Improved Classical Cryptanalysis of SIKE in Practice

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. The main contribution of this work is an optimized implementation of the van Oorschot-Wiener (vOW) parallel collision finding algorithm. As is typical for cryptanalysis against conjectured hard problems (e. g. factoring or discrete logarithms), challenges can arise in the implementation that are not captured in the theory, making the performance of the algorithm in practice a crucial element of estimating security. We present a number of novel improvements, both to generic instantiations of the vOW algorithm finding collisions in arbitrary functions, and to its instantiation in the context of the supersingular isogeny key encapsulation (SIKE) protocol, that culminate in an improved classical cryptanalysis of the computational supersingular isogeny (CSSI) problem. In particular, we present a scalable implementation that can be applied to the Round-2 parameter sets of SIKE that can be used to give confidence in their security levels.

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

1 Introduction

The supersingular isogeny key encapsulation (SIKE) proposal [11] – the actively secure version of Jao and De Feo's SIDH key exchange [12] – is one of 17 second round candidate public key encryption or key establishment proposals submitted to the post-quantum cryptography standardization process initiated by the U.S. National Institute of Standards and Technology (NIST). It is 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 [12, §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. On a quantum computer, Tani's algorithm [31] relies on a generalization of Grover's search algorithm by Szegedy [30] and uses quantum walks on Johnson graphs to solve the claw finding problem in $O(\sqrt[3]{|A||B|})$ time. Following Jao and De Feo, the SIKE team used these asymptotics to specify three Round-1 parametrizations that were intended to meet the requirements for the NIST security categories 1, 3 and 5 defined in terms of resources needed for AES key search [21, p. 18].[4]

⋆ 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.

[4] NIST specifies five target security categories. A cryptosystem meets the requirements of categories 1, 3 and 5 if any attack that breaks the relevant security definition requires computational resources comparable to or greater than those needed for key searches on AES-128, AES-192 and AES-256. Categories 2 and 4 are defined by the resources for collision searches on SHA-256 and SHA-384.

Prior to 2018, the literature on SIDH (starting with Jao and De Feo's original paper [12]) has consistently cited a meet-in-the-middle algorithm for claw finding as the best known classical algorithm for solving the CSSI problem. In 2018, Adj, Cervantes-Vázquez, Chi-Domínguez, Menezes and Rodríguez-Henríquez [1] made a significant step towards a better understanding of the problem's concrete classical complexity. They show that, while the meet-in-the-middle algorithm has the lowest known classical runtime, its storage requirements are so large (for instances of cryptographic size) that its application is not meaningful in any reasonable model of cryptanalytic computation. Indeed, the best classical AES key search algorithms only require a modest amount of storage, so a fair and correct analysis (with respect to 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 storage capacity that is considered "prohibitively costly for the foreseeable future" [1, §5], i.e., 2^{80} *units* of storage, and analyze the runtime of relevant algorithms subject to this capacity. They conclude that despite its higher running time, the van Oorschot-Wiener (vOW) parallel collision finding algorithm [32] has significantly lower space requirements and is the best classical algorithm for the CSSI problem. Thus, its concrete complexity should instead be used to assess the security of SIDH/SIKE against (known) classical attacks. Their analysis ultimately shows that the SIKE team used rather conservative classical security estimates and that significantly smaller parameters can be used to achieve the requisite level of classical security.

Jaques and Schanck [13] 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/SIKE. Along with Tani's algorithm, they also consider a direct application of Grover search [9] to claw finding. Similar to the classical analysis of Adj *et al.*, they conclude that the SIKE proposal's quantum security estimates were too conservative. In fact, Jaques and Schank's analysis shows that the best known quantum algorithms do not achieve a significant advantage over the classical vOW algorithm. In some attack scenarios, it is the classical security that 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 [13, §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." Thus, the precise, real-world complexity of the vOW parallel collision search algorithm is paramount in the discussion of (current and future) parameters for SIDH/SIKE.

Based on the above cryptanalytic results, the parameter sets in the SIKE specification were adjusted in Round 2 of the NIST standardization process. The specification now contains the parameter sets SIKEp434, SIKEp503, SIKEp610 and SIKEp751 targeting the NIST security categories 1, 2, 3 and 5, respectively.

Contributions. We present an implementation of the van Oorschot-Wiener algorithm that is intended 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 improving the instantiations specific to the contexts of SIDH and SIKE. Besides significantly optimizing the efficiency of the underlying finite-field and elliptic-curve arithmetic by incorporating the state-of-the-art formulas, we present several optimizations related to the structure of the isogeny graph.

Beyond being able to reproduce our results, we hope that our implementation[5] can function as the basis for further experiments to assess the security of isogeny-based cryptography, and that it can be used for other applications of the collision finding algorithm. In fact, we provide two implementations: an optimized C code base for both generic collision finding as well as solving the CSSI problem, and an upcoming C++ version designed for modularity, and to allow easy porting

[5] The source code has been released under a free license at https://github.com/microsoft/vOW4SIKE.

to alternative collision finding settings at little cost to efficiency (e. g. for the hybrid attack on lattice-based schemes [10], symmetric cryptography, or highly distributed setups).

Our extensions and improvements to the vOW implementation and analysis in [1] include:

- *Faster collision checking.* One of the main steps in the vOW algorithm is to check whether a given collision is the *golden collision* (see §2). Experimentally, our optimized version of generic vOW found that it constitutes close to 20% of the entire algorithm (aligning with van Oorschot and Wiener's analysis [32, §4.2]). We give a novel, much more efficient method, which is based on a cycle-finding technique by Sedgewick, Szymanski and Yao [27]. It temporarily uses a small amount of local storage (which can be input dynamically as a parameter) during the random walks to accelerate collision checking – see §3.4.
- *SIKE-specific optimizations.* Although the best algorithm for the general CSSI problem is generic (i.e. there are no better known algorithms that exploit its underlying mathematical structure), we take advantage of multiple optimizations that apply to the concrete instantiations in the SIKE specification [11]. Firstly, we show how to exploit the choice of the starting curve as a subfield curve, by defining random walks on (conjugate) classes of j-invariants; such a modified walk is analogous to the walk that exploits the negation map in Pollard's rho algorithm for the ECDLP [36] – see §3.1. Secondly, we show how to exploit that, in SIKE, the isomorphism class of the output curve is not randomized (this possibility was already pointed out by De Feo, Jao and Plût [7]), by using the leakage of the dual of the final isogeny – see §3.1. We quantify the precise security loss suffered by these choices.
- *Precomputation.* Generic collision finding algorithms like vOW are often implemented to target high-speed symmetric primitives. In contrast to those applications, for the CSSI problem, the computation of large-degree isogenies is the overwhelming bottle-neck of the random walks. Therefore, 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 isogeny tree – see §3.3.
- *Experimental results.* For all of the improvements mentioned above, we demonstrate their feasibility by analyzing the runtime of the implementation. In doing so, we re-confirm the analyses of van Oorschot and Wiener [32] and Adj *et al.* [1] in the context of SIDH (with a factor 2 improvement) and extend them to SIKE – see Table 1. Furthermore, we go beyond the setting of small parameters and propose an alternative way of predicting the vOW runtime for actual Round 2 parameters, in particular SIKEp434, giving an *upper* bound on their security level – see §5.1.

Finally, in Appendix B we consider the case when the vOW algorithm is used to target k unrelated public keys simultaneously. We extend the theoretical analysis 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

After defining the CSSI problem in §2.1, we describe the classical meet-in-the-middle claw finding algorithm in §2.2. It is both simpler than, and helps motivate, the description of the vOW parallel collision finding algorithm in §2.3. The complexity analysis of the generic vOW algorithm is given in §2.4.

2.1 The CSSI Problem

Herein, we restrict to the popular scenario whereby an instance of SIDH/SIKE is parameterized by 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. Since $p \equiv 3 \bmod 4$, we fix $\mathbb{F}_{p^2} = \mathbb{F}_p(i)$ with $i^2 + 1 = 0$ throughout. 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 [29, p. 146]. Each supersingular j-invariant belongs to the same isogeny class [17].

In this paper, isogenies are non-constant rational maps between two elliptic curves that are also group homomorphisms. We work only with *separable* isogenies, meaning that the degree of any given isogeny is equal to the size of its kernel. Any subgroup $G \subset E$ determines a unique isogeny (up to isomorphism) whose kernel is G; this isogeny can be computed using Vélu's formulas [34].

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. The graph $\mathcal{G}_\ell$ is connected and, with the exception of the nodes corresponding to j-invariants 0 and 1728, an $(\ell + 1)$-regular multigraph which satisfies the Ramanujan expansion property (see [7, §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 discuss the special node with j-invariant 1728 in §3.1.

