Radboud University Nijmegen

Faculty of Science

Crypto security optimizations

Making cutting edge algorithms useful in real life scenarios

Thesis MSc Computer Science

Author:
Giacomo Bruno

Student number:
S1034730

Supervisor:
prof. dr. Lejla Batina

Second reader:
prof. dr. Wieb Bosma

November 2021

Abstract

Abstract. In 2011 the SIKE algorithm was first proposed as key encapsulation mechanism (KEM) ready for a future dominated by quantum computers. Later it was optimized and submitted to the NIST competition for post quantum cryptography, but it remains one of the most expensive algorithms, computation wise, amongst the ones taking part in the competition. Despite this, it offers some advantages in its key sizes and ciphertext sizes, making it a great candidate for usage in low power micro controllers. To make this KEM work best on such devices some device specific optimizations are required. In chapter 5 we are going to show how an optimized implementation for STM32F4 Discovery equipped with 32 bit ARM cortex-m4 micro controllers is made and show its advantage in performance against the implementation available in the latest NIST submission.

Internally, SIKE makes use of the SIDH algorithm. In 2019 Craig Costello proposed an alternative approach to SIDH that could show promising performance given the right parameters and could even replace SIDH in the right scenarios. A necessary parameter that is yet to be found is a smooth prime number of sufficient size. In chapter 4 we are going to see various approaches to finding such numbers and see an optimized implementation of one such algorithm.

Acknowledgements

During my work on this thesis, much of my work would have been more complex and abstruse were it not for the help I have received in the process. In particular I want to thank my supervisor Lejla Batina, who followed me through this endeavor and put me in contact with other amazing people. I want to thank Michael Naehrig, Maria Corte-Real Santos, Craig Costello, and Michael Meyer for their help and support in looking for more and diverse ways to optimize the neighbors search algorithm, as well as Joost Renes for his initial contribution in the development of the neighbors search program, and continuous support over time.

Contents

Notation List 6

1 Introduction 7
1.1 History . . . 7
1.2 Organization . . . 11

2 Modern cryptography 12
2.1 Diffie-Hellman key exchange . . . 12

3 Preliminaries 15
3.1 Elliptic curve cryptography . . . 15
3.2 Post-Quantum cryptography . . . 17
3.3 Isogeny based cryptography . . . 18
3.4 Supersingular isogeny key exchange . . . 20
3.5 B-SIDH . . . 22
3.6 Supersingular Isogeny Key Encapsulation . . . 23

4 Finding smooth twins 24
4.1 Methods of finding smooth neighbours . . . 24
4.2 Optimizing the Extended neighbours method . . . 27
4.3 Initial optimizations . . . 28
4.4 Further reducing computations . . . 30
4.5 Parallelism . . . 33
4.6 Good neighbors . . . 37
4.7 Better data structures . . . 38
4.8 All together . . . 39

5 SIKE, cortex-m4 implementation 42
5.1 Initial general implementation . . . 42
5.2 FPU registers as additional storage . . . 43
5.3 Karatsuba's multiplication . . . 43
5.4 Karatsuba's multiplication implementation . . . 44
5.5 503bit implementation in detail . . . 47
5.6 Results . . . 50

6 Conclusion and Future Work **53**

Notation List

E	elliptic curve
E/K	elliptic curve over field K
$\#E$	number of points on the elliptic curve E
G	cyclic group
g	generator of a cyclic group
$\mathbb{F}_p$	finite field of prime order p
$\mathbb{F}_q$	finite field with q elements
$\mathbb{F}_{p^k}$	finite field with p^k elements, with p prime
Φ	isogeny between elliptic curves
$j(E)$	j-invariant of the elliptic curve E

1 Introduction

1.1 History

In the long history of human civilization technology has been our tool to obtain more whilst working less and in fact, when developing new tools, the main focus has always been on either what that tool does, or on how much it cost for the tool do its job. The moment a tool does something useful, whether it be a necessity or a commodity, and at the same time its cost is less than the expense of doing the same task without that tool, is the moment when a new product spreads all around the people.

The discovery of technology from the first humans back in the paleolithic age can be said to have created the first cracks separating humans from the rest of the animal kingdom. From that point on wards it's possible to see example of many new and more diverse tools spreading among the people according to their usefulness.

At the beginning of our species tools for hunting were made, then fire was discovered, clothing, and eventually farming was developed with its tools. With the progress of language came writing and with it long distance communication. In the presence of wars, it became clear very soon that being able to send and receive messages to and from the troops gave an almost insurmountable strategic advantage and the ability to intercept such messages will move this advantage to the opposite side.

There is therefore a necessity to be able to protect the message with the assumption that someone whom intercepts it won't be able to make use of it. Cryptography is born.

Like many technologies developed for war, the general public would also benefit and make use of it in due time. Such is the case for Cryptography, from an instrument to hide messages for military communications, to a tool used by the masses to hide personal information, embarrassing conversations, share or hide secrets...

Cryptography has become the tool that, in our digital environment,

let us enjoy a feeling of security and confidentiality. But this tool does not come for free, there is an associated cost, whether it be time, money, or training, that needs to paid to make use of this tool.

The strength and focus of the digital environment that has been developed in the past century is its ease of use and unmatched speed. It takes almost no effort to send a message to anyone on the other side of the globe and the message reaches its destination in less than a second. In its travel to the other side of the planet, we are mostly sure the message is safe thanks to cryptography.

We have become so used to rely on this technology that any other medium for communication has pretty much become mostly irrelevant. If we were to send a paper mail to someone far away, it would take at least a few days to reach its destination, the message would most likely not be encrypted in any way and therefore it's not possible to be sure that someone other than the recipient has not read your message.

When sending a digital message, whether that's just to chat with someone or to log in your bank account, the software is in charge of encrypting that message and making sure that it reaches its destination. While it's not always 100% sure that the encryption is done in a useful manner, we have many more guarantees and the time needed to accomplish the complete task is orders of magnitude less. We got used to this process that runs automatically after we put our finger on the "send" or "log in" button: the message gets encrypted, sent through a bunch of routers around the globe and reaches a destination were it gets decrypted and shown to the recipient. All of this in less than a second of time.

Over the years many algorithms to perform encryption and decryption have become standards and are widely used on many applications and a subdivision in methodology appeared. It's been decided to differentiate types of crypto algorithms based on whether the same code can be used to both encrypt and decrypt a message or there are two different codes for each purpose. We are talking respectively of symmetric algorithms and asymmetric ones(also known as **public-**

key algorithms). The algorithms present in my thesis are public-key algorithms. The sub field of public-key cryptography is not only responsible for developing these algorithms but also for finding ways to bypass them and in fact, at the present moment, many of these standard algorithms have been found to not be as safe as it was thought.

The method driving public-key cryptography is the use of mathematical "hard" problems to hide in plain sight an encrypted message. In this context an hard mathematical problem consists of a problem whose solution requires an incredibly expensive computation, be it memory of time wise.

The algorithms for solving these problems are available to anyone but the amount of time required to get a solution when the key is a big number (more than 300 bits) is so much that in fact there are many cases of such problems remaining unsolved from from 1990s.

A great example of this is the RSA Factoring Challenge[1], where is given a semiprime number n, composed by the product of two prime numbers, and it's asked to find the two prime factors used to generate it. The last number to be factored from this challenge is a number of 829bits and the solution was found in February 2020[2].

In practice, public key systems, are rather clumsy when used to transmit long messages. They become instead extremely useful when used in conjunction with symmetric cryptography, where the main issue is sharing the secret key. Using a public-key algorithm, the secret key of the symmetric algorithm is shared in a public channel safely. Additionally this process is made even simpler by Key Encapsulation Mechanisms (KEM).

A Key Encapsulation Mechanisms (KEM) uses a public key system to share a symmetric key, that itself is derived from an element of the underlying finite group of the public key system.

Making use of these problems it's possible to bypass the encryption only after thousands of years worth of computations (by today's processors). Back in 1980s the first ideas for quantum computers were starting to surface and in 1994 it was shown that, through a quan-

tum computer, it's possible to factorize numbers in a very short time. Number factorization is one of the "hard" problem that is widely used in many algorithms today to guarantee privacy and the coming of quantum computers would deny the security of any information encrypted with algorithms that relied on factorization.

Obviously, researcher did not sit on their hands waiting for a security apocalypse to happen. Work has been done to develop new algorithms that rely on different "hard" problems that are not easily solved even through an quantum computer. At the present time there are more than five directions being taken against it and the one I will focus on is called "supersingular elliptic curve isogeny cryptography".

The contribution of this thesis is not in the development of such an algorithm but instead on its implementation and optimization, in particular I'm going to show an approach to finding parameters capable of speeding up the recently introduced B-SIDH algorithm. Through these optimizations it was possible to find a bigger prime than the one previously found with the same algorithm, and it was done in a fraction of the time it would have previously took. The second part of my contribution is in the form of an optimization of the implementation of the SIDH/SIKE algorithm for Cortex-M4 micro controllers obtaining almost a ten-fold increase in speed in large number multiplications.

1.2 Organization

This thesis is divided in two parts, the first part composed of two chapters:

- **Chapter 2**: An introduction to modern public key-key cryptography.
- **Chapter 3**: A showcase of the contemporary response in the world of crypto to quantum computers and a more in depth explanation of the algorithms relevant to this thesis (B-SIDH and SIDH).

The second part of the thesis focuses on my contributions:

- **Chapter 4**: An in depth showcase of the optimizations implemented in the program responsible for finding the necessary number to a breakthrough in B-SIDH's performance.
- **Chapter 5**: A thorough explanation of the architecture specific optimizations introduced in SIKE/SIDH.

