Parallel Isogeny Path Finding with Limited Memory

Emanuele Bellini[1], Jorge Chavez-Saab[1,3], Jesús-Javier Chi-Domínguez[1], Andre Esser[1], Sorina Ionica[2], Luis Rivera-Zamarripa[1], Francisco Rodríguez-Henríquez[1,3], Monika Trimoska[4], and Floyd Zweydinger[5]

[1] Technology Innovation Institute, UAE
{emanuele.bellini,jorge.saab,jesus.dominguez,andre.esser, luis.zamarripa,francisco.rodriguez}@tii.ae

[2] Université de Picardie Jules Verne, France sorina.ionica@u-picardie.fr
[3] CINVESTAV-IPN, Mexico francisco@cs.cinvestav.mx
[4] Radboud University, The Netherlands monika.trimoska@ru.nl
[5] Ruhr University Bochum, Germany floyd.zweydinger@rub.de

Abstract. The security guarantees of most isogeny-based protocols rely on the computational hardness of finding an isogeny between two supersingular isogenous curves defined over a prime field $\mathbb{F}_q$ with q a power of a large prime p. In most scenarios, the isogeny is known to be of degree ℓ^e for some small prime ℓ. We call this problem the Supersingular Fixed-Degree Isogeny Path (*SIPFD*) problem. It is believed that the most general version of *SIPFD* is not solvable faster than in exponential time by classical as well as quantum attackers.

In a classical setting, a meet-in-the-middle algorithm is the fastest known strategy for solving the *SIPFD*. However, due to its stringent memory requirements, it quickly becomes infeasible for moderately large *SIPFD* instances. In a practical setting, one has therefore to resort to time-memory trade-offs to instantiate attacks on the *SIPFD*. This is particularly true for GPU platforms, which are inherently more memory-constrained than CPU architectures. In such a setting, a van Oorschot-Wiener-based collision finding algorithm offers a better asymptotic scaling. Finding the best algorithmic choice for solving instances under a fixed prime size, memory budget and computational platform remains so far an open problem.

To answer this question, we present a precise estimation of the costs of both strategies considering most recent algorithmic improvements. As a second main contribution, we substantiate our estimations via optimized software implementations of both algorithms. In this context, we provide the first optimized GPU implementation of the van Oorschot-Wiener approach for solving the *SIPFD*. Based on practical measurements we extrapolate the running times for solving different-sized instances. Finally, we give estimates of the costs of computing a degree-2^{88} isogeny using our CUDA software library running on an NVIDIA A100 GPU server.

Keywords: isogenies · cryptanalysis · GPU · golden collision search · Meet-in-the-Middle · time-memory trade-offs · efficient implementation

1 Introduction

Let E_0 and E_1 be two supersingular isogenous elliptic curves defined over a finite field $\mathbb{F}_q$, with q a power of a large prime p. Computing an isogeny $\phi\colon E_0 \to E_1$ is believed to be hard in the classical as well as the quantum setting and is known as the Supersingular Isogeny Path (*SIP*) problem. In many scenarios, the isogeny is of known degree ℓ^e for some small prime ℓ and we refer to this variant as the Supersingular Fixed-Degree Isogeny Path (*SIPFD*) problem. Investigating the concrete computational hardness of *SIPFD* and the best approaches to tackle it in multi- and many-core CPU and GPU platforms, is the main focus of this work.

In the context of cryptographic protocols, *SIP* was first studied by Charles, Goren and Lauter [6]. They reduced the collision resistance of a provably secure hash function to the problem of finding two isogenies of equal degree ℓ^n for a small prime ℓ and $n \in \mathbb{Z}$ between any two supersingular elliptic curves. This in turn may also be tackled as a *SIPFD* problem.

Variants of the *SIPFD* problem form the basis of several isogeny-based signatures [17,33]. Further, *SIPFD* has been used as foundation of recently proposed cryptographic primitives such as Verifiable Delay Functions [11,7]. Based on the intractability of the *SIPFD* problem, Jao and De Feo proposed the Supersingular Isogeny-based Diffie-Hellman key exchange protocol (SIDH) [18,10]. Apart from revealing the isogeny degree, SIDH also reveals the evaluation of its secret isogenies at a large torsion subgroup. This weaker variant of *SIPFD* was dubbed as the *Computational Supersingular Isogeny* (CSSI) problem [10]. SIKE [2], a variant of SIDH equipped with a key encapsulation mechanism, was one of the few schemes that made it to the fourth round of the NIST standardization effort as a KEM candidate [25]. Until recently, the best-known algorithms for breaking SIDH or SIKE had an exponential time complexity in both, classical and quantum settings.

However, in July 2022, Castryck and Decru [5] proposed a surprising attack that (heuristically) solves the CSSI problem in polynomial-time. This attack relies on the knowledge of three crucial pieces of information, namely, (i) the degree of the isogeny ϕ; (ii) the endomorphism ring of the starting curve E_0; and (iii) the images $\phi(P_0), \phi(Q_0)$ of Alice's generator points $\langle P_0, Q_0\rangle = E[2^a]$, where the prime $p = 2^a 3^b - 1$ is the underlying prime used by SIKE instantiations. Recall that (ii) and (iii) are only known in the specific case of the CSSI problem, but not in the more general case of the *SIPFD* problem. Furthermore, another attack by Maino and Martindale [21] and yet another one by Robert [27] quickly followed. Maino and Martindale's attack relies on several crucial steps used in [5], but does not require knowledge of the endomorphism ring associated to the base curve. Robert's attack can also break SIDH for any random starting supersingular elliptic curve.

Despite the short time elapsed since the publication of Castryck and Decru's attack, several countermeasures have already been proposed by trying to hide the degree of the isogeny [24], the endomorphism ring of the base curve [4], or the images of the torsion points [15]. At this point, only time will tell if SIDH/SIKE will ever recover from the attacks on the CSSI problem. But even if this never happens, the theoretical and practical importance of the *SIPFD* problem still stands. For instance, the constructions from [17, Section 4], [7], and [20, Section 5.3] do not append images of auxiliary points to their public keys. In turn the Castryck-Decru family of attacks does not apply, making the security of those applications entirely based on the *SIPFD* problem.

Known attacks on the *SIPFD* problem. Even before the publication of the attack in [5], it was wildly believed that the best approaches for solving the CSSI problem are classical and not quantum [19]. Here we present a brief summary of the different assumptions made across the last decade about the cost of solving the *SIPFD* problem. We stress that while all these advances were made with SIKE as main motivation, the fact that they did not make use of the torsion point images means that they still represent the state-of-the-art for attacks against the general *SIPFD* problem. The fastest known algorithm for solving *SIP* has computational complexity $\tilde{O}(\sqrt{p})$ [16,13,22]. However, if the secret isogeny is of known degree ℓ^e, there might exist more efficient algorithms for solving the *SIPFD*. Indeed, in their NIST first round submission, the SIKE team [2] argued that the best classical attack against the CSSI problem was to treat it as an *SIPFD* problem and use a MitM approach with a time and memory cost of $O(\ell^{\frac{e}{2}})$, which is more efficient for SIKE and all instantiations of the *SIPFD* where $\ell^e \leq p$.

By assuming an unlimited memory budget and memory queries with zero time cost, the MitM attack is indeed the best attack against the *SIPFD* problem.¸Nevertheless, in [1], the authors argued that the van Oorschot-Wiener (vOW) golden collision search, which yields a better time-memory trade-off curve, is the best classical approach for large instances. The rationale used is that the $O(\ell^{\frac{e}{2}})$ memory requirement for launching the MitM attack is infeasible for the cryptographic parameter sizes. Since the best known generic attacks against AES use a negligible amount of memory, it is just natural to set an upper bound on the available classical memory when evaluating the cost of solving *SIPFD* instantiations in the context of NIST security levels 1 to 5.

