Advances Towards Efficient Imeplementations of Isogeny Based Signatures

Submitted by

Robert W.V. Gorrie
B.ASc. Computer Science (McMaster University)

Under the guidance of
Douglas Stebila

Submitted in partial fulfillment of
the requirements for the award of the degree of

Masters of Science
in
Computer Science

Department of Computing and Software
McMaster University
Hamilton, Ontario, Canada

Fall 2018

Abstract

Progress in the field of quantum computing has shown that, should construction of a sufficiently powerful quantum computer become feasible, much of the cryptography used on the Internet today will be rendered insecure. In lieu of this, several approaches to "quantum-safe" cryptography have been proposed, each one becoming a serious field of study. The youngest of these approaches, isogeny based cryptography, is oriented around problems in algebraic geometry involving a particular variety of elliptic curves. Supersingular isogeny Diffie-Hellman (SIDH) is this subfield's main contender for quantum-safe key-exchange. Yoo et al. have provided an isogeny-based signature scheme built on top of SIDH. Currently, cryptographic algorithms in this class are hindered by poor performance metrics and, in the case of the Yoo et al. signature scheme, large communication overhead.

In this dissertation we explore two different modifications to the implementation of this signature scheme; one with the intent of improving temporal performance, and another with the intent of reducing signature sizes. We show that our first modification, a mechanism for batching together expensive operations, can offer roughly 1.1% faster signature signing, and roughly 3.5% faster signature verification. Our second modification, an adaptation of the SIDH public key compression technique outlined in [CJL+17], can reduce Yoo et al. signature sizes from roughly 928λ bytes to 640λ bytes at the 128-bit security level on a 64-bit operating system. We also explore the combination of these techniques, and the potential of employing these techniques in different application settings. Our experiments reveal that isogeny based cryptosystems still have much potential for improved performance metrics. While some practitioners may believe isogeny-based cryptosystems impractical, we show that these systems still have room for improvement, and with continued research can be made more efficient - and eventually practical. Achieving more efficient implementations for quantum-safe algorithms will allow us to make them more accessible. With faster and lower-overhead implementations these primitives can be run on low bandwidth, low spec devices; ensuring that more and more machines can be made resistant to quantum cryptanalysis.

Contents

List of Figures

Acknowledgments

First and foremost I would like to thank my supervisor Douglas Stebila, whose expertise and mentorship have motivated and focused me greatly over the past few years. This research would also not have been possible without the work and guidance of researchers and developers at Microsoft Research's NExT Security & Cryptography group as well as Waterloo University's Institute for Quantum Computing.

I would also like to deeply thank my fellow peers and colleagues at McMaster University who, through years of genial conversation, have helped to foster my curiosity, imagination, and passion for science and mathematics.

Robert Gorrie

November 3, 2018
McMaster University

Chapter 1

Introduction

The past 30 years have brought with them astonishing developments in the field of quantum computing. With these developments, quantum computers have been shown to possess computing power beyond that of our classical, binary architectures. Through the continually developing articulation of quantum algorithms, we have witnessed the discovery of algorithms capable of efficiently solving problems which had no prior known subexponential solution [**?**].

Cryptography, a branch of mathematics separate from that of quantum computing, is the study of secure communication systems. Cryptographic systems operate under the presence of an external, unauthorized, and untrusted party (often referred to as the *adversary*,) against whom properties of the communication must be safeguarded. Also critical to the field of cryptography is the practice of proving (or disproving) that a given system is safe and secure.

These distinct fields overlap in a variety of ways. For example, some of the previously mentioned problems, which now have newly discovered subexponential solutions, have historically been used as the backbone for many popular cryptosystems. It was the assumed difficulty of these problems that the security of certain cryptosystems depended on[1]. Thus, the implementation of a sufficiently large quantum computer would be a catastrophic threat to the majority of modern Internet security.

And so, as physicists and engineers race towards error-free and energy efficient implementations of quantum computers, we steadfastly approach a new age for the art and science of Cryptography. The looming threat of large-scale quantum computing has driven the field of "post-quantum" cryptography; the aspiration of which is to develop efficient and secure cryptographic algorithms that are resistant to quantum cryptanalysis.

1.1 Motivation

The following section will discuss or make reference to cryptographic concepts that may be new to the reader. Section 2.1 provides detailed definitions for some of these concepts, and may prove helpful in illuminating some of the coming discussion.

[1]These problems reside in the complexity class known as BQP, or "bounded-error quantum polynomial-time"; one particular problem in this class is the *hidden subgroup problem*, a problem with much historical significance in the design of Cryptographic systems

There are several subfields that currently occupy the research space of post-quantum cryptography. These subfields are each predicated on their own underlying mathematical problems, and more importantly, assumptions on the difficulty of those problems. The following make up some of the most popular subfields of post-quantum cryptography:

- *Lattice-based Cryptography*, based on problems such as LWE and Ring-LWE,
- *Hash-based Cryptography*, building signatures from cryptographic hash functions,
- *Multivariate Cryptography*, systems designed around multivariate polynomials, and
- *Code-based Cryptography*, based on the difficulty of decoding linear codes.

For this dissertation, however, we will focus on a younger subfield of post-quantum cryptography, namely, isogeny-based cryptography.

Isogeny-based Cryptography. Over the course of the past decade, elliptic curve cryptography (ECC) has proven itself indisposable in the world of applied cryptology. While isogeny-based cryptography and ECC are both built up from elliptic curve mathematics, they differ in their fundamental presuppositions.

Also worth noting is that isogeny-based systems are still considerably young. Because of this, some may be hesitant to trust the security of these systems. Additionally, they are often outperformed by other post-quantum alternatives (which we will investigate more closely in a moment). They do, however, appear to have some advantages - in particular their small cryptographic key sizes.

The aim of this dissertation is to improve the efficiency of a particular isogeny-based scheme. We hope to showcase that, through intelligent implementation, isogeny-based protocols still have a lot of improvement potential in terms of run-time and storage performance.

More specifically, for this dissertation we are primarily focused on the performance of an isogeny-based "proof of knowledge" style signature scheme, outlined in great detail by Youngo Yoo et al. in [YAJ$^+$12], which we will henceforth refer to as the "Yoo et al. signature scheme".

1.1.1 Post-Quantum and Classical Performance Comparisons

We will now provide a rough survey of several post-quantum cryptosystems so as to compare their performance (both temporally in terms of exeuction time, and spatially in terms of key and signature sizes) with popular non-quantum-safe systems.

First, another important detail is the manner in which the security of cryptographic systems is measured. A cryptographic system is said to be n-bit secure if the fastest attack on that system is performed in 2^n operations. These attacks often take the form of a brute-force search of the n-bit space in an attempt to find the secret value/key.

We gathered runtime measurements of the Yoo et al. signature scheme from [YAJ$^+$17], runtimes for other post-quantum schemes from [ea18] and [SM18], and runtimes of the classical protocols RSA and ECDSA from the standard OpenSSL distribution. We've compiled the results into Figures 1.1 and 1.2. In these figures, "SIDH" is used to represent the Yoo et al. signature scheme, which (as we will see in the coming Chapter) is largely based on the supersingular isogeny Diffie-Hellman (SIDH) system [**?**].

	Key Gen	Sign	Verify
SIDH	a	b	c
Sphincs (Hash-based)	17,535,886.94	653,013,784	27,732,049
qTESLA (Ring-LWE)	1,059,388	460,592	66,491
Picnic (Hash-based)	13,272	9,560,749	6,701,701
RSA	a	1,113,600	32400
ECDSA	1,470,000	128,928	140,869

Figure 1.1

	Public Key	Private Key	Signature
SIDH	768	48	141,312
Sphincs (Hash-based)	32	64	8,080 - 16,976
Rainbow (Multivariate-based)	152,097 - 192,241	100,209 - 114,308	64 - 104
qTESLA (Ring-LWE)	4,128	2,112	3,104
Picnic (Hash-based)	33	49	34,004 - 53,933
RSA	384	b	384
ECDSA	32	b	32

Figure 1.2: Signature and key sizes for various post-quantum and classical protocols

All of the measurements in these figures reflect implementations which offer 128 bit post-quantum security, with the exception of classical protocols RSA, and ECDSA, where numbers are taken at the 2048 and 256-bit (*classical*) security level, respectively. The performance measurements of protocols found in 1.1 were either (in the case of Sphincs) measured ourselves, in the same setting as measurements for the isogeny-based scheme, or taken as reported in the relevant literature.

The runtime benchmarks found in Figure **??** and Figure 1.1 are all given in terms of CPU cycles, and as such are more architecture independent than timestamp-based measurements. The measurements for signature and key sizes in Figure 1.2 are given in terms of bytes.

Reflecting on Figures **??** 1.1, ¡ talk about comparison results ¿.

As for Figure 1.2, ¡ talk about size comparison results ¿.

1.2 Contributions

We offer two main contributions to the Yoo et al. signature scheme implementation. Both of these contributions, as previously mentioned, are designed with the intent of improving the performance of said protocol: the first offers an improvement in the run-time of the signature scheme and the second offers reduced signature sizes for the scheme. Our work is built ontop of the SIDH C library written by Microsoft Research, and incorporates code written by Yoo and his associates [?][YAJ+17].

All of these contributions can be found and tested at `https://github.com/GorrieXIV/SIDH2.0-SignatureExtension`.

1.2.1 Operation Batching

Our first contribution involves the implementation of a procedure that batches together many occurrences of the same low level operation. This procedure significantly reduces the total count of a particularly expensive operation. We provide C code which incorporates this batching procedure into the Yoo et al. signature scheme code.

In the section detailing this contribution, we offer extensive measurements of the performance increases offered by the inclusion of the batching procedure.

We conclude that the inclusion of our batching technique in the Yoo et al. signature scheme is both secure and offers noteworthy performance improvements in signature signing and verification routines.

1.2.2 Signature Compression

The second contribution we offer is another addendum to the SIDH/Yoo signature library, this time offering a mechanism to compress signature sizes. We embed a particular compression algorithm into the Yoo et al. signature protocol – the compression algorithm in question (contrived by Costello et al. and outlined in [CJL+17]) is intended for compression of SIDH public keys. We have adopted this method and applied it to specific portions of the Yoo et al. signatures, yielding significantly smaller signatures at the cost of extra computation.

This approach to signature compression is mentioned in [YAJ+12], but not implemented. We detail our implementation in Section 4, and analyse both the decrease in signature size and the computational cost of performing comrpession.

1.3 Organization

With the remaining section of this introductory chapter, we will explain some of the structuring and notation used in this dissertation.

1.3.1 Layout

This dissertation is divided into 5 chapters. This chapter and the one that follows contain the relevent preliminary information for understanding the contributions of the thesis. The two following chapters thereafter outline in detail the two contributions of our work. The fifth and final chapter concludes the dissertation while offering any final remarks and suggestions for continued research.

Chapter 2 covers the relevant mathematical background. Within this chapter we also cover the portions of the SIDH C library that are utilized and/or modified in our implementations.

Chapters 3 and 4 are rather similar in structure. Both begin with an introduction of their contribution's components - doing so in a general setting. Following this, the implementation specifics of the chapters contribution are layed out. For these sections, we attempt to convey the implementation details with a level of granularity we find easily accessible, while also providing enough information such that if the reader were to investigate our code they could do so (hopefully) with ease. The final sections of chapters 4 & 5 include the implementation results, benchmarks, and analysis. The main structural difference between these two chapters is that chapter 4 requires additional background.

We found it more appropriate to include this material here, in the introduction to chapter 4, rather than in chapter 2.

The fifth chapter closes out the dissertation with a summary of our progress and measurements. We then spend some time discussing possible avenues for future work. Following this chapter is Appendix A, which details C code for some of the SIDH C library functions which are particularly relevant to our work.

1.3.2 Notation & Style

We will now take some time to formalize the variety of notation and formatting used in this dissertation.

Functions & Procedures. Throughout the dissertation, general functions and procedures are denoted by the use of a **bold font face**. This is true for procedures introduced both formally and informally. Functions that are defined within the SIDH C codebase (either by us or others), however, are denoted by the use of a `monospace font`. This monospace notation is also sometimes used to denote routines or subroutines composed of by a sequence of functions or a portion of code.

