Implementing countermeasures for attacks on Supersingular Isogeny Diffie-Hellman (SIDH)

Malo Ranzetti

School of Computer and Communication Sciences

Semester Project

June 2023

Responsible
Prof. Serge Vaudenay
EPFL / LASEC

Supervisor
Dr. Boris Fouotsa
EPFL / LASEC

Abstract

Isogenies between supersingular elliptic curves are useful to construct cryptographic schemes that may be resilient in a post-quantum cryptographic era. One particular scheme proposed is the Supersingular Isogeny Diffie-Hellman (SIDH) key exchange. Since its proposal in 2011, it was seen as a promising candidate [6]. However in 2022 it was shown that one can mount a devastating polynomial time attack against SIDH [3, 12]. Countermeasures to this attack have been proposed by Fuotsa, Moriya and Petit [8], but imply an explosion in the size of the scheme parameters.

This project describes the implementation of one countermeasure, namely M-SIDH, for which we implement parameter generation and key exchange for an arbitrary security parameter λ. After evaluating the system performance, we come to the conclusion that for most practical purposes, these new proposed schemes demand an extreme amount of computational power relative to the security they provide. Making them a viable cryptosystem would require a more efficient algorithm to compute isogenies of large separable degrees. Parameter generation is also affected as we would likely need a third party to pre-compute parameters long in advance.

Keywords: M-SIDH, SIDH, Post-Quantum Cryptography, Elliptic Curves

Maintained repository: github.com/mrztti/lasec-MSIDH

Contents

Chapter 1

Introduction

In recent years, the study of isogenies between supersingular elliptic curves has gained significant attention in the field of post-quantum cryptography. Isogenies provide a powerful mathematical tool for constructing cryptographic schemes that are believed to be resistant against attacks by quantum computers. One of the previously most promising schemes in this area was the Supersingular Isogeny Diffie-Hellman (SIDH) key exchange, which was first proposed in 2011 [6].

The SIDH key exchange relies on the properties of supersingular elliptic curves and isogenies to establish a shared secret between two parties over a public channel. However, in 2022, a devastating polynomial time attack against SIDH was discovered [3, 12]. As a result, countermeasures have been proposed to mitigate this attack, such as the M-SIDH scheme, introduced by Fuotsa, Moriya, and Petit [8].

This project focuses on the implementation of M-SIDH and its evaluation in terms of performance. M-SIDH addresses the vulnerability exploited by the attack on SIDH by modifying the computation of the public keys to prevent the attacker from obtaining crucial information. The implementation includes the parameter generation and key exchange for an arbitrary security parameter λ. The system's performance is evaluated to assess its practicality for real-world applications.

During the implementation and evaluation process, it becomes apparent that the computational requirements of M-SIDH are prohibitively high for most practical purposes. The schemes proposed by Fuotsa, Moriya, and Petit demand a significant amount of computational power relative to the level of security they provide. Additional progress in developing efficient algorithms for computing isogenies with large separable degrees is necessary to enhance the viability of these cryptosystems. Moreover, the generation of parameters would probably entail pre-computation conducted by a trusted third party well ahead of time (probably as a service, so that the parameter computation burden is removed from the exchanging parties). This is also for reasons mentioned in subsection 3.3.1, specifically that it may be needed for the starting curve of the scheme to be constructed by a trusted third party.

In this paper, we present the design and implementation details of M-SIDH, along with the modifications made to the original scheme. We discuss the efficiency challenges faced during the implementation as well as the optimizations made to the mathematical programming framework. The limitations and implications of M-SIDH are also discussed, highlighting why there is a need for more efficient algorithms and more design work on the parameter generation process.

Chapter 2

Background

A detailed and thorough mathematical background can be found in the original M-SIDH proposal paper [8]. Here, we give an intuitive description of the main mathematical tools relating to the project, in the hope of making this project more available without full understanding of some high-level mathematical concepts.

2.1 Diffie-Hellman Key Exchange

The Diffie-Hellman (DH) key exchange is a well documented and widely used mathematical construction which allows two parties to agree on a cryptographic key across a public channel. The scheme relies on the ability of the participating parties to construct secret, commutative trapdoor functions. Figure 2.1 illustrates the idea behind the scheme [11].

While the most popular implementation of this scheme is based upon modular exponentiation in the group $\mathbb{Z}_p$ with a large prime p as the secret trapdoor functions, SIDH uses an elliptic curve as a starting element and elliptic curve isogenies as the one-way functions.