For any n with $p \nmid n$, the set of n-torsion points, $E[n] = \{P \in E(\bar{\mathbb{F}}_p) : [n]P = 0_E\}$, satisfies $E[n] \cong \mathbb{Z}_n \oplus \mathbb{Z}_n$. Let $(\ell, e) \in \{(2, e_2), (3, e_3)\}$. Following [7, Problem 5.2] (see also [1, §2.4]), we define a simplified version of the CSSI problem that underlies the SIDH and SIKE protocols within the above context as follows.[6]

Definition 1 (CSSI). *Given two supersingular elliptic curves E and E/G defined over $\mathbb{F}_{p^2}$ such that up to isomorphism there exists a unique 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 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 [12] (and assuming for simplicity that e is even), we can build two trees of curves: the leaves of the first determine the set of all isomorphism classes $\ell^{e/2}$-isogenous to that of E, those of the second the set of all 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 is only one class that lies in both sets [12, §5.1]. It corresponds to the node in the middle of the path from E to E/G, and once it is found, the CSSI problem is solved by composing[7] the $\ell^{e/2}$-isogeny emanating from E with the dual of that emanating from E/G. Assuming that all $(\ell + 1)\ell^{e/2-1}$ classes emanating from one of the sides can be computed and stored, solving the CSSI problem this way takes $O(\ell^{e/2})$ time.

It was not until the work of Adj *et al.* [1] 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 [1] 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. At any given time, an attacker can now only afford to store a small fraction of the $O(\ell^{e/2})$ nodes emanating from one side, try all nodes from the other side, and repeat this process until the CSSI problem is solved. Adj *et al.* therefore conclude that, for CSSI instances of cryptographic relevance, the meet-in-the-middle algorithm is more costly than the vOW algorithm described in the sequel.

[6] As in [1, §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 [7, Problem 5.2] by omitting the auxiliary torsion points because the algorithms considered here are independent of the information given by these points.

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

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 $E_1 = E/G$. Each $(i, y) \in S$ represents a kernel subgroup on the elliptic curve E_i. For example, for $\ell = 2$, Adj *et al.* [1, §4.4] 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 $\langle R_i \rangle \subset E_i$ with

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

Let $h : S \to E_0(\mathbb{F}_{p^2}) \cup E_1(\mathbb{F}_{p^2}), (i, y) \mapsto R_i$ and 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 $\langle R_i \rangle$ emanating from E_i, evaluates the j-invariant $j(E_i/\langle R_i \rangle)$, and maps it back to S using a function g. In order to make f behave like a (pseudo-)random function on S, the function $g : \mathbb{F}_{p^2} \to S$ is chosen to be (pseudo-)random.

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 $|S|/2$ [32, §4.2]. For SIDH, we rely on the function h described above, while for SIKE, h is defined 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, we follow convention [32, 1] and refer to it 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 assume that f is a truly random function on S for which we aim to find a single golden collision.

The vOW algorithm requires a proportion θ of the points in $|S|$ to be *distinguished points*. Whether or not a point is distinguished can be decided by any efficiently computable function $S \to \{0,1\}$, so long as it ensures that close to $\theta \cdot |S|$ of the $|S|$ points are deemed distinguished. The algorithm searches for collisions of f by performing many iterative walks in parallel as follows. Each walk starts at a random point $x_0 \in S$ 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.[8]

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. Optimized parametrizations geared towards real-world CSSI instantiations will have $w \ll \theta \cdot |S|$, i. e. one cannot store enough triples to account for all of the distinguished points. This gives rise to three scenarios when we attempt to store a given triple in memory. The first is that the memory at the given address is empty, in which case we write the triple there and continue; the second is that the memory is occupied by a triple with 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 point, in which case we have a collision and we must now check whether or not it 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 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 $x_0 \neq x'_0$ 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 two undesirable occurrences can arise during their algorithm. First, a trail can collide with the starting point of another trail, which is called a *Robin Hood*. In practice, they note that θ is small enough that this occurs rarely. If it does, we replace

[8] In our scenario, many collisions are encountered before the golden collision is found. Starting new trails (rather than continuing on from distinguished points) avoids falling into cycles and repeatedly detecting the same collisions [32, p.6, Footnote 5].

the triple in memory by the triple found last. Second, a walk can enter into a cycle that does not contain a distinguished point. In [32], 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 explained in [32, §4.2], there exist roughly $|S|/2$ collisions for f, and on average we have to find this many collisions before we encounter the golden collision. However, not all collisions occur equally likely; for any given f, the golden collision may have a very low probability of detection. For example, one or both of the two points that constitute the golden collision could have very few trails leading into them, or in the extreme case, none at all; if so we would have to be extremely lucky to find the collision, i. e. by randomly choosing the two points 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 use f_n with $n \in \mathbb{Z}$ instead of f, where the subscript indicates the different function versions.

2.4 Complexity Analysis of van Oorschot-Wiener

Van Oorschot and Wiener give a complexity analysis for finding a golden collision [32, §4.2], but 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. We reproduce this *flawed* analysis, since we refer back to it throughout.

Recall that w triples (x_0, x_d, d) can be stored in memory. Whenever 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 $\sqrt{8N/w}$. As $N/2$ collisions are required (on average) 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 [32, §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 [32]. 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 Adj *et al.* for SIDH.

Equation (1) shows that the memory size of w distinguished points has a crucial influence on the runtime of the vOW algorithm. It is therefore important to store distinguished points as compactly as possible. If the property for a point to be distinguished is a number of leading or trailing zeroes in its bit representation, these zeroes do not have to be stored, shortening the bit length of x_d in the triple (x_0, x_d, d). Given a distinguished point rate θ, the number of zeroes would 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.

This deduction of the total runtime assumes 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 one such function f_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 runtime 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 runtime without having to complete the full algorithm. For that reason, we run the same experiment to estimate the impact of improved collision checking (see Fig. 4 in §3.4).

3 Parallel Collision Search for Supersingular Isogenies

In this section we describe optimizations that we employ when specializing the van Oorschot-Wiener algorithm to SIKE. We discuss improvements based on the SIKE design in §3.1 and explain the specific instantiation of the vOW algorithm in §3.2. Finally, we show how to use 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. In this section, we discuss their impact on the vOW algorithm and 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 Montgomery form [20] $E_A : y^2 = x^3 + Ax^2 + x$ with constant $A \in \mathbb{F}_{p^2}$. The Montgomery form allows for very efficient arithmetic, which is why it has been used in the SIKE proposal. Further note that, if $\{U, V\}$ is a basis of $E[m^{e_m}]$, then the points $\phi(U), \phi(V)$ are given as well. But 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 simply think of a challenge as being given by the curve E_A.

Since isogenies of degree ℓ^{e_ℓ} are determined by cyclic subgroups of size ℓ^{e_ℓ}, 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.* [1], essentially defining a random function on the set of cyclic subgroups.

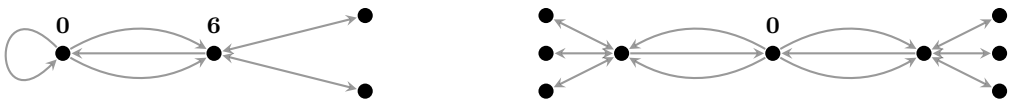
Moving to SIKE, we observe that an important public parameter of its specification is the starting curve E_0. Since $p = 2^{e_2} \cdot 3^{e_3} - 1$ is congruent to 3 modulo 4 for $e_2 > 1$, the curve $y^2 = x^3 + x$ is supersingular for any choice of (large) e_2 and e_3, and this curve was chosen as the starting curve in the Round-1 SIKE specification. In Round 2, the starting curve has been changed to $y^2 = x^3 + 6x^2 + x$.

Choice of secret keys. Any point R of order ℓ^{e_ℓ} on E_0 satisfies $R = [s]P + [r]Q$ for $r, s \in \mathbb{Z}_{\ell^{e_\ell}}$, where both s and r do not vanish modulo ℓ. The SIKE specification [11, §1.3.8] assumes s to be invertible and simply sets $s = 1$. This choice simplifies implementations by making the secret key a sequence of random bits that is easy to sample. When $\ell = 2$, an appropriate choice of P, Q allows to avoid exceptional cases in the isogeny arithmetic [26, Lemma 2]. The main consequence of this is that the key space has size[9] ℓ^{e_ℓ} as opposed to $(\ell + 1)\ell^{e_\ell - 1}$.

[9] Technically, the specification 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 for $\ell = 3$ and we ignore it in our discussion.

The initial step. Our first observation is 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. For $\ell = 3$ the curve has in-degree 2, while its adjacent nodes have in-degree 5; see 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 $E_6 : y^2 = x^3 + 6x^2 + x$ when $\ell = 2$. 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 reason for this behavior 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.

Remark 1. The presence of the distortion map on the node with $j = 1728$ thus leads to loops and double edges in the graph, which reduces the entropy of the private and public keys. This security reduction for SIDH or SIKE can be easily circumvented by moving the starting node from E_0 to E_6 (with $j(E_6) = 287496$), which avoids the loop and double edge for $\ell = 2$. 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. This suggestion has indeed been included in the Round-2 update to the SIKE specification. Note that the Round-1 SIKE specification set up Q as a point of order 2^e defined over $\mathbb{F}_p$ [11, §1.3.3]. Such a point does not exist on E_6, as $E_6[2^e](\mathbb{F}_p) \cong \mathbb{Z}_{2^{e-1}} \times \mathbb{Z}_2$. This only implies that the description of Q is longer as it lies in $E_6(\mathbb{F}_{p^2}) \setminus E_6(\mathbb{F}_p)$.

It is not obvious how the nodes of E_6 and E_0 are connected in the 3-isogeny graph, there is no reason to believe they are close. Therefore, we believe moving to E_6 alleviates issues with double edges in the 3-isogeny graph as well.