When referring to a function in any general sense, we will write only its name using the aforemention convention. By contrast, when we refer to the result of a function executed over input $x_1, ..., x_n$, we append on the function identifier the set of parameters enclosed in parathesis (e.g. **GenericFunction**($x_1, ..., x_n$) **or** `GenericFunction(`$x_1, ..., x_n$`)`.

It is also worth noting that we frequently refer to these abstract, bold-identified functions as *procedures*, whereas we try to reserve use of the term *function* for C-defined functions. When giving precise definitions of procedures, we opt for a pseudocode/algorithmic approach. For functions, on the otherhand, we enclose our definitions in an environment with a light-gray background. Below we illustrate these two different approaches:

Algorithm 1 – **ProcedureExample**($\{a_0, a_1, ..., a_b\}$, c)

```
1: if c ≤ b then
2:     return a_c
3: else
4:     return −1
```

```
void function_example (int* a, int b, int c) {
    if (c <= b) {
        return a[c];
    } else {
        return -1;
    }
}
```

Cryptography Conventions. Cryptographic protocols, as per the usual convention, are written and defined in terms of tuples of algorithms. In denoting general protocols, we frequently use a capital Pi (Π) subscripted with some informative title. Following this format, Π_{sig}.**KeyGen** might represent the key generation algorithm found in some signature protocol. If the context is clear, we may refer to an algorithm/procedure such as this simply by its name (e.g. **KeyGen**), dropping the leading protocol identifier.

Math Conventions. In denoting isogenies (and other functions between elliptic curves) we will opt to use upper-case greek letters. Elliptic curves discussed in a general setting are refered to, when possible, as E; if a more unique identifier is necessary, E with a unique subscript is used. For example, E_{Alice} might refer to a curve created by Alice.

When writting log we assume base 2, unless noted otherwise.

Chapter 2

Technical Background

This chapter will cover the following preliminary topics: cryptographic primitives, isogenies and their relevant properties, supersingular isogeny Diffie-Hellman (SIDH), the Fiat-Shamir construction for digital signatures (and its quantum-safe adaptation), the current landscape of isogeny-based signature schemes, and finally select C implementations of the isogeny-based protocols with which we are concerned.

In the first section of this chapter we will take some time to introduce a few ideas from modern cryptography. We will cover key exchange, identification schemes, and signature schemes - all at as high of an abstraction level as possible. Readers familiar with these topics can skip this section without harm.

Our discussion of isogenies will begin with some basic coverage of the underlying algebra. We will provide the material necessary for the remaining sections as we build up in the level of abstraction; working our way through groups, finite fields, elliptic curves, and finally isogenies and their properties.

Once we have presented the necessary algebra, we will illustrate the specifics of the supersingular isogeny Diffie-Hellman key-exchange protocol. We will spend most of this time dedicated to a modular deconstruction of the protocol, looking at the high-level procedures and algorithms which will be necessary for understanding in detail the signature protocol to come. This subsection will end with a briefing and analysis of the closely related zero-knowledge proof of identity (ZKPoI) protocol proposed in the original De Feo et al. paper [FJP12], as it is the foundation for the isogeny-based signature scheme presented by Yoo et al [YAJ$^+$12].

In section 2.3 we will discuss the Fiat-Shamir transformation [Kat10]; a technique which, given a secure interactive identification scheme, creates a secure digital signature scheme. We will also look at the quantum-secure adaptation published by Unruh [Unr14], as applying a non-quantum-resistant transform to a quantum-resistant primitive would be rather frivolous.

Section 2.4 will be dedicated to covering current isogeny-based signature schemes - the topic about which this dissertation is mainly concerned. We will discuss the signature scheme of Yoo et al., which is a near direct application of Unruh's work to the SIDH zero-knowledge proof of identity.

Finally, the last section of this chapter will introduce the SIDH C library released by Microsoft Research, on top of which the core contributions of this thesis are implemented. We will also look at the implementation of the to-be-discussed signature scheme, which is a proof-of-concept built on top of the Microsoft API.

2.1 Cryptographic Primitives

Cryptographic primitives can be thought of as the basic building blocks of cryptographically secure applications and protocols. The idea of which being that if individual primitives are provably (or believeably) secure, we can be more confident in the security of the application as a whole.[1]

To quickly recap some basic information security, there are serveral different security properties a cryptographic primitive may aim to offer:

- *Confidentiality*: The notion that the information in question is kept private from unauthorized individuals.
- *Integrity*: The notion that the information in question has not been altered by unauthorized individuals.
- *Availability*: The notion that the information in question is available to authorized individuals when requested.
- *Authenticity*: The notion that the source of the information in question is verified.
- *Non-repudiation*: The notion that the source of the information in question **cannot** deny having originally provided the information.

The security of a particular cryptographic primitive is measured by two components. The first, referred to often as a "security guarantee", measures what conditions constitute a successful attack on the primitive. The second, known as the "threat model", makes assumptions about the computational powers that the adversary holds. The best practice in forming security proofs is to aim for security with respect to the most easily broken security guarantee and the most challenging possible threat model. The combination of a security guarantee and threat model is known as a *security goal*.

Each of the primitives to come are designed to offer some security to the communication between a given pair of entities. We will refer to these entities as Alice and Bob. The schemes we are concerned with in this dissertation are strictly *public key* (also known as *asymmetric key*) schemes. In public key primitives, each user possesses a *public* key (visible to every user in the network) as well as a *private* key, which only they have access to.

The first class of primitives we will discuss, *key exchange* protocols, provide a means by which Alice and Bob can come to the agreement of some secret value. The goal of a key exchange protocol is for Alice and Bob, communicating over some open, insecure channel, to reach mutual agreement of the secret value while also ensuring the *confidentiality* of that value. The secret value is referred to as a *secret* or *shared* key and is intended for use in other cryptographic primitives.

Identification schemes are a class of primitives that aim to ensure *authenticity* of a given entity. If Alice is communicating with Bob and she wants to verify that Bob is who he claims to be, the two can utilize a secure identification scheme. After identification protocols we will look at signature schemes, which are somewhat of an extension of the former. Signature schemes aim to provide *authenticity* on every message sent from Bob

[1] This is not to say that software which implements provably secure primitives is guaranteed to be secure. In security, it should be expected that the weakest link in the system is the first to be exploited, and these weak links often lie in careless implementation details.

to Alice, as well as *non-repudiability* and *integrity* of those messages.

Random Oracle Model. Before continuing with our discussion of primitives, it is worth covering briefly a framework in cryptography known as the random oracle model. A "random oracle" is a theoretical black box which, for every unique input, responds with a truly random output. That is, if a query is made to a random oracle h with input x (written $h(x)$) multiple times, h with respond with the same (random) output every time.

For certain constructions to be proven secure, it is sometimes necessary or helpful to assume the existence of random oracles. While this assumption may seem greviously optimistic, *hash functions* are a widely diployed family of functions which are believed to approach the nature of random oracles to some degree. Much of the security of modern cryptography depends on the security of such hash functions.

2.1.1 Key Exchange

A key exchange protocol, which we will denote as Π_{kex}, can be represented in some contexts by a pair of polynomial time algorithms **KeyGen** and **SecAgr**: $\Pi_{kex} = (\textbf{KeyGen}, \textbf{SecAgr})$. Alice and Bob will each run both of these procedures. The first they will run on the same input, 1^λ, a bit string of λ 1's. The second, short for "secret agreement", they will run on both their outputs of **KeyGen** and their peers.

Execution of Π_{kex} between Alice and Bob involves the following:

(i) Alice and Bob run $\textbf{KeyGen}(1^\lambda)$: A probabilistic algorithm with input 1^λ and output (sk, pk). Typically pk is the image of $f(sk)$, where f is some *one-way* function. We will denote the outputs of **KeyGen** for Alice and Bob as $(sk_{\text{Alice}}, pk_{\text{Alice}})$ and $(sk_{\text{Bob}}, pk_{\text{Bob}})$ respectively.

(ii) Alice and Bob exchange (over an insecure channel) their public keys pk_{Alice} and pk_{Bob}.

(iii) Alice runs $\textbf{SecAgr}(sk_{\text{Alice}}, pk_{\text{Bob}})$: A deterministic algorithm with input sk_{Alice} and pk_{Bob} and output $k_{\text{Alice}} \in \{0,1\}^\lambda$. Bob runs $\textbf{SecAgr}(sk_{\text{Bob}}, pk_{\text{Alice}})$ to obtain $k_{\text{Bob}} \in \{0,1\}^\lambda$.

Π_{kex} is said to uphold *correctness* if $k_{Alice} = k_{Bob}$ for all honestly derived keypairs $(sk_{\text{Alice}}, pk_{\text{Alice}})$ and $(sk_{\text{Bob}}, pk_{\text{Bob}})$. Because we deal only with correct Π_{kex}, we refer to the output of Π_{kex} as simply k.

Figure 2.1 illustrates an execution of the Diffie-Hellman key exchange protocol which relies on the difficulty of the *discrete logarithm* problem for its one-way function f.

The security goal typical of a key exchange protocol is that an adversary with access to the session transcript (threat) cannot discern the resulting shared secret key from a randomly generated value (security guarantee).

2.1.2 Interactive Identification Schemes

Imagine Alice wishes to confirm the identity of Bob. The motivation for interactive identification schemes is to provide Bob with some mechanism for proving to Alice (or

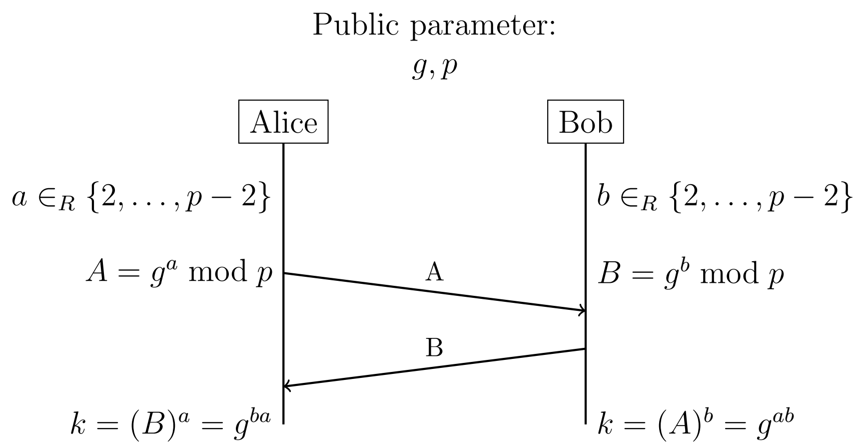


Figure 2.1: Alice and Bob's execution of Diffie-Hellman key exchange.

any other party) that he has knowledge of some secret which **only** Bob could possess. The goal, of course, being to accomplish this without openly revealing the secret, so that it can continue to be used as an identifier for Bob.

An identification scheme (or otherwise "proof of identity" protocol) Π_{id} is composed of by the tuple of polynomial-time algorithms (**KeyGen**, **Commit**, **Prove**, **Verify**) and some set ω. Π_{id} is an interactive protocol requiring two parties. The *prover* (Bob, for example) executes **KeyGen**, **Commit**, and **Prove**. The *verifier* (Alice, in our example) executes **Verify** following Bob's actions.

Execution of Π_{id} between Alice and Bob proceeds as follows:

(i) Bob runs **KeyGen**(1^λ): A probabilistic algorithm with input 1^λ and outputs Bob's keypair (sk, pk). Bob sends his public key pk to Alice.

(ii) Bob runs **Commit()**: a probabilistic algorithm with output *com*. *com* is referred to as a "commitment". Bob sends *com* to Alice.

(iii) Alice sends a randomly generated "challenge" value $ch \in \omega$. Alice sends ch to Bob.

(iv) Bob runs **Prove**$(sk$**,** com**,** $ch)$: A deterministic algorithm with input sk and ch, and output *resp*. *resp* is the "response" to Alice's challenge.

(v) Alice runs **Verify**$(pk$**,** com**,** ch**,** $resp)$: A deterministic algorithm with input pk, com, ch, and $resp$, and output $b \in 0, 1$. Bob has successfully proven his identity to Alice if $b = 1$.

If Alice accepts Bob's response, and $b = 1$, then we refer to the tuple $(com, ch, resp)$ as an *accepting transcript*. This general construction for identification protocols is illustrated in Figure 2.2, where the prover is referred to as $\mathcal{P}$ and $\mathcal{V}$ denotes the verifier.