2.2 Elliptic curves, Supersingularity & Isogenies

Elliptic curves are characterized by their equation, given in the form (Weierstrass equation):

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \tag{2.1}$$

When we take the given equation over a commutative ring, such as a finite field $\mathbb{F}_q$, we obtain a

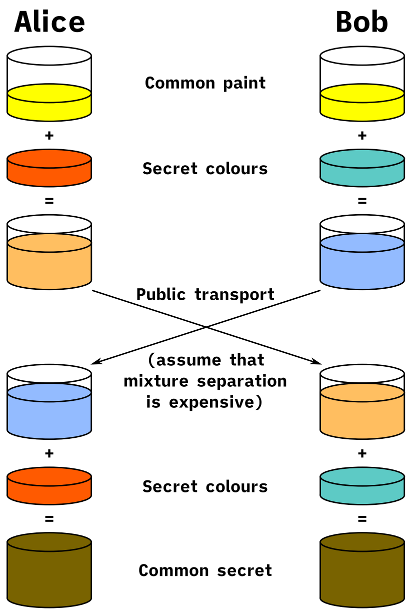

Figure 2.1: Illustration of the concept behind the DH key exchange [11]

set of points that constitutes an Abelian group. In the general case, this group is not guaranteed to have the same order as the original commutative ring and furthermore it may be hard to calculate (although we can find lower and upper bounds using Hasse's theorem). This is one of the reasons why supersingular elliptic curves are handy: for a prime p, a supersingular curve E over the finite field $\mathbb{F}_q$ defines an Abelian group of points of order $p+1$. We can also further extend this to the case where we take the finite field $\mathbb{F}_{p^2}$: now we have an Abelian group isomorphic to $\mathbb{Z}_{p+1} \times \mathbb{Z}_{p+1}$ [5] for which we can find a basis $\langle P, Q \rangle$ with both points having order $p+1$.

An isogeny is essentially a morphism ("map") between two curves (or possibly to itself). Isogenies are determined (up to isomorphism) by their kernel. The degree of an isogeny can be seen as the cardinality of its kernel. One interesting point is that the higher the isogeny degree, the more computationally complex it becomes to compute. We can also separate isogenies of composite order into a chain of isogenies of prime order in the factors of the original isogeny (which we use to calculate large isogenies efficiently).

Isogenies between elliptic curves are not a commutative operations "as-is", but if we know the image of basis points of the isogeny kernel, we can fix the image curve such that the operation becomes "commutative" for our needed cryptographic purposes.

2.3 Supersingular Isogeny Diffie-Hellman

As previously mentioned, SIDH replaces computations in the $\mathbb{Z}_p$ group with arithmetic over elliptic curves defined over a finite field $\mathbb{F}_q$ where $q = p^2$.

The **public parameters** include the curve, the integer p and importantly the values used to generate it A, B which correspond to the degrees of the isogenies used by the two parties. The torsion points of the A, B-torsion subgroup of the curve over the finite field are also made public. Concretely, this means that both parties disclose a basis for the respective starting kernel of their secret isogeny. Now, the public keys consist of the image of the base curve E_0 given by a secret isogeny of high degree, for each party. Along with the curve, each party must transmit the image of the other party's torsion points through their secret isogeny. Using these point images, the parties can apply their isogeny (albeit on a different pre-image domain) a second time on the received public key to each obtain a final curve. The shared secret, the j-invariant of the resulting curves, is then the same for both parties and can be used as a shared secret key.

Full specification: For the complete and exact details of the protocol, see the original SIDH paper [6].

2.4 Overview of the attack on SIDH

In 2022, Castryck and Decru proposed a devastating attack on SIDH [3]. Additional work by Maino, Martindale and Robert led to a polynomial attack for an arbitrary starting curve E_0 [12]. The attack leverages the following key point: there exists an efficient oracle, that for a given step in the factored isogeny construction from the starting curve, decides if it is the correct step taken in the secret isogeny. In SIDH the secret isogenies can be separated into steps of 2- or 3-isogenies. Note that for an isogeny of degree n, there are n possible corresponding possible curves to be checked (if one omits the dual isogenies for each step).

This oracle is constructed using a complex theorem (Kani's theorem) [3]. However it requires two main conditions to work [8]:

- The degree A of the secret supersingular isogeny $\phi : E_0 \to E$ must be known.
- The images $\phi(P), \phi(Q)$ of the torsion basis $\langle P, Q\rangle$ of the B-torsion $E_0[B]$ (where $B > A$ and are coprime) must be known.