To increase interest in studying the CSSI problem Costello published in [8] two Microsoft $IKE challenges, a small and a large one using a 182- and a 217-bit prime number, respectively. These two CSSI instances are known as $IKEp182 and $IKEp217 challenges.[6] A few months later, the solution of $IKEp182 was announced by Udovenko and Vitto in [30]. The authors treated this challenge as an instance of *SIPFD*, and then used a MitM approach largely following the description given in [9] along with several clever sorting and sieving tricks for optimizing data queries for their disk-based storage solution. The authors reported that their attack had a timing cost of less than 10 core-years, but at the price of using 256 TiB of high-performance network storage memory.

[6] The precise specifications can be found in https://github.com/microsoft/SIKE-challenges.

It is obvious that this memory requirements quickly render the strategy unfeasible for larger non-toy instances. As mentioned in [1], there exists a time-memory trade-off variant of the MitM algorithm (*cf.* Section 2.2), which was adopted by Udovenko and Vitto to bring the storage requirements of their attack down to about 70 TiB.

However, determining the best algorithmic choice for solving instances of given size under a certain memory budget and computational platform remains so far an open problem. In this work we present a framework predicting that both MitM variants are outperformed by the vOW golden collision approach even for moderately large *SIPFD* instances. We then substantiate our claims by extrapolating results of our implementations, accounting for practical effects such as memory access costs.

Our contributions In [1] it was found that vOW is a better approach than MitM to tackle large *SIPFD* instances. However, the small Microsoft challenge $IKEp182 was broken, before the Castryck-Decru attack was known, using a MitM strategy [30]. As discussed in [30], it remains unclear for which instance sizes and memory availability, vOW outperforms MitM. In this work we answer this question from a theoretical and practical perspective. Theoretically, we give a precise estimation of the costs of both strategies including most recent algorithmic improvements. Practically we substantiate our estimations via optimized implementations and extensive benchmarking performed in CPU and GPU platforms.

Moreover, in the case of CPU platforms, we present a detailed framework that for a fixed memory budget and prime size, predicts when a pure MitM approach, batched (limited memory) MitM or vOW approach becomes the optimal design choice for attacking *SIPFD* (see Section 3 and Figure 2). The predictions of our model are backed up by practical experiments on small *SIPFD* instances and extrapolations based on the obtained practical timings of our implementations.

We additionally provide the first optimized GPU implementation of the vOW attack on *SIPFD*, outperforming a CPU based implementation by a factor of almost two magnitudes. We provide medium sized experimental data points using our GPU implementation including extrapolations to larger instances. More concretely, our implementation solves *SIPFD* instances with isogeny degree 2^{88} with primes of bit size 180 (comparable to the instance solved in [30]) using 16 GPUs each equipped with only 80 GiB of memory in about 4 months. Based on our experimental results we conclude that vOW is the preferred choice for any larger *SIPFD* instances on reasonable hardware.

Our CPU and GPU software libraries are open-source and available at `https://github.com/TheSIPFDTeam/SIPFD`.

Outline. The remainder of this work is organized as follows. In Section 2 we present a formal definition of *SIPFD* and relevant mathematical background. We also give a detailed explanation of the MitM and vOW strategies. In Section 3 we present a careful estimation of the cost of the MitM and vOW strategies and their corresponding trade-offs in the context of the *SIPFD*. In Section 4 we present our implementation of the CPU-based MitM attack and the vOW strategy on a multi-core GPU platform.

2 Preliminaries

2.1 Elliptic curves and isogenies

Let $\mathbb{F}_p$ be the prime field with p elements and let E be a supersingular elliptic curve defined over $\mathbb{F}_{p^2}$. A common choice, convenient for implementations, is to choose p such that $p \equiv 3 \bmod 4$, and take $\mathbb{F}_{p^2} = \mathbb{F}_p[i]/(i^2+1)$ the quadratic extension of $\mathbb{F}_p$. Moreover, we will assume that E is given by a Montgomery equation:

$$E\colon y^2 = x^3 + Ax^2 + x, \quad A \in \mathbb{F}_{p^2} \setminus \{\pm 2\}.$$

The set of points satisfying this equation along with a point at infinity $\mathcal{O}_E$ form an abelian group. The point $\mathcal{O}_E$ plays the role of the neutral element. In general, we write the sum of d copies of P as $[d]P$ and if k is the smallest scalar such that $[k]P = \mathcal{O}_E$, we say that P is an order-k elliptic curve point. The d-torsion subgroup, denoted by $E[d]$, is the set of points $\{P \in E(\overline{\mathbb{F}}_p) \mid [d]P = \mathcal{O}_E\}$. If $\gcd(p,d) = 1$, then $E[d]$, as a subgroup of E, is isomorphic to $\mathbb{Z}/n\mathbb{Z} \times \mathbb{Z}/n\mathbb{Z}$. The j-invariant of the curve E is given by $j(E) := \frac{256(A^2-3)^3}{A^2-4}$. It has the useful property that two curves are isomorphic if and only if they have the same j-invariant.

An isogeny $\phi\colon E \to E'$ is a rational map (roughly speaking a pair of quotient of polynomials) such that $\phi(\mathcal{O}_E) = \mathcal{O}_{E'}$. This implies that ϕ is a group homomorphism (see for instance [32]). By a theorem of Tate [28], an isogeny defined over $\mathbb{F}_{p^2}$ exists if and only if $\#E(\mathbb{F}_{p^2}) = \#E'(\mathbb{F}_{p^2})$. If there is an isogeny ϕ between E and E', then we say that the two curves are *isogenous*. The kernel $\ker\phi$ is the set of points in the domain curve E which are mapped to the identity point $\mathcal{O}_{E'}$. If we restrict to separable maps only, any isogeny ϕ is uniquely determined by its kernel up to an isomorphism. We say ϕ is a d-isogeny or an isogeny of degree d whenever $\#\ker\phi = d$. Any isogeny can be written as a composition of prime-degree isogenies and the degree is multiplicative, in the sense that $\deg(\phi_1 \circ \phi_2) = \deg(\phi_1)\deg(\phi_2)$.

For each isogeny $\phi\colon E \to E'$ there also exists a dual d-isogeny $\hat{\phi} : E' \to E$ satisfying $\phi \circ \hat{\phi} = [d]$ and $\hat{\phi} \circ \phi = [d]$, where $[d]$ is the isogeny $P \to [d]P$ on E and E' respectively. Isogenies of degree d are computed in time $O(d)$ by using Vélu's formulas, or for sufficiently large d in $O(\sqrt{d})$, using the more recent $\sqrt{\text{é}}$lu's formulas [3]. In practice, if the degree is d^e with d small, one splits a d^e-isogeny as the composition of e d-isogenies each computed with Vélu's formula.

We are now ready to state a formal definition of the *SIPFD* problem.

Definition 1 (*SIPFD* problem). *Let p, ℓ be two prime numbers. Consider E and E' two supersingular elliptic curves defined over $\mathbb{F}_{p^2}$ such that $\#E(\mathbb{F}_{p^2}) = \#E'(\mathbb{F}_{p^2})$. Given $e \in \mathbb{N}$ find an isogeny of degree ℓ^e from E to E', if it exists.*

Concretely, in the remainder of this work we assume that the secret isogeny is of degree 2^e i.e., we fix $\ell = 2$, and define it over $\mathbb{F}_{p^2}$. Moreover, we assume $p = f \cdot 2^e - 1$ for some odd cofactor f and that $\#E(\mathbb{F}_{p^2}) = (p+1)^2$. This allows us to have the 2^e-torsion subgroup defined over $\mathbb{F}_{p^2}$. Finally, we work with instances where e is half the bitlength of the prime p. While all these conditions are more specific than the general *SIPFD* problem, they are efficiency-oriented decisions that are common practice in isogeny-based protocols and we do not exploit them beyond that.