In terms of security, it is common to show that an identification scheme is secure against *impersonation* under a *passive attack*. Proving such security implies that an adversary who eavesdrops on arbitrarily many executions of Π_{id} between a verifier $\mathcal{V}$ and a prover $\mathcal{P}$ cannot successfully impersonate $\mathcal{V}$.

We may at times speak of *canonical* identification schemes. An identification scheme Π occuring between a prover $\mathcal{P}$ and a verifier $\mathcal{V}$ is labelled canonical if it satisfies all of the following constraints:

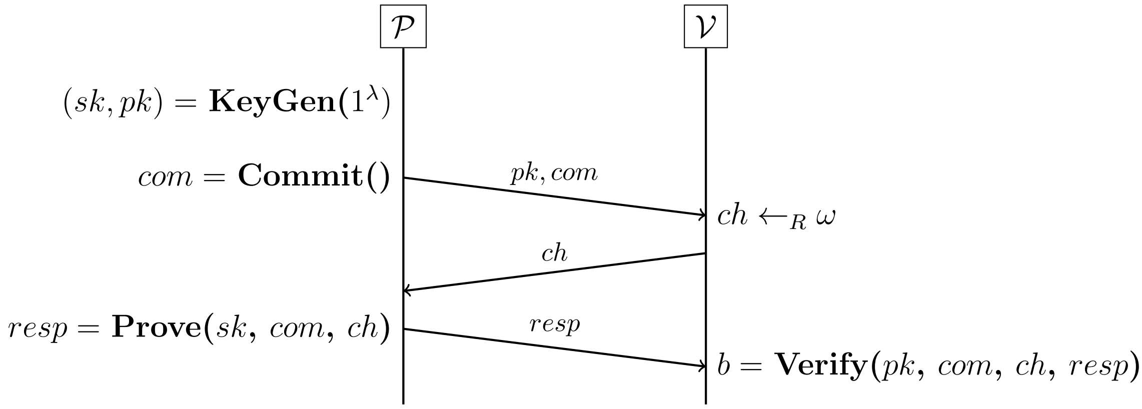


Figure 2.2: A general interactive identification scheme with prover $\mathcal{P}$ and verifier $\mathcal{V}$.

- Π consists of an initial message (or "commitment") *com* sent by $\mathcal{P}$, a challenge *ch* sent by $\mathcal{V}$, and a final response *resp* sent by $\mathcal{P}$.
- *ch* is chosen uniformly at random from some set ω.
- *com* is generated by some probabilistic function $\mathbf{R}$ taking $\mathcal{P}$'s secret key as input. For any secret key *sk* and fixed string $c\bar{o}m$, the probability that $\mathbf{R}(sk) = c\bar{o}m$ is negligible.

These constraints gurantee that Π will have two important features. First, that any third party given the transcript and prover's public key can efficiently determine whether the verifier will accept. Second, that the probability that *comm* repeats in polynomially many executions of Π is negligible.

Lastly, it should be mentioned that there exist variations upon this type of primitive wherein Alice is not required to send Bob a specific challenge value. These are known as *non-interactive* identification schemes, or non-interactive proofs of identity (NIPoI). These non-interactive approaches to solving the problem of identity and *authentication* further bridge the gap between identification protocols and signature schemes.

2.1.3 Signature Schemes

We define a signature scheme as the tuple of algorithms $\Pi_{sig} = (\mathbf{KeyGen}, \mathbf{Sign}, \mathbf{Verify})$. Some execution of Π_{sig} between Alice and Bob for a particular message m sent from Bob to Alice involves the following... First, before any message is to be signed, the Bob must run the following:

- Bob runs $\mathbf{KeyGen}(1^\lambda)$: A probabilistic algorithm with input 1^λ and output (sk, pk).

Then, for every message m Bob wishes to authenticate and send to Alice:

(i) Bob sends his public key *pk* to Alice over an authenticated channel if he has not yet done so.

(ii) Bob runs $\mathbf{Sign}(sk, m)$: A probabilistic algorithm with input *sk* (Bob's secret key) and m (the message Bob wishes to authorize) and output σ, known as a *signature*.

(iii) Bob sends m and σ to Alice.

(iv) Alice runs **Verify**(pk, m, σ): A deterministic algorithm with input pk (Bob's public key), m, and σ and output $b \in \{0, 1\}$. Alice has confidence in the *integrity* and origin *authenticity* of m if $b = 1$.

As previously alluded to, it is worth noting that signature protocols and identification schemes are closely related. In essence, they are rather similar; but with two main differences. The first is rather comparable to the aforementioned difference between interactive identification schemes and non-interactive identification schemes. The second arises as a result of aiming to authenticate Bob on any particular message m. To achieve this, the signature scheme needs to be run every time Bob wishes to send a message to Alice. The details of this comparison are intentionally left vague, as it will from a topic of close inspection in section 2.4.

The strongest security goal for a signature scheme Π_{sig} is expressed as *existential unforgeability* under an *adaptive chosen-message attack*. If this goal is provably satisfied, an adversary with the ability to sign arbitrary messages will not be able to forge any conceivable and valid signature.

2.2 Algebraic Geometry & Isogenies

Groups & Varieties. A **group** is a 2-tuple composed of a set of elements and a corresponding group operation (also referred to as the group *law*). Given some group defined by the set G and the operation $\cdot$ (written as $(G, \cdot)$) it is typical to refer to the group simply as G. If $\cdot$ is equivalent to some rational mapping[2] $f_G : G \to G$, then the group $(G, \cdot)$ is said to form an **algebraic variety**. A group which is also an algebraic variety is referred to as an **algebraic group**.

G is said to be an *abelian* group if, in addition to the four traditional group axioms (closure, associativity, existence of an identity, existence of an inverse), G satisfies the condition of commutitiviy. More formally, for some group G with group operation $\cdot$, we say G is an abelian group iff $x \cdot y = y \cdot x \ \forall x, y \in G$. An algebraic group which is also abelian is referred to as an **abelian variety**.

Definition 1 (Abelian Variety)**.** for some algebraic group G with operation $\cdot$, we say G is an abelian variety iff $x \cdot y = y \cdot x \ \forall x, y \in G$.

For some group $(G, \cdot)$, some $x, y \in G$, and some rational mapping $f_G : G \to G$, let the following sequence of implications denote the classification of $(G, \cdot)$:

$$\text{group} \xRightarrow{x \cdot y = f_G(x,y)} \text{algebraic group} \xRightarrow{x \cdot y = y \cdot x} \text{abelian variety}$$

Morphisms. Let us again take for example some group $(G, \cdot)$. Let's also define some set $S_{(G,\cdot)}$ which contains every tuple (x, y, z) for group elements x, y, z which satisfy $x \cdot y = z$.

$$S_{(G,\cdot)} = \{x, y, z \in G | x \cdot y = z\}$$

Take also for example a second group $(H, *)$ and some map $\phi : G \to H$. ϕ is said to be *structure preserving* if the following implication holds:

$$(x, y, z) \in S_{(G,\cdot)} \Rightarrow (\phi(x), \phi(y), \phi(z)) \in S_{(H,*)}$$

[2]A rational map is a mapping between two groups which is defined by a polynomial function with rational coefficients.

A **morphism** is simply the most general notion of a structure preserving map. More specifically, in the domain of algebraic geometry, we will be dealing with the notion of a **group homomorphism**, defined as follows:

> **Definition 2** (Group Homomorphism)**.** For two groups G and H with respective group operations $\cdot$ and $*$, a group homomorphism is a structure preserving map $h : G \to H$ such that $\forall u, v \in G$ the following holds:
>
> $$h(u \cdot v) = h(u) * h(v)$$

From this simple definition, two more properties of homomorphisms are easily deducible. Namely, for some homomorphism $h : G \to H$, the following properties hold:

- h maps the identity element of G onto the identity element of H, and
- $h(u^{-1}) = h(u)^{-1}, \forall u \in G$

Recall that for some morphism (or function) $h : G \to H$, we refer to G as the domain and H as the codomain.

Furthermore, an *endomorphism* is a special type of morphism in which the domain and the codomain are the same groups. We denote the set of enomorphisms defineable over some group G as $\mathrm{End}(G)$. The *kernel* of a particular homomorphism $h : G \to H$ is the set of elements in G that, when applied to h, map to the identity element of H. We write this set as $ker(h)$, and it is much analogous to the familiar concept from linear algebra, wherein the kernel denotes the set of elements mapped to the zero vector by some linear map.

2.2.1 Fields & Field Extensions

A **field** is a mathematical structure which, while being similar to a group, demands additional properties. Fields are defined by some set F and two operations: *addition* and *multiplication*. In order for some tuple $(F, +, \cdot)$ to constitute a field, it must satisfy an assortment of axioms:

Addition axioms:

- (closure) If $x \in F$ and $y \in F$, then $x + y \in F$.
- $+$ is commutative.
- $+$ is associative.
- F contains an element 0 such that $\forall x \in F$ we have $0 + x = x$.
- $\forall x \in F$ there is a corresponding element $-x \in F$ such that $x + (-x) = 0$.

Multiplication axioms:

- (closure) If $x \in F$ and $y \in F$, then $x \cdot y \in F$.
- $\cdot$ is commutative.

- $\cdot$ is associative.
- F contains an element $1 \neq 0$ such that $\forall x \in F$ we have $x \cdot 1 = x$.
- $\forall x \neq 0 \in F$ there is a corresponding element $x^{-1} \in F$ such that $x \cdot (x^{-1}) = 1$.

Additionally, a field $(F, +, \cdot)$ must uphold the *distributive law*, namely:

$$x \cdot (y + z) = x \cdot y + x \cdot y \text{ holds } \forall x, y, z \in F$$

While these axioms are known to be satisfied by the sets $\mathbb{Q}$, $\mathbb{R}$, and $\mathbb{C}$ with typically defined $+$ and $\cdot$, our focus will be on a particular class of field known as a *finite field*. Finite fields, as the name suggests, are fields in which the set F contains finitely many elements - we refer to the number of elements in F as the *order* of the field.

Let us take some prime number p. We can construct a finite field by taking F as the set of numbers $\{0, 1, ...p-1\}$ and defining $+$ and $\cdot$ as addition and multiplication *modulo* p. Finite fields defined in this fashion are denoted as $\mathbb{F}_p$, and have order p.

$$\forall x, y \in \mathbb{F}_p,\ x + y = (x + b) \mod p, \text{ and}$$
$$\forall x, y \in \mathbb{F}_p,\ x \cdot y = (x \cdot b) \mod p$$

For any given field K there exists a number q such that, for every $x \in K$, adding x to itself q times results in the additive identity 0. This number is referred to as the *characteristic* of K, for which we write $\mathrm{char}(K)$. Finite fields are the only type of field for which $\mathrm{char}(K) > 0$. Furthermore, if the field in question is finite and has prime order, then the order and the characteristic are equivalent.

A particular field K' is called an *extension field* of some other field K if $K \subseteq K'$. The complex numbers $\mathbb{C}$, for example, are an extension field of $\mathbb{R}$. A given field K is *algebraically closed* if there exists a root for every non-constant polynomial defined over K. If K itself is not algebraically closed, we denote the extension of K that is by $\overline{K}$.

An algebraic group G_a is defined over a field K if each element $e \in G_a$ is also an element of the field K, and the corresponding f_{G_a} is defined over K. To show that a particular algebraic group G_a is defined over some field K we will henceforth denote the group/field pairing as $G_a(K)$. Naturally, in the case where our field is a finite field of order p, we write $G_a(\mathbb{F}_p)$.

These algebraic structures are all important for building up to the concept of an *isogeny*. The lowest-level object we will be concerned with when discussing the forthcoming isogeny-based protocols will typically be elements of abelian varieties. The lowest-level structure in the SIDH C codebase is a finite field element.

Montgomery Arithmetic. We will now briefly discuss a technique for efficiently performing modular arithmetic. This method is widely deployed in cryptosystems centered around finite fields, and is abundantly used in the $\mathtt{SIDH_C}$ library that we will shortly be examining.

In 1985, Peter Montgomery introduced a method for efficiently computing the modular multiplication of two elements a and b. The technique begins with the construction of some constant R, whose value depends solely on the modulus N and the underlying computer architecture.[3]

[3]The specifics of how R is constructed are beyond the scope of this dissertation; if they feel so inclined, the reader should refer to [Mon85].

