Bachelor's Thesis Computing Science

Radboud University Nijmegen

Improving AprèsSQI's cost model for verification

Author:
George-Nicolas Nădejde
s1051528

First supervisor/assessor:
Dr. Simona Samardjiska

Daily supervisor:
MSc Krijn Reijnders

Second assessor:
Prof. Dr. Lejla Batina

August 12, 2024

Abstract

In the ongoing NIST competition for additional post-quantum signatures, **SQIsign** stands out as the only candidate that uses *isogenies* in its construction. The scheme is particularly suitable for cryptographic exchanges on small devices such as micro-controllers or key cards, as it uses ideally-sized keys according to NIST's standards. Additionally, the scheme's verification performance, critical for ensuring fast and secure communication between such devices, was recently described by **AprèsSQI**, in an effort to make the original signature more practical.

AprèsSQI is a version of SQIsign that proposes several improvements on verification, using the number of $\mathbb{F}_p$-operations as a cost metric for performance. Namely, the model consists of the amount of multiplications $\mathbf{M}$, squaring operations $\mathbf{S}$, and additions/subtractions $\mathbf{a}$. This theoretical approach approximates the practical performance of verification, independent of the device and implementation used.

In this thesis, we check the efficiency of the $\mathbb{F}_p$-operations cost model proposed by AprèsSQI by implementing their verification improvements on a version of SQIsign that optimizes the finite field arithmetic. Our results show that their model with $\mathbf{S} = 0.8\mathbf{M}$, $\mathbf{M} = 1$, and $\mathbf{a} = 0\mathbf{M}$ overpredicts the performance of verification in practice. Therefore, we propose a new cost model that reflects the practical results more accurately: $\mathbf{S} = 0.8\mathbf{M}$, $\mathbf{M} = 1$ and $\mathbf{a} = 0.23\mathbf{M}$. In particular, we show that additions and subtractions are more significant than AprèsSQI initially considered.

Moreover, we use our cost model to predict the practical performance of the improvements described by AprèsSQI.

Contents

Chapter 1

Introduction

Quantum computers will eventually change a significant part of the current technology setting. The processing power of these machines will modify our perception of current challenges across various domains, which range from computer science [12] and chemistry [3], to finance [16]. And indeed, throughout the decades, researchers consistently agreed on the inevitable arrival and impact of quantum computers [21, 12, 1].

Cryptography represents a vital component not only for securing the internet overall, but also for protecting our online privacy and personal data. Large enough quantum computers will obliterate current cryptographic standards (e.g. DH, ECDH, or RSA) and eventually, these will be rendered obsolete. In 2017, NIST initiated a call for quantum-proof signatures to prevent such scenarios, resulting in the selection of two lattice-based schemes (Dilithium and Falcon) and one hash-based (SPHINCS$^+$) for standardization [15]. To diversify their portfolio and reduce dependence on the security of lattices [15], NIST also announced within the same post, an additional call for short signatures with fast verification, preferably *"not based on structured lattices"*.

Currently, there are seven signature families still in the competition[14]. These are displayed in Table 1.1. The candidate that stands out is **SQIsign** [10], as it is the unique signature that uses *isogenies* in its construction. Defined by an ideal key size and good verification performance, SQIsign represents an interesting candidate for NIST's signature competition. Small signatures and fast verification are critical for cryptographic exchanges on small or embedded devices, micro-controllers, or even key cards [5, p. 2]. As the amount of such assets will only increase in the future, SQIsign has the potential to play a very significant role in the post-quantum world.

Initially, the performance of verification in SQIsign was not impressive. However, despite being a quite recent entry, researchers have already found ways to substantially improve verification [5, 11, 9], making it much more practical. The authors of AprèsSQI [5] recently described several improve-

Type	Number of Signatures
Code based	6
Isogeny-based	1
Lattices	7
MPC	7
Multivariate	10
Symmetric-based	4
Other Signatures	5

Table 1.1: Candidates of Round 1 Additional Signatures

ments by adjusting the signing process and by employing various techniques from the literature. Accompanying the paper, the researchers also provided a proof-of-concept implementation of these optimizations using Sage and Python. Benchmarking was primarily performed on this codebase and on SQIsign's official NIST implementation [2], using the number of $\mathbb{F}_p$-operations as a cost metric for performance.

1.1 Contributions

In this thesis, we intend to extend their work by providing an improved $\mathbb{F}_p$-operations cost model of SQIsign's verification, that reflects its practical performance more accurately. We achieve this by implementing some of the proposed improvements of AprèsSQI in a version of SQIsign that optimizes the finite field arithmetic [9], and measuring their performance in clock cycles. Then, we compare our results with AprèsSQI's theoretical measurements in terms of percentual speedups and derive a new $\mathbb{F}_p$-operations cost model that reflects these practical speedups more precisely. Finally, using our new model, we provide predictions on the practical performance of all SQIsign variants proposed by AprèsSQI and analyze the new speedups we obtain[1].

1.2 Related Work

SQIsign is a post-quantum cryptographic scheme that was brought out to the public by De Feo, Kohel, Leroux, Petit, and Wesolowski [10] in 2020. From that moment on, SQIsign established itself in the post-quantum world by promoting very small signature and public key sizes compared to its competitors and fast verification, key features for cryptography of small

[1] In this thesis, we do not consider higher-dimensional SQIsign.

devices. However, SQIsign does pack quite significant setbacks as well: the signing procedure is complex and difficult to understand and perhaps most importantly, it is very slow compared to other PQC schemes. In 2022, De Feo, Leroux, Longa, and Wesolowski [9] improved the overall performance of the scheme and published an implementation with efficient finite field arithmetic. The codebase written in C is publicly available[2] and is suited for precise benchmarking, due to the low-level optimizations. Therefore, we will use it as a base for our implementations. Subsequently, Lin, Wang, Xu, and Zhao [11] introduced valuable improvements for every procedure, but more particularly, their verification optimizations led to an impressive 18.94% speedup in performance. We will often refer to their work throughout our thesis.

Dartois et al recently published an improved version titled SQISignHD [8], which further simplifies the original SQIsign and enhances signing performance. However, these improvements negatively influence the verification procedure which, as the authors acknowledged, was not their main research focus.

Finally, Corte-Real Santos, Eriksen, Meyer, and Reijnders [5] further improved the speed of the verification procedure and published a proof-of-concept repository[3] written in Python and Sage. However, the authors measured the performance based on the $\mathbb{F}_p$-operations standard model (i.e. $\mathbf{S = 0.8 \cdot M}$, that is, a square operation is considered 20% less expensive than a multiplication). Therefore, more precise measurements are required to confirm the effectiveness of their improvements. In this thesis, we will implement some of these optimizations in the repository that accompanies [9] and benchmark the performance of the verification procedure using clock cycles.

1.3 Organization of the thesis

The thesis is organized as follows. Section 2 presents the necessary background knowledge and a high-level description of SQIsign. For ease of reading, we also included a list of notations used throughout the thesis. Section 3 illustrates the improvements in a three-step manner: theory, implementation, and performance, alongside our performance predictions. Finally, an analysis of the results is shown in section 4.

1.4 Implementation availability

The code including the optimizations described in this thesis is publicly available. Additionally, we also make available a Python implementation

[2] `https://github.com/SQISign/sqisign-ec23`
[3] `https://github.com/TheSICQ/ApresSQI`

to be used as reference throughout the thesis. The code includes finite field (extension) arithmetic, x-only arithmetic on Montgomery curves, and isogeny chain computations. These can be accessed using the following links:

`https://github.com/georgenadejde/SQIsignOpt`
`https://github.com/georgenadejde/Finite-Field-Arithmetic-`

List of Notation

The following list describes the notations used in this thesis.

$\mathbb{F}_p$	finite field with p elements	9
$\mathbb{F}_{p^2}$	finite field extension	10
$-P$	inverse of a point P on an elliptic curve	10
$P + Q$	addition of points on an elliptic curve	10
$\text{char}(\mathbb{F}_p)$	characteristic of a finite field	10
$\mathcal{E}$	denotes an elliptic curve	10
$\mathcal{O}$	the point at infinity	10
$\#\mathcal{E}(\mathbb{F}_p)$	number of points on the elliptic curve $\mathcal{E}$, defined on $\mathbb{F}_p$	11
$\mathbb{A}^2_K$	the affine plane with elements in K	11
$\mathbb{P}^2$	the projective plane	11
$[k]P$	point multiplication by k on an elliptic curve	12
$\mathcal{E}(\mathbb{F}_p)$	set of rational points on curve $\mathcal{E}$ with coordinates in $\mathbb{F}_p$	13
$\mathcal{E}_A$	elliptic curve in Montgomery form	12
$[m]$	multiplication-by-m map	13
$\text{N}(x)$	the norm of $x \in \mathbb{F}_{p^2}$	13
$\text{j}(\mathcal{E})$	the j-invariant of an elliptic curve	13
$\left(\frac{x}{p}\right)$	Legendre's symbol of $x \in \mathbb{F}_p$	13
$\deg(\varphi)$	degree of an isogeny	14

$\ker(\varphi)$	kernel of an isogeny	14
$\mathbb{Z}_m$	cyclic group $\mathbb{Z}/m\mathbb{Z}$	14
$\mathcal{E}/G$	codomain curve of an isogeny with kernel G	14
$\mathcal{E}[m]$	m-torsion subgroup of an elliptic curve	14
$A_{\mathcal{E}}$	A parameter on an elliptic curve $\mathcal{E}$ in Montgomery form	15
$\text{ord}(P)$	order of a point on an elliptic curve	16
$\text{bits}(s)$	Number of bits of s $\in \mathbb{N}$.	25
$x(P)$	x-coordinate of point P	35

Chapter 2

Preliminaries

2.1 Finite Fields

The reader should already be familiar with common algebraic structures such as abelian groups or rings. Therefore, in the following paragraphs, we will assume the basic properties of such constructions.