As a consequence, countermeasures target either one of these conditions.

2.5 Proposed countermeasures

Two countermeasures have been proposed in response to the attack on SIDH [8]: M-SIDH (masked torsion points) and MD-SIDH (masked degree). However, we only focus on the M-SIDH variant, as for the same level of security, it makes use of isogenies of much lower degree[1].

M-SIDH proposes a defense against the attack (section 2.4) by removing one of the necessary ingredients for its success: the images of the torsion points. Indeed, the SIDH protocol is modified by ensuring that the direct images of the torsion points are not available to the attacker, but that the curves resulting from the executed protocol still have the same j-invariant.

Two main modifications are done in order to achieve this. The first one is that we scale the images of the torsion points by an integer α sampled randomly as follows:

$$\alpha \in_R \mu_2(N) := \left\{x \in \mathbb{Z}/N\mathbb{Z} \text{ st. } x^2 \equiv 1 \pmod N\right\} \tag{2.2}$$

The correctness of this scheme is proven in [7] and comes down to the fact that the scaled torsion points generate the same kernel as the direct images. However, if the attacker can find the scaling factor α, then they can run the attack.

There is an existing strategy to recover α^2 modulo the isogeny degree [7]. Therefore, the security of M-SIDH relies on the fact that the multiplicity of the square roots of α modulo either isogeny degrees grows exponentially according to a security parameter λ. Therefore we must construct each isogeny degree as a product of distinct primes linearly in the parameter λ (because the amount of square roots depends on the amount of prime factors of the modulus), so that exhaustive search of α becomes hard. Unfortunately, this means that the size of p grows faster than factorial in λ (indeed, the product of the first n primes is greater or equal to the product of the first n consecutive integers). This has consequences in terms of computation efficiency of the protocol, as we will see in this project.

[1]The prime p used in MD-SIDH for NIST security level 1 is around as large as the prime used for M-SIDH security level 5 (around 13,000 bits) [8]. As we see in section 4.2, key exchange for this large prime is unreasonable for most modern applications.

Chapter 3

Design and Implementation

The main goal of this project is to provide a proof of concept regarding practical use of M-SIDH. It is not a fully efficient and optimized implementation of M-SIDH, but does allow us to evaluate the practical cost on usability of applying a countermeasure on the original SIDH protocol. Although the most was done in the given time frame to implement as many efficiency optimizations as possible, there seems to be space to push the protocol some more to faster speeds as discussed in section 4.3.

3.1 Use of SageMath Library

As a proof of concept, the project uses SageMath [13] for better readability and modularity. This framework will allow us to benefit from many already implemented computation strategies and provides decent computational efficiency. For example, SageMath implements an algorithm to compute isogenies of high, smooth degree, which is key to implement SIDH efficiently.

3.2 Implementation of SIDH

We provide an implementation of SIDH using the same primitives and abstractions as our implementation of M-SIDH. This was done in order to have a fair comparison of the efficiency difference between the two strategies. The implementation comes equipped with a generalized interface abstraction to simulate Diffie-Hellman key exchange schemes.

3.3 Implementing M-SIDH

3.3.1 Modifications compared to proposed scheme

Some modifications were made or attempted during the implementation compared to the scheme as strictly proposed in [8]. For example, the high level description of the public parameter selection algorithm as given in the original paper does not yield the expected primes described. The fault was identified as a wrong condition with respect to the security parameter λ. Indeed, it seems that instead of *"If $\lambda < t - n + 1$, we restart with a larger t"* [pg. 21], the correct condition should be $\lambda > t - n + 1$. While a direct proof is not included here, the implementation provides a script that illustrates this statement in practice (`t_selection_proof.sage`).

Another modification which was made was to omit the 2^2 factor present in the public parameter A as given in the original paper [8]. Note that this change was also made in another implementation [10]. This was done to enable use of the `velusqrt` algorithm [1], as to use this algorithm a requirement is that A and B must be of separable and odd degree. After many attempts, it seems that something isn't right in the Sage `velusqrt` implementation as discussed in section 4.3, and so this change was dropped.

A final addition for parameter generation was to include a check relative to the base curve used. Indeed, if we take the example of the elliptic curve with j-invariant 1728, we need that $p \equiv 3 \bmod 4$ as an additional condition if we want our curve to be supersingular. We therefore add this check in the cofactor f generation step. Note that similar varying conditions might be necessary depending on the choice of the base curve.