With the retrieval of R, $aR \mod N$ and $bR \mod N$ are constructed and referred to as the Montgomery *representation* of a and b respectively. Montgomery multiplication outlines an algorithm for computing $abR \mod N$ (the Montgomery product of a and b), from which $ab \mod N$ can be recovered through conversion back to standard representation. Once in Montgomery representation, other arithmetic can be performed (including field element inversions) in order to leverage the performance improvement offered by Montgomery modular multiplication – converting back to regular representation when necessary.

Applying Montgomery multiplication has the added benefit of decreasing the amount of field element inversions that need to be computed. Because of this, the technique is particularly relevant to this dissertation. We continue this discussion in section 3.2.

2.2.2 Elliptic Curves

An elliptic curve is an algebraic curve defined over some field K, the most general representation of which is given by

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6.$$

This representation encapsulates elliptic curves defined over any field. If, however, we are dicussing curves defined specifically over a field K such that $\text{char}(K) > 3$[4], then the more compact form $y^2 = x^3 + ax + b$ can be applied (see Figure 2.3 for a geometric visualization). In this dissertation we will default to this second representation, as the schemes with which we are concerned will always be defined over $\mathbb{F}_p$ for some large prime p.

We can define a group structure over the points of a given elliptic curve (or any other smooth cubic curve). If we wish to define a group in accordance to a particular curve, we do so with the following notation:

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

Wherein E denotes the group in question, the elements of which are all the points (solutions) of the curve. Throughout much of this section, the words *point* and *element* can be used interchangeably.

The Group Law. The group operation we define for E, denoted $+$, is better understood geometrically than algebraically. Consider the following.

Given two elements P and Q of some arbitrary elliptic curve group E, we define $+$ geometrically as follows: drawing the line L through points P and Q, we follow L to its third intersection on the curve (which is guaranteed to exist), which we will denote as $R = (x_R, y_R)$. We then set $P + Q = -R$, where $-R$ is the reflection of R over the x-axis: $(x_R, -y_R)$. This descriptive definition of $+$ is suitable for all situations *except* for when L is tangent to E or when L is parallel to the y-axis. These cases will be covered in a short moment. See Figure 2.3 for an illustrated representation of this process.

The group operation $+$ is referred to as *pointwise addition*. In order for $(E, +)$ to properly form a group under pointwise addition, it must satisfy the four group axioms:

[4] See [Sil].

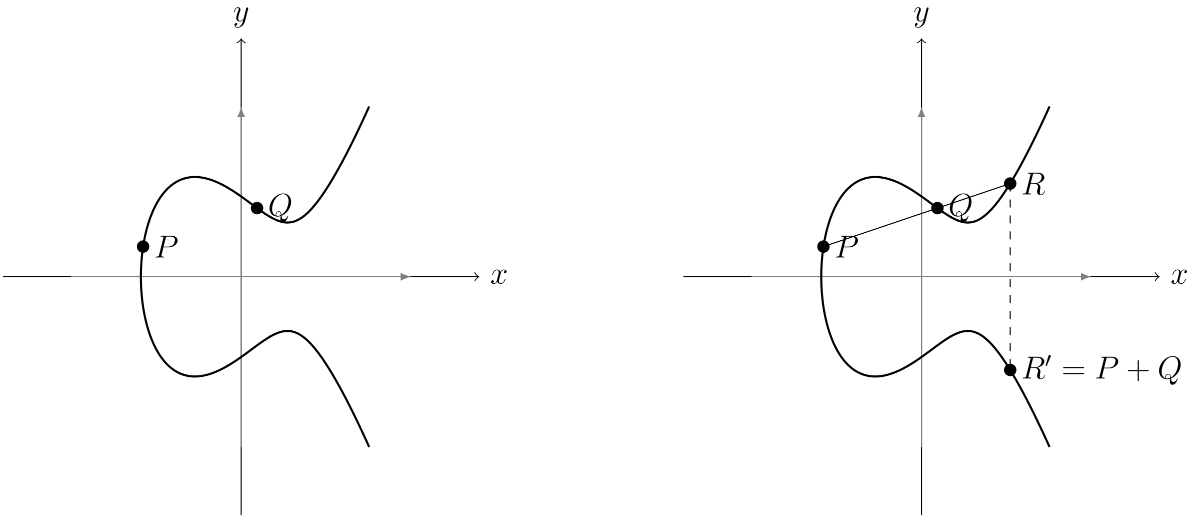


Figure 2.3: + acting over points P and Q of $y^2 = x^3 - 2x + 2$.

- *Closure*: Because elliptic curves are polynomials of degree of 3, we know any given line passing through two points P and Q of E will pass through a third point R. The exceptions here are twofold. First, when $P = Q$ and thus our line is tangent to E, and second, when $Q = -P$ and our line is parallel with the y-axis. We resolve the first case nicely by defining $P + P$ by means of taking L to be the line tangent to E at point P. In the second case, $P + (-P)$, by group axiom, should yield the identity element of the group. We will define this element and resolve this issue below.

- *Identity*: The identity element of elliptic curve groups, denoted as $\mathcal{O}$, is a specially defined point satisfying $P + \mathcal{O} = \mathcal{O} + P = P$, $\forall P \in E$. Because of the inclusion of this special element, we have that $\#(E(K))$ is equal to 1 + the number geometric points on E defined over K. This of course is only a noteworthy claim when K is a finite field (otherwise there are already infinitely many elements in E).

- *Associativity*: For all points P, Q, and R in E, it must be the case $((P+Q)+R = P+(Q+R))$ holds. It is rather easy to see visually why this is true for geometrically defined points in E (see Figure 2.4). Additionally, we can trivially show that this holds when any combination of P, Q, and R are $\mathcal{O}$ by applying the axiom of the identity.

- *Inverse*: Due to the x-symmetry of elliptic curves, every point $P = (x_P, y_P)$ of E has an associated point $-P = (x_P, -y_P)$. If we apply + to P and $-P$, L assumes the line parrallel to the y-axis at $x = x_P$. As discussed above, in this case there is no third intersection of L on E. In light of this, $\mathcal{O}$ can be thought of as a point residing infinitely far in both the positive and negative directions of the y-axis. $\mathcal{O}$ is equivalently referred to as the *point at infinity* (see Figure 2.4).[5]

[5] One might suspect that the inclusion of this (apparently) non-algebraic element $\mathcal{O}$ suggests that + is not a rational-map. The operator + *can* be shown to be a rational-mapping if we define our elliptic curve groups in three-dimensional projective space.

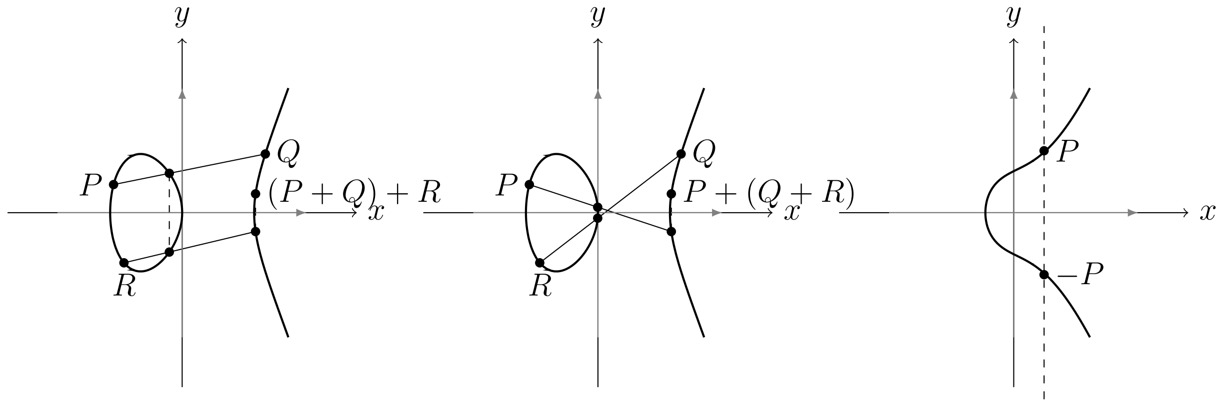


Figure 2.4: associativity illustrated on $y^2 = x^3 - 3x$ (left & center) and $P + (-P) = \mathcal{O}$ illustrated for $y^2 = x^3 + x + 1$ (right).

Of course, there are relatively simple formulas for algebraically defining point-wise addition and inverse computation. We have opted to describe these operations geometrically simply for ease of communication.

Additionally, we shorthand $\overbrace{P + P + ... + P}^{n}$ as nP, analogous to scalar multiplication.

Consequently, because groups defined over elliptic curves in this fashion are commutitive, they also constitute abelian varieties.

When referring to curves as abelian varieties defined over a field, we will write them as $E_\alpha(K)$, for some curve E_α and some field K. If we are only concerned with the geometric properties of the curve, or curves as distinct elements of some group structure, it will suffice to write E_α. Moving forward from here, we will assume all general curves discussed are capable of definition over some finite field $\mathbb{F}_p$.

The *r-torsion group* of E is the set of all points $P \in E(\overline{\mathbb{F}}_q)$ such that $rP = \mathcal{O}$. We denote the r-torsion group of some curve as $E[r]$.

Supersingular Curves. An elliptic curve can be either *ordinary* or *supersingular*. There are several equivalent ways of defining supersingular curves (and thus the distinction between them and ordinary curves) in a general setting, but each of these goes well beyond our scope. In the context of curves defined over finite fiels, however, the following succinct definition holds:

Definition 3 (Supersingular Curve)**.** Let E be an elliptic curve defined over the finite field $\mathbb{F}_p$. E is said to be supersingular if $\#(E(\mathbb{F}_p)) = p + 1$.[a]

[a]Readers are welcomed to investigate [Cos] for further details.

For the remainder of this paper, unless otherwise noted, all elliptic curves in discussion will supersingular.

Projective Space. While elliptic curves are naturally defined in two-dimensional affine space, there are many benefits to expressing them through three-dimension projective coordinates. First and foremost, expressing curves in projective space allows us to reason geometrically about $\mathcal{O}$. This is done by defining a curve E such that it resides in some two-dimension subspace of 3-space, the point $\mathcal{O}$ then resides at some point in 3-space outside of the residing plane of E.

Representing a curve in 3-space requires some substitution of x and y coordinates, a typical forma for achieving this is the following:

$$x = X/Z \quad y = Y/Z \quad Z = 1$$

Such a representation of elliptic curve points offers more computationally efficient arithmetic over points. This is conceptually similar to the previously mentioned Montgomery arithmetic regarding finite field elements. Other substitutions offer different computational advantages, but the implementations we will discuss make use of this typical approach.

2.2.3 Isogenies & Their Properties

Definition 4 (Isogeny)**.** Let G and H be algebraic groups. An isogeny is a morphism $h : G \to H$ possessing a finite kernel.

In the case of the above definition where G and H are abelian varieties (such as elliptic curves,) the isogeny h is homomorphic between G and H. Because of this, isogenies over elliptic curves (and other abelian varieties) inherit certain characteristics.
For an isogeny $h : E_1 \to E_2$ defined over elliptic curves E_1 and E_2, the following holds:

- $h(\mathcal{O}) = \mathcal{O}$, and
- $h(u^{-1}) = h(u)^{-1}, \forall u \in G$

If there exists some isogeny ϕ between curves E_1 and E_2 then E_1 and E_2 are said to be *isogenous*. All supersingular curves are isogenous only to other supersingular curves. The equivalent statement holds for ordinary curves. With this in mind, we can concieve a sort of graph structure connecting all isogenous curves, these graphs pertaining to either the supersingular or ordinary variety of curves.

We write $\text{End}(E)$ to denote the ring formed by all the isogenies acting over E which are also endomorphisms. Note that m-repeated pointwise addition of a point with itself can equivalently be modelled by an endomorphism, we denote the application of such an endomorphism to a point P as $[m]P$, such that $[m] : E \to E$ and $[m]P = mP$.

An important facet of isogenies is that they can be uniquely identified by their kernel. If S is the group of points denoting the kernel of some isogeny ϕ with domain E, we write $\phi : E \to E/S$. Because the subgroup S sufficiently identifies ϕ, any given generator of S equivalently identifies ϕ. Therefore, if R generates the subgroup S we can write $\phi : E \to E/\langle R \rangle$. Moreover, we will have a specific interest in isogenies with kernels defined by some *torsion subgroup*.