The final step. Recall that our elliptic curves are represented in Montgomery form and that isogenies of degree 2^{e_2} are computed as a sequence of 4-isogenies. As already noted in [7, §4.3.2], the choice of arithmetic in SIKE implies that the points $(1, \pm\sqrt{A + 2}) \in E_A$ lie in the kernel of the dual of the secret isogeny. Hence, the final step can be immediately recomputed from the public key. Consequently, $E_A/\langle (1, \pm\sqrt{A + 2}) \rangle$ is isogenous to E_0 by an isogeny of degree 2^{e_2-2}, and to 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} for $\ell = 2$.

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.

Remark 2. To address the issue of leaking the final kernel, we notice that for any $\bar{A} \in \mathbb{F}_{p^2}$ with $j(E_{\bar{A}}) = j(E_A)$ we have

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

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 (2) is equivalent to randomizing the kernel of the final isogeny. Unfortunately, selecting $\bar{A}$ to be anything other than $\pm A$ seems to require an expensive square root. For this reason, we do not suggest full randomization, but emphasize that the random selection of one of $\pm A$ leads to a single bit of randomization at essentially no computational effort. As a result, one would only leak the kernel of the final 2-isogeny (with kernel $(0, 0)$) instead of the last 4-isogeny (see Fig. 2).

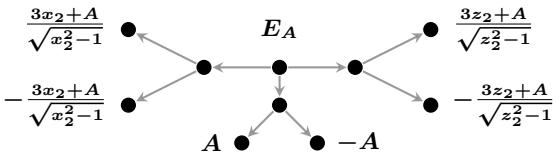


Fig. 2: A sequence of 2-isogenies starting from the curve E_A. Each leaf node is labeled with the Montgomery coefficient $\bar{A}$ with $j(E_{\bar{A}}) = j(E_A)$ such that the isogeny from E_A to that node has kernel $(1, -)$. In particular, it is clear that selecting one of $\pm A$ reduces the leakage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

The Frobenius endomorphism. Every isomorphism class can be represented by an elliptic curve E defined over $\mathbb{F}_{p^2}$ and has an associated Frobenius map $\pi : E \to E^{(p)}, (x, y) \mapsto (x^p, y^p)$. For any kernel $\langle R \rangle \subset E$, we have

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

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. 3 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 isogeny degree is approximately $\sqrt{p}$, making it unlikely for endomorphisms of that degree to exist. As such, the leaves of trees such as in Fig. 3 most probably are all distinct.

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 advantage. 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 $\#E_6(\mathbb{F}_p) = 2^{e_2} \cdot 3^{e_3}$ [29, Exercise V.5.10]. Since the $\mathbb{F}_p$-rational endomorphism ring of E_6 is isomorphic to one of $\mathbb{Z}[\pi]$ or $\mathbb{Z}[(1 + \pi)/2]$ [8, Proposition 2.4], a result by Lenstra [15, Theorem 1(a)] tells us that

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

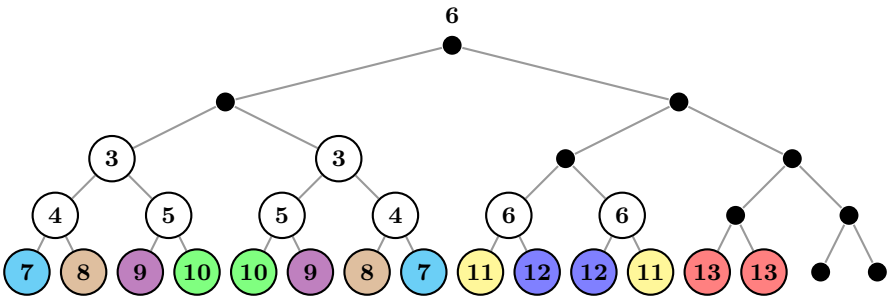


Fig. 3: Part of the 2-isogeny graph for any large $p = 2^{e_2} \cdot 3^{e_3} - 1$ starting at $E_6 : y^2 = x^3 + 6x^2 + x$. Black dots represent curves defined over $\mathbb{F}_p$, j-invariants in the same equivalence class are denoted by equal numbers. All edges represent 2-isogenies. In particular,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

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 [28]) since the group order 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 be uniquely 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}.

Overall, 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. While the Round-2 SIKE specification has moved away from $j = 1728$, the curve E_6 still has a Frobenius endomorphism. 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 less bit of randomness is needed to compute an isogeny of the same degree (see e. g. Fig. 3, where the final step could always move to the left node). These issues can be avoided by moving to a curve where the Frobenius map is not an endomorphism. While this prevents the Frobenius trick, it is a subtle issue (see Remark 3).

Remark 3. The curve $E_0 : y^2 = x^3 + x$ has a known endomorphism ring [29, III.4.4], which is helpful in certain attack scenarios [24]. 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, the walk itself cannot be public and it is unclear how to verifiably achieve this.

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). We point out a subtle mistake in the algorithm (appearing already in the original paper [32] and also used in the work of Adj *et al.* [1]) and show how to overcome it. The solution involves using a different notion of distinguishedness, and it allows us to achieve the average vOW runtime for a *fixed instance*. This allows us to focus on one particular instance, where we are then able to use precomputation in order to analyze the algorithm's behavior (when 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 $S = \{0, 1, \ldots, 2^{e-1} - 1\}$. 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 f_n maps through classes (of size 1 or 2) in $\mathbb{F}_{p^2}$, we first define a canonical representative of the class. Since the conjugate of $j = a + b \cdot i \in \mathbb{F}_{p^2}$ is $\overline{j} = a - b \cdot i$, we say that j is *even* whenever $\mathrm{lsb}(b) = 0$. Using $\gg$ to denote the rightshift operator, we define the function h from S to the set of supersingular j-invariants by

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

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 AES in CBC mode using the AES-NI instruction set (see §4), with the initialization vector and plaintext set to 0 and the key determined by n.

Note that the Frobenius map π is an endomorphism on E_6, but not (necessarily) on E_A. Given $r \in \{0, 1, \ldots 2^{e-2} - 1\}$, 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. Kernels of the form $U + [r]V$ determine 2^{e-2}-isogenies from E_A, all of which lead to distinct, non-conjugate j-invariants. So h maps bijectively into a set of size $2^{e-1} - 1$, with only a single collision given by the isogeny from E_6 to E_A.

Distinguished points and memory. Assume the memory to have size w a 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.* [1, §4.4] determine the memory position of a triple (r_0, r_d, d) using the $\log w$ least significant bits of $\mathrm{MD5}(3, r_d)$. Moreover, the value r_d is distinguished if and only if $\mathrm{MD5}(2, r_d) \leq 2^{32}\theta \mod 2^{32}$ (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. 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 (r_0, r_d, d) found by f_n and (s_0, s_e, e) found by f_{n+1} (say), with $s_e = r_d$, will be classified as a valid collision, triggering the backtracking subroutine. This will fail since f_n and f_{n+1} give rise to different random functions, leading to work going to waste. 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 [32, §4.2], this is only true if one takes the average over all collisions. However, for SIKE (and whenever one wants to find a specific collision), its input values are 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) also occurs for r and s. The runtime will be above average if one or both of r and s are distinguished. This is because the algorithm samples a new starting value every time it reaches r or s, only computing $f_n(r)$ or $f_n(s)$ when they are sampled as initial values. Since distinguishedness is independent of n, this behavior propagates throughout all the f_n.

We give a solution to all of these problems. First, we note that elements of S are uniform bit strings of length $e-1$. Since the value r_d of the triple is always the output of the (random) step function, we simply let the $\log w$ least significant bits determine the memory location. More precisely, the triple (r_0, r_d, d) is stored in the memory location indexed by $(r_d + n) \mod w$. Notice that we choose the location to be dependent on n. Therefore, if two triples $(r_0,\ r_d,\ d)$ and $(s_0,\ s_e,\ e)$ with $s_e = r_d$ are distinguished under functions f_n and f_m respectively (with $n \neq m$), they will be stored at different locations $(r_d + n) \mod w \neq (s_e + m) \mod w$, sparing us the backtracking. Moreover, any other value $(t_0,\ t_c,\ c)$ that is stored during function version f_m at the address of $(r_0,\ r_d,\ d)$ will have $t_c \neq r_d$, and will not be a collision, sparing us the backtracking. Of course, a memory address could be written to during both f_n and f_{n+w} and never in between. But for reasonable values of n and w this is highly unlikely, and it would only incur in the (relatively small) cost of checking for an invalid collision when it happens.

Secondly, we define a better distinguishedness property. Since it should be independent of the memory location, we use the value of $r_d \gg \log w$. As usual, using all of 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

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

With that, every element 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, it holds true heuristically.

We observe that 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 [32, §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 at most B. Adding and subtracting $n \cdot B$ modulo $2^{e-1-\log w}$ when writing to and reading from memory has negligible overhead.

We note that making distinguishedness depend on the function version also causes a triple $(—,\ r_d,\ —)$ to be unlikely to be distinguished often (where time is measured in function versions), giving time to the algorithm to overwrite a stored triple $(r_0,\ r_d,\ d)$ with a different triple $(s_0,\ s_e,\ e)$ with $s_e \neq r_d$, reducing the change of invalid collisions. Since both f_n-dependent memory location and distinguishedness are cheap to realise, we keep both.