Important remark: As noted in [8], the security of M-SIDH relies in part on the fact that the endomorphism of the starting curve E_0 should be unknown to an attacker. To achieve this, the cited paper proposes the possibility that a trusted third party could generate a random trusted supersingular starting curve using a random walk in the graph of supersingular isogenies. For simplicity's sake, we do not include this important point in the parameter generation implementation, but we do mention some implications in the concluding remarks.

3.3.2 Implementation efficiency hurdles

From early on in the project, it became clear that the biggest efficiency hurdles towards lower computation times in the implementation of M-SIDH were the calculations of generators of the base curve, sampling roots of unity of $\mathbb{Z}_n$ and the isogeny computations of the two parties. An enormous amount of tweaking and perfecting was made to finally be able to arrive at decent computation times. The first naive implementation of M-SIDH at the beginning of the project used to have computation times in the order of 5 minutes for both parameter generation and key exchange with

$\lambda = 32$. The current implementation is now around 10 times faster for the same security parameter.

3.4 Improving the Efficiency of M-SIDH

3.4.1 Improving parameter generation

Whereas SIDH parameters are considered to be entirely public and reusable, it is interesting to focus on the efficiency of parameter generation of M-SIDH (and MD-SIDH as well) since a functional implementation could potentially require the use of arbitrary curves with non-public endomorphism rings [8]. The bottleneck computation of parameter generation comes from the computation of basis points of the torsion subgroups of the chosen curve. The strategy we employ is based on the paper by Costello, Longa and Naehrig [5]: we randomly sample a point on the curve until we find one of order $p+1 = A \cdot B \cdot f$, which we denote P. We do the same for a distinct point Q. We use the properties of Weil pairings to check the linear dependence between the two points, and adjust our point Q until we have two linearly independent points that form a basis of $E(F_{p^2})$. We finally multiply these points by $\frac{p+1}{A}, \frac{p+1}{B}$ respectively to obtain the desired torsion points.

Checking the order of a point

Because the base curve is supersingular, we know that $E(F_{p^2}) \cong \mathbb{Z}_{p+1} \times \mathbb{Z}_{p+1}$. Instead of computing the exact order of a point, it is sufficient to check that the order of a possible point is exactly $p' = p+1 = A \cdot B \cdot f$. To do this, we construct the following vector:

$$LP^* = \left[\frac{p'}{l_i} \cdot P \text{ for } l_i \text{ the prime factors of } p'\right]$$

If none of the elements of the vector are the point at infinity, then P is of order p'. Indeed, we can argue from the previously mentioned isomorphism that the maximal order of any point in the curve group has a maximum order of $p+1$ and must also divide this value. Therefore, we can rule out all possible order values by checking if P multiplied by any of the largest divisors of p' (ie. all the $\frac{p'}{l_i}$) results in the point at infinity.

This technique also allows us to pre-compute values that we reuse to construct the following vector used in later steps:

$$LP = \left[\frac{p'}{{l_i}^{e_i}} \cdot P \text{ for } {l_i}^{e_i} \text{ the prime factors of } p'\right]$$

Indeed, we can obtain LP^* by further multiplying elements with non-zero prime factor exponents (there are typically very few of them) by the corresponding remaining factor. We use the same techniques to generate a point Q.

Checking linear independence

To check the linear independence of two points P,Q we use the previously constructed vectors LP, LQ derived from section 3.4.1. We then compute the Weil pairing between each pair of values from theses vectors. If the two points are linearly independent, none of the pairing values should be equal to 1. If we do encounter a pair of values for which the pairing is 1, then we know that we must adjust our candidate Q. Adjusting the point Q is faster than generating a new random point Q, checking that the order is $p+1$ and generating LQ once more. The adjustment is made the following way: let LP_i correspond to the failed pairing value:

1. Sample a new random point R on the curve, then multiply it by $\frac{p+1}{{l_i}^{e_i}}$.
2. Compute the Weil pairing of R and LP_i, and restart if it still is 1.
3. Set $Q = Q - R$

After doing this for every failed value, we can recompute the LQ and check for peace of mind that our adjusted point Q is now linearly independent from P. This algorithm is a better choice since the calculation of the vectors LP, LQ become the biggest bottlenecks when generating basis points for very large p. If we were to sample a new Q every time the check fails, we would have to calculate LQ a large number of times before we would eventually fall on a valid point with enough luck. This algorithm is also much faster than a naive call to the SageMath `gens()` function as we leverage the specific construction of the group.