Let P, Q be a basis of $E[2^e]$. Then the kernel of any 2^e-isogeny can be written as either $\langle P + [\mathrm{sk}]Q \rangle$, with $\mathrm{sk} \in \{0, \dots, 2^e - 1\}$ or $\langle [\mathrm{sk}][2]P + Q \rangle$, with $\mathrm{sk} \in \{0, \dots, 2^{e-1} - 1\}$. For simplicity, in our implementation we work with isogeny kernels are always of the form $\langle P + [\mathrm{sk}]Q \rangle$. There is little loss of generality, since attacking the remaining kernels would only require re-labelling the basis and re-running the algorithm.

An isogeny $\phi : E_0 \to E_1$ of degree ℓ^e can be written as a composition $\phi = \phi_1 \circ \phi_0$ of two isogenies of degree $\ell^{e/2}$ (assuming an even e for simplicity), where $\phi_0 : E_0 \to E_m$ and $\phi_1 : E_m \to E_1$ for some middle curve E_m. Since there exists a dual isogeny $\hat{\phi}_1 : E_1 \to E_m$, one can conduct a Meet in the Middle (MitM) attack by exploring all the possible $\ell^{e/2}$-isogenies emanating from E_0 and E_1, and finding the pair of isogenies that arrive to the same curve E_m (up to isomorphism). The largest attack recorded on the *SIPFD* problem, conducted by Udovenko and Vitto[7] [30], used this strategy to break an instance with $\ell = 2$ and $e = 88$.

2.2 Meet in the Middle (MitM)

Let us briefly recall the MitM procedure to solve the *SIPFD* for $\ell = 2$. We first compute and store all $2^{e/2}$-isogenous curves to E_0 in a table T (identified via their j-invariants). Then we proceed by computing each $2^{e/2}$-isogenous curve to E_1 and check if its j-invariant is present in table T. Any matching pair then allows to recover the secret isogeny as outlined in the previous section.

Complexity. Let $N := 2^{e/2}$. The worst-case time complexity of the MitM attack is $2N$ evaluations of degree-$2^{e/2}$ isogenies, while in the average case $1.5N$ such evaluations are necessary. The space complexity is dominated by the size of the table to store the N j-invariants and scalars.

In a memory restricted setting, where the table size is limited to W entries, the MitM attack is performed in batches. In each batch, we compute and store the output of W isogenies from E_0, then compute and compare against each of the N isogenies from E_1 without storing them. The number of batches is N/W where each batch performs W isogenies from E_0 and N isogenies from E_1, yielding a total of $\frac{N}{W}(N + W)$ evaluations of $2^{e/2}$-isogenies.

[7] This work was realized as an attack on SIKE, but does not exploit the torsion point images and can be regarded as an attack on *SIPFD* in general.

Depth-First Search methodology. In 2018, Adj *et al.* [1, §3.2] showed that computing the isogenies from each side in a depth-first tree fashion yields performance improvements. The improvement stems from the iterative construction of the $2^{e/2}$-isogenies as $e/2$ degree-2 isogenies. Here, whenever two isogenies share the same initial path, the depth-first approach avoids re-computation of those steps.

In order to adapt to the limited-memory scenario, let us assume that the available memory can hold $W = 2^{\omega}$ entries. Then each batch of isogenies from E_0 can be obtained by following a fixed path for the first $e/2 - \omega$ steps, and then computing the whole subtree of depth ω from this node.

Also, the attack is easy to parallelize. Assuming 2^c threads are used, all trees can be branched sequentially for c steps to obtain 2^c subtrees, each of which is assigned to a different core. This methodology for evaluating trees in batches and with multiple cores is summarized in Figure 1.

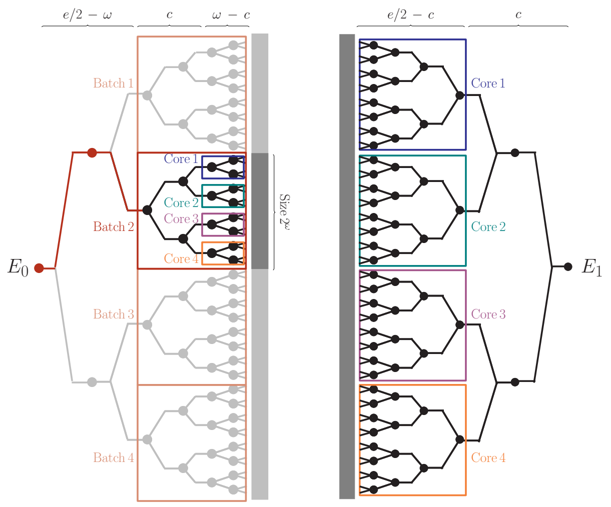

Fig. 1: The batched Meet-in-the-Middle depth-first approach: finding a 2^e-isogeny between E_0 and E_1 with 2^c cores and 2^{ω} memory. In this example, $e = 12$, $c = 2$, and $\omega = 4$.

Since a binary tree of depth ω has $2^{\omega+1} - 2$ edges, each batch is computing $e/2 - \omega + 2^{\omega+1} - 2$ isogenies of degree 2 for the side corresponding to E_0, and the whole tree with $2^{e/2+1} - 2$ isogenies for the side corresponding to E_1. The expected cost corresponding to half of the batches is then

$$\frac{1}{2}2^{e/2-\omega}\left(2^{e/2+1}+2^{\omega+1}+e/2-\omega-4\right)\approx 2^{e-\omega}$$

computations of 2-isogenies.

2.3 Parallel Collision Search

Given a random function $f : S \to S$, van Oorschot and Wiener's method [26] is a parallel procedure to find collisions in f. The main idea of the algorithm is to construct in parallel several chains of evaluations $x_i = f(x_{i-1})$, starting from random seeds x_0. Further, a small fraction of the points in S is called *distinguished* based on an arbitrary criterion (e.g. that the binary representation of $x \in S$ ends with a certain number of zeros). A chain continues until it reaches a distinguished point. Then this point is compared against a hash table including all previously found distinguished points. Further, to avoid infinite loops, chains are aborted after their length exceeds a specified threshold.

Two chains ending in the same distinguished point indicate a collision between those chains. This collision can be efficiently reconstructed if the seeds x_0, x_0' and the lengths d, d' of the colliding chains are known. Therefore, assuming $d > d'$ we take $d-d'$ steps on the longer chain (starting from x_0). From there on we take simultaneous steps on both chains, checking after each step if the collision has occurred. Hence, the hash table stores for each found distinguished point the triplet (x_0, x_d, d) indexed by x_d.

Complexity. Let N be the size of the set S, θ the proportion of points that are distinguished, and W the amount of distinguished triplets that we can store. Since each chain has an average length of $1/\theta$, the chains represented by the stored triplets (once the hash table is completely filled) include an average of W/θ points. Therefore the probability that a given evaluation of f collides with any of these points is $W/N\theta$. After a collision takes place, the chain needs to continue for an additional $1/\theta$ steps on average before it reaches a distinguished point and the collision is detected. At this point, the two involved chains must be reconstructed from the start to find the exact step at which the collision occurred, yielding a total of $N\theta/W + 3/\theta$ evaluations of f to find a collision. The optimal choice for θ is $\sqrt{3W/N}$ yielding a cost of $2\sqrt{3N/W}$ per collision. Note, however that this analysis assumes a table that already contains W triplets. To capture the transition effect of the table filling up, van Oorschot and Wiener [26] model $\theta = \alpha\sqrt{W/N}$ for a parameter α that is experimentally measured to be optimal at $\alpha = 2.25$. The resulting cost per collision is found to be linear in $\sqrt{N/W}$ as long as $2^{10} < W < N/2^{10}$.

Note that any random function from S to itself is expected to have $N/2$ collisions, however, many applications, including the *SIPFD*, require looking for one specific collision that we refer to as the "golden collision" [31,12,14,23]. This means that the attack has to find $N/4$ different collisions on average before stumbling upon the golden collision, bringing the total cost to $\mathcal{O}(\sqrt{N^3/W})$ function evaluations.