A **finite field** is simply a special type of commutative ring. More precisely, a finite set that forms an abelian group with an additive and a multiplicative operation, respectively. Therefore, each element has unique additive and multiplicative inverses, except for the zero element (0) for the latter.

2.1.1 Finite Field $\mathbb{F}_p$

However, we always set the prime of the field to be $q = p^e$, where p is a large prime.

Further, we define the following operations on $\mathbb{F}_p$, corresponding to the names of the functions we used in our code:

- `fp_add(x,y)` $= (x + y) \mod p$, where $x, y \in \mathbb{F}_p$.
- `fp_sub(x,y)` $= (x + (-y)) \mod p$, where $x, y \in \mathbb{F}_p$ *s.t.* $y + (-y) \equiv 0 \pmod p$.
- `fp_mul(x,y)` $= (x * y) \mod p$, where $x, y \in \mathbb{F}_p$.
- `fp_div(x,y)` $= (x * y^{-1}) \mod p$, where $x, y \in \mathbb{F}_p$ *s.t.* $y \cdot y^{-1} \equiv 1 \pmod p$.

2.1.2 Finite field extension $\mathbb{F}_{p^2}$

In our research, we use the finite field extension $\mathbb{F}_{p^2} = \mathbb{F}_p(i)$, with $i^2 = -1$ and $p \equiv 3 \mod 4$. Thus, the elements of $\mathbb{F}_{p^2}$ will be of the form $a + bi$, with

$a, b \in \mathbb{F}_p$. This allows for its specific operations to be defined in terms of the ones above.

Let $c_1 = a + bi$ and $c_2 = c + di$, where $a, b, c, d \in \mathbb{F}_p$. Then:

- $c_1 + c_2 = (a + c) + (b + d)i$
- $c_1 - c_2 = (a - c) + (b - d)i$
- $c_1 \cdot c_2 = (ac - bd) + [(a + b)(c + d) - (ac + bd)]i$
- $c_1/c_2 = \frac{(a+bi)\cdot(c-di)}{c^2+d^2}$

Notice that we can do an `fp2_mul` by using only three `fp_mul`'s.

Because the implementation of the division operation is not as straightforward compared to the other operations, we included a pseudocode version in Appendix A for reference.

Definition. The **characteristic** of a field represents the smallest number of times one needs to add the multiplicative identity to get the additive identity.

Particularly, for the aforementioned fields, the following holds:

$$\text{char}(\mathbb{F}_p) = \text{char}(\mathbb{F}_{p^2}) = p$$

2.2 Elliptic Curves

Elliptic curves are one of the most exciting aspects of modern cryptography. The computation speed, small key size, and powerful mathematical properties are the main reasons why they are the preferred choice for current implementations of cryptographic schemes.

Formally, an **elliptic curve** is a pair $(\mathcal{E}, \mathcal{O})$, where $\mathcal{E}$ represents a non-singular curve and $\mathcal{O}$ a point called the *point at infinity* [19]. A non-singular curve assures us that every point on the curve has a well-defined tangent line [20, p. 21], a crucial aspect when performing arithmetic operations on elliptic curves. In the same context, $\mathcal{O}$ acts as the identity element, being tightly linked with Bézout's theorem:

Theorem 2.2.1 (**Bézout's theorem**)**.** *Let $\mathcal{E}$ be an elliptic curve and $\mathcal{L}$ a line $y = ax + b$. Then $\mathcal{E}$ intersects $\mathcal{L}$ in exactly three points (counting multiplicities).*

If we add three such points found on $\mathcal{E} \cap \mathcal{L}$, we end up at the point at infinity:

$$P + Q + R = \mathcal{O}.$$

From this, we can derive the addition law of two elliptic curve points:

$$P + Q = -R,$$

where $-R$ is obtained by inverting the y-coordinate. This law makes $\mathcal{E}$ an *abelian group* with the identity element $\mathcal{O}$. For the full proof, see [19, p. 51].

In this thesis, we only use elliptic curves with a special property: **supersingularity**, as defined in [17, p. 5].

Definition. An elliptic curve $\mathcal{E}$ defined over a finite field $\mathbb{F}_q$ with characteristic p is called *supersingular* if and only if:

$$p \mid \#\mathcal{E}(\mathbb{F}_q) - q - 1$$

We will assume from now on that every elliptic curve mentioned is supersingular[1].

2.2.1 Affine vs projective coordinates

In the two-dimensional plane, one typically represents a point using *affine coordinates*, e.g. $P = (x, y)$, where $x, y \in \mathbb{R}$. More generally, we denote such a plane as the *affine K-plane*, where K represents a finite field. It is defined as follows:

$$\mathbb{A}^2_K = \{(x, y) \mid x, y \in K\}$$

However, it is often more convenient to work in the *projective plane*:

$$\mathbb{P}^2 = \{(X : Y : Z) \mid X, Y, Z \in K\} \setminus (0 : 0 : 0) \mod \sim,$$

with the equivalence relation $\sim$ defined by:

$$(a : b : c) \sim (a' : b' : c') \iff \exists \lambda \in K : (a : b : c) = (\lambda a' : \lambda b' : \lambda c')$$

Hence, as opposed to the affine space, the projective space uses triples instead of tuples to identify points.

Remark 2.2.1. The affine space defines unique points on an elliptic curve. However, triples in the projective space may correspond to the same affine point.

Converting an affine point to projective coordinates is done as follows:

$$(x, y) \mapsto (x : y : 1),$$

Conversely,

$$(X : Y : Z) \mapsto (X/Z, Y/Z),$$

[1] Generally, there is some confusion around the terms *singular* and *supersingularity*. Although they might seem related, they are in fact not. In isogeny-based cryptography, a supersingular elliptic curve is, by definition, an elliptic curve, which again, by definition, is non-singular [19, p. 145]

whenever $Z \neq 0$.

Cryptographers tend to prefer projective coordinates over affine due to the ability to delay divisions until the very end of the procedure. That is, instead of performing a division immediately, we defer it by multiplying it into the denominator Z. To retrieve the resulting x-coordinate, we simply divide X/Z once, at the end of the execution. Since divisions are expensive and occur regularly in cryptography, projective coordinates help minimize the amount of division we do and thus significantly improve the execution time of the procedures.

2.2.2 Montgomery Curves

The elliptic curves that we will be focusing on are *Montgomery curves*. A curve in Montgomery form has the following equation [2]:

$$\mathcal{E}_A : y^2 = x^3 + Ax^2 + x,$$

where A is a parameter in $\mathbb{F}_{p^2}$.

What sets it apart from other curves is its scalar multiplication algorithm called the **Montgomery Ladder** (referred to as `xMUL` from now on); that is, computing

$$\underbrace{P + P + \cdots + P}_{k \text{ times}} = [k]P$$

using a technique similar to the more common *double and add* algorithm, except it *only* uses the *x-coordinate*, thus completely discarding the y-coordinate. A simple idea that facilitates very fast curve arithmetic and one of the main reasons why Montgomery curves are preferred in ECC or isogeny-based cryptography [6, p. 4].

When implementing `xMUL`, one typically uses two additional auxiliary functions: `xDBL`: $\mathrm{x}(P) \mapsto [2]\,\mathrm{x}(P)$ and `xADD`: $(\mathrm{x}(P), \mathrm{x}(Q), \mathrm{x}(P-Q)) \mapsto \mathrm{x}(P+Q)$. We included a pseudocode implementation of `xMUL` in Appendix A.

2.2.3 Finding a random rational point

Not every x-value in $\mathbb{F}_{p^2}$ we can come up with has a corresponding y-coordinate on a Montgomery curve $\mathcal{E}_A$, unless the value $x^3 + Ax^2 + x \in \mathbb{F}_{p^2}$ for our chosen x is indeed a square. However, checking that an element $e \in \mathbb{F}_{p^2}$ is a square is not as straightforward as one may imagine.

Definition. The set of points on an elliptic curve $\mathcal{E}_A$ defined over $\mathbb{F}_{p^2}$ is called the set of **rational points** and is denoted by $\mathcal{E}(\mathbb{F}_{p^2})$.

[2] Typically, a Montgomery curve is determined by two parameters: A and B, the latter being the coefficient of y^2. However, since the arithmetic and the j-invariant (we will introduce this notion later) formulas may only depend on A, we set $B = 1$ and ignore it when defining a Montgomery curve.

In our thesis, we often need to generate a random rational point. To achieve this, we apply the following procedure:

1. Generate a random $x \in \mathbb{F}_{p^2}$.
2. Compute the norm of $x \overset{\text{not}}{=} \mathrm{N}(x) \in \mathbb{F}_p$.
3. Check $\left(\frac{\mathrm{N}(x)}{p}\right) = 1$.