3.4.2 Sampling of square roots of unity in $\mathbb{Z}_n$

The naive implementation of sampling a random element of $\mathbb{Z}_n$ and checking if it is indeed a square root of unity is highly inefficient. In the implementation given in [8], the only values of n for which we need to implement this sampling function are $n = A, B$. Because we have the factoring of these two values available to us, we can implement μ_2 efficiently (function `mewtwo` in codebase). We create an empty list to store values. We proceed by sampling a random bit b uniformly for each factor k of n (available to us. Then, if $b = 0$ we add 1 to our list, otherwise if $b = 1$ we add $k-1$. Finally we use the CRT between the items in our list in their corresponding factors as moduli to deduce our final result.

3.4.3 Implementation usage guide

The implementation provides a main entry point `run.py` which must be run using a recent version of Sage [13] and given arguments. Note that to obtain the performance as evaluated in this report, one must modify the Sage source code for factored isogeny computation as described in section 4.3. The outputs of all tests are saved to a CSV file.

Example use cases

- Show help:
  ```
  sage run.py -h
  ```
- Run SIDH implementation on p751 parameters:
  ```
  sage run.py -t sidh -c p751
  ```
- Generate M-SIDH parameters for $\lambda = 128$ as given in [8]:
  ```
  sage run.py -g128
  ```
- Generate M-SIDH parameters for arbitrary λ given as an argument [8]
  ```
  sage run.py -g <lambda>
  ```
- Test 2 rounds of M-SIDH using the parameters for $\lambda = 128$:
  ```
  sage run.py -t msidh -r 2 -f MSIDHp128.pickle
  ```
- Test 10 rounds of M-SIDH using the parameters for arbitrary $\lambda = 32$:
  ```
  sage run.py -t msidh -r 10 -f MSIDH_AES-32.pickle
  ```

Chapter 4

Evaluation

The evaluation of this implementation consists of a time based analysis. In most modern cryptosystems, it can be argued that the most decisive factor to determine usefulness is computation time proportional to security level. We will evaluate both the generation of parameters and the key exchange.

System specifications: The evaluation was performed on a M2 pro Apple silicon (12 cores 3.3GHz, 16GB RAM)

4.1 Parameter Generation

As we can see in section 4.1, the results for parameter generation are already quite alarming. Indeed, parameter generation for classical NIST-1 level security already takes around 2000 seconds. As we see in Figure 4.1, the parameter generation time is exponential relative to the security parameter. Fitting an exponential curve to the results obtained (Figure 4.2) we can estimate that generating parameters for $\lambda = 192, 256$ should take more than 5 hours and more than a day respectively. For time's sake, we will therefore omit the evaluation for these values (especially since the next section shows that cost of performing the key exchange grows even faster than parameter generation).

4.2 Key exchange

After evaluation, we obtain similar results for the full key exchange protocol in the sense that the time needed to complete the protocol grows exponentially as the security parameter (and thus p) grows. Even using the smallest parameter proposed in [8] yields computation times that are way beyond most practical uses of the cryptosystem (for this implementation at least). The total sum of