Application to the *SIPFD* problem To attack the *SIPFD* problem and find the kernel of a degree-2^e isogeny between E_0 and E_1, we assume for simplicity that e is even and define $S = \{0,1\} \times \{0,\ldots,2^{e/2}-1\}$ so that $N = 2^{e/2+1}$. We also define the map $g : S \to \mathbb{F}_{p^2},\ (c,k) \mapsto j(E_c/\langle P_c + [k]Q_c\rangle)$, where (P_c, Q_c) are a predefined basis of the $2^{e/2}$-torsion on either side as before. As explained in Section 2.1, the function g yields a bijection between S and the set of $2^{e/2}$-isogenies with kernel $\langle P_c + [k]Q_c\rangle$ from the curves on either side. A collision $g(c,k) = g(c',k')$ with $c \neq c'$ implies two isogenous curves starting on opposite sides and meeting at a middle curve (up to isomorphism).

To apply the parallel collision search, we need a function f that maps S back to itself. Hence, we have to work with the composition $f = h \circ g$ where h is an arbitrary function mapping j-invariants back to S. This composition introduces several fake collisions that are produced by the underlying hash function while there is still only one (golden) collision that leads to the secret isogeny.

Note that for a certain (unlucky) choice of hash function h the golden collision might not be detectable.[8] Therefore, we have to periodically switch the hash function h. More precisely, we switch the function whenever we found a certain amount C of distinguished points. If we model $C = \beta \cdot W$ for some constant β, then each hash function will have a probability of $2\beta W/N$ for finding the golden collision. Experimentally, van Oorschot and Wiener [26] found $\beta = 10$ to perform best, and the average running time of the attack is measured to be $(2.5\sqrt{N^3/W})/m$, where m is the number of processors computing paths in parallel.

3 Accurate formulas for vOW and MitM

So far, we have provided theoretical cost functions for the golden collision search in terms of the number of evaluations of the function f, and for the batched depth-first MitM in terms of the number of 2-isogeny evaluations. We now provide a more detailed cost model in terms of elliptic curve operations to make these costs directly comparable. These formulas give a first indication of which memory regime favors which algorithm and, further, they form the starting point for parameter selection in our implementation.

3.1 Meet in the Middle

For the depth-first MitM, we have counted only the 2-isogeny evaluations but the total cost involves also obtaining the kernel points of each isogeny and pushing the basis points through the isogeny. As described in [1], the total cost of processing a node at depth d can be summarized as:

- $2^{e/2-d}$ point doublings to compute the kernel points
- 2 isogeny constructions to compute the children nodes

[8] For instance, one of the points that leads to the golden collision might be part of a cycle that does not reach a distinguished point.

– 1 point doubling, 1 point addition, and 6 isogeny evaluations to push the basis through the isogenies.

Nodes at the second-to-last level represent an exception since once we obtain the leaves, we no longer require pushing the bases and instead we need to compute the j-invariant.

Let us refer by `ADD, DBL, ISOG, EVAL, JINV` to the cost of a point addition, point doubling, 2-isogeny construction, 2-isogeny evaluation at a point, and j-invariant computation, respectively. The total cost of computing a tree of depth $e/2$ is then

$$\begin{aligned}\mathrm{DFS}(e/2) &= \sum_{d=0}^{e/2-2} 2^d \left((2^{e/2-d}+1)\mathtt{DBL} + 2\mathtt{ISOG} + 1\mathtt{ADD} + 6\mathtt{EVAL}\right)\\ &+ 2^{e/2-1}\left(2\mathtt{DBL} + 2\mathtt{ISOG} + 2\mathtt{JINV}\right)\\ &= 2^{e/2-1}\left((e+1)\mathtt{DBL} + 4\mathtt{ISOG} + 1\mathtt{ADD} + 6\mathtt{EVAL} + 2\mathtt{JINV}\right) + O(1).\end{aligned}$$

The expected time of the whole MitM attack using 2^ω memory entries, which computes a tree of depth ω on one side and a tree of depth $e/2$ on the other side for each batch, is then

$$\begin{aligned}\mathrm{MitM}(e,\omega) &= \frac{2^{e/2}}{2\cdot 2^\omega}(DFS(\omega) + DFS(e/2))\\ &\approx (2^{e-\omega-2} + 2^{e/2-2})\left(\mathtt{DBL} + 4\mathtt{ISOG} + 1\mathtt{ADD} + 6\mathtt{EVAL} + 2\mathtt{JINV}\right)\\ &\quad + (2^{e-\omega-2}e/2 + 2^{e/2-2}\omega)\mathtt{DBL}.\end{aligned}$$

3.2 Golden Collision Search

For the golden collision search, the cost of an evaluation of the random function, given a scalar $k \in \mathbb{Z}_{2^{e/2}}$ and a bit $c \in \{0,1\}$, consists of

– computing the kernel point $P_i + [k]Q_i$,
– constructing a single $2^{e/2}$-isogeny with said kernel and
– computing the j-invariant of the output curve.

The first step is usually done with a three-point Montgomery ladder which has an average cost of $\frac{e}{2}(\mathtt{DBL}+\mathtt{ADD})$. For the second step, it is shown in [10] that a "balanced" strategy for computing a $2^{e/2}$-isogeny costs about $\frac{e}{4}\log(e/2)\mathtt{DBL} + \frac{e}{4}\log(e/2)\mathtt{EVAL} + \frac{e}{2}\mathtt{ISOG}$. Hence, the total expected sequential time of the golden collision search is

$$\begin{aligned}\mathrm{GCS}(e,\omega) &= 2.5\cdot 2^{3(e/2+1)/2-\omega/2}\\ &\quad\times\left(\frac{e}{4}\log(e/2)(\mathtt{DBL}+\mathtt{EVAL}) + \frac{e}{2}\mathtt{ISOG} + \mathtt{JINV} + \frac{e}{2}(\mathtt{DBL}+\mathtt{ADD})\right).\end{aligned}$$

3.3 Simplified Cost Models for Montgomery Curves

Assuming that we use Montgomery curve arithmetic, then the cost of curve operations can be expressed in terms of field additions, multiplications, squares and inverses ($\mathtt{A}$, $\mathtt{M}$, $\mathtt{S}$, $\mathtt{I}$, respectively) as follows (compare to [2])

$$\mathtt{DBL} = 4\mathtt{A} + 4\mathtt{M} + 2\mathtt{S}, \quad \mathtt{ADD} = 6\mathtt{A} + 4\mathtt{M} + 2\mathtt{S}, \quad \mathtt{ISOG} = \mathtt{A} + 2\mathtt{S} \quad \text{and}$$
$$\mathtt{EVAL} = 6\mathtt{A} + 4\mathtt{M}, \qquad \mathtt{JINV} = 8\mathtt{A} + 3\mathtt{M} + 4\mathtt{S} + \mathtt{I}.$$

Moreover, we assume $\mathtt{M} = 1.5\mathtt{S} = 100\mathtt{A} = 0.02\mathtt{I}$ which we have obtained experimentally from our quadratic field arithmetic implementation. The cost models can then be written in units of $\mathtt{M}$ as

$$\mathrm{MitM}(e,\omega)/\mathtt{M} \approx \frac{22799}{600}\left(2^{e-\omega} + 2^{e/2}\right) + \frac{403}{300}\left(2^{e-\omega}\cdot e/2 + 2^{e/2}\cdot\omega\right) \tag{1}$$

and

$$\mathrm{GCS}(e,\omega)/\mathtt{M} \approx 2.5\cdot 2^{3(e/2+1)/2-\omega/2}\left(\frac{4181}{75} + \frac{1211}{200}e + \frac{283}{120}e\log(e/2)\right) \tag{2}$$

For a given value of e and a memory budget ω, we can now determine which algorithm is favorable. Figure 2 visualizes three different regions. For $\omega \geq e/2$ the full MitM attack without batching can be applied. The batched MitM attack is found to have a narrow area of application at the border of the region where the golden collision search is optimal, which dominates the largest part of the limited-memory area.