Lemma 1 (Uniquely identifying isogenies)**.** *Let E be an elliptic curve and let Φ be a finite subgroup of E. There are a unique elliptic curve E' and a seperable isogeny $\phi : E \to E'$ satisfying ker$(\phi) = \Phi$.*

2.3 Supersingular Isogeny Diffie-Hellman

This section will aim to accomplish two things. First, we will briefly explain the isogeny-level & key-exchange-level procedures of the SIDH protocol. Second, we will illuminate how these procedures map onto Microsoft Research's C implementation of SIDH. In this regard, this section can be considered an attempt to meld two domains of SIDH functions & procedures, in hopes of easing the navigation from the SIDH protocol to Microsoft's C implementation, and vice versa.

The original work of De Feo, Jao, and Plut [FJP12] outlines three different isogeny-based cryptographic primitives: Diffie-Hellman-esque key exchange, public key encryption, and the aforementioned zero-knowledge proof of identity (ZKPoI). Because all three of these protocols require the same initialization and public parameters, we will begin by covering these parameters in detail. Immediately after, we will analyze the key exchange at a relatively high level. Our goal of this section is to explain in detail the algorithmic and cryptographic aspects of the ZKPoI scheme, as this forms the conceptual basis for the signature scheme we will be investigating. We begin with the key exchange protocol because its sub-routines are integral to the Yoo et al. signature implementation.

For the discussion that follows, we will assume every instance of an SIDH protocol occurs between two parties, A and B (eg. Alice & Bob,) for which we will colorize information particular to A in red and B in blue. This will include private keys & public keys as well as the variables and constants used in their generation.

Public Parameters. As the name suggests, SIDH protocols work over supersingular curves (with no singular points). Let $\mathbb{F}_q = \mathbb{F}_{p^2}$ be the finite field over which our curves are defined, $\mathbb{F}_{p^2}$ denoting the quadratic extension field of $\mathbb{F}_p$. p is a prime defined as follows:

$$p = \ell_A^{e_A} \ell_B^{e_B} \cdot f \pm 1$$

Wherein ℓ_A and ℓ_B are small primes (typically 2 & 3, respectively) and f is a cofactor ensuring the primality of p. We then define globally a supersingular curve E_0 defined over $\mathbb{F}_q$ with cardinality $(\ell_A^{e_A} \ell_B^{e_B} f)^2$. Consequently, the torsion group $E_0[\ell_A^{e_A}]$ is $\mathbb{F}_q$-rational and has ${\ell_A}^{e_A - 1}(\ell_A + 1)$ cyclic subgroups of order $\ell_A^{e_A}$, with the analogous statement being true for $E_0[\ell_B^{e_B}]$. Additionally, we include in the public parameters the bases $\{P_A, Q_A\}$ and $\{P_B, Q_B\}$, generating $E[\ell_A^{e_A}]$ and $E[\ell_B^{e_B}]$ respectively.

This brings our set of global parameters, G, to the following:

$$G = \{p, E_0, \ell_A, \ell_B, e_A, e_B, \{P_A, Q_A\}, \{P_B, Q_B\}\}$$

2.3.1 SIDH Key Exchange

This subsection will illustrate an SIDH key exchange run between party members Alice and Bob. The general idea of the protocol is summerized in the diagram below. In the scheme, **private keys** take the form of isogenies defined with domain E, and **public keys** are the associated co-domain curve of said isogenies.[FJP12] For the entirety of this section we will denote isogenies by their function symbols, but when we come section 2.6 to will show how we can efficiently represent isogenies in a computational environment.

$$\begin{array}{ccc} E_0 & \xrightarrow{\phi_A} & E_0/\langle A\rangle \\ \phi_B \downarrow & & \downarrow \phi'_B \\ E_0/\langle B\rangle & \xrightarrow[\phi'_A]{} & E_0/\langle A, B\rangle \end{array}$$

The premise of the protocol is that both parties each generate a random point (A or B in the diagram,) which, according to proposition 1, identifies some distinct isogeny $\phi_A : E_0 \to E/\langle A\rangle$ (or equivalent for B). Alice and Bob then exchange codomain curves and compute

$$\phi_A(E_0/\langle B\rangle)$$
or
$$\phi_B(E_0/\langle A\rangle).$$

From these isogenies, Alice and Bob arrive at their shared secret agreement: the mutual codomain curve of $\phi_A(E_0/\langle B\rangle)$ (equivalently $\phi_B(E_0/\langle A\rangle)$), denoted E_{AB}.

Below we've outlined the SIDH key exchange protocol $\Pi_{\text{SIDH}} = (\textbf{KeyGen}, \textbf{SecAgr})$ in a descriptive manner. We do not provide algorithmic definitions for all of these procedures, but algorithmic details for some of these are covered partly in Sections 2.6 and 3.2. C code for functions that are not covered in these Sections but are noneless relevant can be found in Appendix A.

KeyGen(λ): Alice chooses two random numbers $m_A, n_A \in \mathbb{Z}/\ell_A^{e_A}\mathbb{Z}$ such that $(\ell_A \nmid m_A) \vee (\ell_A \nmid n_A)$. Alice then computes the isogeny $\phi_A : E_0 \to E_A$ where $E_A = E_0/\langle [m_A]P_A, [n_A]Q_A\rangle$ (equivalently, $ker(\phi_A) = \langle [m_A]P_A, [n_A]Q_A\rangle$). Bob does the same for random elements $m_B, n_B \in \mathbb{Z}/\ell_B^{e_B}\mathbb{Z}$.

Alice then applies her isogeny to the points which Bob will use in the creation of of his isogeny: $(\phi_A(P_B), \phi_A(Q_B))$. Bob performs the analogous operation. This leaves us with the following private and public keys for Alice and Bob:

$$\begin{aligned} sk_A &= (m_A, n_A) \\ pk_A &= (E_A, \phi_A(P_B), \phi_A(Q_B)) \\ sk_B &= (m_B, n_B) \\ pk_B &= (E_B, \phi_B(P_A), \phi_B(Q_A)) \end{aligned}$$

PK Exchange: After Alice and Bob successfully complete their key generation, they perform the following over an insecure channel:

- Alice sends Bob $(E_A, \{\phi_A(P_B), \phi_A(Q_B)\})$
- Bob sends Alice $(E_B, \{\phi_B(P_A), \phi_B(Q_A)\})$

Again, we remind the reader that we will show how curves such as E_A and E_B can be represented efficient and compactly in a computing environment when we come to our section on implementations of isogeny-based systems (2.6).

SecAgr(sk_1,pk_2): After reception of Bob's tuple, Alice computes the isogeny $\phi'_A : E_B \rightarrow E_{AB}$ and Bob acts analogously. Alice and Bob then arrive at the equivalent image curve:

$$E_{AB} = \phi'_A(\phi_B(E_0)) = \phi'_B(\phi_A(E_0)) = E_0/\langle [m_A]P_A + [n_A]Q_A, [m_B]P_B + [n_B]Q_B\rangle$$

From this they can derive their shared secret k as the common j-invariant of E_{AB}.

We have included a graphical illustration of the entire SIDH key exchange process in Figure 2.5, wherein solid lines denote private computations, and dashed lines denote information sent over an insecure channel.

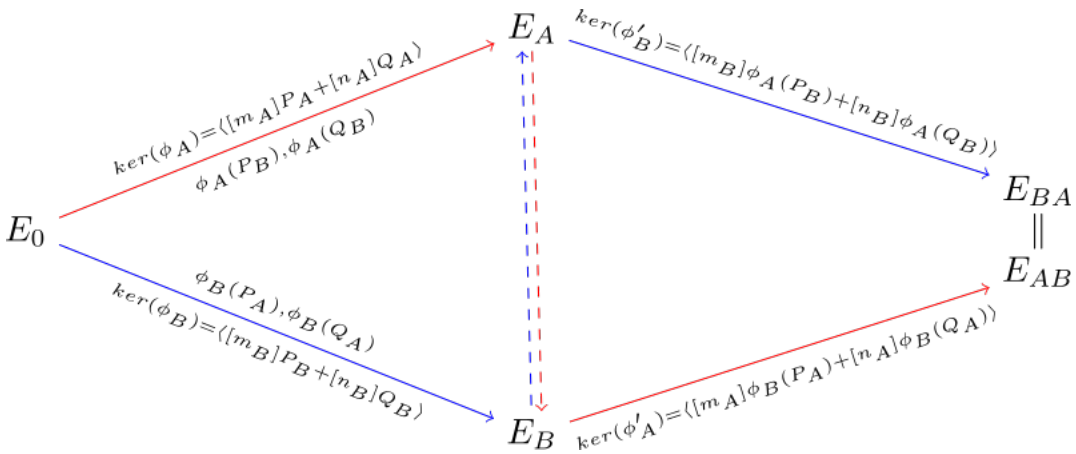


Figure 2.5: SIDH key exchange between Alice & Bob [FJP12]

2.3.2 Zero-Knowledge Proof of Identity

Recall the earlier discussed notion of an identification scheme. A canonical identification scheme $\Pi_{\text{SID}} = (\textbf{KeyGen}, \textbf{Prove}, \textbf{Verify})$ can be derived somewhat analogously to the SIDH protocol, and is outlined in the original work of De Feo et al.

Say Bob has derived for himself the key pair (sk_B, pk_B) with $sk_B = \{m_B, n_B\}$ and $pk_B = E_B = E_0/\langle [m_B]P_B + [n_B]Q_B\rangle$ in relation to the public parameters E_0 and $\ell_B^{e_B}$. With E_0 and E_B publicly known, Π_{ZKPoI} revolves around Bob trying to prove to Alice that he knows the generator for E_B without revealing it.

To achieve this, Bob internally mimicks an execution of the key exchange protocol Π_{SIDH} with an arbitrary "random" entity Randall.

KeyGen: Key generation is performed exactly as in Π_{SIDH}, the only difference being that in Π_{ZKPoI} only the prover (Bob, in our example,) needs to generate a keypair.

Commitment: Bob generates a random point $R \in E_0[\ell_A^{e_A}]$ $(R = [m_R]P_A + [n_R]Q_A)$ along with the corresponding isogenies necessary to compute the diagram below in full (if Alice were acting as the prover in Π_{ZKPoI}, then she would choose $R \in E_0[\ell_B^{e_B}]$). Bob sends his commitment *com* as $(com_1, com_2) = (E/\langle R\rangle, E/\langle B, R\rangle)$ to Alice.

$$\begin{tikzcd}
E_0 \arrow[r, "\phi_B"] \arrow[d, "\psi_R"'] & E_0/\langle B\rangle \arrow[d, "\psi'_R"] \\
E_0/\langle R\rangle \arrow[r, "\phi'_B"'] & E_0/\langle B, R\rangle
\end{tikzcd}$$

Response: Alice chooses a bit b at random and sends her challenge $ch = b$ to Bob.

Prove(sk, ch): If Alice's challenge bit $ch = 0$ then Bob reveals the isogenies ψ_R and ψ'_R (to do this, he can simply reveal the generators of the kernels of ψ_R and ψ'_R; R and $\phi_B(R)$ respectively). This proves he knows the information necessary to form a shared secret with Randall *if and only if* he happens to know the private key $B = \{[m_B]P_B + [n_B]Q_B\}$. If $ch = 1$, Bob reveals the isogeny ϕ'_B. This proves that Bob knows the information necessary to form a shared secret with Randall *if and only if* he knows Randall's secret key R.