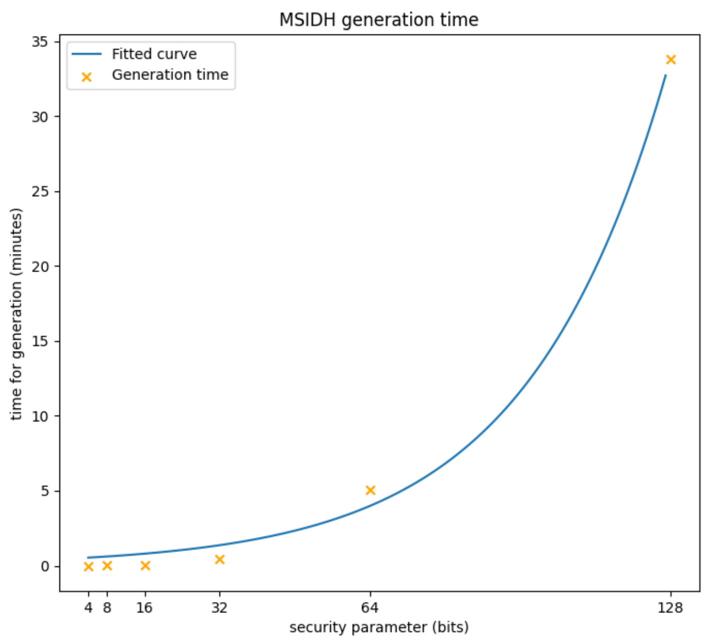

Figure 4.1: Plot of parameter generation times

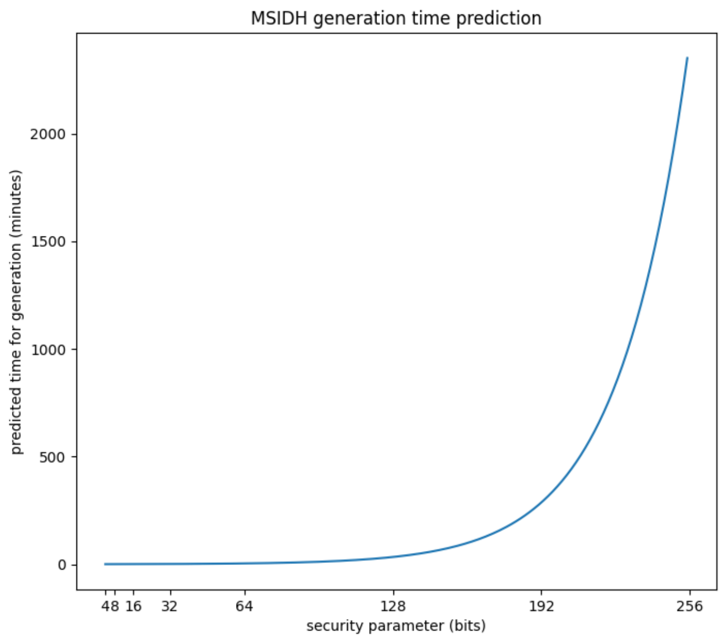

Figure 4.2: Prediction of generation times

λ **(bits)**	**Generation time (s)**
4	0.09961
8	0.5185
16	1.901
32	25.55
64	303.6
128	2026

Table 4.1: Evaluation of parameter generation (5 repetitions)

λ **(bits)**	**Key exchange time (s)**
4	0.1681 ± 0.0028
8	0.8171 ± 0.013
16	4.183 ± 0.070
32	28.27 ± 0.47
64	252.5 ± 4.2
128	3000 ± 50

Table 4.2: Evaluation of parameter generation (20 repetitions)

the computation time, as shown in section 4.2, is of around 3000 seconds (50 minutes).

These results are not so far from the results obtained in [10]. As a point of reference, completing one full exchange using SIDH as implemented in this project takes around 10 seconds for NIST security level 5 (curve `p751`). If we consider that both parties can compute the isogenies simultaneously, we can consider that this means a practical exchange could happen in 25 minutes. From the results, plotted in Figure 4.3, we can see that the key exchange is even more computationally expensive than the parameter generation. We show an estimate in Figure 4.4 of the computation time for NIST level 3 and 5 of around 16 hours and more than a week respectively.

4.3 Discussion on the limitations of the SageMath framework

In this section, we discuss the difficulties faced during the implementation of this project using SageMath. Comparing the results with the ones achieved in [10] (reproduced on the same machine as the one used to evaluate this implementation) shows that a more efficient, low-level implementation of the computation of isogenies as described in [5] may yield faster speeds for the key exchange phase. This is one of the downsides of Sage being sometimes too high level for a given goal.

The SageMath source code (9.8) used to calculate isogenies also seems to have some efficiency issues. One bug in the source code that was found was that the order of the kernel point was