2 Modern cryptography

Before the digital era, cryptography was a simple manipulation of traditional characters that gained its security by hiding what manipulations were being done. This is called security by obscurity.

With the coming of computers, and the departure from security by obscurity, cryptographers started to take advantage of the newfound processing power and put into practice publicly known mathematical algorithms to encode information.

Until June 1976 cryptography mainly consisted of encryption mechanisms that use the same key, shared between those who need to decrypt the cipher text. This is called **symmetric cryptography**.

The problem of this method is that the parties involved in the communication either need to know the secret key a priori or they are forced to communicate the key unsafely.

This problem highlights the need at the time for a different method of encryption that could guarantee the safety of the communication without the shortcomings of symmetric cryptography and the solution comes in the form of **asymmetric cryptography**, or **public-key cryptography**.

2.1 Diffie-Hellman key exchange

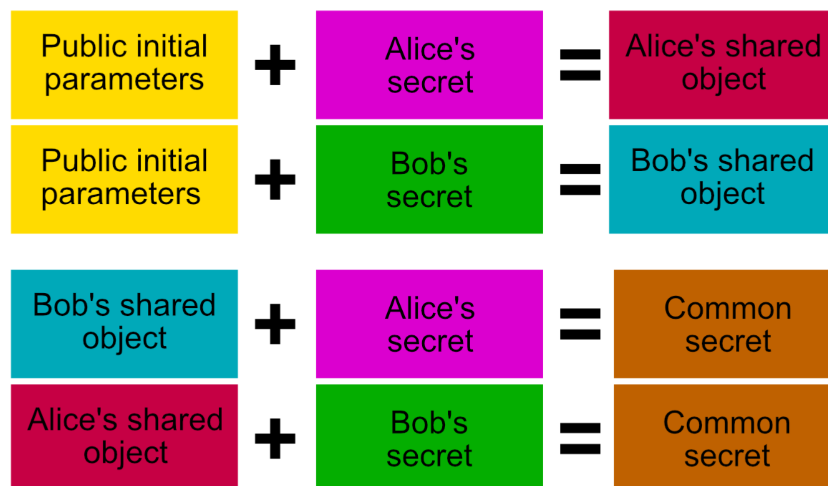
In this section I'm going to show the first devised method that signified the birth of asymmetric cryptography:
Diffie-Hellman key exchange.

In 1976, Whitfield Diffie and Martin Hellman published the first known work showing a method of safely exchanging cryptographic keys in a public channel[3].

In the Diffie-Hellman key exchange (D-H), two entities (Bob and Alice) each have a secret and a public object. They use the other's public object to generate a new object that they then share to each

other. Finally they use their private object to generate a new private object that's the same for both of them and that will be used for encryption/decryption.

Notice how I'm talking about objects and not something more precise. That is because what the objects and how they are used to generate new objects is not important to the general scheme. What is important is that, given the public objects and the first two generated objects that are shared between Bob and Alice, going backwards and finding the private objects is an "hard mathematical problem".



In the original implementation multiplicative groups of modulo p, where p is a prime number, was used to make this exchange.

Diffie-Hellman key exchange importance comes from establishing this general method of sharing secrets, thanks to this, when a "hard mathematical problem" is not hard anymore and computing the private objects backwards becomes "easy", one can swap the objects in this algorithm for different ones in an effort to maintain the same(or better) level of security as before whilst maintaining the same general schema.

More specifically the objects used in Diffie-Hellman schema are **cyclic groups**.

Definition 2.1. A cyclic group is a set of invertible elements with a single associative binary operation between them. This set needs to also contain an element g such that every other element of the set can be obtained by (repeatedly) applying the group operation to g or its inverse. This element g is called the **generator** of the group.

Definition 2.2. The order of a cyclic group G is the number of elements in G.

Historically, with the use of symmetric-key algorithms, where one key is used to both encrypt and decrypt messages, it was necessary for the two parties involved to both have that key available and the process of sharing the key between them came to be called key-exchange.
Before Diffie-Hellman key exchange was introduced, the big problem to these algorithms was precisely how to share the secret key in a public and unsafe communication channel.
With Hellman's PhD student Ralph Merkle's ideas, in 1976, a new method, that was later known as called Diffie-Hellman key exchange (D-H),was first introduced to share a secret key safely in a public channel.
It's worth mentioning that the same idea behind these public channel key exchanges actually goes back to 1973 when Malcolm J. Williamson, James H. Ellis, Clifford C. Cocks from GCHQ(Government Communications Headquarters in UK) first came up with the idea but, because they were working for British intelligence they could not publish any of their work.

At the present moment, the D-H key exchange saw its uses with multiplicative groups and through elliptic curve cryptography (ECDH). With the advent of quantum computers, both of these implementation of D-H key exchange cannot be considered to be safe anymore. To overcome this new hurdle, researchers started looking towards supersingular elliptic curve isogeny graphs.

3 Preliminaries

3.1 Elliptic curve cryptography

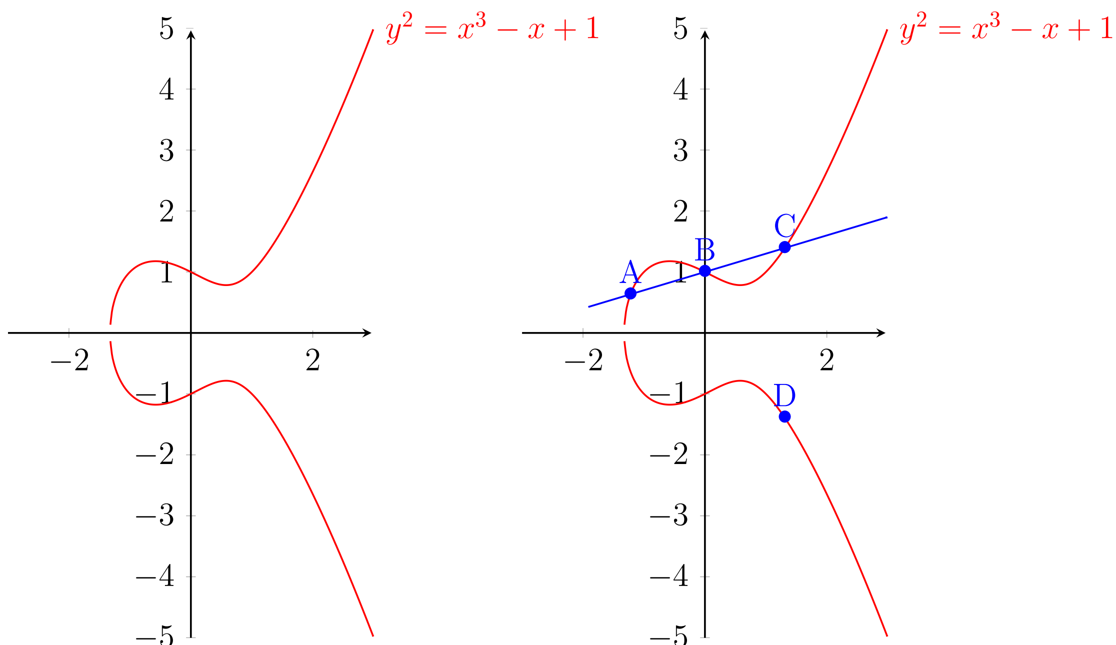
Elliptic curve cryptography (ECC) is a type of public-key cryptography based on Elliptic curves defined over finite fields.

An elliptic curve is a smooth projective algebraic curve of genus one having a specified point O, or more simply a set of point satisfying equations of shape:

$$y^2 = x^3 + ax + b$$

An important property of elliptic curves for cryptography is that they are horizontally symmetrical: any point on the curve can be reflected over the x axis (by negating the y coordinate) and the resulting point is still part of the curve.

Another important property is that a line intersecting two points on the curve will also intersect a third point on the curve. If a line is tangent in one point of the curve then it intersects one other point in the curve.



Using these properties the point addition operation on elliptic curves

is defined: given two points in the curve then their addition is the reflection of the point obtained from the intersection between the line intersecting both addendum points ($A + B = D$) as shown in the figure).

If you repeatedly do this addition n times using always A as an addendum and the result of each addition as the second addendum it's impossible to figure out the value of n by just the value of A and the final result of the repeated addition unless you do this process anew.

In cryptography n represents the private key and finding n is called the "elliptic curve discrete logarithm problem".

This is the "hard" mathematical problem making ECC work, but this is a problem that can be solved using Shor's algorithm on a hypothetical quantum computer. Not quantum computers would break ECC cryptography, but it would also be easier to break ECC than to break RSA algorithms, in terms of amount of qubits required for the quantum computer to have.

3.2 Post-Quantum cryptography

We have seen how Quantum computers are going to be ruining cryptography as we know it. Before diving into one of the possible solutions (Isogeny based cryptography), I want to mentions other techniques and approaches aiming to put on a fight against for the future computers.

At the present moment five new fields of cryptography have started to develop:

- **Lattice-based cryptography**: Nth Degree Truncated Polynomial Ring Unit (NTRU) cryptosystems, Goldreich–Goldwasser–Halevi (GGH) cryptosystems.
- **Multivariate cryptography**: Rainbow singature scheme.
- **Hash-based cryptography**: Lamport signature, Merkle signature scheme, XMSS, SPHINCS.
- **Code-based cryptography**: Niederreiter cryptosystems, McEliece cryptosystems.
- **Supersingular elliptic curve isogeny cryptography**: SIDH, B-SIDH.

The National Institute of Standards and Technology (NIST)[4] is currently organizing a competition to update their standards to include post-quantum cryptography and amongst the competitors there are many of the algorithms mentioned above and other (CRYSTALS-KYBER, SABER, BIKE, NTRU Prime, SIKE, ...). The competition started in 2016 and we are currently at the third submission and the drafts for the new standards are planned to be released between 2022 and 2024.