In the following two graphs, bold arrows are used to indicate the information revealed by Bob. The graph on the left corresponds to Bob's actions when $ch = 0$, the graph on the right shows the information revealed when $ch = 1$.

$$\begin{tikzcd}
E_0 \arrow[r, dash, dashed, "\phi_B"] \arrow[d, Rightarrow, "\psi_R"'] & E_0/\langle B\rangle \arrow[d, Rightarrow, "\psi'_R"] \\
E_0/\langle R\rangle \arrow[r, dash, dashed, "\phi'_B"'] & E_0/\langle B, R\rangle
\end{tikzcd}
\qquad
\begin{tikzcd}
E_0 \arrow[r, dash, dashed, "\phi_B"] \arrow[d, dash, dashed, "\psi_R"'] & E_0/\langle B\rangle \arrow[d, dash, dashed, "\psi'_R"] \\
E_0/\langle R\rangle \arrow[r, Rightarrow, "\phi'_B"'] & E_0/\langle B, R\rangle
\end{tikzcd}$$

Note that Bob cannot at once reveal all of the information necessary to convince Alice that he knows B. If he reveals R, $\phi_B(R)$, and ϕ'_B all in one go, he incidentally reveals his secret key $B = [m_B]P_B + [n_B]Q_B$. This is because Bob reveals ϕ'_B by revealing the generator of $ker(\phi'_B)$, namely:

$$(B, R) = ([m_B]P_B + [n_B]Q_B, [m_R]P_A + [n_R]Q_A)$$

How Π_{ZKPoK} handles this is by having Bob and Alice run **Prove()** and **Verify()** for λ iterations, with a different $(com, ch, resp)$ transcript generated for every instance. This way, if Bob is able to provide a *resp* that satisfies Alice's *ch* for every iteration, she can be sufficiently confident that Bob has knowledge of B. By taking this approach, Alice gains no knowledge of Bob's secret key (e.g. "zero-knowledge").[6]

Verify(pk, com, ch): Like the proving procedure, verification is a conditional function depending on the value of b:

- if $ch = 0$: return 1 *if and only if* R and $\phi_B(R)$ have order $\ell_A^{e_A}$ and generate the kernels of isogenies from $E_0 \to E_0/\langle R\rangle$ and $E_0/\langle B\rangle \to E_0/\langle B, R\rangle$ respectively.

[6]This iterative approach to a zero-knowledge proof of knowledge is well illustrated by the "Ali Baba Cave" anecdote: `https://en.wikipedia.org/wiki/Zero-knowledge_proof#Abstract_examples`.

- if $ch = 1$: return 1 *if and only if* $\psi_R(B)$ has order $\ell_B^{e_B}$ and generates the kernel of an isogeny over $E_0/\langle R\rangle \to E_0/\langle B, R\rangle$.

This scheme constitutes what is known in the literature as a *zero knowledge* proof of identity. It is referred to as such because Alice, acting as the verifier, does not gain any information about Bob's secret key sk.

2.4 Fiat-Shamir Construction

The Fiat-Shamir construction (also frequently referred to as the Fiat-Shamir transform, or Fiat-Shamir hueristic,) is a high-level technique for transforming a canonical identification scheme into a secure signature scheme.

The construction is rather simple. The idea is to first transform a given interactive identification protocol Π_{ID} into a *non-interactive* identification protocol. To achieve this, instead of allowing input from the verifier $\mathcal{V}$, we have our prover $\mathcal{P}$ generate the challenge ch by itself. In order for the verifier to be able to check that ch was generated honestly, we define $ch = H(com)$, where H is some secure hash function. If we model H as a random oracle, $H(com)$ is assumed truly random; from this it can be shown that it is just as difficult for an impersonator of $\mathcal{P}$ to find an accepting transcript $(com, H(com), resp)$ as it would be for them to successfully impersonate $\mathcal{P}$ in Π_{ID}.

Now that we've paired Π_{ID} with H to achieve a non-interactive identification scheme Π_{NID}, we need only to factor in some message m from $\mathcal{P}$ to have constructed a signature scheme Π'_{ID}. This can be achieved by including m in our calculation of the challenge: $ch = H(com, m)$. Therefore, given theorem 1, if $(com, H(com), resp)$ is an accepting transcript of Π_{NID}, then $(com, H(com, m), resp)$ is a secure signature for the message m. Of course, because $H(com, m)$ can be constructed by any passively observing party, it is redundant to include; and so $(com, resp)$ constitutes a valid signature for m. A proof of theorem 1 can be found in [Kat10]. The security of the Fiat-Shamir construction was first proven by Pointcheval & Stern in [PS96].

Theorem 1 (Fiat-Shamir Security)**.** *Let* $\Pi_{ID} = (\textbf{\textit{KeyGen}}, \textbf{\textit{Commit}}, \textbf{\textit{Prove}}, \textbf{\textit{Verify}})$ *be a canonical identification scheme that is secure against a passive attack. Then, if H is modeled as a random oracle, the signature scheme Π'_{ID} that results from applying the Fiat-Shamir transform to Π_{ID} is* classically *existentially unforgeable under an adaptive chosen-message attack.*

We will write $\mathbf{FS}(\Pi)$ to denote the result of applying the Fiat-Shamir transformation to some identification protocol Π.

2.4.1 Unruh's Post-Quantum Adaptation

In 2014, Ambainis et al. showed in [ARU14] that classical security proofs for "proof of knowledge" protocols are insecure in the quantum setting. This is due to a technique used in the proof of FST's security whereby the random oracle is subject to "rewinding": the proof simulates multiple runs of FST with different responses from the random oracle [ARU14].

Following this insight, Unruh proposed in [Unr14] a construction based off that of Fiat & Shamir which he proved to be secure in both the classical and quantum random oracle models.

Unruh's construction demands a small addition to the proof and verification procedures. In **Prove**, for every possible challenge value $ch_0, ch_1, ..., ch_n$, Unruh's construction demands that a hash of the corresponding responses $resp_0, resp_1, ..., resp_n$, along with the possible challenge values themselves, be included as input to the hash function H computing the actual challenge. While Unruh originallly presented this technique in a generalized setting with n possible challenge values, In figure **??** we detail a version that assumes there are only two possible challenge values. This is done in an attempt to more closely reflect the zero-knowledge proof of identity scheme presented in [FJP12].

The construction is given in the form of two procedures: $\mathbf{Prove_{Un}}$ and $\mathbf{Verify_{Un}}$. Given the proving procedure $\mathbf{P}_\Pi$ of some canonical identification scheme Π, $\mathbf{Prove_{Un}}$ can be constructed and forms the basis for the **Sign** procedure of a quantum-safe signature scheme $\mathbf{Un}(\Pi)$. Analagously, given the verification procedure $\mathbf{V}_\Pi \in \Pi$, $\mathbf{Verify_{Un}}$ details the outline of signature verification in $\mathbf{Un}(\Pi)$.

Similar to above, we will write $\mathbf{Un}(\Pi)$ to denote the result of applying Unruh's construction to some identification protocol Π.

Algorithm 2 – $\mathbf{Prove_{Un}(P_\Pi)}$

if $User = Alice$ **then**
    Pick a random point S of order $\ell_A^{e_A}$
if $User = Bob$ **then**
    Pick a random point S of order $\ell_B^{e_B}$
Compute the isogeny $\phi : E \to E/\langle S \rangle$
$pk \leftarrow (E/\langle S \rangle, \phi(P_{User}), \phi(Q_{User}))$
$sk \leftarrow S$
return (sk, pk)

Algorithm 3 – $\mathbf{Verify_{Un}(V_\Pi)}$

if $User = Alice$ **then**
    Pick a random point S of order $\ell_A^{e_A}$
if $User = Bob$ **then**
    Pick a random point S of order $\ell_B^{e_B}$
Compute the isogeny $\phi : E \to E/\langle S \rangle$
$pk \leftarrow (E/\langle S \rangle, \phi(P_{User}), \phi(Q_{User}))$
$sk \leftarrow S$
return (sk, pk)

2.5 Isogeny-based Signatures

Since publication of the SIDH suite, there have been several attempts at providing authentication schemes using the same primitives. The post-quantum community had demonstrated undeniable signatures [JS14], designated verifier signatures [STW12], and undeniable blind signatures [SC16] all within the framework of isogeny-based systems. It was not until the work of Yoo et al. (([YAJ+12])), however, that an isogeny-based protocol for general authentication was shown as demonstrably secure. This protocol, particularly its C implementation, is where we have decided to focus our efforts.

Now that we've seen the zero-knowledge proof of identity (ZKPoI) from [FJP12] as well as Unruh's quantum-safe Fiat-Shamir adaption, we have presented all of the material necessary for an indepth analysis of the isogeny-based signature scheme presented by Yoo et al. The signature protocol, which we'll denote as Σ', is an application of Unruh's construction to the SIDH ZKPoI. In this section we will refer to the SIDH ZKPoI as Σ.

Σ' is defined in the traditional manner, by a tuple of algorithms for key generation, signing, and verifying: $\Sigma' = (\textbf{KeyGen()}, \textbf{Sign()}, \textbf{Verify()})$. **KeyGen()** in Σ' is defined identically to the key generation found in SIDH key exchange. **Sign()** and **Verify()** are defined by applying Unruh's transformation to **Prove()** and **Verify()**.

For our discussion of the signature scheme, we will make use of the naming conventions used in Section 2.3. That is, we will discuss Σ' as occuring between entities Bob and Alice, with Bob imitating the role of an arbitrary third party Randall during **Sign**.

The public parameters used in Σ' are the same as outlined above for all of the protocols found in [FJP12]. Namely, we have $p = \ell_A^{e_A}\ell_B^{e_B} \cdot f \pm 1$ where $\ell_A^{e_A} = 2$, $\ell_B^{e_B} = 3$, and f is a cofacter such that p is prime. We also set as parameter the curve E such that $\#(E(F_{p^2})) = (\ell_A^{e_A}\ell_B^{e_B})^2$. And again, we include the sets of points $\{P_A, Q_A\}$ and $\{P_B, Q_B\}$ generating $E[\ell_A^{e_A}]$ and $E[\ell_B^{e_B}]$ respectively. We have chosen E over the previously used E_0 simply for ease of notation.

2.5.1 Algorithmic Definitions

It will be useful for us to outline in more detail the procedures of Σ', at the very least to ease the transition into our discussion of the C implementation. In this subsection we will look at isogeny-level algorithmic definitions for **KeyGen()**, **Prove()**, and **Verify()**, and then look at how these procedures can be expressed in terms of the procedures of Π_{SIDH}.

KeyGen(λ, $User$): As previously mentioned, key generation in Σ' is identical to Σ:**KeyGen(λ)**, which in turn is identical to Π_{SIDH}:**KeyGen(λ)**. We've included a parameter $User$ equaling either *Alice* or *Bob* – this denotes whether the user running the procedure uses blue or red constants. We've also obfuscated the lower level details in regards to how points are generated and how isogenies can be constructed. We write P_{User} and Q_{User} for P_A & Q_A or P_B & Q_B, depending on $User$. The result is the following:

Algorithm 4 – **KeyGen(λ, $User$)**

1: **if** $User$ = *Alice* **then**
2:     Pick a random point S of order $\ell_A^{e_A}$
3: **if** $User$ = *Bob* **then**
4:     Pick a random point S of order $\ell_B^{e_B}$
5: Compute the isogeny $\phi : E \to E/\langle S \rangle$
6: $pk \leftarrow (E/\langle S \rangle, \phi(P_{User}), \phi(Q_{User}))$
7: $sk \leftarrow S$
8: **return** (sk, pk)

Transcribing this to the procedures of Π_{SIDH} we arrive (quite trivially) at:

Algorithm 5 – $\mathbf{KeyGen}(\lambda,\ User)$ via Π_{SIDH}

$(sk,\ pk) \leftarrow \Pi_{\text{SIDH}}$:$\mathbf{KeyGen}(\lambda,\ User)$
return $(sk,\ pk)$

For $\mathbf{Sign}(sk,\ m)$ and $\mathbf{Verify}(pk,\ m,\ \sigma)$ we assume Bob to be the signer and Alice to be the verifier. Consequently, we will write the signer's key pair $(sk,\ pk)$ as $(B,\ \phi_B)$. Algorithms for which the roles are reversed can be constructed simply by replacing red constants with their blue correspondants, and vice-versa.

$\mathbf{Sign}(sk,\ m)$: The sign procedure, as a consequence of the Unruh construction, makes use of two random oracle functions $\mathbf{H}$ and $\mathbf{G}$. In the sign algorithm below, make note of how Bob computes both commitments and their corresponding responses for every iteration i before he computes the challenge values (the bits of J). He then uses the 2λ bits of J to decide which responses to include in σ.