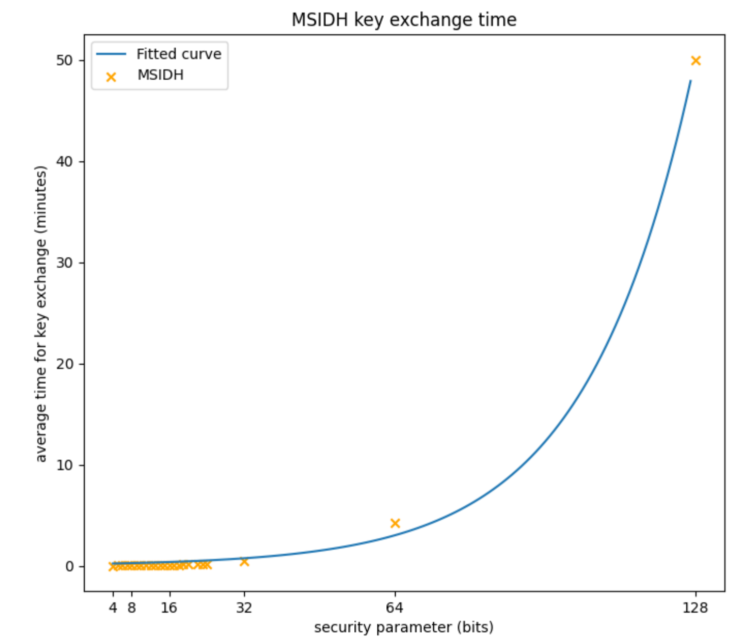

Figure 4.3: Plot of key exchange times

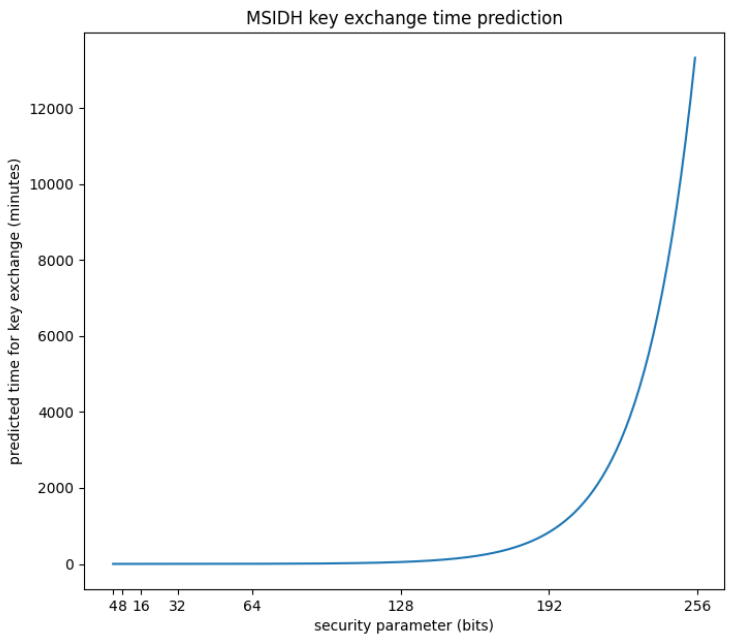

Figure 4.4: Prediction of key exchange times

calculated at every step of the factored isogeny calculation. A modified version of the source code was implemented[1]. Because the order of the isogenies was already known (A, B), function arguments were also included to directly pass the order of the kernel point and skip an (expensive) order calculation.

Furthermore, the decomposed isogenies do not leverage computation using `velusqrt`. An attempt to modify this was made, but there seems to be a performance issue with the Sage implementation of the algorithm. Indeed, the "naive" Vélu algorithm seems to have much better performance[2], even for $l > 1000$. In conclusion, the final modifications made to the source code was to remove the order calculation as well as disabling all proofs in SageMath (which was causing a major hangup by checking primality of p multiple times along the calculations).

One other example of a problem encountered came as an attempt to implement multi-core parallel computation for the key generation: currently Sage may seem to be facing some incompatibility with multi-threading[3]. This also has consequences for key exchange, since it could be possible to leverage multiple cores to divide the workload for factored isogenies.

Overall, it would seem like the biggest hope for M-SIDH (and MD-SIDH with even larger isogeny degree) to reach faster practical use speeds would be the development of a low-level, efficient and tailor made implementation of an isogeny computation of degrees in the specific form of these protocols.

[1]A copy of the modified source code is included in `hom_composite.py`

[2]This is contrary to what is indicated in the reference guide **here**

[3]Running parameter generation on multiple threads yielded a stack trace similar to the one mentioned **here**, seemingly due to some compatibility issues with the `cypari2` framework

Chapter 5

Related Work

Closely related work includes [10] (Lin, Lin, Cai, Wang, Zhao). It provides an implementation of M-SIDH which includes key compression to reduce the size of the keys. After examining the source code, it seems that the code is not modular, and includes hard coded values at many points of the implementation. An attempt was made at modularizing the code to leverage the low level elliptic curve arithmetic functions implemented in this project, however the lack of documentation and abstraction made the task much too complex. After thoughtful implementation and optimization of the SageMath source code, this project reached similar run times to [10], while being generalized to any correct parameter set.

It is also interesting to compare the efficiency of other alternative isogeny based post-quantum cryptosystems: C-SIDH [4] is one such system. Running an implementation of this scheme [9] on the same machine as used to evaluate the implementation of M-SIDH, we find that key exchange for NIST level 1 takes around 15 seconds. Some recent paper even propose and evaluate the use of C-SIDH in TLS communication [2], an application far from conceivable with the current performance of M-SIDH.