3.3 Isogeny based cryptography

Before advancing into Supersingular elliptic curve isogeny cryptography, it's necessary to have some knowledge on isogeny based cryptography first.

Definition 3.1. An **isogeny** is a morphism of algebraic groups that is surjective and has a finite kernel

In the case of elliptic curves (E_1, E_2), an isogeny between them is a rational morphism $\Phi : E_1 \to E_2$ that maps the point at infinity O of E_1 to the point at infinity O of E_2. Elliptic curves have a distinguished point(O) and a group structure therefore the isogeny Φ preserves this structure.

Definition 3.2. Given two elliptic curves E_1 and E_2, let $\Phi : E_1 \to E_2$ be an isogeny between them over a field k and let $k(E_1)$ and $k(E_2)$ be function fields of E_1 and E_2.
The **degree** of Φ is defined as deg $\Phi = [k(E_1) : \Phi * (k(E_2))]$, where $\Phi * (k(E_2))$ is a sub-field obtained by the composition of Φ and $k(E_2)$.

Elliptic curves are the basis of elliptic curves cryptography (ECC), but because an attacker needs to solve an instance of the discrete logarithm problem they are not safe from quantum computer attacks.

Isogeny based cryptography is the first evolution of ECC, [5] and it's security stems from the problem of finding an isogeny between two elliptic curves, which is not made substantially easier to solve by quantum computers.

The first algorithm making use of isogenies is Couveignes Rostovtsev Stolbunov key exchange (CRS) but the problem with this algorithm is that it's unacceptably slow, taking several minutes for a single key exchange.

Additionally in 2010[6] it was showed that solving an instance of the abelian hidden-shift problem was enough to break the security of the CRS scheme and this problem has a known sub-exponential time quantum algorithm solving it[7].

Later CSIDH[8] was introduced, solving the speed problem of CRS, but it fails to address the abelian hidden-shift problem showed in 2010.

The current evolution of ECC, supersingular elliptic curve isogeny cryptography, aims to solve this problem by using all the supersingular elliptic curves in a finite field of p^2 elements.

Definition 3.3. Supersingular elliptic curves are a special class of elliptic curves over a field of characteristic $p > 0$ with large endomorphism rings.

3.4 Supersingular isogeny key exchange

Supersingular isogeny Diffie-Hellman key exchange (SIDH) is an algorithm analogous to the Diffie-Hellman key exchange algorithm, but based on walks on a Supersingular Isogeny Graph.

Definition 3.4. Supersingular isogeny graph are a class of expander graph having

- **nodes**: supersingular elliptic curves over finite fields
- **labels**: j-invariants of the curves
- **edges**: isogenies between the curves

The supersingular elliptic curves in Montgomery form represented with:

$$E_a : y^2 = x^3 + ax^2 + x \tag{1}$$

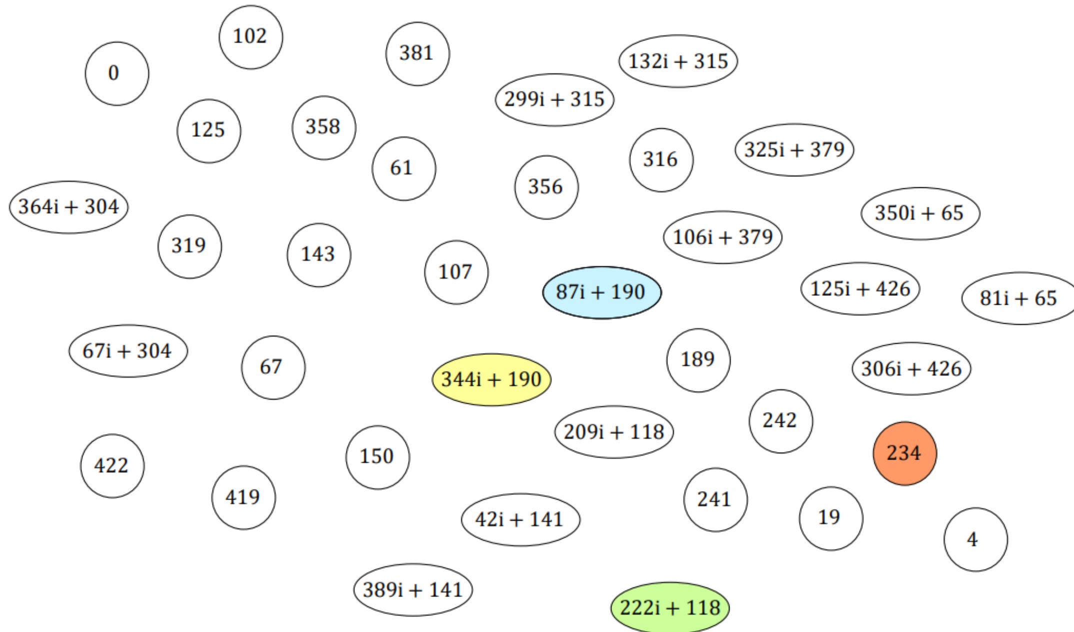
and j-invariants are:

$$j(E_a) = \frac{256(a^2 - 3)^3}{(a^2 - 4)} \tag{2}$$

Definition 3.5. A j-invariant is a modular function, a complex analytic function on an upper half-plane $\mathcal{H}$ satisfying some equation with respect to the group action (edges), defined on the upper half-plane of complex numbers($\mathcal{H} \equiv \{x + iy \mid y > 0; x, y \in \mathbb{R}\}$).

The curves used in SIDH use primes of the form $p = 2^m 3^n - 1$ and each isomorphism is represented by a supersingular elliptic curve with group order$\#E(\mathbb{F}_{p^2}) = (p + 1)^2 = (2^m 3^n)^2$

Using the example given in Craig Costello's paper on SIDH [9], using $p := 431$ we have the following graph:



In a traditional Diffie-Hellman algorithm, we have a cyclic group G with public generator g. Analogously the blue node with j-invariant $87i + 190$ acts as the generator g. Alice would then choose a secret integer that moves her along a subset of the graph until she reaches the green node. This node will be used as a public key to be sent to Bob. In the same way Bob moves from the blue node to the yellow node and he shares this node to Alice. The both of them then move from the received public key using their secret key and reach the red node. These movements are done using the isogenies that were the edges of our graph.

3.5 B-SIDH

In SIDH we have seen that only curves with group order $\#E(\mathbb{F}_{p^2}) = (p+1)^2$ are used.
B-SIDH [10], additionally, uses curves of group order $\#E(\mathbb{F}_{p^2}) = (p-1)^2$. The curves of these two sets are not isomorphic nor isogenous to one another in $\mathbb{F}_{p^2}$ but they become isomorphic in $\mathbb{F}_{p^4}$, and therefore they share the same j-invariant in $\mathbb{F}_{p^2}$

The whole point of B-SIDH, as explained in Craig Costello's paper on the matter, is remove the restriction of SIDH on the two sets of quadratic twists. The advantage over SIDH is that the primes and the public keys would be significantly smaller than those required in SIDH.

The only claims in regards to B-SIDH's performance are that on Alice's side there will be a clear improvement due to the size of the prime being smaller. On Bob's side there will almost always be a "colossal slowdown"

In that same paper have been shown that the two main obstacles in making the performance of B-SIDH competitive with SIDH/SIKE are

- faster methods of computing l-isogenies (isogenies of degree l)
- finding large enough primes p having p+1 and p-1 be as smooth as possible.

The first part of this thesis will hinge on the second obstacle to B-SIDH.

3.6 Supersingular Isogeny Key Encapsulation

The Supersingular Isogeny Key Encapsulation mechanism (SIKE) is the only post-quantum key encapsulation mechanism based on supersingular elliptic curves and their isogenies. It was proposed in 2011 by Jao and De Feo[11].

When compared to other algorithms submitted to NIST's post quantum competition it is one of the slowest, requiring the most number of clocks. Its advantage comes in the size of keys it uses and the size of the encrypted message. Because of this, SIKE is a good fit for resource constrained devices as the Cortex-M4.

In SIKE, like in SIDH, the supersingular elliptic curve E_0/F_{p^2} is public, the prime p of shape $l_A^{e_A} l_B^{e_B} \pm 1$ is known and dependant on the implementation. The values of l_A and l_B are fixed to 2 and 3, while e_A and e_B are dependent on the security level of the implementation.

Parameter Set	Security Level	Public key (B)	Chipher Text (B)	Shared Secret (B)
434bit	1	330	346	16
503bit	2	378	402	24
610bit	3	462	486	24
751bit	5	564	596	32

SIKE has four different primes to ensure NIST security level 1, 2, 3, and 5, respectively of length 434, 503, 610 and 751 bits. The computations over such huge numbers is a challenging problem especially for resource constrained devices. The acceleration of these operations is the main focus to improve SIKE and its timing, and energy consumption.

4 Finding smooth twins

In this section I talk about the methods and ideas used to find greater smooth prime numbers whilst taking meaningfully less time than before eventually discovering a larger result than the one previously discovered through the same algorithm.

The reason we are looking for a large smooth prime is to make B-SIDH useful in real scenarios and to do so a big enough prime number is needed. This prime number p needs to have for neighbors $b = p - 1$ and $B = p + 1$ such that both b and B are smooth numbers for a relatively low smoothness. The prime needs to be in between 200 and 300 bits of length and the smoothness needs to be too not much higher than 1000.

Definition 4.1. A z-smooth number is a number only divisible by numbers less or equal to z. The number z is called smoothness.

Finding primes with this property is an expensive task, and the fastest method currently known first finds a smooth neighbour pair.

Definition 4.2. A smooth neighbour pair is a pair of two numbers m and $m + 1$ that are both z-smooth.