Remark 4. 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. They are elements $(i, b, k) \in \{1, 2\} \times \{0, 1, 2\} \times \{0, \ldots, 2^{e_2/2} - 1\}$ [1, §4.4] which are represented as $(3 + e_2/2)$-bit strings where the least significant bit determines i and the 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

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

Here, 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 problem 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: 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 $h(r)$ 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, and evaluates it under g_n (which is comparatively fast). This improves the speed of our benchmarks significantly, while not affecting any outcomes regarding a precise analysis of the vOW algorithm.[10]

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.* [1]. 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 software solves the SIDH instances using less than half the number of steps that were taken for the same instances in [1]. The primes used in Table 1 are

$$23 \cdot 2^{32} \cdot 3^{20} - 1\,, \quad 31 \cdot 2^{36} \cdot 3^{22} - 1\,, \quad 71 \cdot 2^{40} \cdot 3^{25} - 1\,, \quad 37 \cdot 2^{44} \cdot 3^{27} - 1\,,$$
$$13 \cdot 2^{48} \cdot 3^{30} - 1\,, \qquad 2^{52} \cdot 3^{33} - 1\,, \quad 57 \cdot 2^{56} \cdot 3^{35} - 1\,.$$

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**				
e_2	$\log w$	Exp.		[1]	**This**		Exp.		[1]	**This**	
		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 of the vOW step function is the (repeated) evaluation of isogenies of degree $\ell^{e_\ell/2-1}$. To alleviate this burden, one can partially precompute the isogeny tree by computing all possible isogenies of a fixed degree Δ and storing a table of the image curves together with some torsion points (that help to complete the isogenies from these intermediate

[10] Of course, this strategy is not useful for a distributed attack on an actual cryptographically sized problem instance. It only aids the efficiency of small-sized experiments in order to get a better understanding of the algorithm.

curves. Such precomputation presents a trade-off between memory and computation time for the step function). We elaborate on the method in detail. As it applies to the general case of SIDH, we discuss that first and then specialize to SIKE instances with $\ell = 2$.[11]

Let E be a supersingular curve and $P, Q \in E$ be such that $\langle P, Q\rangle = E[\ell^d]$, for some $d > 0$ (typically $d \approx e_\ell/2$). 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, it suffices to store only two points on $E/\langle R_\Delta\rangle$. If $\ell \nmid s$, we can assume that $s = 1$ and $R = P + [r]Q$ for $r \in \mathbb{Z}_{\ell^d}$. Then $R_\Delta = [\ell^{d-\Delta}]P + [r_0 \cdot \ell^{d-\Delta}]Q$ and the precomputed table only needs to contain entries of the form

$$\left[E/\langle R_\Delta\rangle, P_\Delta = \phi_\Delta(P + [r_0]Q), Q_\Delta = \phi_\Delta([\ell^\Delta]Q)\right] \tag{3}$$

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 $\ell \mid s$, 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 $(d-\Delta)\log(d-\Delta)$ [7, §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 (3). 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 $\langle P + [r \gg 1]Q\rangle$ on E_6 if $\mathrm{lsb}(r) = 0$, and an isogeny with kernel $\langle U + [r \gg 1]V\rangle$ 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.[12] As a result, all isogenies used throughout the whole implementation have fixed degree $e-2-\Delta$. The isogeny cost reduces from $(e-1)\log(e-1)$ to $(e-2-\Delta)\log(e-2-\Delta)$ and choosing Δ such that $e-2-\Delta \equiv 0 \bmod 2$ allows the use of 4-isogenies as in SIKE. Table 2 demonstrates the effect of precomputation on the SIKE step function.

[11] The extreme case, when the full isogeny tree from one side is precomputed, corresponds to the meet-in-the-middle algorithm as described by Adj *et al.* [1].

[12] 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.

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

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 kernel representation. Adj *et al.* [1, 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. [1, Fig. 1]). This method is much more efficient than the naive way of computing full $2^{e_2/2}$-isogenies for all possible kernel points. Obviously, it 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. Depending on the specific problem instances and communication properties of the network, the memory required for precomputation could alternatively be used as part of the main memory that stores distinguished points. In other words, precomputed tables might take away a certain amount of memory 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 $\tau(\Delta)$, and for simplicity that every processor precomputes the same depth tree. For example, for the SIDH case of Adj *et al.* [1] 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, this size is $\tau(\Delta) = 2 \cdot 2^{\Delta} = 2^{\Delta+1}$.

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 vOW main memory is reduced from w to $w - 12 \cdot \tau(\Delta) \cdot m$ points (when each of the m processors stores its own table). Thus, the number of function iterations 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 step function cost by a factor $\sigma(\Delta, e)$. We have $\sigma(\Delta, e) = (e-\Delta)\log(e-\Delta)/(e\log e)$ for SIDH (given e_2 is even), while for SIKE (separating the two equally likely cases where we start from E_6 resp. E_A)

$$\sigma(\Delta, e) = \frac{1}{2}\left(\frac{(e-2-\Delta)\log(e-2-\Delta)}{(e-2)\log(e-2)} + \frac{(e-2-\Delta)\log(e-2-\Delta)}{(e-1)\log(e-1)}\right). \tag{4}$$

The total runtime of the van Oorschot-Wiener algorithm decreases if

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

Remark 5. 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. Furthermore, depending on the design of the main memory, each machine may have memory available that cannot contribute to it 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. Example 1 shows that speed-ups for cryptographic parameters can be obtained with very small tables, making this scenario more realistic.

Example 1. Let $p = 2^{216} \cdot 3^{137} - 1$ and $(e, m, w) = (108, 2^{64}, 2^{80})$, following the setup of [1, Remark 6]. For both SIKE and SIDH, the (near) optimal pre-computation depth is $\Delta = 6$ and each processor pre-computes a local table that takes up space for $12 \cdot \tau(\Delta)$ distinguished elements; this requires around 41 resp. 62 kilobytes of memory per processor (totalling 2.34% resp. 3.52% of the full memory w). In both cases, the step function cost is reduced by a factor $\sigma(\Delta, e) \approx 0.93$. For SIKE, we decrease the runtime of the full algorithm by a factor approximately 0.94, for SIDH, by 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 per processor (totalling approximately 3.75% of the total memory w). The cost of the algorithm is reduced by a factor 0.88. For SIDH we obtain a table of size 4.0 megabytes (5.63% of the total memory). The runtime is decreased by a factor 0.89.

3.4 Fast Collision Checking

As discussed in Remark 5, in a distributed implementation processors are likely to have local memory that cannot contribute to the main memory (that which is used for storing distinguished point triples). We now describe another way to use such memory to significantly improve the overall runtime of van Oorschot-Wiener. Analogous to §3.3, even if memory is consumed that could otherwise be used to store distinguished points, we argue that dedicating a moderate amount of storage to this faster collision checking reduces the overall runtime.

Recall from §2.3 that a single walk in the vOW 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 the triple (x_0, x_d, d) collides with a triple, say (y_0, y_e, e), previously stored in main memory and that it is not a mere memory collision. To check if we have found the golden collision, we need to locate the indices $i < d$ and $j < e$ for which $x_i \neq y_j$ and $f(x_i) = f(y_j)$. 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 [32, p. 9]. Our goal is to lower the overall runtime by reducing the number of function iterations for retracing.

Saving intermediate values. Suppose that apart from the global memory for keeping distinguished points, a processor has access to enough local memory to store $t - 1$ additional points intermittently (more on what this means in a moment). On a walk from x_0 to x_d, it now stores $t + 1$ points in total. These points $(x_{d_0} = x_0, x_{d_1}, \ldots, x_{d_t} = x_d)$, where $0 = d_0 < d_1 < \cdots < d_t$, can now be used together with (y_0, y_e), to locate the collision more efficiently.

We start by copying y_0 to y', e to e' and iterate steps $y' \leftarrow f(y'), e' \leftarrow e' - 1$. When y' is the same distance away from the distinguished point as the closest of the saved points, say x_{d_j} (i.e. j is minimal with $e' = d_t - d_j$), 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. This is repeated until y' collides with one of the saved points,

say x_{d_k}. Note that equality checks only occur with the x_{d_i} and not at every step as in the original collision checking function. Once the minimal index k with $y' = x_{d_k}$ is detected, we know that the collision must take place between $x_{d_{k-1}}$ and x_{d_k}. At this point, the original collision checking function without saving intermediate points can be called on the 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`.

What have we gained? First of all, the trail with stored points is not retraced at all, only in the final call to the original collision checking, on a single subinterval of length $d_k - d_{k-1}$, which in general is much shorter than the original trail length d_t. 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. 4 shows how the number of function steps for checking and locating collisions is reduced when running vOW 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. However, additional gains become smaller when increasing this number because, when the maximal number of intermediate points approaches the average trail length, almost every point can be stored and adding more memory does not add more intermediate points, nor influence the distance between them.