Chapter 6

Conclusion

In this project, we have shown that it is possible to implement M-SIDH as a countermeasure to attacks on SIDH. We provided a standalone, generalized implementation which can be used according to any security parameter λ. M-SIDH involves working over a finite field of much larger size, with isogenies of much larger degrees, compared to SIDH. Understandably, this implies exponentially more computation time for the same level of security, even after optimizing the implementation as much as possible. Even with state of the art algorithms, the key exchange takes almost an hour for NIST security level on a modern machine. Compared to other proposed isogeny based post-quantum cryptosystems, this is impractical. Even if there could be some hope in the development of more efficient isogeny computation, or in building a decentralized trusted system that pre-computes scheme parameters in advance, it is hard to say whether M-SIDH (and as such MD-SIDH) could ever become performance competitive with post-quantum cryptosystems.

Bibliography

[1] Daniel J. Bernstein, Luca De Feo, Antonin Leroux, and Benjamin Smith. *Faster computation of isogenies of large prime degree.* Algorithmic Number Theory Symposium 2020, Document ID: 44d5ade1c1778d86a5b035ad20f880c08031a1dc. Homepage: `https://velusqrt.isogeny.org/`. 2020. URL: `https://velusqrt.isogeny.org/velusqrt-20200616.pdf`.

[2] Fabio Campos, Jorge Chavez-Saab, Jesús-Javier Chi-Domínguez, Michael Meyer, Krijn Reijnders, Francisco Rodríguez-Henríquez, Peter Schwabe, and Thom Wiggers. *On the Practicality of Post-Quantum TLS Using Large-Parameter CSIDH.* Cryptology ePrint Archive, Paper 2023/793. `https://eprint.iacr.org/2023/793`. 2023. URL: `https://eprint.iacr.org/2023/793`.

[3] Wouter Castryck and Thomas Decru. *An efficient key recovery attack on SIDH.* Cryptology ePrint Archive, Paper 2022/975. 2022. URL: `https://eprint.iacr.org/2022/975`.

[4] Wouter Castryck, Tanja Lange, Chloe Martindale, Lorenz Panny, and Joost Renes. *CSIDH: An Efficient Post-Quantum Commutative Group Action.* Cryptology ePrint Archive, Paper 2018/383. `https://eprint.iacr.org/2018/383`. 2018. URL: `https://eprint.iacr.org/2018/383`.

[5] Craig Costello, Patrick Longa, and Michael Naehrig. *Efficient algorithms for supersingular isogeny Diffie-Hellman.* Cryptology ePrint Archive, Paper 2016/413. 2016. URL: `https://eprint.iacr.org/2016/413`.

[6] Luca De Feo, David Jao, and Jérôme Plût. *Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies.* Cryptology ePrint Archive, Paper 2011/506. 2011. URL: `https://eprint.iacr.org/2011/506`.

[7] Tako Boris Fouotsa. *SIDH with masked torsion point images.* Cryptology ePrint Archive, Paper 2022/1054. 2022. URL: `https://eprint.iacr.org/2022/1054`.

[8] Tako Boris Fouotsa, Tomoki Moriya, and Christophe Petit. *M-SIDH and MD-SIDH: countering SIDH attacks by masking information.* Cryptology ePrint Archive, Paper 2023/013. 2023. URL: `https://eprint.iacr.org/2023/013`.

[9] Gora Adj, Jesús-Javier Chi-Domínguez, and Francisco Rodríguez-Henríquez. *SIBC Python library.* 2021. URL: `https://github.com/JJChiDguez/sibc/`.

[10] Kaizhan Lin, Jianming Lin, Shiping Cai, Weize Wang, and Chang-An Zhao. *Public-key Compression in M-SIDH.* Cryptology ePrint Archive, Paper 2023/136. 2023. URL: `https://eprint.iacr.org/2023/136`.

[11] University of Duisburg-EssenSVG version: Flugaal Original schema: A.J.jacquie MacLaine /Vinck. *File:Diffie-Hellman Key Exchange.svg - Wikimedia Commons.* 2020. URL: `https://commons.wikimedia.org/wiki/File:Diffie-Hellman_Key_Exchange.svg`.

[12] Damien Robert. *Breaking SIDH in polynomial time.* Cryptology ePrint Archive, Paper 2022/1038. 2022. URL: `https://eprint.iacr.org/2022/1038`.

[13] The Sage Developers. *SageMath, the Sage Mathematics Software System (Version 9.8).* 2023. URL: `https://www.sagemath.org`.