Once a neighbour pair is found, m is multiplied by two and if $2 * m + 1$ is a prime number then it's also a z-smooth prime number.

4.1 Methods of finding smooth neighbours

Lenstra's method. Using Størmer's thorem [12] it is known that for a given smoothness z the set of smooth neighbors is finite and it's possible to find all these neighbour pairs using the Pell equation.

A Pell equation is a Diophantine equation of type:

$$x^2 - dy^2 = 1$$

$$x^2 - dy^2 = -1$$

The standard method of solving these equations is through continued fractions, by expressing $\sqrt{d}$ in its continued fraction form.

Continued fractions of a square root are cyclic, meaning that the fractions at a certain point start repeating themselves. One can truncate the fraction at the end of a cycle to obtain an approximate solution to the square root.

To solve the Pell equation $x^2 - 14y^2 = 1$...explain here continued fraction

After we reach the first cycle we can calculate the fraction to obtain a number of the form $\frac{a}{b}$. In the example from before we would get that $\sqrt{14} = \frac{15}{4}$. We let $x = a$ and $y = b$ and obtain a solution to the equation. This solution obtained from the truncation up to the first cycle of the continued fraction of the square root of d is called the fundamental solution.

Størmer's initial method for finding all smooth neighbors given a certain smoothness z involves calculating 3^k Pell equations, where k is the number of primes in $[2 - z]$. This method was later on improved and simplified by D. H. Lehmer[13] In his method you only need to calculate the following Pell equation

$$x^2 - 2qy^2 = 1$$

for each z-smooth square free number q other than 2. Since each q is generated as a product of a subset of the primes in $[2 - z]$ with size k, $2^k - 1$ Pell equations need to be calculated.

Since every smooth neighbor is of the form $\frac{(x_i - 1)}{2}$ and $\frac{x_i + 1}{2}$ we can check for smoothness only the solutions to the Pell equation above that also are of this form.

Because this method requires the usage of continued fractions, it's exponentially slow and it will not run in polynomial time. Meaning that it's not useful for finding neighbors pairs for large smoothness $(z > 150)$.

PTE-Method. This method showcased by Craig Costello[14] is based on solving Prouhet-Tarry-Escot (PTE) problems where for multisets $\{a_1, ..., a_n\}$, $\{b_1, ..., b_n\}$ $a_1^i + ... + a_n^i = b_1^i + ... + b_n^i$ holds for all $0 \leq i \leq n-1$

From this multiset we obtain the following polynomials:

$$a(x) = \prod_{i=1}^{n}(x - a_i) \qquad b(x) = \prod_{i=1}^{n}(x - b_i)$$

It's proven that $a(x)$ and $b(x)$ differ only by a constant $C \in \mathbb{Z}$. If you can find $d \in \mathbb{Z}$ such that $a(d) \equiv b(d) \equiv 0$ *modulus* C then $a(d)$ and $b(d)$ are the smooth neighbors you were looking for.

This method is great for limiting the search of a smooth prime to a required bit length as it does not need to build up growing sets of smooth pairs that would over time hinder the performance but it's not as reliable. The benefit you get from the constructive approach is that you get all, or most, of the smooth pairs up to the required smoothness.

Extending neighbours method. This is a constructive approach that starts from an initial set of z smooth numbers (easily all numbers $\leq z$) and generates new smooth neighbours and keeps repeating this process until no new neighbour are found anymore. Given two z-smooth numbers (b and B) it's possible to find a new z-smooth number by solving the following equation:

$$\frac{\beta}{\beta + 1} = \frac{b}{b + 1} \times \frac{B + 1}{B}$$

4.2 Optimizing the Extended neighbours method

The main idea is that, given a smoothness z and two numbers b and B that are z-smooth by solving this equation we find a new z-smooth number β [15]

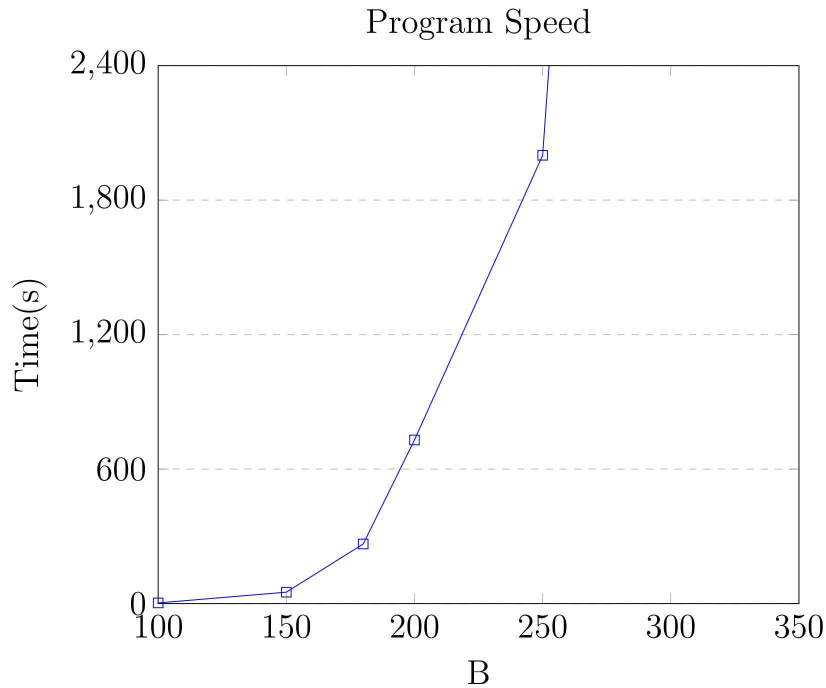
$$\frac{\beta}{\beta+1} = \frac{b}{b+1} \times \frac{B+1}{B} \tag{3}$$

We can apply this equation to sets of z-smooth numbers and obtain an algorithm that keeps enlarging this set until no longer possible.

The first step of the algorithm is to decide on a starting set S to work from. We can easily fill S with numbers less or equal to the smoothness z.

The second step, which is usually referred to as "the iteration", consist of taking all possible pairs (b, B) with $(b < B)$ of this set S and then finding the solution to (1) with each pairing. After we have found all solutions, we store the new ones into the set S.

Finally we repeat the last step until we get no new results.



As we can see, this algorithm experience an exponential growth in time needed as the smoothness increases.

4.3 Initial optimizations

From this initial description we can easily see that the number of computations we do on each iteration is n^2 where n is the number of elements currently in S. Furthermore each iteration *forgets* what pair it already tried and does the same computation again.

The first optimization that can be done is exactly focused on this aspect.

Instead of having only a single set S, populated in the same way as before, we also have an additional set T that will contain only all the **new results** obtained from the previous iteration. Then the pairs for each iteration are created from one element of T and one element of S.

This simple operation removes **all** redundant computations.

Thinking about the computations themselves, is there a way to make them faster? Looking at equation **(1)**, what comes easy to do is

1. $x = b * (B + 1)$
2. $y = B * (b + 1)$
3. calculate canonical form of $\frac{x}{y}$
4. check whether $x - y = 1$
5. save y if the check passes.

One can optimize this by removing one multiplication but there actually is a much better approach that does not require to calculate the canonical form of $\frac{x}{y}$, which is the most expensive part of this calculation.

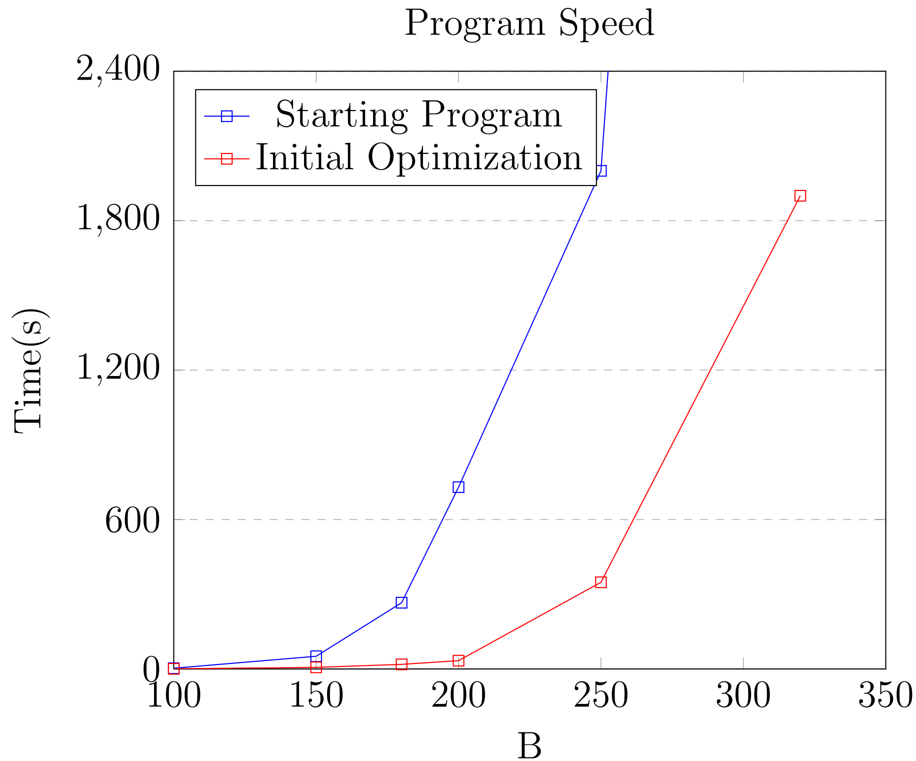
Here we can see the intermediate steps before reaching the optimized equivalent procedure.