For details, see Appendix A.

Theorem 2.2.2. *Let $\mathcal{E}_1$ and $\mathcal{E}_2$ be two elliptic curves. Then $\mathcal{E}_1$ is isomorphic to $\mathcal{E}_2$ if and only if*

$$\mathrm{j}(\mathcal{E}_1) = \mathrm{j}(\mathcal{E}_2),$$

where $\mathrm{j}(\mathcal{E})$ denotes the *j-invariant* of an elliptic curve $\mathcal{E}$. A j-invariant simply acts as a representative of the isomorphism class of an elliptic curve, and not as a unique identifier.

The j-invariant of a Montgomery curve can be computed using only the A parameter of the curve (for the formula on Montgomery curves, see A.1). In isogeny-based cryptography specifically, one important use of the j-invariant is to check if two elliptic curves are isomorphic.

2.3 Isogeny-based Cryptography

Let $\mathcal{E}$ and $\mathcal{E}'$ be two elliptic curves. A non-constant morphism $\Phi : \mathcal{E} \to \mathcal{E}'$ which satisfies $\Phi(\mathcal{O}) = \mathcal{O}'$, is called an **isogeny**. Since an elliptic curve is defined as an abelian group, it follows that an isogeny is also a *group homomorphism* [4, p. 19]. Namely, given $P, Q \in \mathcal{E}$, the following relation holds:

$$\Phi(P + Q) = \Phi(P) + \Phi(Q)$$

For completeness, we will also mention that an isogeny can either be *separable* or *ordinary*. However, we will only be focusing on the former for this thesis.

Some well-known isogenies are the multiplication-by-m maps: $[m] : \mathcal{E} \to \mathcal{E}$, which compute $P \mapsto [m]P$, for $P \in \mathcal{E}$.

Similar to isomorphisms, an isogeny $\varphi : \mathcal{E}_1 \to \mathcal{E}_2$ of degree n also has a unique inverse called a *dual isogeny*, defined as $\varphi' : \mathcal{E}_2 \to \mathcal{E}_1$, and with the following composition properties:

$$\varphi \circ \varphi' = [n]_{\mathcal{E}_1} \qquad \varphi' \circ \varphi = [n]_{\mathcal{E}_2}{}^{3}$$

[3] $[n]_{\mathcal{E}_p}$ denotes the multiplication-by-n map on $\mathcal{E}_p$.

Example. For a standard Montgomery curve $\mathcal{E}_A$, the multiplication-by-2 map [2] for the x-coordinate is defined as follows [6, p. 4]:

$$x \mapsto \frac{(x^2-1)^2}{4x(x^2+ax+1)},$$

where a represents the curve's parameter in affine coordinates.

The points of the form $(x, 0)$ whose x-coordinates make the denominator vanish, together with $\mathcal{O}$, form the *kernel* of the map. Moreover, these points have order 2 on $\mathcal{E}_A$ [6, p. 4]. Later, we will give an interesting property that ties the kernel to a particular group structure.

2.3.1 The kernel

Definition. Let $\Phi : \mathcal{E} \to \mathcal{E}'$ be an isogeny. Then we can define the **kernel** of Φ as follows:

$$\ker(\Phi) = \{P \in \mathcal{E} \mid \Phi(P) = \mathcal{O}\}$$

An important property of a separable isogeny $\mathcal{E}$ is that it is in one-to-one correspondence with finite subgroups of $\mathcal{E}$. More precisely, for any subgroup $\mathrm{G} \subseteq \mathcal{E}(\mathbb{F}_{p^2})$, there exists a unique isogeny $\Phi : \mathcal{E} \to \mathcal{E}'$ (up to post-composition with an isomorphism), such that

$$\ker(\Phi) = G \qquad \text{and} \qquad \deg(\Phi) = \#\ker(\Phi)$$

Since the points in the kernel get mapped to infinity on $\mathcal{E}'$, we often denote $\mathcal{E}'$ as $\mathcal{E}/G$. Therefore, one can say that the kernel defines separable isogenies *uniquely* [17].

2.3.2 m-torsion points

Definition. The **m-torsion** represents the set of points sent to $\mathcal{O}$ under the multiplication-by-m map [6, p. 6]. Formally:

$$\mathcal{E}[m] = \ker([m] : \mathcal{E} \to \mathcal{E})$$

Coming back to multiplication-by-m maps, it has been shown that the kernel of these maps follows a quite specific pattern.

Let us return to the doubling map presented earlier. As noted in [6, p. 5], the following relation holds:

$$\mathcal{E}([2]) \cong \mathbb{Z}_2 \times \mathbb{Z}_2,$$

which translates to having the *2-torsion* be precisely *three cyclic subgroups of order 2*. The author further generalizes the result for supersingular curves, showing that the pattern holds for any multiplication-by-m map, with m not a power of p:

$$\mathcal{E}([m]) \cong \mathbb{Z}_m \times \mathbb{Z}_m$$

2.3.3 2-isogenies

Finding an isogeny may seem a difficult task at first. However, *Vélu's* formulas [22] make the process quite straightforward. Without these, it would be quite challenging to implement isogeny-based schemes reliably [18]. They take as input an elliptic curve $\mathcal{E}$ and its kernel K and produce the image curve $\mathcal{E}/K$ and the isogeny $\phi : \mathcal{E} \to \mathcal{E}/K$, such that $\ker(\phi) = K$. A relevant fact to mention here is that these formulas are simply rational functions with their degree equal to the size of K. Consequently, their implementation is explicit and simple and takes linear time in the size of the kernel.

Example. Let us choose the kernel G $= \{\mathcal{O}, (\alpha, 0)\}$ for an arbitrary Montgomery curve $\mathcal{E}_A$. G represents a cyclic subgroup of order 2. Using Vélu's formulas, we obtain the following 2-isogeny that sends a point from $\mathcal{E}_A$ to $\mathcal{E}/G$ [6, p. 6]:

$$x \mapsto \frac{x(\alpha x - 1)}{x - \alpha}$$

where α represents the x coordinate of a point with order 2 on $\mathcal{E}_A$, that has the affine coordinates $(\alpha, 0)$. Further, we can use the following formula to get the A parameter of the codomain's curve:

$$A_{\mathcal{E}/G} = 2(1 - 2\alpha^2)$$

One can compute 2-isogenies on any given Montgomery curve using only these two formulas.

In practice, we encounter isogenies of degree 2^e, for $e > 200$. Since computing them directly would negatively influence the computation time due to the large size of the kernel [6, p. 9], we want to separate them into a *chain* of smaller degree isogenies. More precisely, isogenies of degree 2.

Suppose we want to compute a 2^n-isogeny $\Phi : \mathcal{E}_1 \to \mathcal{E}_n$. As explained above, we compute it using a chain of 2-isogenies. Let the first 2-isogeny be $\varphi_1 : \mathcal{E}_1 \to \mathcal{E}_2$. Such isogenies have a specific kernel that is equal to the set of points $\{\mathcal{O}, K_2\}$, where $\mathrm{x}(K_2) = \alpha$, the same value we had specified above. Therefore, K_2 must have order 2 on $\mathcal{E}_1$.

To ensure the existence of such a point for each isogeny in the chain, we first find a point P with order 2^e on $\mathcal{E}_1$. By multiplying it with the scalar 2^{e-1}, we obtain the point K_2 of order 2 and thus compute the kernel of φ_1:

$$\ker(\varphi_1) = \{\mathcal{O}, K_2\}.$$

We then input $\mathcal{E}_1$ and $\ker(\varphi_1)$ into *Vélu's* formulas, which yield the map and $\mathcal{E}_2$. The next step is to push P through φ_1, resulting in a point that resides on $\mathcal{E}_2$:

$$P_2 := \varphi_1(P)$$

Proposition. Let $\varphi : \mathcal{E}_1 \to \mathcal{E}_2$ be a 2-isogeny and $K \in \mathcal{E}_1$ with $\text{ord}(K) = 2^n$, such that K lies above a non-trivial element in the kernel [4]. Then

$$\text{ord}(\varphi(K)) = 2^{n-1}$$

Proof. Let

$$K_2 = [2^{n-1}]K. \tag{2.1}$$

We can then write

$$\ker(\varphi) = \{\mathcal{O}, K_2\} \tag{2.2}$$

By definition (see Section 2.3), an isogeny is a group homomorphism. Consequently,

$$[2^{n-1}]\varphi(K) = \varphi([2^{n-1}]K).$$

Using 2.1 in the equation above yields

$$[2^{n-1}]\varphi(K) = \varphi(K_2),$$

From 2.2, it results that

$$[2^{n-1}]\varphi(K) = \mathcal{O}. \tag{2.3}$$

Since K has order 2^n on $\mathcal{E}_1$, it follows that any point $[2^e]K$, for $e < n-1$, does not lie in the kernel. Consequently, the isogeny φ does not map these points to $\mathcal{O}$. More particularly, we know that

$$[2^{n-2}]\varphi(K) \neq \mathcal{O} \tag{2.4}$$

Using 2.3 and 2.4, we conclude that

$$\text{ord}(\varphi(K)) = 2^{n-1}.$$

■

Using the proposition above, we find that

$$\text{ord}(P_2) = \text{ord}(P)/\deg(\varphi_1) = 2^{n-1}.$$

We then use P_2 to calculate the next K_2 of order 2 on $\mathcal{E}_2$ and proceed similarly as with φ_1. Therefore, the proved proposition ensures we can always find the kernel of an isogeny for all 2-isogenies in the chain. More particularly, it follows that we need to compute precisely n such 2-isogenies to yield our desired isogeny Φ of degree 2^n.

