility or size (they often provide short word-sized output), while being very fast. We implemented an XOF based on xxHash [Col], and run multiple experiments. In Tables 9, 10, we provide a comparison of our results using the AES-based XOF vs the xxHash-based one, showing expeirments using the latter to be slightly more than 50% faster (in the number of cycles required) while displaying the same asymptotic behaviour.

Regarding random number generation for the attack (for all step functions), we considered two options. The first was to follow Adj *et al.*'s example and use C's `rand`, which on POSIX.1-2001 is based on the Linear Congruential Generator (LCG) [mpp]. The second was to implement a PRNG based on AES-CTR using the AES-NI instructions. We reimplemented POSIX.1-2001's `rand` to match our PRNG API and to produce the same numbers on Windows. While it resulted in slightly faster code, using the LCG has a story of deceiving cryptanalysts using it to key RC4 by introducing small cycles in the key space which were later picked up by their analysis [Pat19].

Table 9: Reproduction of Table 3 from [ACVCD+19], using our a AES-based generic random function on Atomkohle. Experiments are run using 20 cores.

			Expected		**Average**			**Median**		
$\log \lvert S\rvert$	$\log w$	#runs	$\#f_n$	$\log \sqrt{\lvert S\rvert^3/w}$	$\#f_n$	$\log \sqrt{\lvert S\rvert^3/w}$	cycles	$\#f_n$	$\log \sqrt{\lvert S\rvert^3/w}$	cycles
18	9	1000	230.40	23.82	220.01	23.93	28.23	157.00	23.45	27.75
20	10	1000	460.80	26.32	429.44	26.32	30.44	325.00	25.92	30.04
22	11	1000	921.60	28.82	832.31	28.76	32.79	577.50	28.23	32.27
24	13	1000	921.60	30.82	873.88	30.78	34.72	622.50	30.29	34.23

Table 10: Reproduction of Table 3 from [ACVCD+19], using our a xxHash-based generic random function on Atomkohle. Experiments are run using 20 cores.

			Expected		**Average**			**Median**		
$\log \lvert S\rvert$	$\log w$	#runs	$\#f_n$	$\log \sqrt{\lvert S\rvert^3/w}$	$\#f_n$	$\log \sqrt{\lvert S\rvert^3/w}$	cycles	$\#f_n$	$\log \sqrt{\lvert S\rvert^3/w}$	cycles
18	9	1000	230.40	23.82	221.77	23.88	26.92	159.50	23.40	26.44
20	10	1000	460.80	26.32	450.04	26.35	29.13	323.50	25.87	28.65
22	11	1000	921.60	28.82	872.30	28.80	31.32	605.00	28.27	30.79
24	13	1000	921.60	30.82	928.12	30.86	33.35	681.50	30.41	32.91

In light of the risk of something along those lines happening, and given the marginal speedup it provided, we only used AES-CTR for our experiments.

F Kernel Reconstruction for the Full Order Isogeny

To solve the CSSI problem as presented in Definition 1, we are asked to compute the isogeny ϕ of degree ℓ^e between the given supersingular elliptic curves E and E/G, or equivalently to determine a generator R for its cyclic kernel subgroup G. However, both the meet-in-the-middle and the van Oorschot-Wiener algorithm as presented in [ACVCD+19] and here return two isogenies of degree $\ell^{e/2}$ with cyclic kernels that map from E and E/G to a common curve that lies *in the middle*. This section describes how R can be computed from generators of the kernels for those two isogenies.

We discuss the algorithm in the following setting, which is slightly more general than the specific scenarios for SIDH and SIKE above. Let p, e_2, e_3 and $(\ell, e_\ell) \in \{(2, e_2), (3, e_3)\}$ be as in § 2.1. Given two supersingular elliptic curves $E_{(1)}$ and $E_{(2)}$ over $\mathbb{F}_{p^2}$ such that there exists an isogeny of degree ℓ^e with cyclic kernel between them. Suppose, we know a third supersingular elliptic curve $E_{(3)}$ and isogenies $\phi_1 : E_{(1)} \to E_{(3)}$ and $\phi_2 : E_{(2)} \to E_{(3)}$ with $\deg \phi_1 = \ell^{e_{(1)}}$ and $\deg \phi_2 = \ell^{e_{(2)}}$ such that $e_{(1)} + e_{(2)} = e$, $\ker \phi_1 = \langle R_1 \rangle$ and $\ker \phi_2 = \langle R_2 \rangle$.