1. g = GCD(x, y)
2. check that $\frac{x}{g} - \frac{y}{g} = 1$
3. this is equivalent to saying "check that $B - b = g$"
4. additionally $\text{GCD}(bB + b, bB + B) = \text{GCD}(bB + b, B - b) = g$
5. therefore we can simplify $B - b = g$ to $((bB + b)\ modulo\ (B - b)) = 0$

This is the optimized procedure, completely removing the calculation of any GCD, and calculating the resulting number only in the case that that number satisfy our restrictions.

1. $x = b * (B + 1)$
2. $\Delta = B - b$
3. $d = x\ modulo\ \Delta$
4. check that d is equal to zero
5. save $\frac{x}{\Delta} - 1$ if the check passes

In conclusion we have gone from n^2 computations to $n * k$ computations where n is the number of elements in S and k is the number of elements in T (with always $k < n$). In addition we made each computation more lightweight.



4.4 Further reducing computations

From this point onward, each optimization applied is going to result in a slight loss of total results found. The focus becomes to balance this loss with enough speed to justify it.

The first thing that was done, was reducing the numbers of pairs that are used in each iteration. Not every pair produces a new result, and some pairs are better than others. Initially it was decided to create pairs (b, B) such that $b < B < k * b$ where k is an arbitrary constant usually $\in [1, 2]$. With decreasing value of k the number of pairs reduces, and the speed goes up. At the same time the number of results become lower.

It was found with increasing values of b, B k can decrease without having a big impact on the results. The same is true the other way around, with small values for b, B k needs to be bigger.

The biggest problem in this approach is the overhead in calculating $k * b$ and finding this number inside the set S. Additionally it needs to be considered that this calculation needs to be done twice for each number in the set T.

One way to lessen this burden is to subdivide the set T in chunks and do this operations only for the first and last element of each chunk, thus reducing the overhead costs.

From the same line of reasoning, a somewhat easier but less reliable approach comes about that would remove completely the overhead of finding $k * b$, **constant ranges**.

With constant ranges you are trying to guess ahead of time how many numbers there are in between b and $k * b$ and iterate through the set S for that amount of numbers when creating pairs.

Another approach that was explored was reducing the amount of work the program does when it starts getting less results: once a new iteration yields less new results than the previous one, instead of keeping around the whole S set for pairings, only use the results from the last two iterations to generate pairs. This approach was found to be almost a free efficiency upgrade, with very little cost in the amount of results obtained.

One can even decide to use a combination of constant ranges and this pairing reduction: when only two small sets are used for pairings, increase increase the constant range so that possibly more results are squeezed out of the last operations.

One can also play around with **variable ranges**: the same as constant ranges except they are different for each iteration based on the size of S and T. Using pairs (b, B) such that $b < B < k * b$ as we saw before can be considered a special kind of variable ranges, dependant instead on the number being used for the pairing.

Figure 1: Number of neighbours found (higher is better)

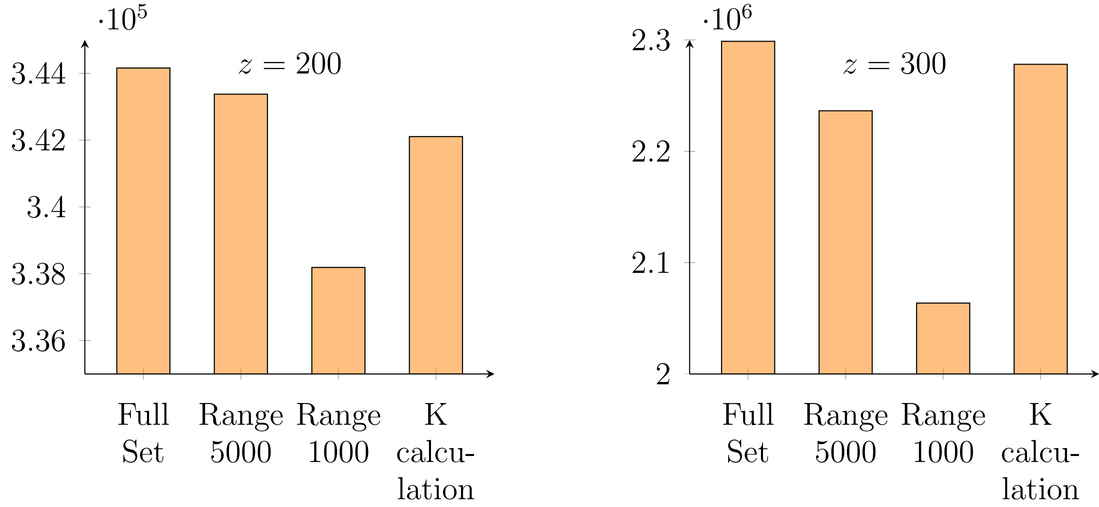
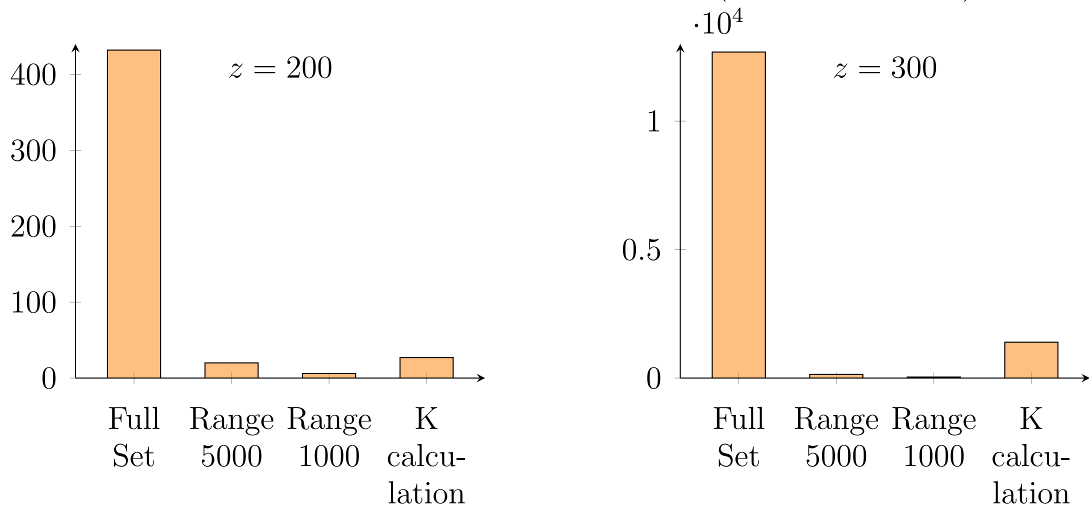


Figure 2: Time required on 32 cores (lower is better)



4.5 Parallelism

There are different approaches one can utilize with this algorithm in regards to parallelism.

The simplest one would be to have a task for each number in the set T and have the task do all the necessary work to pair $t \in T$ correctly with $s \in S$. The problem with this approach when constant ranges are not being used is that the overhead on creating the pairs is applied to every element of T and is a massive decrease in performance. Additionally task management costs time and resources, making a task for each element of T would result in the tasks fighting over resources and slowing the whole process.

It's been decided a fixed amount of task, decide at compile time, that should be equal to the amount of threads the machine running the code can sustain.

The process of "creating pairs" has been replaced by giving the elements in T a range of elements to pair with in S. For example if we have a number x the ranges r_1 and r_2 associated with it are going to be

$$r_1 = size(\{y \in S | y < x \land y > x/k\})$$
$$r_2 = size(\{y \in S | y > x \land y < x * k\})$$

By using these ranges we can then iterate through S starting from element x and create pairs in this way. Additionally we can approximate that close numbers in T are going to have close ranges r_1 and r_2 and therefore create "batches" of element in T using the same ranges: r_1 computed from the first element of the batch and r_2 computed from the last element of the batch.

As one can imagine, having batches of too small a size does not improve the situation compared to not having batches at all. Also having batches too big in size become counterproductive: the ranges are going to become less and less accurate, often becoming way bigger than they need to be and therefore generating a large amount of "useless" pairs.

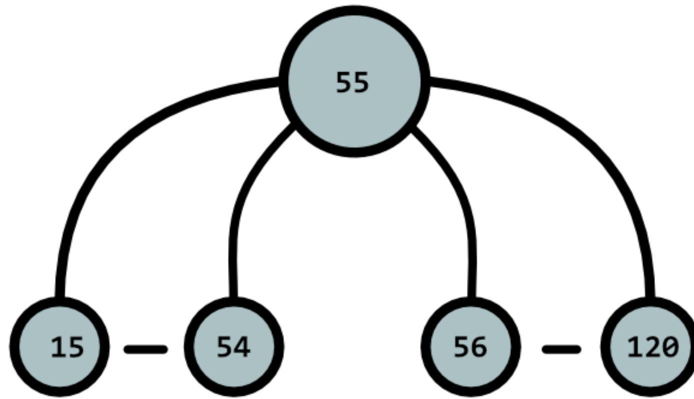
After having calculated all the ranges for all the batches, each batch is given to another task responsible for computing all the new neighbors using the pairing generated by iterating through S.

Through trial and error, it's been decided to make the size of the batches dependant on the size of T whilst always staying in between 10 and 100 as this resulted in a good ration of speed and results found.

Because of this, the process of generating batches, associating pairs, and computing new smooth numbers is gonna be done multiple times until the whole set T has been emptied.