The entire procedure described above can be visualized in an **isogeny graph**. The elliptic curves are represented as nodes, while the 2-isogenies are represented as edges. Consequently, computing a chain of isogenies is equivalent to taking steps in the isogeny graph.

[4] One says a point P *lies above* a point K if $\exists s \in \mathbb{N}$, such that $[s]P = K$.

2.3.4 SQIsign

SQIsign is one of the most compact PQC schemes in the ongoing NIST competition for additional post-quantum signatures. Despite a fast verification and small key sizes, signing with SQIsign is quite complex and slow. We present a simple outline of how SQIsign works below. It should be noted that what follows is an informal description of the procedures, with many details omitted in order to maintain a focus on verification.

Setup: Find a prime $p \equiv 3 \mod 4$ and $\mathcal{E}_0(\mathrm{F}_{p^2})$, a supersingular elliptic curve.

Key generation: Compute an isogeny $\varphi : \mathcal{E}_0 \to \mathcal{E}_A$. Then take φ as the *secret key* and $\mathcal{E}_A$ as the *public key*.

Commitment: The prover generates a random commitment, that is, an isogeny $\varphi_{comm} : \mathcal{E}_0 \to \mathcal{E}_1$ and then sends $\mathcal{E}_1$ to the verifier.

Challenge: The verifier generates a random challenge, that is, an isogeny $\varphi_{chal} : \mathcal{E}_1 \to \mathcal{E}_2$ and sends φ_{chal} to the prover.

Response: The prover uses φ_{comm} and φ_{chal} to compute an isogeny $\sigma : \mathcal{E}_A \to \mathcal{E}_2$ and then sends σ to the verifier.

Verification: The verifier checks if σ is indeed an isogeny from $\mathcal{E}_A$ to $\mathcal{E}_2$.

The slowdown in verification comes from computing an isogeny σ of a large degree $2^e = 2^{1000}$. For the prime we are using, we can only have rational points of order 2^f with $f = 75$. Therefore, we can only split σ into blocks of 2^f-isogenies. Ideally, we would like the number of blocks to be as small as possible, because computing an isogeny requires costly operations, as we will see below.

Generally, as described in [5], verification reduces to the following procedures:

`FindBasis`: Compute a deterministic basis $\langle P, Q\rangle$ of $E[2^f]$.

`FindKernel`: Using the basis $\langle P, Q\rangle$ and $s \in \mathbb{Z}/2^f\mathbb{Z}$ (given in the signature), compute the kernel generator $K = P + [s]Q$.

`FindIsogeny`: Using K, compute the isogeny $\phi : \mathcal{E} \to \mathcal{E}/K$ and push Q through the isogeny, that is, compute $\phi(Q)$.

We apply these procedures sequentially, for each isogeny in the chain. Namely, in our case, we split σ into isogenies of degree 2^f:

$$E_1 \xrightarrow{\varphi_1} E_2 \xrightarrow{\varphi_2} E_3 \xrightarrow{\varphi_3} \cdots \xrightarrow{\varphi_{n-1}} E_n,$$

such that:

$$\sigma = \varphi_{n-1} \circ \varphi_{n-2} \circ \cdots \circ \varphi_2 \circ \varphi_1.$$

Naturally, for each isogeny φ_i, $i \in \{1, 2, \ldots n-1\}$, we find a basis $\langle P_i, Q_i \rangle$ using `FindBasis`. We then compute their corresponding kernels via `FindKernel` and finally, get the isogeny-maps by `FindIsogeny`.

As stated in [5], the twin process of finding a basis and kernel is expensive, considering the length of the isogeny chain and the limitations regarding the number of blocks we can divide it into. Therefore, improving the performance of these procedures would significantly influence the efficiency of SQIsign's verification.

Chapter 3

Research

3.1 Research context

The thesis intends to analyze how verification improvements presented in AprèsSQI [5] hold up in a practical context. We achieve this by implementing the following optimizations:

1. Non-square x-coordinates
2. `xMUL` with adds

The first one is by far the most impactful out of the ones described in the thesis. The second fits nicely since it is applied on small coordinates which the first improvement ensures. Further, we compare their performance in terms of AprèsSQI's [5] more theoretical cost model based on the amount of $\mathbb{F}_p$-operations, with the more practical-oriented and precise cost model using clock cycles.

3.2 Technical details

For our implementation, we used the repository that accompanies [9]. We will denote the original implementation by `SQIsignOpt`. The code was especially suitable due to the low-level optimizations of the $\mathbb{F}_p$ arithmetic. This allowed us to benchmark the performance of verification more precisely in a practical context. However, we were also limited in the primes we could use, as the codebase is restricted to only two primes: p_{3923} and p_{6983}. We chose to implement the optimizations on the former. Our code is available at `https://github.com/georgenadejde/SQIsignOpt`. We ran benchmarks on a 1.8GHz Intel Core i7-8550U processor with Turbo Boost disabled. All of our tests were run using the existing benchmark tool provided by the repository. By default, the tool generates 5 random keys, signs 5 random messages under each key, and runs verification 10 times. This yields 250

runs in total. Our statistics are based on the average of 10 such tests, thus equivalent to 2500 verification samples. We also make available the scripts we had used for testing.

For reference, we provide a Python implementation that highlights the $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$ arithmetic, Montgomery curves operations, and isogeny computations. The code is available at `https://github.com/georgenadejde/Finite-Field-Arithmetic-`.

3.3 Non-square x-coordinates

In this section, we present an important optimization for computing the verification isogeny. Particularly, it enhances the performance of `FindBasis`. Our goal is to compare the performance of our implementation with the original described by [11] and with the measurements in $\mathbb{F}_p$-operations illustrated by [5].

3.3.1 Finding a deterministic basis for $E[2^f]$

Originally, finding P and Q such that they form a deterministic basis for $E[2^f]$ reduces to the following steps:

1. **Find two points P and Q**
   - Sample x-coordinates of the form $x_k = 1 + k \cdot i$, where $k \in \mathbb{F}_p$ and $k \geq 1$.
   - Check if the corresponding point lies on the curve.
2. **Check** $\text{ord}(P) = \text{ord}(Q) = 2^f$
   - Compute $P \leftarrow \left[\frac{p+1}{2^f}\right] P$ and $Q \leftarrow \left[\frac{p+1}{2^f}\right] Q$.
   - Verify that $[2^{f-1}]P \neq \mathcal{O}$ and $[2^{f-1}]Q \neq \mathcal{O}$.
3. **Check linear independence**
   - Verify that $[2^{f-1}]P \neq [2^{f-1}]Q$.
   - Otherwise: discard Q and re-sample.

As one may observe, this procedure is quite complex and computationally heavy. Fortunately, Lin, Wang, Xu, and Zhao [11, § 5.1, Th. 3] described an optimization that enhances the performance of verification by almost 19%. In [5], the improvement is described more generally and is benchmarked using the $\mathbb{F}_p$-operations cost model.

More precisely, [11] showed that in the case that $[2^{f-1}]P = (0, 0)$, we can speed up the generation of the second torsion point Q by sampling with

non-square x-coordinates. AprèsSQI's [5] generalizes this result, explaining how, given a point P of order 2^f above a 2-torsion point (thus not necessarily above $(0,0)$), we can determine the second point Q of order 2^f. Moreover, to ensure linear independence check, we find a point Q with 2^f-torsion that is *not* above $(0,0)$. By using this improvement, we can safely skip half of step 2 and the entire step 3, which enhances the performance significantly.

For more details, see the proofs in [5, [§ 5.1, Th. 2] and [11, [§ 5.1, Th. 3].

Consequently, to generate the second torsion point Q, we only have to do the following:

1. Sample non-square $x_Q \in \mathbb{F}_{p^2}$.
2. Check that $x_Q^3 + Ax_Q^2 + x_Q$ is a square.
3. `xMUL` Q by the cofactor.

Moreover, since non-squareness is only dependent on the prime p and not on an elliptic curve [5], we can pre-generate a list of these non-square coordinates for every prime.

3.3.2 Implementation

In this section, we present details about the implementation of the above optimization, which we denote as `SQIsignOpt(LXWZ)`. In terms of modifications, we only adjusted the function `deterministic_second_point()` from the original `SQIsignOpt`.

First, we proceed to generate a list of non-square coordinates using the sampling function presented in Algorithm 1. The output of this function allows us to then instantiate an array with possible candidates Q's. We decided to include 15 such points in the array. Since our isogeny has degree 2^{1000} and our $f = 75$ (i.e. the number of blocks), it means we need to compute 13 isogenies of degree 2. Unless we are extremely unlucky, the 15 candidates from the array should accommodate for finding a suitable point that reside on each curve in the chain. Our testing confirms this.