Computing the kernel of $\hat{\phi}_2$. First, we compute the dual of ϕ_2, which is an isogeny $\hat{\phi}_2 : E_{(3)} \to E_{(2)}$. Given the kernel point R_2 of ϕ_2, we find a basis $\{R_2, Q_2\}$ of $E_{(2)}[2^{e_{(2)}}]$. This can be done by randomly selecting Q_2 of the right order and checking that the Weil pairing of R_2 and Q_2 has full order or by using parts of the deterministic basis generation algorithms used for public-key compression described for example in [AJK+16, CJL+17, ZSP+18]. A generator for the kernel of $\hat{\phi}_2$ is then given as $\hat{R}_2 = \phi_2(Q_2)$.

Composing isogenies. Next, we find a kernel for the composition $\phi = \hat{\phi}_2 \circ \phi_1$. It is generated by a point R of order ℓ^e such that

$$R_1 = [\ell^{e_{(2)}}]R, \tag{4}$$
$$\langle \hat{R}_2 \rangle = \langle \phi_1(R) \rangle. \tag{5}$$

Let $\{P, Q\}$ be a basis for $E_{(1)}[\ell^e]$, then $\{P_1 = [\ell^{e_{(2)}}]P, Q_1 = [\ell^{e_{(2)}}]Q\}$ is a basis for $E_{(1)}[\ell^{e_{(1)}}]$. Assume[9] that we know that $R_1 = P_1 + [r]Q_1$ where $r \in \{0, 1, \ldots, \ell^{e_{(1)}} - 1\}$. We set $R = P + [r + \ell^{e_{(1)}} s]Q$ for a yet unknown $s \in \{0, 1, \ldots, \ell^{e_{(2)}} - 1\}$. Then, clearly condition (4) is satisfied. It now remains to determine s such that condition (5) holds.

The value for s can be determined iteratively, coefficient by coefficient in its ℓ-adic representation. Let $s = \sum_{i=0}^{e_{(2)}-1} s_i \ell^i$. Start with $i = 0$, and determine s_0 modulo ℓ such that the point $\phi_1(R^{(0)})$ lies in the subgroup generated by $\hat{R}_2$, where $R^{(0)} = [\ell^{e_{(2)}-1}](P + [r + \ell^{e_{(1)}} s_0]Q)$. This can be done by computing the Weil pairing $e_{\ell^{e_{(2)}}}(\phi_1(R^{(0)}), \hat{R}_2)$ and checking whether it is equal to 1 (cf. [Mil04, Prop. 12]). Once a suitable value for s_0 is found, continue with s_1. Find the value of s_1 modulo ℓ that satisfies $e_{\ell^{e_{(2)}}}(\phi_1(R^{(1)}), \hat{R}_2) = 1$, where $R^{(1)} = [\ell^{e_{(2)}-2}](P + [r + \ell^{e_{(1)}}(s_0 + \ell s_1)]Q)$. We can iteratively determine s_i by checking the pairing condition for the point $R^{(i)} = [\ell^{e_{(2)}-i-1}](P + [r + \ell^{e_{(1)}}(s_0 + \ell s_1 + \cdots + \ell^i s_i)]Q)$. At the end of this process, the point $R = P + [r + \ell^{e_{(1)}} s]Q$ has full order ℓ^e and satisfies both conditions (4) and (5), which means it is a generator for the kernel of the ℓ^e-isogeny ϕ.

[9] The other case, where the factor in front of Q_1 can be scaled to 1 is analogous and we omit the details.