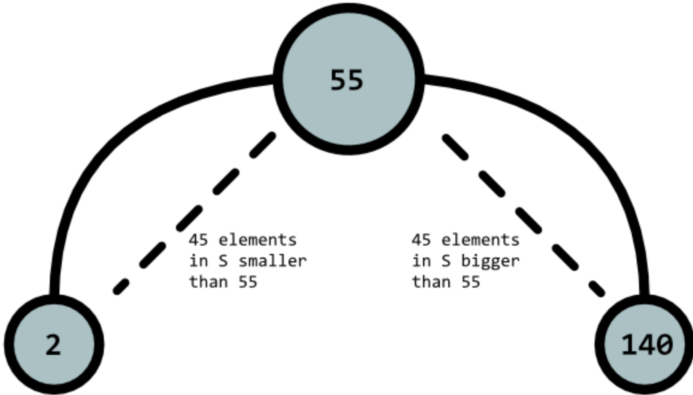
Here two examples of how a batch is made and used, in the first example there are batches without ranges, that are made of two subsets of S, one for number bigger than $t \in T$ and one lower, and a subset of T. In this way there is more control in the number of pairs generated by each batch.

Batch Number	Batch 1	Batch 2
T Elements	[30-60]	[61-120]
S Elements (top)	[31-120]	[62-440]
S Elements (bot)	[15-29]	[31-61]



{(15,55);(16,55);...;(54,55)} + {(55,56);(55,59);...;(55,120)}

The second example instead shows how a batch works with ranges. Each batch is associated a range of elements of S to pair with each $t \in subset\ of\ T$. There is going less accurate control in the number of pairings being generated but the overhead of calculating the two subsets of S is removed, leading to an important benefit in performance.



4.6 Good neighbors

When finding solutions to (1), it was found that some numbers yield more results while others less.

Smoothness	Number	#Twins Found
100	36875124	59
100	33669999	63
100	1534811544	7
100	486351112	6
200	1100322575	9
200	36875124	262
200	33669999	144
200	1534811544	72
200	486351112	37
200	1100322575	104

It can be seen in this table, that a number giving few results for a certain smoothness might, at a different smoothness, yield more results therefore we cannot make a precalculated table valid for all smoothnesses to store either all the good neighbors or the bad ones.

Thanks to this notion, in each run of the program we can "discard" some of the results and only keep the "good" ones for making pairs in future iterations.

In each iteration, we have pairs (s, t) with $s \in S$ and $t \in T$. Because of how the code was structured, the pairs are computed somewhat orderly, meaning that all pairs with the same t will be computed together. It comes easy here to keep count of how many results each t yields and remove all t that do not yield enough results from S and store them separately in another set. In this way we reduce the number of pairs we have in every future iterations at a very minimal cost in number of results.

We unsuccessfully attempted to find a characterization for these numbers by looking at their factors and by looking at the d of the Pell equation that can generate them. For now no good way of character-

izing such numbers has been found.

Work in this direction is still ongoing, but showing some results.

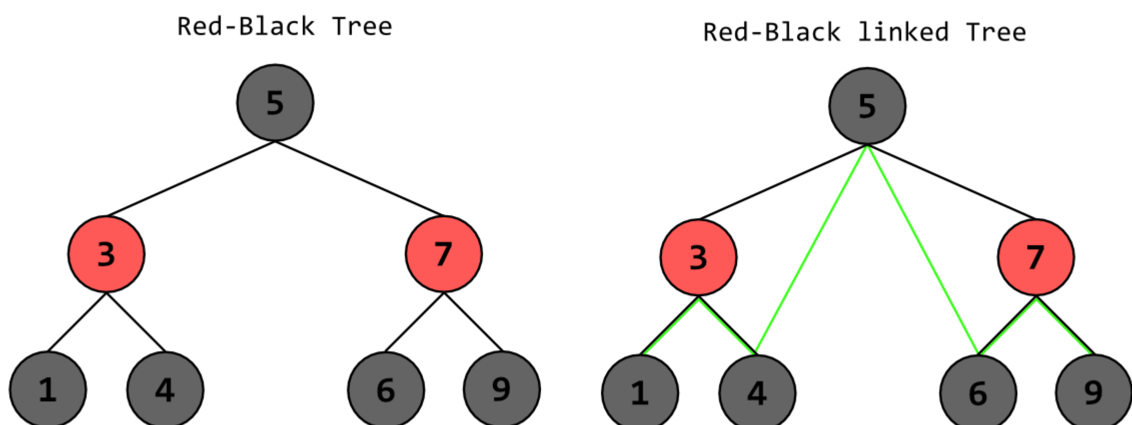
4.7 Better data structures

Another part of the algorithm that was mainly overlooked for the majority of the project is the data structure representing the sets S and T.

Initially S was a simple array. But this required doing additional operations for ordering and was inefficient in search and allocation.

The code at this point got moved from plain C to C++ to make use of the available standard library to simplify development. At this point S had become an $std :: set$, a ordered data structure with $O(logn)$ search.

Last but surely not least, the std::set was replaced by red-black trees with linked leafs. This change removed some of the overhead that std::set has for being a type agnostic container, but more importantly made iterating through the set, either from the beginning or from an arbitrary node, both forwards and backwards more efficient.



It can be easily seen that iterating from one node to the next in

the given order is a constant operation with linked trees thanks to the edge connecting elements next to each other. Normal Red-Black trees don't have this connection and need to do more operations to reach the next(or previous) element in the order.

This is extremely useful and impactful in the current use case, because it's necessary to iterate through N nodes in order starting from any arbitrary node in the tree. This operation is done for every element and the upgrade in performance reaches up to 0.25x speedup with only a small price in memory (having to store the connection between nodes).

4.8 All together

Smoothness	#Threads	K1‖2	C.Ranges	Range	#Results	Time
200	32	None	No	None	346192	143s
200	32	2.0 ‖ 1.5	No	None	342102	24s
200	32	1.8 ‖ 1.3	No	None	340813	20s
200	32	None	Yes	10000	343957	29s
200	32	None	Yes	5000	343337	18s
300	32	None	No	None	2316631	6000s
300	32	1.8 ‖ 1.3	No	None	2262253	1003s
300	32	2.0 ‖ 1.5	No	None	2278102	1379s
300	32	None	Yes	10000	2265800	228s
300	32	None	Yes	5000	2236364	124s

Splitting the execution of the algorithm and using different parameters can be a great approach to finding most results. It's possible to from an initial set of precalculated numbers for a smaller smoothness so that, when it's affordable to slow the process down, we can maximize the number of results obtained, and when it's expensive, we can focus on speed and give up some of the results. Having an initial set containing most (or all) result for a certain smoothness will then have a meaningful impact on successive executions with higher smoothness and speed-focused optimizations active. This will reduce the loss of

results whilst maintaining an acceptable speed.

By using small constant ranges (1000), and having a precalculated initial set of 599-smooth numbers, we managed to calculate a 127 bit long smooth prime number with smoothness = 887. This was obtained when running the program on a 64 core machine for around 6 hours, with a given smoothness bound of 1029.

The number found is:

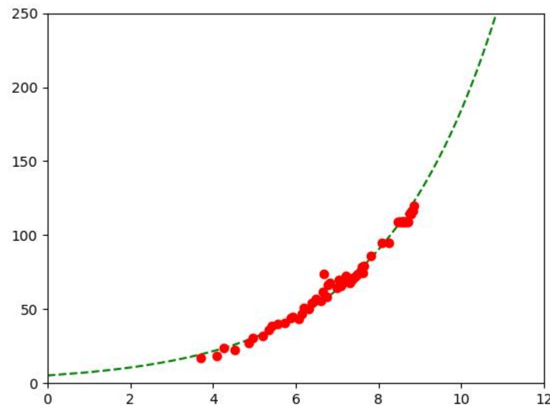
$$140925620568672748356470714008628611199$$

And its neighbours are:

$140925620568672748356470714008628611198 = 2^7 \times 3^2 \times 5^2 \times 7^4 \times 11^2 \times 13^2 \times 29^2 \times 37^2 \times 73^2 \times 83^2 \times 107^2 \times 113^2 \times 127^2$

$140925620568672748356470714008628611200 = 2^1 \times 17^1 \times 19^1 \times 23^1 \times 31^1 \times 41^1 \times 79^1 \times 103^1 \times 109^1 \times 139^1 \times 173^1 \times 191^1 \times 227^1 \times 239^1 \times 383^1 \times 409^1 \times 487^1 \times 499^1 \times 887^1$

From the obtained results we can plot the bit length of the biggest prime found at different smoothness. We can then see that there is an upward trend in bit length with an increase in smoothness. We can expect to be able to find smooth prime of 200 bits of length at around smoothness equal to 2^{10}.



As seen above, running the algorithm with all the optimizations activated only produced a 127bit length prime, more than 60 bit less than the expected number. With this experiment it's not possible to know whether a 200bit prime exists and we missed it due to the loss of results caused by the optimization or said number actually does not exist and we need to go for a higher smoothness to have chances of finding a 200bit prime.

To show how many results might have been lost during an execution we made more runs with different optimization parameters. Other than the parameters we also tried using different initial sets of neighbours. An interesting result we found was that, with the same exact parameters running the program to find 1029-smooth primes as above but instead of starting from an initial set of the 599-smooth neighbours the starting set was of 559-smooth neighbours. This simple change produced millions less results and could not find the biggest prime we now have.

5 SIKE, cortex-m4 implementation

The Cortex-M4 is an high performance processor developed by arm to address the digital signal control markets that an efficient, easy-to-use blend of control and signal processing capabilities.[16]

The Cortex-M4 processor is a low power device and in fact it is a very efficient device. It comes with the advanced ArmV7-thumb instruction set, which will prove irreplaceable for the purpose of my work. Additionally it features a Floating point processing unit (FPU), which will come in handy later thanks to the additional on chip storage that it provides.

Because the SIKE algorithm has very small key sizes and cipher text sizes compared to the other algorithms proposed at the NIST competition, small and resource constrained devices are some of the best platforms where to run SIKE. On the other side SIKE is also very computationally expensive, therefore, for it to be a good fit for such machines, it needs to be heavily optimized to be considered usable in real life scenarios.