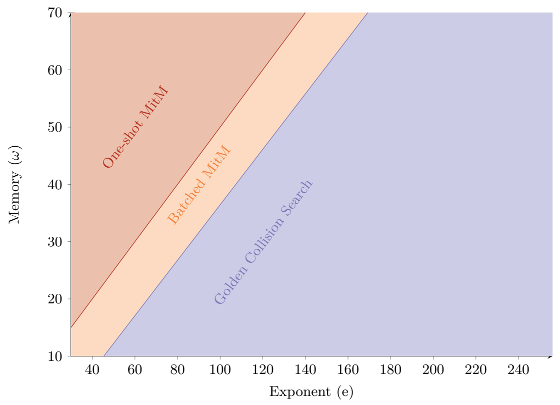

Fig. 2: Regions in the (e,ω) space where each attack is optimal for solving a SIPFD problem of size 2^e with memory limited to 2^w entries.

We would like to stress, that this comparison is based on idealized models involving only underlying field arithmetic operations. They do not take into account any practical effects, as e.g. memory access timings or parallelization issues. Nevertheless, it gives a first indication of the superiority of the golden collision search in the limited memory setting.

4 Practical Results on solving the *SIPFD*

In this section we present our experimental results with a focus on our GPU implementation of the van Oorschot and Wiener golden collision search. But first, let us start with an experimental validation of our theoretical estimates of the MitM algorithm and its batched version from Section 3.1.

4.1 Practical Results of our MitM CPU Implementation

We have implemented the batched depth-first MitM attack and run experiments on an AMD EPYC 7763 64-Core processor at 2.45 GHz, running 32 threads in parallel.

The j-invariants in each batch are stored in RAM, along with the corresponding scalar sk. Each processor maintains an array with j-invariants that have been calculated and sort lexicographically, to reduce the number of memory accesses when searching for the collision.

For measuring the performance of the batched depth-first MitM with the memory parameter ω, we fix a small instance with exponent $e = 50$ and benchmark the attack for ω with $18 \leq \omega \leq 25$. These timings are compared to Equation 1, using a separate benchmark for the cost of `M`, i.e. a multiplication operation of our implementation. As shown in Figure 3, the experimental measurements are found to adhere to the model up to an overhead factor of about 2, which is explained by the memory access times and sorting overheads that are not accounted for in Equation 1.

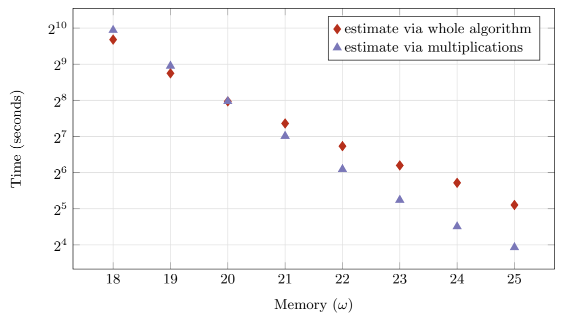

Fig. 3: Completion time of the MitM attack for an exponent $e = 50$ using 32 physical processors and different memory bounds compared to the prediction in Equation 1.

We then tested the attack for increasing values of e while limiting the memory to $\omega \leq 28$. For $e > 56$, the batched MitM must be used and we have estimated the complexity of the whole attack by completing a single batch. As expected, Figure 4 shows that the slope of the cost changes drastically once we enter the limited-memory region. The overhead factor between the experimental results and the theoretical model is always found to be less than 2.6. We conclude that Equation 1 can be used to estimate the cost of the attack for larger parameters without significant overhead.

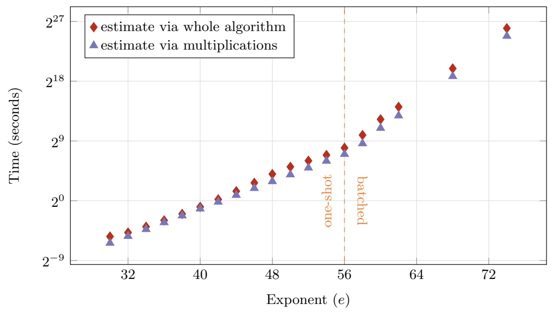

Fig. 4: Completion time of the MitM attack for various exponent sizes.

For comparison, the instance solved by Udovenko and Vitto in [30] was in the unlimited-memory setting using $e = 88$ and $\omega = 44$. Based on our model and adjusting to their clock frequency, we obtain an estimate of 9.47 core-years for the attack. This is close to Udovenko and Vitto's experimental result of 8.5 core-years, despite the fact that they used network storage.

4.2 Practical considerations for our vOW GPU implementation

Let us give a brief explanation of the GPU architecture we used, followed by a summary of practical features of our implementation.

GPU architecture. An NVIDIA CUDA device allows to execute thousands of threads in parallel. Following the Single Instructions Multiple Thread (SIMT) paradigm, a collection of 32 threads is bundled in *warps* that can only perform the same instruction on different data. One of the main challenges when programming CUDA devices is to decrease the memory latency, i.e., the time the threads are waiting for the data to be loaded into the corresponding registers. Therefore all CUDA devices have a multiple-level memory hierarchy incorporating memory and caches of different size and speed.

The NVIDIA A100 has an 80GB sized main memory, connected to other GPUs in the same cluster via a high throughput bus called *NVLINK*. However, for performing computations, data must be propagated through the two levels of caches down to the registers. Each thread has only a very limited amount of these registers. Whenever more registers are addressed than physically available, the memory must be outsourced to other memory levels, causing latency and stalls. Further, whenever more threads are requested than the hardware can handle concurrently, a scheduling is performed, by swapping active threads against queued ones. As a consequence, caches must be invalided, which leads to further memory latency. However, there is usually an optimal number of concurrent threads such that memory latency can be minimized by an optimal scheduling.

GPU potential of vOW. Note that the major task performed inside the vOW algorithm is the computation of chains of evaluations of the given function on different inputs. Therefore, it fits into the SIMT paradigm and can effectively be parallelized on the GPU. Further, since the devices are inherently memory-constrained, they profit from the good asymptotic trade-off curve of the vOW collision search.

Practical features

Hash function. For performance improvements, we heuristically model hash functions with ℓ-bit output as the projection to the first ℓ bits of the input. To obtain a randomized version we xor a fixed random nonce to the output. That is, for a given nonce $\mathbf{r} \in \mathbb{F}_2^\ell$ the hash function $h_\mathbf{r} \colon \mathbb{F}_2^* \mapsto \mathbb{F}_2^\ell$ is defined as $h_\mathbf{r}(\mathbf{x}) := (x_1, \ldots, x_\ell) + \mathbf{r}$. This is justified by the fact that the inputs usually inherit already enough randomness, which is confirmed in our experiments.

Memory optimizations. The bit-size of every triplet (x_0, x_d, d) is roughly $e + \log(20/\theta)$, since x_0, x_d encode $2^{e/2}$-isogenies and the length of each chain is $d < 20/\theta$. However, due to our hash function choice, we can omit $\log W$ bits of x_d referring to its address in the table, plus another $\log(1/\theta)$ bits from the fact that it is a distinguished point, giving a size of roughly $e + \log(20) - \log W$ bits per triplet.

PTX assembly. We provide core functionalities of our GPU implementation in PTX (Parallel Thread eXecution) assembly, which is the low level instruction set of NVIDIA CUDA GPUs. This includes our own optimized $\mathbb{F}_p$ arithmetic. In this context, we provide optimized version of both the schoolbook and the Karatsuba algorithm for integer multiplication, as well as the Montgomery reduction.