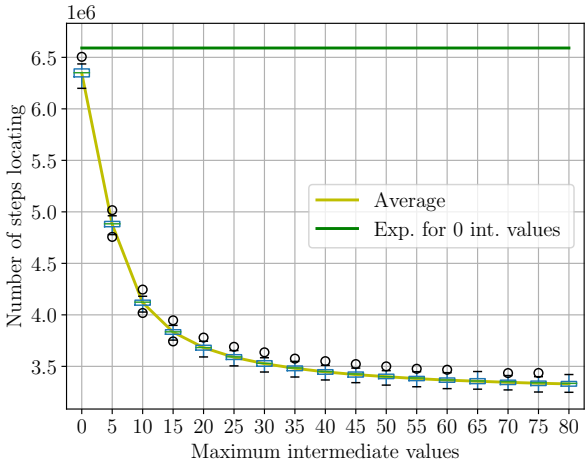


Fig. 4: Number of steps used for locating a collision as a function of maximum amount of intermediate values allowed for the AES-based random function with $\log |S| = 30$, $\log w = 15$. Averaged over 64 function versions, using 28 cores and run on Atomkohle.

Remark 6. There is potential for further improvement by allowing storage for $2t - 2$ points. As above, the $t - 1$ points $(x_{d_0}, \ldots, x_{d_{k-2}}, x_{d_{k+1}}, x_{d_t})$ are stored while walking the trail. But during collision checking against (y_0, y_e, e), $t-1$ additional intermediate points are stored when retracing the trail from y_0. When the collision is encountered, the latter points take the place of the x_{d_i} and $(y_0, y_e) \leftarrow (x_{d_{k-1}}, x_{d_k})$. Storage for the $t-1$ elements $(x_{d_0}, \ldots, x_{d_{k-2}}, x_{d_{k+1}}, x_{d_t})$ can be reused for keeping intermittent points when retracing the trail from the new y_0. Repeating this procedure, we recurse until $y_e = f(y_0)$, at which point we check for the golden collision. Algorithm 1 (see Appendix A), describes this recursion precisely.Note that splitting the space for $2t-2$ points in half eases the exposition, but might be suboptimal. 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.

How to save points intermittently. It remains to describe how the $t-1$ intermediate points are stored. Given the expected trail length of $1/\theta$, one could store points at regular intervals of length $1/(t\theta)$. However, walks much longer than $1/\theta$ would lead to a much larger distance between the final intermediate point and the distinguished point; walks much shorter than $1/\theta$ would lead to unused memory that could have decreased the average gap between intermediate points. In the ideal scenario, a full set of $(t-1)$ additional points is stored and they are as close to being equally spaced as possible when the distinguished point is reached. Since trails randomly vary in length, the best approach involves overwriting previously placed points in such a way that the distances between points grow with the trail length.

We modify an algorithm for finding cycles in random walks by Sedgewick, Szymanski and Yao [27]. In the first t steps of the trail, the allocated memory is exhausted by storing a point at every step, so that $(d_0, d_1, \ldots, d_{t-1}, d_t) = (0, 1, \ldots, t-1, t)$, 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 steps away from another point will be deleted and replaced by a point that is twice as far away from the last; by definition, δ is simultaneously doubled and all of the points in the list are δ away from each other. This process is applied at lines 19 and 21 of Algorithm 1 (see Appendix A).

4 Implementation

We produced two implementations of the van Oorschot-Wiener algorithm, one in C, optimized for efficiency, and a more modular one in C++. The C implementation makes use of the Microsoft SIDH library [18] for field and curve arithmetic when running the attack against SIDH and SIKE instances. 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 [22]. For simplifying batch experiments we wrote Python wrappers to our code using SWIG [4].

The experiments are run on two different machines. The first, referred to as Atomkohle, contains two Intel(R) Xeon(R) E5-2690 v4 CPUs running at 2.60GHz that both have 14 physical cores (so 28 in total). The second, referred to as Solardiesel, contains two Intel(R) Xeon(R) Gold 6138 CPUs at 2.00GHz that have 20 cores each (40 in total). Unless specified otherwise, all measurements and statistics reported in this paper have been produced using the C implementation and are compiled with gcc version 6.3.0.

Optimized Implementation. The C 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 [32]) and linear speed-up on larger sets(see Appendix D), while also displaying our improvements in the SIDH and SIKE settings (e. g. as shown in Table 1).

Modular Implementation. While for all SIDH and SIKE experiments we used our C implementation on individual multi-core machines, it would be interesting to deploy the van Oorschot-Wiener algorithm in alternative settings. For example, running attacks with more cores distributed over the internet could change the balance between the cost of a step function evaluation and the cost of memory access, and would certainly present memory topology and core synchronization challenges, see Appendix C. Furthermore, collision-finding techniques play a role in the cryptanalysis of other encryption schemes, e. g. NTRU [10, 33], where memory constrained cryptanalytic experiments could be useful. Since it could be tricky to adapt our C code to such varied settings, we also produced a C++ implementation with the goal of obtaining a more modular, developer-friendly, code base. Test results on a fast, generic, random function showing that it matches the

Table 3: Reproduction of Table 3 from [1], using our C++ implementation, using an AES-based generic random function on Atomkohle. Experiments are run using 20 cores. $\#f_n$ is the number of different random functions used per instance.

			Expected		**Average**		
$\log \lvert S \rvert$	$\log w$	#runs	$\#f_n$	$\log \sqrt{\lvert S \rvert^3/w}$	$\#f_n$	$\log \sqrt{\lvert S \rvert^3/w}$	cycles
18	9	1000	230.40	23.82	204.74	23.83	30.23
20	10	1000	460.80	26.32	420.01	26.27	30.57
22	11	1000	921.60	28.82	898.79	28.86	33.05
24	13	1000	921.60	30.82	850.49	30.74	34.89

expected asymptotics can be found in Table 3. Ideally, it should not be too difficult to write "drivers" for access to different forms of memory (say, storage over the internet rather than local RAM), or different sets S and step functions f_n.

Selecting a XOF and PRNG. One goal of actually implementing vOW is to verify the runtime against the asymptotic theoretical values, using a fast random function. Adj *et al.* [1] chose to use an MD5-based random function for this purpose. We have instead opted for a custom XOF based on AES-CBC mode using AES-NI instructions. This provides much better performance on modern hardware, while guaranteeing cryptographic properties of the function. Regarding our PRNG, we use AES-CTR mode with AES-NI instructions. In Appendix F, we discuss the alternative options that were considered.

In Table 4 we reproduce [32, 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 [32, Table 1], using the AES-based XOF on Solardiesel, i. e. the number of function steps required to find the golden collision divided by $\lvert S \rvert^{3/2}/w^{1/2}$. 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

5 Analysis of SIKE Round-2 Parameters

In the Round 2 of the NIST standardization effort, the analyses of Adj *et al.* [1] and Jaques and Schanck [13] have prompted the introduction of two new parameter sets to the SIKE submission, SIKEp434 and SIKEp610, as well as a security reassessment of the parameter sets SIKEp503 and

SIKEp751. The four sets are based on the primes p434 $= 2^{216}3^{137} - 1$, p503 $= 2^{250}3^{159} - 1$, p610 $= 2^{305}3^{192} - 1$ and p751 $= 2^{372}3^{239} - 1$, and target security categories 1, 2, 3 and 5, respectively.

This section provides concrete classical security estimates for these parameter sets, in two different ways; the first follows an approach similar to the one by van Oorschot and Wiener and Adj *et al.*. We count the average number of oracle calls to run the vOW algorithm and multiply them by the complexity of the oracle itself, measured in x64 instructions. This leads to a more informed estimate than provided by Adj *et al.* and Jaques and Schanck, but the final result remains the same – see §5.1. A downside of this approach is that although it captures much of the algorithm's cost, it ignores some potentially significant parts. In particular, it does not account for the cost of memory access (assumed free) or the practical difficulty of scaling across different cores (assumed linear), see [1, §5, Remark 6]. We present an alternative method in §5.2.

5.1 Concrete Security of SIKE Round-2 Parameters

In Table 5 we use the Round-2 SIKE implementation to estimate the number of x64 instructions necessary to compute *half-size* isogenies. More specifically, we provide estimations for $2^{\lfloor e_2/2 \rfloor - 2}$-isogenies in Table 5a and for $3^{\lfloor e_3/2 \rfloor}$-isogenies in Table 5b. 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. A lower bound on the runtime of the vOW algorithm can now simply be obtained by multiplying the costs of the above isogeny oracles with the number of times they are called, which we summarize in Table 6. Our analysis concludes that the number of classical gates required for (i) vOW on SIKEp434 is at least 2^{143}, (ii) vOW on SIKEp503 is at least 2^{170}, (iii) vOW on SIKEp610 is at least 2^{210}, and (iv) vOW on SIKEp751 is at least 2^{262}. Note that the counts for (i) and (iii) closely agree with classical gate counts by Jaques and Schanck, who are also rather conservative in their costing of the isogeny functions – see [13, §7.1].

5.2 Concrete Security of SIKEp434

Finally, we focus our attention on arguably the most interesting cryptanalytic target, namely the SIKE Round-2 category-1 parameter set SIKEp434 with claimed (classical) security comparable to AES-128. Although the analysis in the previous section shows agreement between our estimates and those in the literature, all approaches so far have one thing in common: communication and memory access costs are not taken into account. As these become non-negligible when the memory and the number of cores grow — already mentioned in the context of SIDH/SIKE by Adj. *et al.* [1, Remark 6] — one can wonder how significant they are. Since such costs are often difficult to capture in theoretical models, we take a more practical approach.