This is why there have been interest in a cortex-m4 specific implementation of SIKE.

5.1 Initial general implementation

On the SIKE website is available a package containing the submitted implementations of the algorithm, which include a reference, generic, implementation, and optimized implementations for ARM64, AMD64, ARM32 Cortex-M4, VHDL, and the compressed version of the algorithm.

Unfortunately the Cortex-M4 optimizations available in the package only has some optimization for the addition and subtractions operations, while the rest is the same as the generic implementation.

In this thesis cortex-m4 FPU registers will be used to reduce memory loads/store usage and three thumb instructions available to the

platform are going to be used to implement Karatsuba's algorithm without suffering from the overhead of the multiple additions typical of this algorithm.

5.2 FPU registers as additional storage

The Cortex-M4 platform provides thirty-two 32bit registers (or sixteen 64 bit registers) that can be used to store any intermediate value without having to resort to use memory and therefore speeding up multiple operations.

While doing Karatsuba's multiplication, these registers are going to be extensively used to store both inputs and intermediate results avoiding to have to load the same inputs multiple time and to waste time accessing memory.

Thanks to this approach a lot less memory accesses are needed but, as we increase size of the factors in the multiplication, it becomes harder to avoid loading the same memory more than once.

The problem of this method is that it's not really scalable and it can lead to over complicated code that is very difficult to read and as difficult to write.

Nevertheless, as the size of the factors increase, it's obvious that the assembly code, regardless of using or not these FPU registers, becomes more complex.

5.3 Karatsuba's multiplication

Karatsuba's algorithm for multiplication makes use of a divide and conquer approach where the factors are expressed as their lower and higher part and their product is a combination of multiplications/additions of these lower and higher parts.

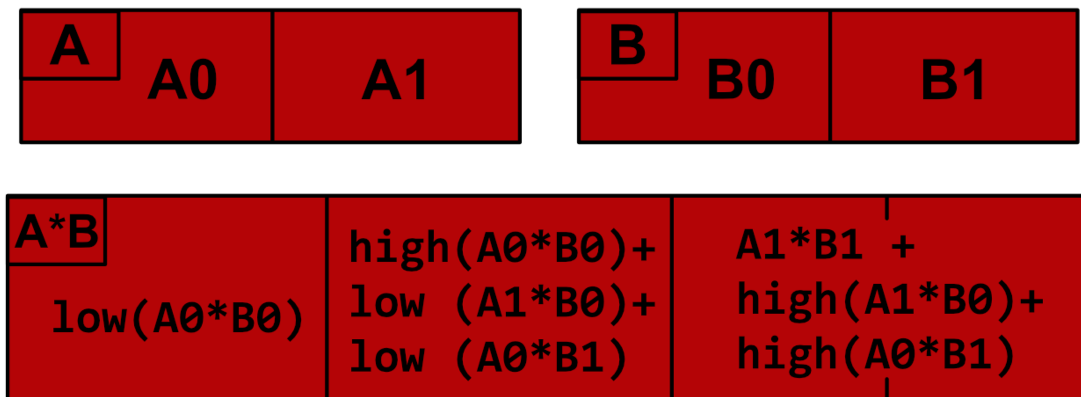
Through this approach the complexity of multiplications becomes

$O(n^{log_2 3})$ making it faster than the naive approach, that has complexity $O(n^2)$.

The only problem in this algorithm that makes it not that useful in most cases is that there is an overhead in the recursion and that the reduced number of multiplications is replaced by an increased number of additions.

In fact, the algorithm is used in the libgmp library only in a certain range of numbers.

As stated before, Karatsuba's algorithm represents the two factors by splitting them into a lower and higher part. If I have a number A, I'm going to call A0(or low(A)) and A1 (or high(A)) the lower and higher part respectively.



As we can see from the picture above, we need to do four multiplications and four additions. By going through the equations it's seen that the number of multiplications can actually be reduced to three and while this could seem to be better, because of the instructions available on the m4 platform it actually is not the case.

5.4 Karatsuba's multiplication implementation

The Cortex-M4 supports instructions sets Thumb and Thumb-2. In particular with the second ISA, we can implement all multiplications and additions of the Karatsuba's algorithm with only three instruc-

tions, furthermore these instructions can do both multiplication and addition at the same time.

Definition 5.1. A Karatsuba base case in 32bit cortex-m4 code is the case in which the factors are small enough that they can be multiplied directly by the processors without the need for further recursion (on a new split of the factors).

Karatsuba's algorithm does four multiplications (of half the size of the factors) and four additions, but can also be expressed as three multiplications and two subtractions. With the instructions available doing four multiplications and four additions results in a total of four instructions, instead, using the refined formula, we would need (need to count) instructions.

Code optimized for thumb2 instructions set.

```
1	# CLEAR R9 AND R10
2	MOV R9, #0
3	MOV R10, #0
4
5	# LOW(A) * LOW(B) STORE RES IN R7,R8
6	UMULL R7, R8, R3, R5
7	# HIGH(A) * LOW(B) + R8 STORE RES IN R8 R9
8	UMLAL R8, R9, R4, R5
9	# LOW(A) * HIGH(B) + R8 STORE RES IN R8 R10
10	UMLAL R8, R10, R3, R6
11	# HIGH(A) * HIGH(B) + R9 + R10 STORE RES IN R9 R10
12	UMAAL R9, R10, R4, R6
```

Unoptimized code.

```
1	# CLEAR R9 AND R10
2	UMULL R7, R8, R3, R5
3	UMULL R9, R10, R4, R6
4
5	ADDS R11, R3, R4
6	ADCS R0, #0, #0
7
```

```
8       ADDS R14, R5, R6
9       ADCS R1, #0, #0
10
11      UMULL R3, R4, R11, R14
12
13      SUBS R4, R4, R8
14      SBCS R4, R4, R10
15
16      # and more ...
```

As we can see, thanks to the instructions available, doing multiplications whilst doing additions and at the same time taking care of eventual carries is very simple and very efficient on Cortex-M4. Here is important to note that the cost of one multiplication (UMLAL, UMULL, UMAAL) is one cycle, which is the same as one addition or one subtraction. It becomes trivial to see that the implementation with the least amount of instructions coincides with the fastest and most efficient implementation.

Because the length of the factors is known there is no need to implement Karatsuba's recursively, in fact the implementation is going to unroll the recursion whilst trying to avoid any unnecessary calculations.

There are four different implementations needed, namely one for 434, 503, 610, and 751 bit length factors.

- 434bit → array of fourteen 32bit numbers
- 503bit → array of sixteen 32bit numbers
- 610bit → array of twenty 32bit numbers
- 751bit → array of twenty four 32bit numbers

Except for the 503bit case, all other cases flow towards uneven splittings in Karatsuba's recursion. In such cases one can choose different ways to split in two the factors and there is no performance difference from way to way. The only difference to take in consideration is the ease of development: some splittings are more confusing than other

to write down, especially in assembly.

5.5 503bit implementation in detail

The implementation for primes of 503bits is the simplest and most straight forward as a 503bit number uses sixteen 32bit registers to store and can therefore be split evenly throughout the algorithm.

First of all I'm going to make a distinction on three possible cases for multiplication:

- 1: normal multiplication $a * b$
- 2: multiplication and sum: $a * b + c$
- 3: multiplication and 2 sums: $a * b + c + d$

These three cases are exactly what we need to do one Karatsuba multiplication (A×B = C):

- First do case 1 on the low parts of A and B to obtain C0 and C1, respectively the lower part of this multiplication and the higher part of the multiplication
- Then do case 2 on the high part of A and low part of B, adding C1 obtained before overwriting C1 with the low part of the new result and storing the high part on C2
- Another time apply case 2, but this time on the low part of A and the high part of B, again adding C1. We overwrite C1 with the low part of the result and store the high part on C3.
- Finally apply case 3 on the high parts of A and B, adding C2 and C3. Store the results in C2 and C3 and the complete result C is available as C0 C1 C2 C3.

These three operations are perfectly implemented by the thumb2 instructions UMULL, UMLAL, and UMAAL.

- **UMULL**: multiply two 32bit registers and store the 64bit result in two other 32bit register

- **UMLAL**: multiply two 32bit registers, add one 64bit number(from two 32bit registers), and store the 64bit result in the two 32bit registers used for the addendum
- **UMAAL**: multiply two 32bit registers, add two 32bit registers, and store the 64bit result in the two 32bit registers used for the two addendum.

Here we can see how these three instructions are used to implement the various operations needed by Karatsuba's multiplication.