For sampling, although the original function uses slightly different sequences, we decided to follow the original sequence described by NIST [2] to generate the x_Q's:

$$x_Q = 1 + k \cdot i,$$

choosing k as small as possible. Since points are represented in projective coordinates, we set $X_Q = 1 + k \cdot i$ and $Z = 1$.

Algorithm 1 Generate sample point

Input: $k \in \mathbb{N}$
Output: A candidate point Q with non-square x-coordinate.

1: $Q \leftarrow$ `samplePoint`(k) ▷ $x_Q = 1 + k \cdot i$
2: **if not** `fp2_issquare`(x_Q) **then** ▷ check x_Q is non-square
3: `arrayAdd(Q)` ▷ add point to the array
4: **end if**
5: `update(k)` ▷ $k \leftarrow k + 1$

We add the sampled values that turned out to have a non-square x-coordiante to an array lwxz, which is initialized in `deterministic_second_point()`. After checking if the point resides on the curve, we multiply it by the cofactor $\frac{p+1}{2^f}$. This is because a random point always has an order that divides $p+1$ and by sampling it with non-square x-coordinate, we obtain a point of order $2^f \cdot g$, for some unknown g that divides the cofactor. Therefore, scalar multiplying by the cofactor ensures that our point has order 2^f. The entire procedure is presented in Algorithm 2.

Algorithm 2 `deterministic_second_point`

Input: cofactor $\in \mathbb{F}_p$, $A \in \mathbb{F}_{p^2}$, lwxz $= [Q_1, Q_2, \dots]$
Output: $Q \in \mathbb{F}_{p^2}$. such that $Q \in E[2^f]$.

1: **for** Q_i **in** lwxz **do**
2: **if** `is_on_curve`(Q_i, A) **then**
3: $Q \leftarrow$ `xMUL`$(\text{cofactor}, Q_i, A)$ ▷ $[\text{cofactor}]Q_i$
4: **return** Q
5: **end if**
6: **end for**

3.3.3 Performance

In this section, we present benchmarks comparing `SQIsignOpt` and `SQIsignOpt(LWXZ)`. The results are summarized in Table 3.1.

Table 3.1: Timings comparing the original implementation `SQIsignOpt` and `SQIsignOpt(LWXZ)`. Results are expressed in millions of clock cycles.

Prime	Implementation	Cycles (mil)
p_{3923}	`SQIsignOpt`	32.963
	`SQIsignOpt(LWXZ)`	28.273
Speedup		**14.22%**

We notice a significant difference in performance between the two implementations, yielding a speedup of 14.22%. If we compare it with [11]'s original results on the same prime (and implemented in the same codebase), we notice that the speedup we obtained is slightly lower (18.94%). We believe the difference comes from the additional techniques employed by the authors to further improve their implementation. These were adapted from the literature and not mentioned explicitly in their paper [11, p. 26].

Table 3.2 presents the results obtained by AprèsSQI [5] when implementing the same improvements, measured in number of $\mathbb{F}_p$-operations. The authors used the official NIST submission [2] as their codebase, referred to as `SQIsign(NIST)`, while their implementation including the non-square coordinates is referred to as `SQIsign(LWXZ)`.

Table 3.2: Comparison between NIST's submission implementation [2] and AprèsSQI's [5] implementation of LWXZ. The latter was included in the former's codebase. Results are expressed in terms of 10^3 $\mathbb{F}_p$-multiplications, using $\mathbf{S} = 0.8 \cdot \mathbf{M}$.

Prime	**Implementation**	**$\mathbb{F}_p$-operations**
p_{1973}	`SQIsign(NIST)` [2]	500.4
	`SQIsign(LWXZ)` [5]	383.1
Speedup		**23.44%**

If we compare the two speedups, we notice a difference of less than 10% between the two. The $\mathbb{F}_p$-operations model seems to overestimate the correct performance gain by a factor of *1.65*. However, considering that the two implementations run on different primes (p_{3923} and p_{1973}, respectively), we believe the factor to be slightly smaller in practice. The reason is that if we were to run `SQIsignOpt(LWXZ)` on a smaller prime, it would result in a higher speedup which would lead to a smaller difference in comparison to `SQIsign(NIST)`.

3.4 `xMUL` with adds

In this section, we consider an optimization for `xMUL` described by [5]. Our goal is to see how the improvement maps to a more practical context, that is, how the performance in $\mathbb{F}_p$-operations relates to the one in clock cycles.

3.4.1 Taking advantage of small coordinates

As described in the previous section, finding a second basis point requires an `xMUL` by the cofactor to ensure the point has order 2^f. Since during the

verification phase we need to generate a basis quite frequently, improving `xMUL` would greatly benefit the performance.

In AprèsSQI [5], the authors describe several improvements for `xMUL` which essentially optimize the $\mathbb{F}_{p^2}$-multiplications used in the procedure. For our research, we will focus on one of these. Specifically, we will take advantage of the small x-coordinate of the second torsion point. The previous section highlighted how we can generate such x-coordinates of the form $1+ki$, where $k \in \mathbb{F}_p$ and k is chosen as small as possible. When multiplying by a scalar in `xMUL`, the authors of [5] claim that if we replace an `fp2_mul` by x_Q with $1+k$ additions, we save around $3\mathbf{M}$ per bit of the scalar. This is explained by how `fp2_mul` is originally implemented. As shown in 2.1.2, we can do an `fp2_mul` with 3 `fp_mul`'s and 5 `fp_add`'s[1]. However, we show in the next section that if we replace an `fp2_mul` by additions, it costs $2k+2$ additions instead, for some pre-determined $k \in \mathbb{N}$. Since [5] ignored additions in their cost model, implementing an `fp2_mul` with additions implies saving the 3 `fp_mul`'s that were originally used in its definition.

For reference throughout the section, an example implementation of `xMUL` using `xDBL` and `xADD` is available in Appendix A, more precisely Algorithm 6, Algorithm 5 and Algorithm 4 respectively.

3.4.2 Implementation

Let $S, Q \in \mathbb{F}_{p^2}$, such that

$$S = a + bi \qquad Q = 1 + ki,$$

where $a, b, k \in \mathbb{F}_p$. Then we can write their multiplication as follows:

$$\begin{aligned}(a+bi)\cdot(1+ki) &= a + bi + ki \cdot (a+bi) \\ &= a + bi + kai - bk \\ &= (a - bk) + (b + ak)i\end{aligned} \tag{3.1}$$

From 3.1, we observe that we only need to calculate two $\mathbb{F}_p$-multiplications (bk and ak) and two $\mathbb{F}_p$-additions ($a + (-bk)$ and $b + ak$). This results in precisely $2k+2$ $\mathbb{F}_p$-additions or equivalently, $k+1$ $\mathbb{F}_{p^2}$-additions.

We implemented this idea in `SQIsignOpt(LWXZ)` by adjusting `xMUL`. The existent `xMUL` calls only one function: `xDBLADD`, which combines the functionalities of `xDBL` and `xADD`. Therefore, it has the parameters x_P, x_Q, x_{P-Q} and A. As explained in Appendix A, if the first two change values quite frequently, the third one representing their difference remains constant throughout the execution. In our case, the difference is always the second torsion point that we previously generated. Furthermore, in `xDBLADD`, there

[1]For simplicity, we treat subtractions equally in weight with additions. Therefore, we replace the number of subtractions with the same amount of additions in our measurements.

is only one $\mathbb{F}_{p^2}$-multiplication by x_{P-Q} that we refactor using additions. A pseudocode implementation is given in Algorithm 3.

Algorithm 3 `fp2_mul` with additions

Input: $S, Q \in \mathbb{F}_{p^2}$, with $S = a + bi$ and $Q = 1 + ki$, $a, b, k \in \mathbb{F}_p$
Output: $S \cdot Q \in \mathbb{F}_{p^2}$

1: $sum_1 \leftarrow 0$
2: $sum_2 \leftarrow 0$
3: **for** $i = 1$ to k **do**
4: $\quad sum_1 \leftarrow sum_1 + b$ ▷ a · k
5: $\quad sum_2 \leftarrow sum_2 + a$ ▷ b · k
6: **end for**
7: $sum_1 \leftarrow (-1) \cdot sum_1$ ▷ - b · k
8: $SQ_{re} \leftarrow a + sum_1$ ▷ $a - bk$
9: $SQ_{im} \leftarrow b + sum_2$ ▷ $b + ak$

3.4.3 Performance

In this section, we analyze the performance of `xMUL` using both cost metrics. We denote the original `xMUL` implementation by `xMUL*` and the one that replaces `fp2_mul`'s with additions by `xMUL+`. Finally, their performance comparison helps us derive a more precise $\mathbb{F}_p$-operations model.

As mentioned above, when generating the second torsion point, we use `xMUL` with the cofactor as a scalar. The cofactor is constant throughout the execution and for our specific prime p_{3923}, the cofactor has 188 bits. We denote this by $\text{bits}(s) = 188$.

Moreover, we define the performance of a function T in terms of $\mathbb{F}_p$-operations as $\mathrm{F_pO}(T)$ and in thousands of clock cycles as $\mathrm{CC}(T)$.

$\mathbb{F}_p$-operations model

Further, we present the measurements of both types of `xMUL` in terms of the $\mathbb{F}_p$-operations model, closely following AprèsSQI's [5] cost metrics. Therefore, we consider $\mathbf{S} = 0.8$, $\mathbf{M} = 1$ and $\mathbf{a} = 0$.