We start by noticing that the current complexity estimates are measured in *average number of oracle calls*, where an oracle call corresponds to an isogeny computation (e. g. of degree 2^{106} or 3^{68} for SIKEp434). Given the fact that we now have an optimized implementation of the algorithm itself, a simple alternative is to measure the complexity in *average number of cycles* instead. Much of the heuristic approach of van Oorschot and Wiener [32, §4.2] remains the same; we run a single function version and measure the number of distinct collisions it generates, from which we approximate the runtime of the full algorithm. That is, we assume that each function version behaves approximately the same with respect to the number of distinct collisions it generates, which van Oorschot and Wiener heuristically show to be true for $w \geq 2^{16}$ (the results for different function versions are within 1% of one another). Thus, writing N for the set size and c for the number of distinct collisions generated per function version, every function version has (independent) probability $2c/N$ to find the golden collision and completing the vOW algorithm requires on average $N/(2c)$ versions. If each one requires t cycles to complete, the average total runtime is therefore $tN/(2c)$.

Equivalently, on average we need t/c cycles per generated collision, of which there are $N/2$ in total, leading to the above average runtime. Therefore, one may want to simplify the analysis by

Table 5: Isogeny costs in terms of the total number of x64 instructions $i_{\mathbf{sum}}$, broken down into multiplication instructions $i_{\mathbf{mul}}$, addition, subtraction and logical instructions $i_{\mathbf{asl}}$ and move instructions $i_{\mathbf{mov}}$; **M** denotes multiplication, **S** squaring, **add** addition and **sub** subtraction in $\mathbb{F}_{p^2}$.

	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
SIKEp503	362	189	2582	1858	2047	1480	905376	3332506	2099672	22.60
SIKEp610	434	255	3266	2398	2653	1888	1638294	5433856	3553530	23.34
SIKEp751	548	334	4196	3100	3434	2432	3254832	9365124	9863656	24.42

(a) Costs for a $2^{\lfloor e_2/2 \rfloor - 2}$-isogeny (omitting single 2-isogenies for odd exponent) using an optimal strategy composed of quadrupling and 4-isogeny steps; DBL denotes a point doubling, 4-iso a 4-isogeny computation, and the cost for DBL is assumed to be 4**M**+2**S**+2**add**+2**sub** and for 4-iso it is 6**M**+6**S**+7**add**+4**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
SIKEp503	229	275	3253	2520	4537	2978	1172192	4479442	2875831	23.02
SIKEp610	290	350	4130	3200	5770	3780	2112930	7266720	4861220	23.76
SIKEp751	395	429	5339	4120	7191	4910	4208868	12471749	13228173	24.83

(b) Costs for a $3^{\lfloor e_3/2 \rfloor}$-isogeny (omitting single 3-isogenies for odd exponent) using an optimal strategy composed of point tripling and 3-isogeny steps; TPL denotes a point tripling, 3-iso a 3-isogeny computation and the cost for TPL is assumed to be 7**M**+5**S**+3**add**+7**sub** and for 3-iso it is 6**M**+5**S**+14**add**+5**sub**.

Table 6: Average number of x64 instructions to run vOW on the 2- and 3-torsion for the Round-2 SIKE parameters with memory size $w = 2^{80}$, set size $N = |S| = 2^{e_2/2-1}$ for the 2-torsion and $N = |S| = 3^{(e_3-1)/2}$ for the 3-torsion – see §3. Numbers are shown as the floor of their base-2 logarithms. The number of isogeny computations, #isog, is computed by setting $m = t = 1$ in Eq. (1), and the numbers $i_{\mathbf{sum}}$ of instructions for each isogeny are taken from Tables 5a and 5b. The total number of instructions, **vOW**, is the product of #isog and $i_{\mathbf{sum}}$ and is intended to act as a lower bound on the number of gates required to solve the CSSI problem 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**
SIKEp503	124	147	23	**170**	125	149	23	**172**
SIKEp610	151	187	23	**210**	150	187	23	**210**
SIKEp751	185	238	24	**262**	188	244	24	**268**

generating only very few collisions and approximating the runtime from that. However, we note that t/c is very large in the beginning of the algorithm as the memory starts out being empty, while the distribution of distinguished points in memory becomes biased towards those with lower probability of producing a collision – see [32, §4.2]. It may be possible to run less than a full function version to get a close approximation of t/c, but we consider this out of scope for this work and stick with completing a function version for our estimations.

Looking at the conjectured setup proposed by Adj *et al.* (i. e. memory $w = 2^{80}$, $m = 2^{64}$ cores), when used against SIKEp434 the number of oracle calls grows linearly with $\sqrt{Nw}/m$, where $N = 2^{107}$, while each oracle call takes on the order of 2^{22} x64 instructions (see Table 5a). In the theoretical model where memory accesses are free and the algorithm parallelizes perfectly, the function version can be run with approximately $2^{51.5}$ x64 instructions per core (and to run the full algorithm we need approximately 2^{27} function versions, agreeing with the estimates in Table 6). If each x64 instruction were a single cycle on a machine running at 1 GHz, such a computation would finish in about 37 days. Although it should be noted that such a setup is not realistic, other combinations of resources allow for (theoretically) running a single function version within a reasonable amount of time (say, a year). It is not clear that these runtimes will hold true in practice, as for example distributing the experiment across different machines can cause significant overhead. We consider exploring this overhead, e. g. by analyzing how different network topologies affect the results, a very worthwhile research direction.

In a more constrained environment, i. e. when running experiments on Atomkohle for which we choose $w \in \{2^{16}, 2^{18}, 2^{20}\}$ and $m = 28$, running a single SIKEp434 function version requires millions of years. Instead, we decrease the degree e of the isogeny we try to reconstruct, but do not change the finite field, to a point where experiments run in a few hours. Crucially, if the theoretical analysis of van Oorschot and Wiener holds up for these resources, then the runtime of a function version grows linearly with $\sqrt{N}$ and we can extrapolate the runtime of a single function version for the actual SIKEp434 parameters on such a setup by drawing a line through the data points. Interestingly, the difference between this approximation of the security of SIKEp434 when compared to the theory can be seen as an error measure for the theoretical analysis of vOW (the better the fit, the closer the theory to reality).

More concretely, we choose $e = 28, 30, \ldots, 42$ and measure the cycle counts to complete one function version and the number of distinct collisions that they generate. We use precomputation depth $\Delta = 16$ and to account for the difference of the cost of the oracle (a 2^e-isogeny) we normalize the cycle count by a factor

$$\sigma(\Delta, e) \cdot \zeta(e)\,, \quad \text{with } \zeta(e) = (1/2) \cdot ((e-2)\log(e-2) + (e-1)\log(e-1))$$

the estimated average cost of the oracle and $\sigma(\Delta, e)$ given as in Equation (4). Hence, we have a measure for the average number t/c of cycles required to generate a single collision, which we summarize in Table 7.

For a fixed w, we then extrapolate, using the least squares method, the function that maps $\sqrt{2^{e-1}}$ to the corresponding value in the table. This leads to the three approximation functions

$$\begin{aligned}
z_{16}(e) &= \sigma(\Delta, e) \cdot \zeta(e) \cdot (3.44.. \cdot \sqrt{2^{e-1}} + 19247.78..)\,,\\
z_{18}(e) &= \sigma(\Delta, e) \cdot \zeta(e) \cdot (1.72.. \cdot \sqrt{2^{e-1}} + 6151.88..)\,,\\
z_{20}(e) &= \sigma(\Delta, e) \cdot \zeta(e) \cdot (0.87.. \cdot \sqrt{2^{e-1}} - 928.81..)\,,
\end{aligned}$$

where the factor $\sigma(\Delta, e) \cdot \zeta(e)$ is only there to undo the normalization factor. For any w, the runtime of a single function version for SIKEp434 is then $z_{\log(w)}(e)$ cycles, while the full algorithm has total runtime $2^{e-2} \cdot z_{\log(w)}(e)$ cycles, since $|N| = 2^{e-1}$. Thus, setting $e = 108$, we expect vOW on SIKEp434 to have a runtime of $2^{170.47..}$, $2^{169.47..}$ and $2^{168.50..}$ cycles for $w = 2^{16}$, $w = 2^{18}$ and $w = 2^{20}$ respectively. For comparison, using Equation (1) combined with the approximation of the cost of the isogeny oracle of Table 5a, we expect runtimes $2^{170.71..}$, $2^{169.71..}$ and $2^{168.71..}$ x64 instructions respectively. We observe that these approximations match very closely, confirming

Table 7: Number of cycles (measured in thousands and rounded to the nearest multiple of 10^3) to generate a single collision, for different memory sizes w and isogeny instances of degree 2^{e_2}, where $e = e_2/2$. All numbers are scaled by a factor $\sigma(\Delta, e) \cdot \zeta(e)$.

w	e							
	28	30	32	34	36	38	40	42
2^{16}	69	113	177	391	726	1 331	2 257	5 261
2^{18}	–	57	90	196	362	659	1 122	2 642
2^{20}	–	–	46	99	182	331	557	1 340

that the theoretical estimates lie very close to the practical runtimes for these values of w and m. Indeed, this is no surprise, as such small values should not cause significant overhead.