Data structure. For storing distinguished points we compare the performance of a standard hash table against the Packed Radix-Tree-List (PRTL) proposed in [29]. The PRTL is a hash table that stores a linked list at each address, instead of single elements. This avoids the need for element replacement in case of hash collisions. Further it identifies the address of an element via its prefix (*radix*) and stores only the prefix-truncated element. The *packed* property of the PRTL relates to distinguished point triplets being stored as a single bit-vector, thus, avoiding the waste of space due to alignment. We ran CPU experiments with both data structures to identify the optimal choice prior to translating the code to the GPU setting. Eventually, we adopted the packed property and the use of prefixes, while we found no improvement in performance from using linked lists.

Precomputation. As discussed in [9], the time T_f required for a function evaluation can be decreased via precomputation. For a depth parameter d, one can precompute the 2^d curves corresponding to all the 2^d-isogenies from E_0 and E_1. When computing a $2^{e/2}$-isogeny, the initial d steps are replaced by a table lookup and we end up computing only a $2^{e/2-d}$-isogeny. Note that the memory needed for precomputation grows exponentially with d and so asymptotically it does not play a relevant role. However, for relatively small parameters it can provide valuable savings and speed up our experiments without affecting metrics such as the number of calls to f.

4.3 Practical results of our vOW GPU implementation

In the following we use the practical performance of our implementation together with the known theoretical behavior to extrapolate the time to solve larger instances. In the original work of van Oorschodt-Wiener the time complexity of the procedure was found to be well approximated by

$$\frac{1}{m}(2.5\sqrt{N^3/W}) \cdot T_f, \tag{3}$$

where T_f is the cost per function evaluation. Therefore, we measure the cost T_f of our implementation which then allows us to derive an estimate for arbitrary instances. Further, we compare this estimate against the theoretical estimate via Equation 2 and an estimate based on collecting a certain amount of distinguished points.

Additionally, we verify that our GPU implementation using the functions specified in Section 2.3 has a similar behavior as the CPU implementation using random functions of [26]. This increases the reliability in our estimates, as it shows that the time complexity of our implementation is still well approximated by Equation 3. Let us start with this verification.

Verifying the theoretical behavior. In [26] van Oorschot and Wiener find that on average it takes $\frac{0.45N}{W}$ randomized versions of the function to find the solution, which in our case corresponds to random choices of the hash function (compare to Section 2.3). In their experiments, the function is changed after $\beta \cdot W$ distinguished points have been discovered, where a value of $\beta = 10$ is found to be optimal. Further, chains are aborted after they reach a length of $20\theta^{-1}$, i.e., 20 times their expected length.

Optimal value of β. Let us first verify that an amount of $10 \cdot W$ distinguished points until we abort the collision search for the current version of the function is still a suitable choice for our implementation. Table 1 shows the average running time of our vOW implementation using different values of β. We conclude that the values around $\beta = 10$ give comparable performance, with $\beta = 10$ being optimal in most of the experiments. The results are averaged over 100 ($e = 34$) and 50 ($e = 36$) runs respectively.

e	ω	$\beta = 5$	$\beta = 10$	$\beta = 15$	$\beta = 20$
34	8	405.08	384.74	371.67	**335.88**
	9	244.30	**198.86**	238.60	285.97
	10	173.73	207.37	**136.80**	179.93
36	9	704.65	**567.89**	654.15	599.61
	10	419.87	**373.16**	489.71	542.00
	11	398.72	365.62	**314.26**	290.49

Table 1: Running time in seconds for different values of β.

Expected number of randomized versions of the function. Now that we confirmed the optimal choice of β, we expect that the required amount of random functions until success also matches the one from [26]. In this case, the number of required randomizations of the function until the golden collision is found should follow a geometric distribution with parameter close to $\frac{W}{0.45N}$.

We confirm this distribution in an experiment for $e = 30$, in which case we have $N = 2^{e/2+1} = 2^{16}$ and use a hash table that can store up to $W = 2^7$ distinguished points. We then solved 1000 such instances and recorded for each the number of randomized versions of the function until the solution was found. On average, it took 208.28 versions compared to the approximation of $\frac{0.45N}{W} = 230.4$, despite slightly surpassing the $W \leq N/2^{10}$ limit where the vOW experiments took place. In Figure 5 we visualize the obtained frequencies (triangles) and give as comparison the probabilities of the geometric distribution with parameter $\frac{1}{208.27}$ (diamonds). In this figure we accumulated the frequencies in each interval of size 20 to allow for a better visualization.

Measuring the time per function evaluation. Next we measured the time per function evaluation that the GPU implementation requires on our hardware for different values of e. To pick our parameters, we first set W to the largest power of 2 such that the memory would not surpass our GPU's 80 GB budget, then chose the largest precomputation depth that would fit in the remaining memory. In the smaller instances, the memory and precomputation depth were additionally subject to a cap of $W \leq N/2^8$ and $d \leq e/4$ in the smaller instances. After performing the precomputation, we measured the time per function evaluation as illustrated in Figure 6. The jumps in the graph indicate that the bitsize of the used prime, which is roughly $2e$, exceeds the next 64-bit boundary. In

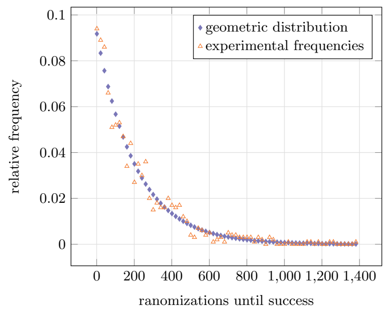

Fig. 5: Number of used randomizations to find the solution for $e = 30$, $W = 2^7$

those cases the prime occupies an additional register, which leads to a slowdown of the $\mathbb{F}_{p^2}$-arithmetic.

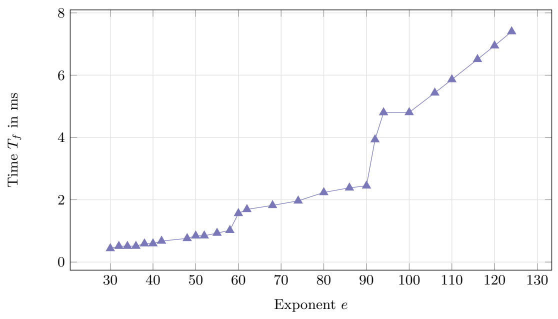

Fig. 6: Cost per function evaluation using 6912 threads in parallel. Each data point is averaged over 4096 evaluations.

Performance estimation using a single GPU Now, the measured timings allow us to estimate the time required by our implementation to solve larger instances. To compute this estimate we use Equation 3 with the measured value for T_f and the number of concurrent threads m used on the GPU. The resulting estimate is shown in Figure 7 (diamonds).

Note that the steeper incline in the estimation for $e > 62$ stems from the fact that for $e = 62$ we reach the maximum number of concurrent threads for our implementation, which we find to be $27,648$ threads. Further, from $e = 80$ onwards we additionally hit our hash table memory limit of $W = 2^{33}$ elements. We summarize in Table 2 optimal configurations for the *SIPFD* instances executed on our single GPU platform.

We also obtain an alternative estimate based on the time to finish one version of the random function in the full implementation of the attack. That is,

e	30	32	34	36	38	40	42	48	50	52	56	62	68	74	80
d	9	9	10	11	11	12	12	14	14	15	16	17	19	21	22
$\log W$	8	9	10	11	12	13	14	17	18	19	21	24	27	30	33

Table 2: Optimal configurations for vOW on single GPU with 80GB memory. Configurations for $e > 80$ match the one of $e = 80$.