In both `xMUL*` and `xMUL+`, the function `xDBLADD` is called several times. However, in the case of `xMUL+`, one $\mathbf{M}$ is replaced by $(k+1)\mathbf{a}$. Without loss of generality, we choose k as the average of its values in 250 runs of verification. More precisely, we observe which second torsion points are chosen from the pre-generated list and average their corresponding k values(for the complete list of values, see the code on `https://github.com/georgenadejde/SQIsignOpt`). Consequently[2], rounded to the nearest integer, we choose $k = 11$. Therefore, counting the number of operations in

[2]We measured the average value of k to be 10.52. For reference, the first 4 values of k for p_{3923} in the corresponding list of second torsion points are 9, 10, 11 and 15.

both variants yields

$$\mathrm{F_pO}(\texttt{xDBLADD*}) = 4\mathbf{S} + 8\mathbf{M} + 8\mathbf{a} \qquad \mathrm{F_pO}(\texttt{xDBLADD+}) = 4\mathbf{S} + 7\mathbf{M} + 20\mathbf{a},$$

thus replacing $1\mathbf{M}$ by $12\mathbf{a}$. However, since the original model ignores additions, the above equations are equivalent to

$$\mathrm{F_pO}(\texttt{xDBLADD*}) = 4\mathbf{S} + 8\mathbf{M} \qquad \mathrm{F_pO}(\texttt{xDBLADD+}) = 4\mathbf{S} + 7\mathbf{M}$$

Additionally, we can extend the measurements to the entire `xMUL`. As explained in the previous sections, `xMUL` calls the underlying function `xDBLADD` precisely $\mathrm{bits}(s) = 188$ times. Hence,

$$\mathrm{F_pO}(\texttt{xMUL*}) = 188 \cdot (4\mathbf{S} + 8\mathbf{M}) \qquad \mathrm{F_pO}(\texttt{xMUL+}) = 188 \cdot (4\mathbf{S} + 7\mathbf{M})$$

The final numbers obtained by replacing the values from the model inside the two equations are presented in Table 3.3.

Table 3.3: Performance of the two `xMUL` implementations.

Implementation	$\mathbb{F}_p$**-operations**
`xMUL*`	2105.6
`xMUL+`	1917.6
Speedup	**8.92%**

Clock cycles model

We compare these theoretical measurements with our benchmark in clock cycles. We computed the average performance of both types of `xMUL` and displayed them in Table 3.4.

The benchmark presents a substantial performance decrease of 15.5% between the two `xMUL` implementations. However, the overall performance is not impacted as significantly, with a downgrade of only 2.65%.

More formally, it follows that

$$\mathrm{CC}(\texttt{xMUL*}) = 354,623 \qquad \mathrm{CC}(\texttt{xMUL+}) = 409,560$$

The results imply that the estimates based on the $\mathbb{F}_p$-operations model are again overpredicting, but this time with higher consequences. Theoretically, one expected a positive performance gain, while in practice, the changes impacted the performance negatively.

Consequently, this means additions are more significant in performance than `AprèsSQI`'s model [5] initially predicted.

Table 3.4: Performance comparison between `xMUL*` and `xMUL+`. Results are expressed in clock cycles.

Implementation	Cycles	Overall (mil)
`xMUL*`	354,623	28.273
`xMUL+`	409,560	29.025
Speedup	**-15.5%**	**-2.65%**

Finding a better model

We want the $\mathbb{F}_p$-operations model to predict the performance in clock cycles as precisely as possible. However, the results presented above in Table 3.3 and Table 3.4 show that the model does not perform as expected.

Further, we present our approach for creating a more precise model based on the amount of **S**, **M**, and **a**, that would reflect the performance difference in clock cycles.

We concluded in the previous section that additions should not be ignored in measuring performance. Therefore,

$$\mathrm{F_pO}(\texttt{xMUL*}) = 188{\cdot}(4\mathbf{S}{+}8\mathbf{M}{+}8\mathbf{a}) \qquad \mathrm{F_pO}(\texttt{xMUL+}) = 188{\cdot}(4\mathbf{S}{+}7\mathbf{M}{+}20\mathbf{a}),$$

with $\mathbf{a} \neq 0$.

Based on the -15.5% speedup in CC performance, we can compute the expected amount of $\mathbb{F}_p$-operations for `xMUL+`:

$$\mathrm{F_pO}(\texttt{xMUL+}) = \mathrm{F_pO}(\texttt{xMUL*}) + 15.5\% \cdot \mathrm{F_pO}(\texttt{xMUL*}) = 2431.9$$

We then proceed to solve the following system of equations:

$$\begin{cases} 188(4\mathbf{S} + 8\mathbf{M} + 8\mathbf{a}) & = 2105.6 \\ 188(4\mathbf{S} + 7\mathbf{M} + 20\mathbf{a}) & = 2431.9 \end{cases} \tag{3.2}$$

Using 3.2, we intend to find a suitable value for **a** that indicates the difference in clock cycles performance between the two `xMUL` variants. Moreover, we want our $\mathbb{F}_p$-operations cost model to reflect the number of multiplications we use. Therefore, we assume $\mathbf{M} = 1$ and $\mathbf{S} = 0.8$. From 3.2, it then follows that

$$\begin{cases} 4\mathbf{S} + 8\mathbf{M} + 8\mathbf{a} & = 11.2 \\ 4\mathbf{S} + 7\mathbf{M} + 20\mathbf{a} & = 12.9 \end{cases}$$

Using the first equation into the second yields

$$12\mathbf{a} - \mathbf{M} = 1.7 \tag{3.3}$$

Substituting $\mathbf{M} = 1$ in 3.3, yields

$$\mathbf{a} = 0.23,$$

rounded to two decimals. Therefore, we conclude

$$\mathbf{S} = 0.8\mathbf{M} \quad \mathbf{M} = 1 \quad \mathbf{a} = 0.23\mathbf{M}.$$

If we apply this model to the performance in $\mathbb{F}_p$-operations of `xMUL+`, we get that

$$\begin{aligned} \mathrm{F_pO}(\texttt{xMUL*}) &= 188 \cdot (4\mathbf{S} + 8\mathbf{M} + 8\mathbf{a}) = 2451.5 \\ \mathrm{F_pO}(\texttt{xMUL+}) &= 188 \cdot (4\mathbf{S} + 7\mathbf{M} + 20\mathbf{a}) = 2782.4, \end{aligned}$$

yielding a predicted speedup of **-13.5%**, which is close to the speedup in practice of -15.5%, and more precise than the original model's prediction of 8.92%.

3.5 Performance predictions

In this section, we present our predictions regarding the performance of the SQIsign variants described in [5]. We estimate the performance in cycles of these implementations relying on our proposed $\mathbb{F}_p$-operations model and benchmark results presented in Section 3.3 and Section 3.4. Note that these are just our general predictions and should not be considered precise indications of their practical performance.

Table 3.5 presents the implementations on different primes described by AprèsSQI in their paper [5], together with their variants. The table depicts smaller speedup factors compared to the ones presented by AprèsSQI. Our speedup factors for the unseeded and seeded versions of p_7 in comparison to `SQIsign(NIST)` are 1.89 and 2.23, contrary to [5]'s 2.37 and 2.80, respectively. Moreover, the same variants in comparison to `SQIsign(LWXZ)` yield 1.62 and 1.92, as opposed to 1.82 and 2.15.

For p_4's uncompressed version, we predict a speedup factor of 3.55 in comparison to `SQIsign(NIST)` and 3.05 in comparison to `SQIsign(LWXZ)`. These numbers are slightly lower compared to AprèsSQI's predictions of 4.46 and 3.41, respectively.

Table 3.5: Performance comparison between different implementations described by [5] in terms of 10^3 $\mathbb{F}_p$-multiplications, together with our predictions expressed in millions of clock cycles.

prime	Implementation	Variant	AprèsSQI's model [5]	Our prediction
p_{1973}	`SQIsign(NIST)` [2]	-	500.4	32.963
	`SQIsign(LWXZ)`	-	383.1	28.273
	`AprèsSQI`	unseeded	276.1	22.827
	`AprèsSQI`	seeded	226.8	18.753
p_7	`AprèsSQI`	unseeded	211.0	17.439
	`AprèsSQI`	seeded	178.6	14.760
	`AprèsSQI`	uncompressed	103.7	8.576
p_4	`AprèsSQI`	unseeded	185.2	15.314
	`AprèsSQI`	seeded	160.8	13.291
	`AprèsSQI`	uncompressed	112.2	9.276

Chapter 4

Conclusions

In this thesis, our goal was to check how the $\mathbb{F}_p$-operations cost model proposed by AprèsSQI [5] corresponds to the practical cost model in clock cycles. We implemented two different improvements on the codebase provided by [9] and measured their performance. Table 4.1 reports our benchmarking results.

Table 4.1: Performance comparison of different `SQIsign` implementations. Results are expressed in clock cycles.