However, we emphasize that this is the first time a theoretical estimate on the security of SIKEp434 is met with serious practical consideration (i. e. without ignoring memory access times and issues with parallelism). If our setup with $w = 2^{20}$ was run on an instance with $e = 108$ it would require (on average) $2^{168.50..}$ cycles to complete. We believe this value could therefore be viewed as an *upper* bound on the security level of SIKEp434. On the other hand, the analyses of Adj *et al.* [1] and Jaques and Schanck [13], assuming $w = 2^{80}$ and $m = 2^{64}$, provide a *lower* bound on the security level. This gap could be closed by computing $z_{\log(w)}$ for larger values of w and m and showing that they agree with the theoretical estimations, which is a valuable effort that should be seriously considered to understand the security of SIKEp434. It is of course not clear that the gap between the upper and lower bound will vanish completely; scaling the setup to large memory and distributed systems will cause significant overhead, which is also noticeable in cryptanalytic efforts in other domains [35].

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

[1] 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 *SAC 2018*, pages 322–343. Springer, 2019.

[2] Reza Azarderakhsh, David Jao, Kassem Kalach, Brian Koziel, and Christopher Leonardi. Key Compression for Isogeny-Based Cryptosystems. In *AsiaPKC 2016*, pages 1–10. ACM, 2016.

[3] 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 Güneysu, 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.

[4] David M. Beazley. SWIG: An Easy to Use Tool for Integrating Scripting Languages with C and C++. In *USENIX Tcl/Tk Workshop*. USENIX Association, 1996.

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

[6] Craig Costello, David Jao, Patrick Longa, Michael Naehrig, Joost Renes, and David Urbanik. Efficient Compression of SIDH Public Keys. In *EUROCRYPT 2017*, pages 679–706. Springer, 2017.

[7] 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.

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

[9] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *STOC '96*. ACM, 1996.
[10] Nick Howgrave-Graham. A hybrid lattice-reduction and meet-in-the-middle attack against NTRU. In Alfred Menezes, editor, *Advances in Cryptology - CRYPTO 2007, 27th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2007, Proceedings*, volume 4622 of *Lecture Notes in Computer Science*, pages 150–169. Springer, 2007.
[11] David Jao, Reza Azarderakhsh, Matthew Campagna, Craig Costello, Luca De Feo, Basil Hess, Amir Jalali, Brian Koziel, Brian LaMacchia, Patrick Longa, Michael Naehrig, Geovandro Pereira, Joost Renes, Vladimir Soukharev, and David Urbanik. SIKE: Supersingular Isogeny Key Encapsulation. Manuscript available at sike.org/, 2017.
[12] David Jao and Luca De Feo. Towards Quantum-Resistant Cryptosystems from Supersingular Elliptic Curve Isogenies. In *PQCrypto 2011*, pages 19–34. Springer, 2011.
[13] Samuel Jaques and John M. Schanck. Quantum cryptanalysis in the RAM model: Claw-finding attacks on SIKE. In *CRYPTO 2019*, pages 32–61. Springer, 2019.
[14] John Kelsey, Shu-jen Chang, and Ray Perlner. SHA-3 derived functions: cSHAKE, KMAC, TupleHash, and ParallelHash. Technical report, NIST, 2016.
[15] Hendrik W. Lenstra, Jr. Complex Multiplication Structure of Elliptic Curves. *Journal of Number Theory*, 56(2):227 – 241, 1996.
[16] Linux man-pages project. rand(3) - linux manual page. http://man7.org/linux/man-pages/man3/srand.3.html. Accessed: 2019-02-11.
[17] 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.
[18] Microsoft. SIDH Library v3.0. Available for download at https://github.com/Microsoft/PQCrypto-SIDH, 2015–2019.
[19] Victor S. Miller. The Weil Pairing, and Its Efficient Calculation. *Journal of Cryptology*, 17(4):235–261, Sep 2004.
[20] Peter L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of computation*, 48(177):243–264, 1987.
[21] National Institute of Standards and Technology. Post-quantum cryptography standardization, December 2016. https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Post-Quantum-Cryptography-Standardization.
[22] OpenMP Architecture Review Board. OpenMP Application Program Interface Version 4.5, November 2015.
[23] Kenneth G. Paterson. Personal communication, 2019.
[24] Christophe Petit. Faster Algorithms for Isogeny Problems Using Torsion Point Images. In *ASIACRYPT 2017*, pages 330–353. Springer, 2017.
[25] John M. Pollard. A monte carlo method for factorization. *BIT Numerical Mathematics*, 15(3):331–334, Sep 1975.
[26] Joost Renes. Computing Isogenies Between Montgomery Curves Using the Action of $(0, 0)$. In *PQCrypto 2018*, pages 229–247. Springer, 2018.
[27] 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.
[28] Daniel Shanks. Class number, a theory of factorization, and genera. In *Proc. Symp. Pure Math*, volume 20, pages 415–440, 1971.
[29] Joseph H Silverman. *The Arithmetic of Elliptic Curves*, volume 106. Springer Science & Business Media, 2009.
[30] Mario Szegedy. Quantum Speed-Up of Markov Chain Based Algorithms. In *FOCS 2004*, pages 32–41. IEEE, 2004.
[31] Seiichiro Tani. Claw finding algorithms using quantum walk. *Theor. Comput. Sci.*, 410(50):5285–5297, 2009.
[32] Paul C. van Oorschot and Michael J. Wiener. Parallel Collision Search with Cryptanalytic Applications. *Journal of Cryptology*, 12(1):1–28, 1999.
[33] Christine van Vredendaal. Reduced memory meet-in-the-middle attack against the ntru private key. *LMS Journal of Computation and Mathematics*, 19(A):43–57, 2016.
[34] Jacques Vélu. Isogénies entre courbes elliptiques. *Comptes Rendus de l'Académie des Sciences des Paris*, 273:238–241, 1971.
[35] Michael J. Wiener. The Full Cost of Cryptanalytic Attacks. *Journal of Cryptology*, 17(2):105–124, 2004.

[36] Michael J. Wiener and Robert J. Zuccherato. Faster Attacks on Elliptic Curve Cryptosystems. In *SAC 1999*, pages 190–200. Springer, 1999.

[37] 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 *PQCrypto 2018*, pages 248–268. Springer, 2018.

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 6 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 corresponding to the two kernel subgroups and returns `true` if they are equal, and `false` otherwise.

Algorithm 1: `FastCollisionCheck`

Input: $\mathbf{x} = (x_{d_0}, x_{d_1}, \ldots, x_{d_t})$, (y_0, y_e) with $x_{d_t} = y_e$, $0 = d_0 < d_1 < \cdots < d_t$.
Output: `true`, x, y, such that $f(x) = f(y)$ or `false`

1 **if** $e = 1$ **then**
2 **while** $d_t - d_{t-1} > 1$ **do**
3 $x_{d_{t-1}} \leftarrow f(x_{d_{t-1}})$
4 $d_{t-1} \leftarrow d_{t-1} + 1$
5 **end**
6 **return** `IsGolden`$(x_{d_{t-1}}, y_0)$
7 **else**
8 **while** $e > d_t$ **do**
9 $y_0 \leftarrow f(y_0)$
10 $e \leftarrow e - 1$
11 **end**
12 **if** $y_0 = x_{d_0}$ **then**
13 **return** `false` `// Robin Hood`
14 **end**
15 $k = \min\{m \mid d_t - d_m \leq e\}$
16 Initialize $z \leftarrow y_0$
17 **repeat**
18 $x \leftarrow x_{d_k}$
19 Initialize fresh trail $\mathbf{z} = (z)$ `// and/or overwrite existing`
20 **while** $e > d_t - d_k$ **do**
21 $z \leftarrow f(z)$ (trail stored in $\mathbf{z}$) `// intermittent storage`
22 $e \leftarrow e - 1$
23 **end**
24 $k \leftarrow k + 1$
25 **until** $z = x$
26 $(y_0, y_e) \leftarrow (x_{d_{k-2}}, x)$
27 $e \leftarrow d_{k-1} - d_{k-2}$
28 $\mathbf{x} \leftarrow \mathbf{z}$ `// overwrite`
29 **return** `FastCollisionCheck`$(\mathbf{x}, (y_0, y_e))$
30 **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 $k = 2$, $k = 3$ 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 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' such that its elements correspond uniformly to subgroups belonging to curves in the set $\{E, E/G_1, E/G_2, \ldots, E/G_k\}$ via the correspondence implicit in the function h. The function $f = g \circ h$ 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 [32], 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 the representation of elements in S' to account for this;having 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. 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 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 [25]), the van Oorschot-Wiener algorithm presents some issues regarding synchronization of the random function being used across machines in any specific window of time. Indeed, this makes the addition of hot plugging 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 [3]. In this section we discuss some of the issues and possible solutions.

Each function version used in the vOW algorithm has a certain "shelf life", given by the number of distinguished points to be found during its use (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. For a single-machine multi-threaded implementation of the algorithm, one may consider having a global counter of the distinguished points already 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. 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 could be used to express the current function version being used across cores.