Algorithm 6 – $\mathbf{Sign}(sk = B,\ m)$

for `i = 1..2`λ **do**
  Pick a random point R of order $\ell_A^{e_A}$
  Compute the isogeny $\psi_R : E \rightarrow E/\langle R\rangle$
  Compute the isogeny $\phi'_B : E/\langle B\rangle \rightarrow E/\langle B, R\rangle$
  $(E_1, E_2) \leftarrow (E/\langle R\rangle, E/\langle R, B\rangle)$
  $com_i \leftarrow (E_1, E_2)$
  $ch_{i,0} \leftarrow_R \{0,1\}$
  $ch_{i,1} \leftarrow 1 - ch_{i,0}$
  $(resp_{i,0}, resp_{i,1}) \leftarrow ((R, \phi_B(R)), \psi_R(B))$
  if $\mathtt{ch}_{i,0} = 1$ **then**
    $\mathbf{Swap}(resp_{i,0}, resp_{i,1})$
  $h_{i,j} \leftarrow \mathbf{G}(resp_{i,j})$
$J_1 \parallel \ldots \parallel J_{2\lambda} \leftarrow \mathbf{H}(\phi_B,\ m,\ (com_i)_i, (ch_{i,j})_{i,j}, (h_{i,j})_{i,j})$
return $\sigma \leftarrow ((com_i)_i, (ch_{i,j})_{i,j}, (h_{i,j})_{i,j}, (resp_{i,J_i})_i)$

If we write out **Sign** using the Π_{SIDH} API, we see that the only real computation is being performed by **KeyGen** and **SecAgr**, and our two random oracles $\mathbf{H}$ and $\mathbf{G}$. The rest of the algorithm is merely organizing the information we've generated into the transcript $(com,\ ch,\ resp)$ and then finally into σ.

$\mathbf{Verify}(pk,\ m,\ \sigma)$: Alice begins her execution of **Verify()** where Bob ended his execution of **Sign()**, with the computation of J. Alice then knows at each iteration what check to perform on Bob's response, based on a conditional branch. You will notice that Bob's secret key B occurs in the negative path of this branch; this is not a security concern because it is actually the point $\psi_R(B)$ that is communicated in σ, from which B cannot be recovered.

Algorithm 7 – **Sign**($sk = B$, m) via Π_{SIDH}

for `i = 1..`2λ **do**
  $(R, \psi_R) \leftarrow \Pi_{\text{SIDH}}$:**KeyGen**($\lambda$, *Alice*)
  $\phi'_B : E/\langle B\rangle \rightarrow E/\langle B, R\rangle \leftarrow \Pi_{\text{SIDH}}$:**SecAgr**($B$, ψ_R)
  $(E_1, E_2) \leftarrow (E/\langle R\rangle, E/\langle B, R\rangle)$
  $com_i \leftarrow (E_1, E_2)$
  $ch_{i,0} \leftarrow_R \{0, 1\}$
  $ch_{i,1} \leftarrow 1 - ch_{i,0}$
  $(resp_{i,0}, resp_{i,1}) \leftarrow ((R, \phi_B(R)), \psi_R(B))$
  if $\mathtt{ch}_{i,0} = 1$ **then**
    Swap($resp_{i,0}$,$resp_{i,1}$)
  $h_{i,j} \leftarrow$ **G**($resp_{i,j}$)
$J_1 \parallel \ldots \parallel J_{2\lambda} \leftarrow$ **H**(ϕ_B, m, $(com_i)_i$,$(ch_{i,j})_{i,j}$,$(h_{i,j})_{i,j}$)
return $\sigma \leftarrow ((com_i)_i, (ch_{i,j})_{i,j}, (h_{i,j})_{i,j}, (resp_{i,J_i})_i)$

Algorithm 8 – **Verify**($pk = \phi_B$, m, σ)

Parse $((com_i)_i, (ch_{i,j})_{i,j}, (h_{i,j})_{i,j}, (resp_i)_i) \leftarrow \sigma$
$J_1 \parallel \ldots \parallel J_{2\lambda} \leftarrow$ **H**(ϕ_B, m, $(com_i)_i$,$(ch_{i,j})_{i,j}$,$(h_{i,j})_{i,j}$)
for `i = 0..`2λ **do**
  check $h_{i,J_i} = G(resp_i)$
  if $ch_{i,J_i} = 0$ **then**
    Parse $(R, \phi_B(R)) \leftarrow resp_i$
    check $(R, \phi_B(R))$ have order $\ell_A^{e_A}$
    check R generates the kernel of the isogeny $E \rightarrow E_1$
    check $\phi_B(R)$ generates the kernel of the isogeny $E/\langle B\rangle \rightarrow E_2$
  else
    Parse $\psi_R(B) \leftarrow resp_i$
    check $\psi_R(B)$ has order $\ell_B^{e_B}$
    check $\psi_R(B)$ generates the kernel of the isogeny $E_1 \rightarrow E_2$
if all checks succeed **then**
  return 1
else
  return 0

What we are checking for in the verification process is whether or not Bob and Randall performed an honest and valid key exchange. And so, if the challenge bit is 0, we can use SIDH key generation to determine that R and ψ_R are a valid key pair and then run SIDH secret agreement with R and Bob's public key ϕ_B to confirm that it properly executes outputting an isogeny with kernel generated by $\phi_B(R)$. If the challenge bit is 1, we can run an instance of SIDH secret agreement to verify that $\psi_R(B)$ generates the kernel of an isogeny with domain E_1 and co-domain E_2 (refer again to the diagrams outlining **Prove** of section 2.3.2).

These observations are formalized in Algorithm 6, where we rewrite Σ':**Verify()** in terms of Π_{SIDH} procedure calls. Note, in line 10 of Algorithm 6, the call Π_{SIDH}:**SecAgr**($\psi_R(B)$, ψ_R). It should be noted that $\psi_R(B)$ is not the proper secret key input used by Bob in **Sign()**, but we will see in the section to follow how we can use $\psi_R(B)$ in the C implementation of **SecAgr** to perform our verification (without compromising Bob's secret key B).

Algorithm 9 – **Verify**($pk = \phi_B$, m, σ) via Π_{SIDH}

Parse $((com_i)_i, (ch_{i,j})_{i,j}, (h_{i,j})_{i,j}, (resp_i)_i) \leftarrow \sigma$
$J_1 \parallel \ldots \parallel J_{2\lambda} \leftarrow \mathbf{H}(\phi_B, m, (com_i)_i, (ch_{i,j})_{i,j}, (h_{i,j})_{i,j})$
for `i = 0..2`λ **do**
    check $h_{i,J_i} = G(resp_{i,J_i})$
    if $ch_{i,J_i} = 0$ **then**
        Parse $(R, \phi_B(R)) \leftarrow resp_{i,J_i}$
        check (R, ψ_R) is a valid output of Π_{SIDH}:**KeyGen**(λ, *Alice*)
        check that Π_{SIDH}:**SecAgr**(R, ϕ_B) successfully outputs an isogeny with co-domain E_2
    else
        Parse $\psi_R(B) \leftarrow resp_{i,J_i}$
        check that Π_{SIDH}:**SecAgr**($\psi_R(B)$, ψ_R) successfully outputs an isogeny with co-domain E_2
if all checks succeed **then**
    return 1
else
    return 0

2.6 Implementations of Isogeny-based Cryptographic Protocols

Having now introduced all of the background material necessary for understanding SIDH and the isogeny-based signature scheme in detail, we will investigate the portions of the SIDH C library which are relevent to our contributions.

The SIDH C library, written by a research team at Microsoft Research, was released in 2016 alongside an article titled *Efficient Algorithms for Supersingular Isogeny Diffie-Hellman* (see [CLN16]). The article in question details several adjustments to the algorithms and data-representations outlined in [FJP12], leading to improved performance and key-sizes. Their library (which we will henceforth refer to as $\texttt{SIDH}_\texttt{C}$) consists of C and assembly implementations of the algorithms outlined in [CLN16]. Much of these functions are tailored to a specific set of parameters allowing for increased performance. The library presents 128-bit quantum security and 192-bit classical security key exchange up to 2.9 times faster than any previous isogeny-based key-exchange system. We will look at some of the details of $\texttt{SIDH}_\texttt{C}$ below.

Before proceeding, it may be advisable to briefly review the section on notation (**??**) if one has not already.

2.6.1 Parameters & Data Representation

Parameters. $\texttt{SIDH}_\texttt{C}$ operates over the underlying basefield $\mathbb{F}_p$ where $p = \ell_A^{e_A} \cdot \ell_B^{e_B} - 1$, with $\ell_A = 2$, $\ell_B = 3$, $e_A = 372$, and $e_B = 239$, giving p a bitlength of 751. Now, recall the

Montgomery representation of a curve:

$$By^2 = Cx^3 + Ax^2 + Cx$$

$\mathtt{SIDH_C}$ uses the public parameter curve E defined in Montgomery form with $A = 0$, $B = 1$, and $C = 1$. The point pairs (P_A, Q_A) and (P_B, Q_B), generating $E[\ell_A^{e_A}]$ and $E[\ell_B^{e_B}]$ respectively, are hard-coded as an array of bytes. These parameters (including related data such as the bitlength of certain constants) are stored in the struct type `CurveIsogenyStaticData` under the variable name `SIDHp751`. This struct, along with many other $\mathtt{SIDH_C}$ data types and representations, will be outlined in the coming subsection.

One priority of the parameter choices found in `SIDHp751` was to approach $\ell_A^{e_A} \approx \ell_B^{e_B}$. This attempted at balancing $\ell_A^{e_A}$ and $\ell_B^{e_B}$ helps to ensure two things: first, that no side of the key exchange is any easier to attack than the other, and second, that the cost of computation is split evenly between parties. This constraint had to be compromise with the primary security concern: that p must have a bit-length providing sufficient classical and quantum security.

Data Structures. There are several custom-defined data structures that are integral to $\mathtt{SIDH_C}$. Below, we will briefly cover the ones which are likely to arise in our discussion:

Field elements

- `felm_t` – buffer of bytes representing elements of $\mathbb{F}_p$.
- `f2elm_t` – pair of `felm_t` representing elements of $\mathbb{F}_{p^2}$.

Elliptic curve points

- `point_affine` – an `f2elm_t` `x` and an `f2elm_t` `y` representing a point in affine space.
- `point_proj` – an `f2elm_t` `X` and an `f2elm_t` `Z` representing a point as projective XZ Montgomery coordinates.
- `point_full_proj` – `f2elm_t` elements `X`, `Y`, and `Z` representing a point in projective space.
- `point_basefield_affine` – an `felm_t` `x` and an `felm_t` `y` representing a point in affine space over the base field.
- `point_basefield_proj` – an `felm_t` `X` and an `felm_t` `Z` representing a point as projective XZ Montgomery coordinates over the base field.

Cryptographic structures

- `publickey_t` – three `f2elm_t`s representing a public key.
  `publickey_t[0]` = user's private isogeny applied to the other party's generator P_x
  `publickey_t[1]` = user's private isogeny applied to the other party's generator Q_x
  `publickey_t[2]` = user's private isogeny applied to $P_x - Q_x$

Curve structures

- `CurveIsogenyStruct` – Structure containing all necessary public parameter data.

- `CurveIsogenyStaticData` – The same as `CurveIsogenyStruct`, but with buffer sizes fixed for `SIDHp751`.

The reader may note that `publickey_t` does not contain any information defining the user's co-domain curve $E/\langle S\rangle$ (with S as the users secret key). It just so happens that in Π_{SIDH} key exchange, the curves $E/\langle A\rangle$ and $E/\langle B\rangle$ are simply intermediary steps (useful for conceptualizing the protocol) and not necessary for computing the shared secret $j(E_{AB})$.

Also worth noting is the lack of a specific data structure for representing curves. As it turns out, curves within Π_{SIDH} can be distinctly represented by their A value alone. As we are working with curves defined over $\mathbb{F}_{p^2}$, we have $A \in \mathbb{F}_{p^2}$ and thus we can succinctly represent any curve with a single `f2elm_t`.

2.6.2 $\texttt{SIDH}_{\texttt{C}}$ Design Decisions

The following are, at a high-level, the algorithmic improvements upon Π_{SIDH} as outlined in [CLN16]. Costella et al. do make additional contributions in their paper, however we will discuss only those contributions which pertain to the performance of SIDH.

Projective Space Arithmetic. As is common in ECC, a vast majority of the procedures of $\texttt{SIDH}_{\texttt{C}}$ operate over elliptic curve points which are defined over *projective space* (recall Section 2.2.2). This widely-deployed technique is used to avoid the substantial cost of field element inversions (computing x^{-1} for some element $x \in \mathbb{F}_{p^2}$). This means the majority of our calculations are performed over `point_proj` structures using *Montgomery arithmetic* (Section 2.2.1) and converted to `point_affine` when necessary. This general design strategy is highly related to our first contribution, which will be elaborated upon in Section 3.