Implementation	`xMUL`	**Overall (mil)**
`Original`	383,413	32.963
`Original(LWXZ)`	354,623	28.273
`Original(LWXZ_Adds)`	409,560	29.025

The $\mathbb{F}_p$-operations cost model eliminates the dependence on the level of optimization of the $\mathbb{F}_p$ arithmetic and the hardware running SQIsign [5], but its general nature implies that it may lack precision. We used an implementation that has highly optimized $\mathbb{F}_p$ arithmetic [9] to rigorously measure some of the verification improvements AprèsSQI proposed, and thus test the efficiency of their cost model. We showed that the $\mathbb{F}_p$-operations model overpredicts the real speedup by a factor of 1.65 for the non-square x-coordinates optimization. In the case of `xMUL` with adds, we established that the theoretical cost model predicts a positive speedup instead of a negative one, as shown in Table 4.2.

Based on these measurements, we were able to adjust the $\mathbb{F}_p$-operations cost model such that it reflects the practical speedup more accurately:

$$\mathbf{S} = 0.8\mathbf{M} \quad \mathbf{M} = 1 \quad \mathbf{a} = 0.23\mathbf{M}$$

.

Thus concluding that additions and subtractions are more significant than AprèsSQI initially considered in their model.

Moreover, we also provide estimates of the performance in practice of the implementations described by AprèsSQI in their paper based on our model. These are not precise enough to reflect the actual performance of these implementations, but they give a general overview of the expected performance.

As of *future work*, we would like to implement the rest of the proposed optimizations in [5] and compare the two cost models further. Additionally, we would like to test our own proposed cost model against these improvements, essentially verifying its effectiveness against a larger pool of implementations and adjusting it accordingly.

Table 4.2: Performance comparison of different `xMUL` implementations. Results are expressed in $\mathbb{F}_p$-operations and clock cycles.

Implementation	**AprèsSQI's model [5]**	**Our model**	**Cycles**
`xMUL*`	2105.6	2451.5	354,623
`xMUL+`	1917.6	2782.4	409,560
Speedup	**8.92%**	**-13.5%**	**-15.5%**

Bibliography

[1] Gilles Brassard. A quantum jump in computer science. In *Computer Science Today*, 1995. URL: https://api.semanticscholar.org/CorpusID:9733468.

[2] Jorge Chavez-Saab, Maria Corte-Real Santos, Luca De Feo, Jonathan Komada Eriksen, Basil Hess, David Kohel, Antonin Leroux, Patrick Longa, Michael Meyer, Lorenz Panny, Sikhar Patranabis, Christophe Petit, Francisco Rodríguez-Henríquez, Sina Schaeffler, and Benjamin Wesolowski. SQIsign: Algorithm specifications and supporting documentation. National Institute of Standards and Technology, 2023. URL: https://csrc.nist.gov/csrc/media/Projects/pqc-dig-sig/documents/round-1/specfiles/sqisign-spec-web.pdf.

[3] Hai-Ping Cheng, Erik Deumens, James K Freericks, Chenglong Li, and Beverly A Sanders. Application of quantum computing to biochemical systems: A look to the future. *Frontiers in chemistry*, 8:587143, 2020. URL: https://www.frontiersin.org/journals/chemistry/articles/10.3389/fchem.2020.587143/full.

[4] Gary Cornell, Joseph H. Silverman, and Glenn Stevens, editors. *Modular Forms and Fermat's Last Theorem*. Springer, New York, NY, 1997. URL: https://link.springer.com/10.1007/978-1-4612-1974-3, doi:10.1007/978-1-4612-1974-3.

[5] Maria Corte-Real Santos, Jonathan Komada Eriksen, Michael Meyer, and Krijn Reijnders. Aprèssqi: Extra fast verification for sqisign using extension-field signing. In *Advances in Cryptology – EUROCRYPT 2024*, pages 63–93, Cham, 2024. Springer Nature Switzerland. URL: https://eprint.iacr.org/2023/1559.

[6] Craig Costello. Supersingular isogeny key exchange for beginners. In *Selected Areas in Cryptography – SAC 2019*, pages 21–50, Cham, 2020. Springer International Publishing. URL: https://eprint.iacr.org/2019/1321.

[7] Craig Costello and Benjamin Smith. Montgomery curves and their arithmetic: The case of large characteristic fields. *Journal of Cryptographic Engineering*, 8, 03 2017. doi:10.1007/s13389-017-0157-6.

[8] Pierrick Dartois, Antonin Leroux, Damien Robert, and Benjamin Wesolowski. Sqisignhd: New dimensions in cryptography. In *Advances in Cryptology – EUROCRYPT 2024*, pages 3–32, Cham, 2024. Springer Nature Switzerland. URL: https://eprint.iacr.org/2023/436.

[9] Luca De Feo, Antonin Leroux, Patrick Longa, and Benjamin Wesolowski. New algorithms for the deuring correspondence. In *Advances in Cryptology – EUROCRYPT 2023*, pages 659–690, Cham, 2023. Springer Nature Switzerland. URL: https://eprint.iacr.org/2022/234.

[10] Luca De Feo, David Kohel, Antonin Leroux, Christophe Petit, and Benjamin Wesolowski. SQISign: Compact Post-quantum Signatures from Quaternions and Isogenies. In Shiho Moriai and Huaxiong Wang, editors, *Advances in Cryptology - ASIACRYPT 2020 - 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea, December 7-11, 2020, Proceedings, Part I*, volume 12491 of *Lecture Notes in Computer Science*, pages 64–93. Springer, 2020. doi:10.1007/978-3-030-64837-4_3.

[11] Kaizhan Lin, Weize Wang, Zheng Xu, and Chang-An Zhao. A faster software implementation of SQISign. Cryptology ePrint Archive, Paper 2023/753, 2023. https://eprint.iacr.org/2023/753. URL: https://eprint.iacr.org/2023/753.

[12] Matthias Möller and Cornelis Vuik. On the impact of quantum computing technology on future developments in high-performance scientific computing. *Ethics and Information Technology*, 19:253 – 269, 2017. URL: https://api.semanticscholar.org/CorpusID:6864503.

[13] Peter L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48(177):243–264, 1987. URL: https://www.ams.org/mcom/1987-48-177/S0025-5718-1987-0866113-7/, doi:10.1090/S0025-5718-1987-0866113-7.

[14] National Institute of Standards and Technology. Post-quantum cryptography: Round 1 additional digital signature candidates, 2024. Accessed: 2024-06-06. URL: https://csrc.nist.gov/projects/pqc-dig-sig/round-1-additional-signatures.

[15] National Institute of Standards and Technology (NIST). Pqc standardization process: Announcing four candidates to

be standardized, plus fourth round candidates, 2022. Accessed: 2024-06-06. URL: https://csrc.nist.gov/News/2022/pqc-candidates-to-be-standardized-and-round-4.

[16] Román Orús, Samuel Mugel, and Enrique Lizaso. Quantum computing for finance: Overview and prospects. *Reviews in Physics*, 4:100028, 2019. URL: https://www.sciencedirect.com/science/article/pii/S2405428318300571.

[17] Lorenz Panny. Elliptic curves and isogenies: The good bits. URL: https://yx7.cc/docs/misc/isog_bristol_notes.pdf.

[18] Maria Corte-Real Santos. Vélu's formulas for sidh, 2020. Accessed: 2024-06-29. URL: https://www.mariascrs.com/2020/11/07/velus-formulas.html.

[19] J.H. Silverman. *The Arithmetic of Elliptic Curves*. Graduate Texts in Mathematics. Springer New York, 2009. URL: https://books.google.ro/books?id=Z90CA_EUCCkC.

[20] Joseph H. Silverman and John T. Tate. *Rational Points on Elliptic Curves*. Springer Cham, 2 edition, Jun 2015. doi:10.1007/978-3-319-18588-0.

[21] Lee Spector. Quantum computing. In *Proceedings of the 10th Annual Conference Companion on Genetic and Evolutionary Computation*, GECCO '08, page 2865–2894, New York, NY, USA, 2008. Association for Computing Machinery. doi:10.1145/1388969.1389082.

[22] J. Vélu. Isogénies entre courbes elliptiques. *Comptes-Rendus de l'Académie des Sciences, Série I*, 273:238–241, juillet 1971.

Appendix A

Appendix

In this section, we present in more technical detail algorithms and formulas we used throughout the thesis.

A.1 Elliptic curve arithmethic

The arithmetic on Montgomery curves differs slightly compared to other variants of elliptic curves, mainly because it only uses the x-coordinate. Therefore, we tend to ignore the y-coordinate of a point and instead represent it solely by its x-coordinate, denoted by $x(P)$, for $P \in \mathcal{E}_A$. As argued in the thesis, we use projective instead of affine coordinates for efficiency reasons. Hence,

$$P = x(P) = X_P/Z_P,$$

for $P \in \mathcal{E}_A$, where $X_P, Z_P \in \mathbb{F}_{p^2}$.

The Montgomery Ladder [13] is a simple and fast method to perform scalar multiplication on different types of elliptic curves. Typically named `xMUL` in the isogeny-based cryptography field, it is implemented using two additional functions: `xADD` and `xDBL`.

For both algorithms, we denote $\mathcal{O}$ and $\mathcal{T}$ to be points in projective coordinates of the following form:

$$\mathcal{O} = (a : 0) \qquad \mathcal{T} = (0 : b),$$

with $a, b \in \mathbb{F}_{p^2}$ and $a, b \neq 0$.