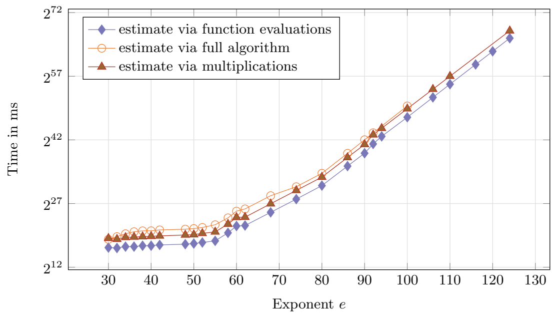

Fig. 7: Estimated time to solve instances of *SIPFD* on a single GPU

we measure the time to obtain $10 \cdot W$ distinguished points and then multiply by the average number $\frac{0.45N}{W}$ of random functions needed. This method should capture the performance more accurately as it includes practical effects such as the memory access costs. For $e \leq 62$ we averaged 100 experiments of completing a random function, while for larger instances we decreased the number of experiments and for $e \geq 76$ we only computed a $1/2^{10}$ fraction of the needed points and scaled the resulting time accordingly. The results of this second estimation are also shown in Figure 7 (circles) and present an overhead factor of about 8. This overhead is likely the result of imperfect parallelization speedups in GPUs, as well as the cost of memory accesses, but it is observed to decrease towards larger instances.

Finally, we benchmarked the average cost of field multiplications in our GPU setup to obtain a third estimate based on Equation 2, which is also presented in Figure 7 (triangles). This estimate closely matches the estimate via the full algorithm, especially for larger instances where distinguished points are rare and memory accesses are more sporadic.

Overall, our measurements support the use of any of the three methods described to obtain accurate extrapolations of the algorithm's running time. For a concrete example, we estimate that a problem with $e = 88$ which corresponds to the instance solved by Udovenko and Vitto in [30], would take about 44 years on a single GPU with 80GB memory limit. While this is not yet very impressive, compared to the 10 CPU years reported in [30], a single GPU is far less expensive and powerful than the 128TB network storage cluster used for that record. Therefore in the following section we give an estimate of the attack when scaling to a multiple GPU architecture.

Multiple GPU estimation. We explored different strategies for parallelizing the vOW algorithm across multiple GPUs. In the first strategy, every GPU independently runs its own instantiation of the algorithm. The advantage of this approach lies its simplicity, which minimizes overhead since no communication between GPUs is necessary. On the downside, it provides only a linear speedup in the number of GPUs, since additional memory resources are not shared. In our second approach, GPUs report distinguished points to the same hash table, which is stored distributed over the global memory of all GPUs. The advantage here clearly lies in the increase of the overall memory, which allows to make use of the good time-memory trade-off behavior inherent to the vOW algorithm. However, this approach introduces a communication overhead due to the distributed memory access. On top of that, the data needs to be send over the slower *NVLINK* instead of the internal memory bus of the GPU.

We performed an extrapolation of the time to solve different sized instances in the distributed setting, similar to the extrapolation via the full algorithm in the single GPU setting. In this experiment, we allocated a hash table able to store up to $W = 2^{34}$ distinguished triplets, which for large instances corresponds to about 200GB, across the memory of four GPUs connected via an *NVLINK* bus. We then measured the time to collect and store a certain amount X of distinguished points. Multiplying this time by $\frac{10 \cdot W}{X} \cdot 0.45 \cdot \frac{2^{e/2+1}}{W} = 4.5 \cdot 2^{e/2+1}/X$, gives an extrapolation of the running time of completing the whole attack.

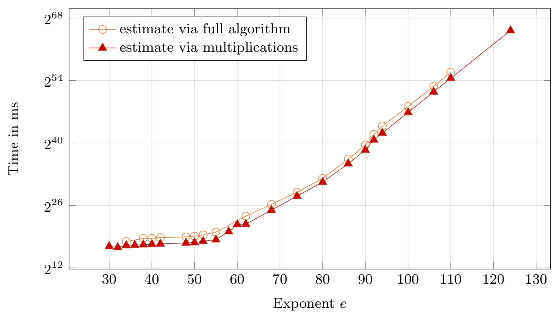

Fig. 8: Estimated time to solve instances of *SIPFD* on 4 GPUs connected via an *NVLINK* bus

Figure 8 visualizes the obtained extrapolations (circles) in comparison to the estimate via the multiplication benchmark (triangles), i.e., using Equation 2. We observe, similar to the single GPU case, a slight underestimation by using Equation 2, which for larger instances vanishes. For the larger instances we obtain an underestimation by a factor of roughly two, which corresponds to the performance difference of the *NVLINK* bus in comparison to the internal memory bus. However, since for larger instances with fixed memory budget the time to compute distinguished points dominates, the factor is expected to vanish.

Hence, we finally conclude that using the distributed memory architecture does not lead to unexpected performance slowdowns.

Comparing both strategies. Let us determine, which of the parallelization strategies is preferable for a specific amount of GPUs. For large instances, the computational cost of the multi-GPU as well as the single-GPU setting, are well approximated by Equation 2. Therefore the speedup when parallelizing via distributed memory using X GPUs is

$$\frac{\mathrm{GCS}(e,\omega)}{\mathrm{GCS}(e,\omega+\log X)/X} = X^{3/2},$$

and, hence, preferable over the strategy via independent executions with a speedup of only X. Also, if comparing the exact numbers obtained from the estimate via the full algorithm in the distributed memory setting and the single GPU setting, we find that the distributed setting offers a better practical performance already for $e \geq 62$.

Extrapolating $e = 88$ *and the way forward.* Based on our practical timings we estimate the time to solve an instance with $e = 88$ on 4 GPUs to about 32 GPU years in comparison to roughly 44 GPU years in the single GPU setting. Moreover, if we scale the attack to 16 GPUs, which is the maximum that the *NVLINK* bus currently supports, we estimate the time to only 5.6 GPU years, which means the experiment would finish in about 4 months. We therefore conclude from our experiments that for larger instances, with a memory budget of 128TB in the MitM case and 80GB per device in the GPU case, the vOW algorithm is the preferred choice.

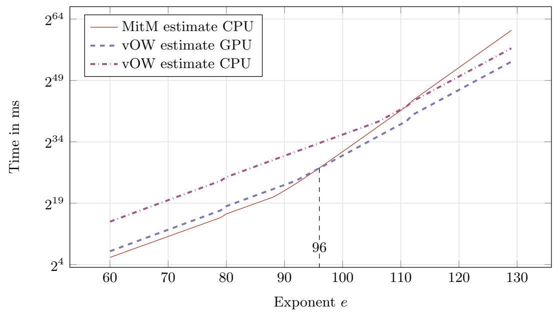

Fig. 9: Estimated time to solve instances of *SIPFD* on 16 NVIDIA Ampere GPUs with 80GB each connected via an *NVLINK* bus in comparison to a cluster with 128TB storage and 256 cores.

In Figure 9 we visualize the result of the estimation via Equation 1 and 2 in both settings assuming 256 cores with 128TB of memory in the CPU case and 16 NVIDIA A100s connected via an *NVLINK* bus in the GPU case. This figure illustrates the estimate for running the MitM on the CPU (solid line) and the vOW on the GPU system (dashed line). We find that under these fixed resources, the break-even point from where vOW offers a better performance lies at $e = 96$. Additionally, we provide the estimate if we instead execute vOW on the corresponding CPU system (dash dotted line). Observe, that even under the unrealistic assumption that the 128TB of memory would allow for efficient random access (for the vOW hash table), it does not outperform the GPU based approach for any instance size. Moreover, even under this memory advantage in case of $e = 96$, the GPU implementation offers a speedup of almost two magnitudes (82x). We conclude that the way forward when tackling larger instances of the *SIPFD* clearly favors vOW implementations on GPU platforms.