First case normal multiplication of two numbers of 32bit each

```
    # CLEAR R9 AND R10
    MOV R9, #0
    MOV R10, #0

    # LOW(A) * LOW(B) STORE RES IN R7,R8
    UMULL R7, R8, R3, R5
    # HIGH(A) * LOW(B) + R8 STORE RES IN R8 R9
    UMLAL R8, R9, R4, R5
    # LOW(A) * HIGH(B) + R8 STORE RES IN R8 R10
    UMLAL R8, R10, R3, R6
    # HIGH(A) * HIGH(B) + R9 + R10 STORE RES IN R9 R10
    UMAAL R9, R10, R4, R6
```

Second case multiplication of two numbers of 32bit each and addition of one 64bit number

```
1     # CLEAR R9 AND R10
2     MOV R9, #0
3     MOV R10, #0
4
5     # R7 and R8 contain the 64bit addendum
6
7     # LOW(A) * LOW(B) + R7 STORE RES IN R7,R8
8     UMAAL R7, R9, R3, R5    # UMLAL here makes no difference
9     # HIGH(A) * LOW(B) + R8 + R9 STORE RES IN R8 R9
10    UMAAL R8, R9, R4, R5
11    # LOW(A) * HIGH(B) + R8 STORE RES IN R8 R10
12    UMAAL R8, R10, R3, R6   # UMLAL here makes no difference
13    # HIGH(A) * HIGH(B) + R9 + R10 STORE RES IN R9 R10
14    UMAAL R9, R10, R4, R6
```

Third case multiplication of two numbers of 32bit each and addition of one two 64bit numbers

```
1     # R7 and R8 contain the first 64bit addendum
2     # R9 and R10 contain the second 64 bit addendum
3
4     # LOW(A) * LOW(B) + R7 + R9 STORE RES IN R7,R8
5     UMAAL R7, R9, R3, R5
6     # HIGH(A) * LOW(B) + R8 + R9 STORE RES IN R8 R9
7     UMLAL R8, R9, R4, R5
8     # LOW(A) * HIGH(B) + R8 + R10 STORE RES IN R8 R10
9     UMLAL R8, R10, R3, R6
10    # HIGH(A) * HIGH(B) + R9 + R10 STORE RES IN R9 R10
11    UMAAL R9, R10, R4, R6
```

As it can be seen, the first two cases require two output registers to be clean and therefore take two more cycles than the third case, which makes use of all the values stored in the output registers.

Multiplying two numbers of 512bits each requires sixteen registers

for each number. The multiplication is divides the multiplication by doing:

- low(A)×low(B) ⟶ C0 C1 (8 reg 1st case)
- high(A)×low(B) + C1 ⟶ C1 C2 (8 reg 2nd case)
- low(A)×high(B) + C1 ⟶ C1 C3 (8 reg 2nd case)
- high(A)×high(B) + C2 + C2 ⟶ C2 C3 (8 reg 3rd case)

This only shows the first split that's being done, internally the multiplication is split until there are only factors made of two 32bit registers. The schema that each of these split multiplications follow is the same as the one above.

5.6 Results

Here we can see how much these optimizations reduce the number of cycles needed to perform various operations. The Original is referring to the reference implementation with the inclusion of optimizations to subtraction and addition operations.

Implementation	p503 $\mathbb{F}_p$ mul	p610 $\mathbb{F}_p$ mul	p751 $\mathbb{F}_p$ mul
Original	14264	22018	31452
This work	1339	2184	3114

It is estimated that, with these optimizations in place, the time required to perform Key Generation, Encapsulation, and De Encapsulation should be reduced by slightly less than three times.

Additionally it's possible to shave off a few cycles by replacing the generic 32bit multiplication with an assembly function of four instructions.

Modifying this function in this way is a simple yet effective trick to improve performance without writing hundreds of new lines of code. This function is called both by the original "mp_mul" function and

by the Montgomery Reduction function "rdc_mont". Since the first one has already been optimized separately, and the performance obtained through this is, as expected, inferior to what we obtain with Karatsuba, I'm going to ignore it there. In the case of Montgomery reduction we can see an increase in speed by around 3%, that while is not very impressive, it basically comes for free.

Here we can see how it's implemented and the timings for the Montgomery Reduction.

```
    push {r3-r4}
    umull r3, r4, r0, r1
    stmia r2, {r3-r4}
    pop {r3-r4}
```

Implementation	p503 rdc_mont	p610 rdc_mont	p751 rdc_mont
Original	10954	16646	23537
This work	10623	16127	22811

These results are similar to the one shown by Hwajeong Seo, Mila Anastasova, Amir Jalali, and Reza Azarderakhsh in their paper for the submission their implementation of SIKE for cortex-m4 during the second round of the NIST competition [17]. Unfortunately I was not able to compile and compare their implementation with my own. It is unclear to me at the moment if the publicly available code includes all of the optimization in their paper and how to get it to actually work.

In their paper they use the Operand Scanning method instead of Karatsuba because of the additional overhead Karatsuba brings when implemented as shown in [18], where three multiplications are done instead of four.

As shown in this thesis, Karatsuba can be implemented using only six to four cycle, depending on whether it's necessary or not to clean up two output registers, to perform multiplication of two 64bit factors.

The results shown in their paper are slightly better than the ones I

obtained and that is most likely due to a better usage of the FPU registers, avoiding unnecessary additional memory accesses or redundant movements from FPU to Core registers and vice versa.

There is more ground to cover to optimize SIKE on cortex-m4 devices, starting from re-implementing various functions in assembly and rewriting the Montgomery Reduction to make use of the advantages of Karatsuba's multiplication in this device. Unfortunately I did not find the time to make introduce these optimizations myself.

6 Conclusion and Future Work

To conclude we have seen many improvements in the research for new and bigger smooth prime numbers, the algorithm themselves have gotten way faster than ever before and new large smooth primes are starting to surface. In this direction it's foreseeable in the near future to find what we need for B-SIDH. Unfortunately we can also see the approaching obstacles and one of them is the amount of memory required to store our constructive sets of smooth neighbours. When computing prime smooths for $z = 1029$ with different parameters than the ones showcased before we run into issues were not only the RAM was not enough anymore to store the necessary data, but it was also not possible to store all this information in the albeit small hard drive of the machine. It will become necessary to find ways to distinguish between necessary and unnecessary data and try to purge as much of that data as possible. Considering that our objective is not to create a complete set of z-smooth primes but to find a single z-smooth prime of more than 200 bits with a $z < 10000$, it becomes not only acceptable, but also necessary to start dismissing some of the results.

With regards to SIKE, the path that needs to be followed to obtain great results on micro-controllers has been already cleared up and pointed by Hwajeong Seo, MilaAnastasova, Amir Jalali, and Reza Azarderakhs in their paper[17]. There are surely more improvements to be made on this implementation by replacing the generic C code with architecture specific assembly code and using different algorithms that can take advantage of the architecture. The first example of this is the function seen above, "rdc_mont", which can be better implemented as seen in [17] and that would speed up the whole Key Generation, Encapsulation and De Encapsulation by another three times. But that is not the only place that could be better with some rewriting, there are other functions, doing either corrections, negations or digit addition and subtraction that could be made more efficient by rewriting them in arm assembly.

References

[1] Burt Kaliski. Announcement of "rsa factoring challenge", 1991. `https://groups.google.com/u/1/g/sci.crypt/c/AA7M9qWWx3w/m/EkrsR69CDqIJ`.

[2] Paul Zimmermann. Factorization of rsa-250, 2020. `https://lists.gforge.inria.fr/pipermail/cado-nfs-discuss/2020-February/001166.html`.

[3] Martin E. Hellman Whitefield Diffie. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976. `https://www.cs.utexas.edu/~shmat/courses/cs380s/dh.pdf`.

[4] Lily Chen, Dustin Moody, and Yi-Kai Liu. Post-quantum cryptography, 2016. `https://csrc.nist.gov/projects/post-quantum-cryptography`.

[5] Jean-Marc Couveignes. Hard homogeneous spaces. Cryptology ePrint Archive, Report 2006/291, 2006. `https://ia.cr/2006/291`.

[6] Alexander Rostovtsev and Anton Stolbunov. Public-key cryptosystem based on isogenies. Cryptology ePrint Archive, Report 2006/145, 2006. `https://ia.cr/2006/145`.

[7] Andrew Childs, David Jao, and Vladimir Soukharev. Constructing elliptic curve isogenies in quantum subexponential time. *Journal of Mathematical Cryptology*, 8(1):1–29, Jan 2014.

[8] Wouter Castryck, Tanja Lange, Chloe Martindale, Lorenz Panny, and Joost Renes. Csidh: An efficient post-quantum commutative group action. In *Advances in Cryptology – ASIACRYPT 2018*, volume 11274 of *Lecture Notes in Computer Science*, pages 395–427. Springer, 2018.

[9] Craig Costello. Supersingular isogeny key exchange for beginners. Cryptology ePrint Archive, Report 2019/1321, 2019. `https://ia.cr/2019/1321`.

[10] Craig Costello. B-sidh: supersingular isogeny diffie-hellman using twisted torsion. Cryptology ePrint Archive, Report 2019/1145, 2019. `https://ia.cr/2019/1145`.

[11] Jao and De Feo. Qcryptov3.2. sidh library. github, 2011. `https://github.com/microsoft/PQCrypto-SIDH/releases/tag/v3.2`.

[12] Carl Størmer. Quelques théorèmes sur l'équation de pell $x^2 - dy^2 = \pm 1$ et leurs applications. Christiania, 1897.

[13] D. H. Lehmer. On a problem of Størmer. *Illinois Journal of Mathematics*, 8(1):57 – 79, 1964.

[14] Craig Costello, Michael Meyer, and Michael Naehrig. Sieving for twin smooth integers with solutions to the prouhet-tarry-escott problem. Cryptology ePrint Archive, Report 2020/1283, 2020. `https://ia.cr/2020/1283`.

[15] J. B. Conrey, M. A. Holmstrom, and T. L. McLaughlin. Smooth neighbors. *Experimental Mathematics*, 22(2):195–202, 2013.

[16] ARM holdings. Cortex-m4. www.arm.com, 2020. `https://www.arm.com/products/silicon-ip-cpu/cortex-m/cortex-m4`.

[17] Hwajeong Seo, Mila Anastasova, Amir Jalali, and Reza Azarderakhsh. Supersingular isogeny key encapsulation (sike) round 2 on arm cortex-m4. Cryptology ePrint Archive, Report 2020/410, 2020. `https://ia.cr/2020/410`.

[18] Philipp Koppermann, Eduard Pop, Johann Heyszl, and Georg Sigl. 18 seconds to key exchange: Limitations of supersingular isogeny diffie-hellman on embedded devices. Cryptology ePrint Archive, Report 2018/932, 2018. `https://ia.cr/2018/932`.