For `xADD`, besides the two points we want to add, we also need a third parameter: the difference between these two points. Although it might seem peculiar to define the addition of points in terms of their difference, they are in fact mathematically indistinguishable. However, we do not need to compute the difference separately. Instead, we use one of the properties of the Montgomery Ladder which ensures that whenever we have to call `xADD` on two points R_0 and R_1, their difference will always be the same: P, precisely the point which we want to multiply by scalar s.

Algorithm 4 Point Addition on Montgomery Curve (`xADD`)

Input: $P = (X_P : Z_P),\ Q = (X_Q : Z_Q),\ P - Q = (X_{P-Q} : Z_{P-Q}) \in \mathcal{E}_A$
Output: $P + Q = (X_{P+Q} : Z_{P+Q}) \in \mathcal{E}_A$

if $(P - Q) = \mathcal{O}$ **or** $(P - Q) = \mathcal{T}$ **then**
    return $\mathcal{O}$
else
    $V_0 \leftarrow \text{fp2\_add}(X_P, Z_P)$
    $V_1 \leftarrow \text{fp2\_sub}(X_Q, Z_Q)$
    $V_1 \leftarrow \text{fp2\_mul}(V_1, V_0)$
    $V_0 \leftarrow \text{fp2\_sub}(X_P, Z_P)$
    $V_2 \leftarrow \text{fp2\_add}(X_Q, Z_Q)$
    $V_2 \leftarrow \text{fp2\_mul}(V_2, V_0)$
    $V_3 \leftarrow \text{fp2\_add}(V_1, V_2)$
    $V_3 \leftarrow \text{fp2\_mul}(V_3, V_3)$
    $V_4 \leftarrow \text{fp2\_sub}(V_1, V_2)$
    $V_4 \leftarrow \text{fp2\_mul}(V_4, V_4)$
    $X_{P+Q} \leftarrow \text{fp2\_mul}(Z_{P-Q}, V_3)$
    $Z_{P+Q} \leftarrow \text{fp2\_mul}(X_{P-Q}, V_4)$
    return (X_{P+Q}, Z_{P+Q})
end if

For point doubling, we will also need the A parameter of the curve. As [5] observed, using A in affine coordinates instead of projective improves the speed of `xDBL` since it requires one less $\mathbb{F}_{p^2}$-multiplication.

Algorithm 5 Point Doubling (`xDBL`)

Input: $P = (X_P : Z_P) \in \mathcal{E}_A,\ A \in \mathbb{F}_{p^2}$
Output: $[2]P = (X_{[2]P}, Z_{[2]P}) \in \mathcal{E}_A$

if $P = \mathcal{O}$ **or** $P = \mathcal{T}$ **then**
    return $\mathcal{O}$
else
    $V_1 \leftarrow \text{fp2\_add}(X_P, Z_P)$
    $V_1 \leftarrow \text{fp2\_mul}(V_1, V_1)$
    $V_2 \leftarrow \text{fp2\_sub}(X_P, Z_P)$
    $V_2 \leftarrow \text{fp2\_mul}(V_2, V_2)$
    $x_{[2]P} \leftarrow \text{fp2\_mul}(V_1, V_2)$
    $V_1 \leftarrow \text{fp2\_sub}(V_1, V_2)$
    $c1 \leftarrow \text{fp2\_add}(A, 2)$
    $V_3 \leftarrow \text{fp2\_mul}(\text{fp2\_div}(c1, 4), V_1)$
    $V_3 \leftarrow \text{fp2\_add}(V_3, V_2)$
    $z_{[2]P} \leftarrow \text{fp2\_mul}(V_1, V_3)$
    return $(x_{[2]P}, z_{[2]P})$
end if

Finally, the Montgomery Ladder combines both `xADD` and `xDBL`. Notice that we use two points R_0 and R_1 that are initialized with P and $[2]P$, respectively. The difference of P will be kept through the whole process, which enables us to easily provide the third parameter for `xADD`. Moreover, similar to other scalar multiplication algorithms such as square and multiply, the scalar needs to be traversed bit by bit:

Algorithm 6 Montgomery Ladder (`xMUL`)

Input: $k = b_{n-1}b_{n-2}\dots b_1b_0$, with $b_i \in \{0,1\}$ and $b_{n-1} = 1$, $P = (X_P : Z_P)$, $A \in \mathbb{F}_{p^2}$
Output: $[k]P = (X_{[k]P}, Z_{[k]P}) \in \mathcal{E}_A$

if $P = \mathcal{O}$ **or** $P = \mathcal{T}$ **then**
    return $\mathcal{O}$
end if
$R_0 \leftarrow P$
$R_1 \leftarrow \texttt{xDBL}(P, A)$
for *bit* in k **do**
    if $bit = 0$ **then**
        $R_1 \leftarrow \texttt{xADD}(R_0, R_1, P)$
        $R_0 \leftarrow \texttt{xDBL}(R_0, A)$
    else
        $R_0 \leftarrow \texttt{xADD}(R_0, R_1, P)$
        $R_1 \leftarrow \texttt{xDBL}(R_1, A)$
    end if
end for
return R_0

For even more details about the arithmetic on Montgomery curves, see the paper by Costello and Smith [7].

Finally, the j-invariant of a Montgomery curve is denoted by the following formula:

$$j(\mathcal{E}_A) = \frac{256(a^2-3)^3}{a^2-4}$$

A.2 Finite field operations

In this section, we present the implementation of division in $\mathbb{F}_{p^2}$. Essentially, the algorithm uses the following idea:

$$\frac{a+bi}{c+di} = \frac{(a+bi)\cdot(c-di)}{c^2+d^2} = [(a+bi)\cdot(c-di)]\cdot(c^2+d^2)^{-1},$$

where $a, b, c, d \in \mathbb{F}_p$.

Algorithm 7 $\mathbb{F}_{p^2}$ division (`fp2_div`)

Input: $c_1 = a + bi,\ c_2 = c + di \in \mathbb{F}_{p^2}$
Output: $c_1/c_2 \in \mathbb{F}_{p^2}$

1: $conj_2 \leftarrow \text{getConj}(c_2)$ ▷ $c - di$
2: $N \leftarrow \text{fp2\_mul}(c_1, conj_2)$ ▷ $(a + bi) \cdot (c - di)$
3: $norm \leftarrow \text{fp2\_mul}(c_2, conj_2)$ ▷ $c^2 + d^2$
4: $D \leftarrow \text{fp\_inv}(norm)$ ▷ $(c^2 + d^2)^{-1}$
5: $R \leftarrow \text{fp2\_mul}(N, D)$
6: **return** R

A.3 Generating a random rational point

In isogeny-based cryptography, we often need to generate random points on elliptic curves. And since we are using fast x-only arithmetic, we only need to generate x-coordinates. However, not all values are valid x-coordinates. An additional check needs to be done using the curve equation. More precisely, we need to show that the point which corresponds to our randomly generated x-coordinate lies on the elliptic curve. To prove this, we plug in the x-coordinate into the curve's equation and check if the right-hand side is a square.

An x-coordinate is a value in $\mathbb{F}_{p^2}$. Therefore, when we plug it into the curve's equation, we get that

$$y^2 = a + bi,$$

for some $a, b \in \mathbb{F}_p$. Computing y^2 is shown in Algorithm 8.

Algorithm 8 Compute RHS of EC equation

Input: $x \in \mathbb{F}_{p^2},\ A \in \mathbb{F}_{p^2}$
Output: $x^3 + Ax^2 + x$

1: $x_2 \leftarrow \text{fp2\_pow}(x, 2)$ ▷ x^2
2: $a\_sq \leftarrow \text{fp2\_mul}(A, x_2)$ ▷ Ax^2
3: $x_3 \leftarrow \text{fp2\_pow}(x, 3)$ ▷ x^3
4: $add_1 \leftarrow \text{fp2\_add}(x_3, a\_sq)$ ▷ $x^3 + Ax^2$
5: $y_2 \leftarrow \text{fp2\_add}(add_1, x)$ ▷ $x^3 + Ax^2 + x$
6: **return** y_2

Further, we need to show that $\exists c, d \in \mathbb{F}_p$ such that

$$a + bi = (c + di)^2.$$

Showing that a complex number is a square represents a difficult problem. However, instead of working in $\mathbb{F}_{p^2}$, we can use a smart idea and instead

work in $\mathbb{F}_p$. Any complex number $a + bi$ has a *norm*, which is equal to

$$(a + bi)(a - bi) = a^2 + b^2,$$

and trivially

$$a^2 + b^2 \in \mathbb{F}_p.$$

Finally, we use Legendre's symbol to check if our $\mathbb{F}_p$ value is indeed a square. Legendre's symbol is defined as follows:

$$\left(\frac{\alpha}{p}\right) = \alpha^{\frac{p-1}{2}} = \begin{cases} 1, & \alpha \text{ is a square.} \\ -1, & \alpha \text{ is not a square.} \end{cases}$$

Therefore, we only need to perform one more exponentiation to complete our algorithm. This is shown in Algorithm 9.

Algorithm 9 Legendre's symbol

Input: $q \in \mathbb{F}_p$
Output: `True` if q is a square, `False` otherwise.

1: $pw \leftarrow \text{fp\_pow}(q, \frac{p-1}{2})$
2: **if** $pw = 1$ **then**
3: **return** `True`
4: **else**
5: **return** `False`
6: **end if**