References

1. Adj, G., Cervantes-Vázquez, D., Chi-Domínguez, J., Menezes, A., Rodríguez-Henríquez, F.: On the cost of computing isogenies between supersingular elliptic curves. In: Cid, C., Jr., M.J.J. (eds.) Selected Areas in Cryptography - SAC 2018 - 25th International Conference, Calgary, AB, Canada, August 15-17, 2018, Revised Selected Papers. Lecture Notes in Computer Science, vol. 11349, pp. 322–343. Springer (2018)
2. Azarderakhsh, R., Campagna, M., Costello, C., De Feo, L., Hess, B., Jalali, A., Jao, D., Koziel, B., LaMacchia, B., Longa, P., Naehrig, M., Pereira, G., Renes, J., Soukharev, V., Urbanik, D.: Supersingular Isogeny Key Encapsulation. Third Round Candidate of the NIST's post-quantum cryptography standardization process (2020), available at: https://sike.org/
3. Bernstein, D.J., Feo, L.D., Leroux, A., Smith, B.: Faster computation of isogenies of large prime degree . ANTS XIV, The Open Book Series **4**, 39–55 (2020)
4. Burdges, J., De Feo, L.: Delay encryption. In: Canteaut, A., Standaert, F. (eds.) Advances in Cryptology - EUROCRYPT 2021, Part I. Lecture Notes in Computer Science, vol. 12696, pp. 302–326. Springer (2021)
5. Castryck, W., Decru, T.: An efficient key recovery attack on sidh (preliminary version). Cryptology ePrint Archive, Paper 2022/975 (2022), https://eprint.iacr.org/2022/975, https://eprint.iacr.org/2022/975
6. Charles, D.X., Lauter, K.E., Goren, E.Z.: Cryptographic hash functions from expander graphs. J. Cryptol. **22**(1), 93–113 (2009)
7. Chávez-Saab, J., Rodríguez-Henríquez, F., Tibouchi, M.: Verifiable isogeny walks: Towards an isogeny-based postquantum VDF. In: AlTawy, R., Hülsing, A. (eds.) Selected Areas in Cryptography. Lecture Notes in Computer Science, vol. 13203, pp. 441–460. Springer (2021)
8. Costello, C.: The case for SIKE: A decade of the supersingular isogeny problem. IACR Cryptol. ePrint Arch. p. 543 (2021), https://eprint.iacr.org/2021/543
9. Costello, C., Longa, P., Naehrig, M., Renes, J., Virdia, F.: Improved classical cryptanalysis of SIKE in practice. In: Kiayias, A., Kohlweiss, M., Wallden, P., Zikas, V. (eds.) Public-Key Cryptography - PKC 2020 - 23rd IACR International Conference on Practice and Theory of Public-Key Cryptography, Edinburgh, UK, May 4-7, 2020, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12111, pp. 505–534. Springer (2020)

10. De Feo, L., Jao, D., Plût, J.: Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. J. Math. Cryptol. **8**(3), 209–247 (2014)
11. De Feo, L., Masson, S., Petit, C., Sanso, A.: Verifiable delay functions from supersingular isogenies and pairings. In: Galbraith, S.D., Moriai, S. (eds.) Advances in Cryptology - ASIACRYPT 2019, Part I. Lecture Notes in Computer Science, vol. 11921, pp. 248–277. Springer (2019)
12. Delaplace, C., Esser, A., May, A.: Improved low-memory subset sum and lpn algorithms via multiple collisions. In: IMA International Conference on Cryptography and Coding. pp. 178–199. Springer (2019)
13. Delfs, C., Galbraith, S.D.: Computing isogenies between supersingular elliptic curves over $\mathbb{F}_p$. Des. Codes Cryptogr. **78**(2), 425–440 (2016)
14. Esser, A., May, A.: Low weight discrete logarithm and subset sum in $2^{0.65n}$ with polynomial memory. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 94–122. Springer (2020)
15. Fouotsa, T.B.: SIDH with masked torsion point images. Cryptology ePrint Archive, Paper 2022/1054 (2022), https://eprint.iacr.org/2022/1054, https://eprint.iacr.org/2022/1054
16. Galbraith, S.D.: Constructing isogenies between elliptic curves over finite fields. LMS J. Comput. Math **2**, 118–138 (1999)
17. Galbraith, S.D., Petit, C., Silva, J.: Identification protocols and signature schemes based on supersingular isogeny problems. In: Takagi, T., Peyrin, T. (eds.) Advances in Cryptology - ASIACRYPT 2017, Part I. Lecture Notes in Computer Science, vol. 10624, pp. 3–33. Springer (2017)
18. Jao, D., De Feo, L.: Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. In: Yang, B. (ed.) Post-Quantum Cryptography - 4th International Workshop, PQCrypto 2011. Lecture Notes in Computer Science, vol. 7071, pp. 19–34. Springer (2011)
19. Jaques, S., Schanck, J.M.: Quantum cryptanalysis in the RAM model: Claw-finding attacks on SIKE. In: Boldyreva, A., Micciancio, D. (eds.) Advances in Cryptology - CRYPTO 2019 - 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2019, Proceedings, Part I. Lecture Notes in Computer Science, vol. 11692, pp. 32–61. Springer (2019)
20. Luca De Feo and Samuel Dobson and Steven D. Galbraith and Lukas Zobernig: SIDH proof of knowledge. IACR Cryptol. ePrint Arch. p. 1023 (2021), https://eprint.iacr.org/2021/1023, to appear in ASIACRYPT 2022
21. Maino, L., Martindale, C.: An attack on SIDH with arbitrary starting curve. Cryptology ePrint Archive, Paper 2022/1026 (2022), https://eprint.iacr.org/2022/1026, https://eprint.iacr.org/2022/1026
22. Maria Corte-Real Santos, C.C., Shi, J.: Accelerating the Delfs-Galbraith algorithm with fast subfield root detection. In: Dodis, Y., Shrimpton, T. (eds.) Advances in Cryptology - CRYPTO 2022 - 42nd Annual International Cryptology Conference, CRYPTO 2022, Santa Barbara, CA, USA, August 15-18, 2022, Proceedings, Part III. Lecture Notes in Computer Science, vol. 13509, pp. 285–314. Springer (2022)
23. May, A.: How to meet ternary lwe keys. In: Annual International Cryptology Conference. pp. 701–731. Springer (2021)
24. Moriya, T.: Masked-degree sidh. Cryptology ePrint Archive, Paper 2022/1019 (2022), https://eprint.iacr.org/2022/1019, https://eprint.iacr.org/2022/1019
25. NIST: NIST Post-Quantum Cryptography Standardization Process. Second Round Candidates (2017), available at: https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions

26. van Oorschot, P.C., Wiener, M.J.: Parallel collision search with cryptanalytic applications. J. Cryptol. **12**(1), 1–28 (1999)
27. Robert, D.: Breaking SIDH in polynomial time. Cryptology ePrint Archive, Paper 2022/1038 (2022), https://eprint.iacr.org/2022/1038, https://eprint.iacr.org/2022/1038
28. TATE, J.: Endomorphisms of abelian varieties over finite fields. Inventiones Mathematicae **2**, 134–144 (1966), http://eudml.org/doc/141848
29. Trimoska, M., Ionica, S., Dequen, G.: Time-memory analysis of parallel collision search algorithms. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2021**(2), 254–274 (2021)
30. Udovenko, A., Vitto, G.: Breaking the SIKEp182 challenge. Cryptology ePrint Archive, Paper 2021/1421. Accepted to the SAC 2022 conference. (2021), https://eprint.iacr.org/2021/1421, https://eprint.iacr.org/2021/1421
31. van Vredendaal, C.: Reduced memory meet-in-the-middle attack against the ntru private key. LMS Journal of Computation and Mathematics **19**(A), 43–57 (2016)
32. Washington, L.C.: Elliptic Curves: Number Theory and Cryptography, Second Edition. Chapman &amp; Hall/CRC, 2 edn. (2008)
33. Yoo, Y., Azarderakhsh, R., Jalali, A., Jao, D., Soukharev, V.: A post-quantum digital signature scheme based on supersingular isogenies. In: Kiayias, A. (ed.) Financial Cryptography and Data Security. Lecture Notes in Computer Science, vol. 10322, pp. 163–181. Springer (2017)