While this works for experiments with small set sizes, 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 [1, 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 realize this functionality, 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 one 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 $\mathcal{W}^{\text{th}}$ 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. 5 we present the result for SIKE with $e_2 = 32$ and $\log w = 9$. We decided to plot the *inverse* of the average wall time (in seconds), since that should show a linear improvement (as it does).

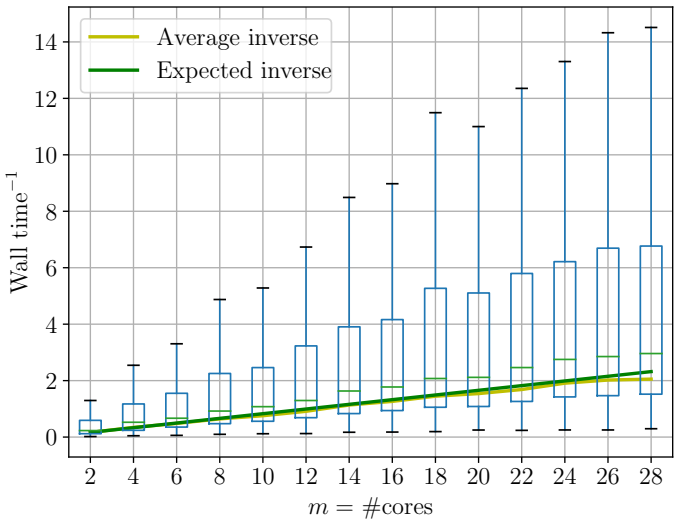


Fig. 5: Inverse wall time as a function of the number of cores used for the attack, averaged over 1000 function versions on Atomkohle. The experiment is run on SIKE with $e_2 = 32$, $\log w = 9$ with the *stakhanov* synchronization strategy, using no precomputation. Outliers are hidden to improve plot 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 numbers 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, it is likely that caching would not be useful. A bare-bones implementation could consist

in an `mmap`'d amount of storage across different machines, with daemons 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 their memory to precompute isogeny tables, for example. 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.

D Linear speedups 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.

E Comparing Sync Strategies

In § C, 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. In Figures 8 and 9, 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.

F 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 that theoretical values are being met in practice. Adj *et al.* use an MD5-based random function for this purpose. Originally, we used an implementation of cSHAKE [14] as XOF to construct the function, but it resulted in poor performance. We have hence moved to an AES-based construction taking advantage of the AES-NI instruction set.

We also considered using non-cryptographic hash functions, usually used for implementing hash tables or check sums. 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 [5], and ran multiple experiments. In Tables 8 and 9, we provide a comparison of our results using the AES-based XOF vs the xxHash-based one, showing experiments using the latter to be slightly more than 50% faster (in the number of cycles required) while displaying the same asymptotic behavior.

To implement 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) [16]. 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 [23]. 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.

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 numbers of cores. Outliers are hidden to improve plot scale.

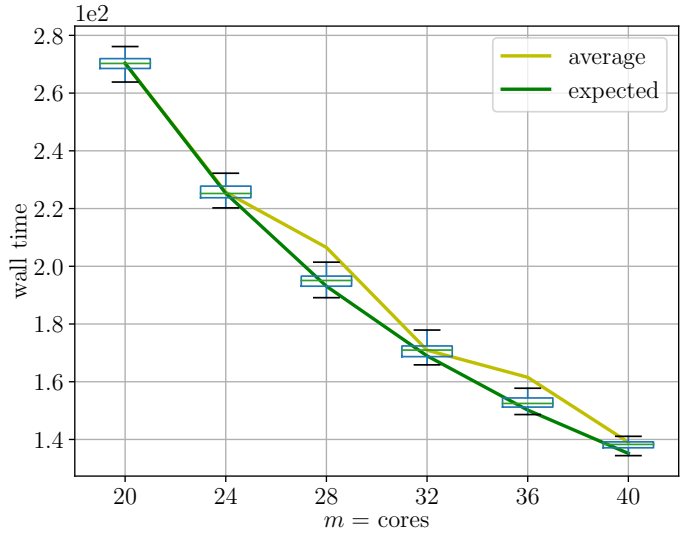


Fig. 7: Box plot for total number of step function calls made as a function of the number of cores used for the attack, averaged over 64 function versions on atomkohle. AES-based random function with $\log |S| = 52$, $\log w = 13$.

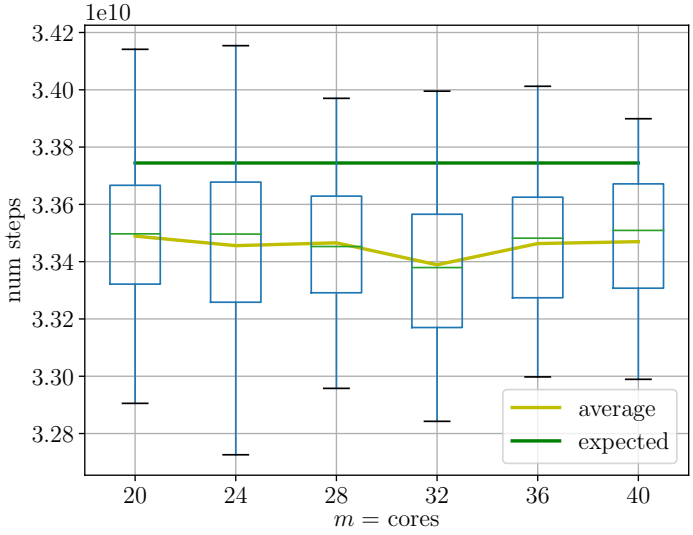


Fig. 8: Box plot for inverse wall time as a function of the number of cores used for the attack, averaged over 1000 function versions on Atomkohle. SIKE with $e_2 = 32$, $\log w = 9$ with windowed sync strategy for $\mathcal{W} = 10$, using no precomputation. Outliers are hidden to improve plot scale. 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.

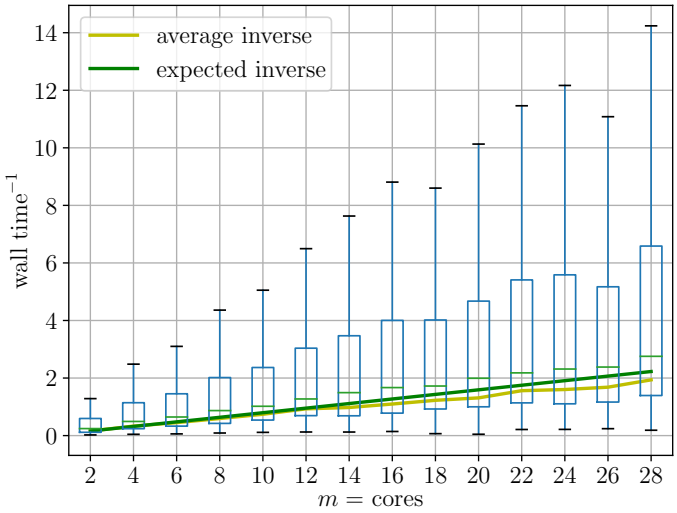


Fig. 9: Box plot for inverse wall time as a function of the number of cores used for the attack, averaged over 1000 function versions on Atomkohle. SIKE with $e_2 = 32$, $\log w = 9$ with the third proposed sync strategy, using no precomputation. Outliers are hidden to improve plot scale. The expected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

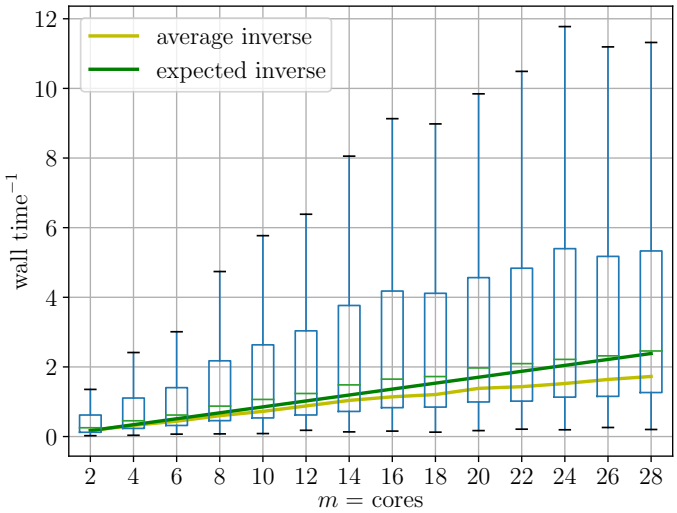


Table 8: Reproduction of Table 3 from [1], using an AES-based generic random function on Atomkohle. Experiments are run using 20 cores.

			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 9: Reproduction of Table 3 from [1], using 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

G 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 [1] 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 [2, 6, 37]. 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

$$R_1 = [\ell^{e_{(2)}}]R, \tag{5}$$

$$\langle \hat{R}_2 \rangle = \langle \phi_1(R) \rangle. \tag{6}$$

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[13] 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 (5) is satisfied. It now remains to determine s such that condition (6) 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. [19, 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 (5) and (6), which means it is a generator for the kernel of the ℓ^e-isogeny ϕ.

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