In addition to traditional point-wise projective arithmetic, Costella et al. showed that isogeny arithmetic can also be carried out in this space. By performing isogeny arithmetic in the projective space, the number of $\mathbb{F}_{p^2}$ inversions in Π_{SIDH}:**KeyGen** and Π_{SIDH}:**SecAgr** can be reduced to 1 and 2, respectively.

Key Representation. Recall the origin of an Π_{SIDH} private key (m, n): the goal is to randomly select a generator of the torsion group $E[\ell_A^{e_A}]$ (or $E[\ell_B^{e_B}]$ for Bob). It is noted in [FJP12] that any generator of the required torsion group is sufficient. It is also noted that m, unless equal to the order of the torsion group, is invertible. Because of this, Alice, for example, can simple compute $R = P_A + [m^{-1}n]Q_A$, thus enabling secret keys to be stored as a single $\mathbb{F}_{p^2}$ element (which is referred to as m). This technicality has been implemented in $\texttt{SIDH}_{\texttt{C}}$, which both saves on storage as well as offers a means for generating secret keys that is more efficient than the trivial scalar multiplication and point-wise addition approach to computing $[m]P + [n]Q$.

Tailor-made Montgomery Multiplication. The parameters of a default $\texttt{SIDH}_{\texttt{C}}$ execution, stored in `SIDHp752`, support efficient arithmetic and grant access to a variety of modular arithmetic optimizations. These optimizations inlude esfs,esdfsef,sefsef.. Moreover, Costella et al. supply a modified version of the Montgomery multiplication algorithm which, when performing over the class of curves outlined by their set of parameters, yields faster modular arithmetic.

2.6.3 Key Exchange & Critical Functions

There are 3 central modules (C files) in $\texttt{SIDH}_\texttt{C}$, all dealing with different levels of abstraction in the Π_{SIDH} protocol. Figure 2.6 illustrates the relationship between these modules and the abstraction levels of isogeny-based key exchange.

Operating at the lowest abstraction level is the module `fpx.c`, wherein functions for manipulating $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$ elements are defined. One level up from `fpx.c` we have `ec_isogeny.c`, containing functions pertaining to elliptic curves and point arithmetic (such as `j_inv(...)` for computing the j-invariant of a curve and `secret_pt(...)` for computing a users secret point S given their private key m). The final, highest abstraction-level module we will discuss is `kex.c`. `kex.c` contains the protocol-level functions for performing Π_{SIDH}, namely `KeyGeneration_A(...)` and `KeyGeneration_B(...)` for generating Alice and Bob's private and public keys, as well as `SecretAgreement_A(...)` and `SecretAgreement_B(...)` for completing the secret agreement from both sides of the key exchange.

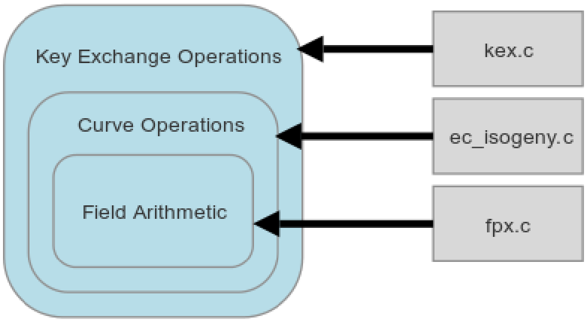


Figure 2.6: Relationship between Π_{SIDH} & $\texttt{SIDH}_\texttt{C}$ modules

For functions defined in `fpx.c` the notational practice is to prepend function names with either `fp` or `fp2`, signifying whether the function is defined for elements of $\mathbb{F}_p$ or $\mathbb{F}_{p^2}$. Additionally, it is common to append `_mont` to the name of functions which utilize Montgomery arithmetic, and thus expect elements in Montgomery representation. Functions in `fpx.c` are largely defined by byte and memory arithmetic, with the exception of slightly higher-level functions (such as field element inversion, `fpinv751_mont(...)`) which are defined in terms of other `fpx.c` functions. Furthermore, for efficiency, functions of `fpx.c` are defined as `__inline` when applicable.

In addition to the `fpx.c` functions we've outlined in Figure **??** there are of course definitions for addition, copying elements, retrieving the zero element, Montgomery multiplication, squaring, and so on and so forth.

`ec_isogeny.c` functions are defined almost exclusively in terms of `fpx.c` functions, with a few occurances of internal function calling. Functions in this module that are significant to our our work are briefly summerized in Figure **??**. The implementation specifics of most other `ec_isogeny.c` functions are not critical to our work, and so have been excluded. The design and efficiency of these algorithms do, however, have a rich background and can be further read about in [FJP12] and [CLN16].

The key exchange procedures found in `kex.c` are composed entirely of calls to `fpx.c` and `ec_isogeny.c` functions, modulo some basic branching logic. All of the functions from this module are relevant to our work - we provide quick debriefings of these functions `fpx.c` functions

Function	Input	Output
`to_fp2mont` Converts an $\mathbb{F}_{p^2}$ element to Montgomery representation	`f2elm_t a`	`f2elm_t ma`
`from_fp2mont` Converts an $\mathbb{F}_{p^2}$ element from Montgomery representation to regular form	`f2elm_t ma`	`f2elm_t a`
`fp2inv751_mont_bingcd` performs *non-constant* time inversion of a $\mathbb{F}_{p^2}$ element	`f2elm_t a`	`f2elm_t a`$^{-1}$
`fp2inv751_mont` performs *constant* time inversion of a $\mathbb{F}_{p^2}$ element	`f2elm_t a`	`f2elm_t a`$^{-1}$

`ec_isogeny.c` functions

Function	Input	Output
`j_inv` computes the j-invariant of a curve with represented in Montgomery form with `A` and `C`	`f2elm_t A` `f2elm_t C`	`f2elm_t jinv`

`secret_pt`
generates the secret
point `R` from
secret key `m`
	`point_basefield P`
	`digit_t m`
	`SIDHp751`
	`int AliceOrBob`
		`point_proj R`

`inv_3_way`
performs simultaneous inversion
of three elements
	`f2elm_t z1`
	`f2elm_t z2`
	`f2elm_t z3`
		`f2elm_t z1`$^{-1}$
		`f2elm_t z2`$^{-1}$
		`f2elm_t z3`$^{-1}$

`inv_4_way`
performs simultaneous inversion
of 4 elements
	`f2elm_t z1`
	`f2elm_t z2`
	`f2elm_t z3`
	`f2elm_t z4`
		`f2elm_t z1`$^{-1}$
		`f2elm_t z2`$^{-1}$
		`f2elm_t z3`$^{-1}$
		`f2elm_t z4`$^{-1}$

`generate_2_torsion_basis` constructs a basis $(\{R1, R2\})$ generating $E[\ell_A^{e_A}]$	`f2elm_t A` `SIDHp751`	`point_full_proj R1` `point_full_proj R2`
`generate_3_torsion_basis` constructs a basis $(\{R1, R2\})$ generating $E[\ell_B^{e_B}]$	`f2elm_t A` `SIDHp751`	`point_full_proj R1` `point_full_proj R2`

kex.c functions

Function	Input	Output
KeyGeneration_A performs key generation for Alice	unsigned char* privateKeyA bool generateRandom	unsigned char* privateKeyA unsigned char* publicKeyA
KeyGeneration_B performs key generation for Bob		unsigned char* privateKeyB unsigned char* publicKeyB
SecretAgreement_A computes the shared secret from Alice's perspective	unsigned char* privateKeyA unsigned char* publicKeyB point_proj kerngen	unsigned char* sharedSecretA point_proj kerngen
SecretAgreement_B computes the shared secret from Bob's perspective	unsigned char* privateKeyB unsigned char* publicKeyA point_proj kerngen	unsigned char* sharedSecretB point_proj kerngen

in Figure ??.

The reader may note that, in Figure ??, privateKeyA (in KeyGeneration_A) and kerngen (in both secret agreements) appear as both inputs and outputs. This is not a mistake. In KeyGeneration_A, if generateRandom = false is passed as an input, then privateKeyA is expected to be set, and the corresponding public key is computed. In secret agreement, if kerngen is set to null then the algorithm proceeds normally. If it is set to a valid point, however, it can be used in place of a secret key input (which in such a case is expected to be null). Both of these details are critical to the design of signature functions as they are described below.

2.6.4 Signature Layer

Yoo et al. provided, along with their publication of [YAJ+12], an implementation of their signature scheme as a fork to $\mathtt{SIDH}_\mathtt{C}$. All of their functions are written specifically for an instance of Σ' where the signer is assuming the B role (meaning that Randall assumes the A role), but their algorithms could be trivially modified to provide versions supporting a signer in the A role. Their contributions to the $\mathtt{SIDH}_\mathtt{C}$ codebase come in the form of the functions listed below.

To begin, isogeny_keygen has a trivial definition; KeyGeneration_B is called and populates the signer's public and private keys. isogeny_keygen returns the success status of the call to KeyGeneration_B.

In their original fork of $\mathtt{SIDH}_\mathtt{C}$, Yoo et al. included these functions in the file kex_tests.c. This file was originally intended for testing the functions of kex.c, and so our fork of the library has placed the signature functions in a new file SIDH_signature.c. We have also included a file sig_tests.c for testing the contents and performance of SIDH_signature.c functions.

If we transcribe the procedures Σ':**Sign** and Σ':**Verify** (as described in Section ??) to the language of the $\mathtt{SIDH}_\mathtt{C}$ API, we have in essence the procedures Sign and Verify given by Algorithms 10 and 11 respectively.

Function	Input	Output
`isogeny_keygen` generates the signers key pair		`unsigned char* privateKeyB` `unsigned char* publicKeyB`

`isogeny_sign`
produces a signature
for a message
	`privateKey`
	`publicKey`
	message m
		`Signature sig`

`sign_thread` performs a single iteration of the for-loop in **Sign**		
`isogeny_verify` checks the validity of a signature	`Signature sig`	`true` or `false`
`verify_thread` performs a single iteration of the for-loop in **Verify**		

Algorithm 10 – $\mathtt{Sign}(sk_B,\ m)$

for `i = 1..`2λ **do**
  $(sk_R = R, pk_R) \leftarrow \mathtt{KeyGeneration\_A(NULL,\ true)}$
  $(E/\langle B, R\rangle, \psi_R(B)) \leftarrow \mathtt{SecretAgreement\_B}(sk_B, pk_R, \mathtt{NULL})$
  $(E_1, E_2) \leftarrow (E/\langle R\rangle, E/\langle B, R\rangle)$
  $(\mathtt{com}[i]_0, \mathtt{com}[i]_1) \leftarrow (E_1, E_2)$
  $(\mathtt{resp}[i]_0, \mathtt{resp}[i]_1) \leftarrow (R, \psi_R(B))$
  $h[i] \leftarrow \mathtt{keccak}(\mathtt{resp}[i]_0)\,|\,\mathtt{keccak}(\mathtt{resp}[i]_1)$
$J_1 \,\|\, \ldots \,\|\, J_{2\lambda} \leftarrow \mathtt{keccak}(\mathtt{com},\ m,\ h)$
return $\sigma \leftarrow ((\mathtt{com}_i)_i, (\mathtt{ch}_{i,j})_{i,j}, (h_i)_i, ((\mathtt{resp})[J_i])$

Algorithm 11 – $\texttt{Verify}(pk = \phi_B,\ m,\ \sigma)$

$J_1 \parallel \dots \parallel J_{2\lambda} \leftarrow \texttt{keccak}(\texttt{com},\ m,\ h)$
for i = 0..2λ **do**
  check $h[i] = \texttt{keccak}(\texttt{resp}[i]_0)\,|\,\texttt{keccak}(\texttt{resp}[i]_1)$
  if $J_i = 0$ **then**
    $R \leftarrow \texttt{resp}[i]_0$
    $pk_R \leftarrow \texttt{KeyGeneration\_A}(\texttt{R},\ \texttt{false})$
    check $pk_R = \texttt{com}[i]_0$
    $E_{RB} \leftarrow \texttt{SecretAgreement\_A}(R,\ \phi_B,\ \texttt{NULL})$
    check $E_{RB} = \texttt{com}[i]_1$
  else
    $\psi_R(B) \leftarrow \texttt{resp}[i]_1$
    $pk_R \leftarrow \texttt{com}[i]_0$
    $E_{BR} \leftarrow \texttt{SecretAgreement\_B}(\texttt{NULL},\ pk_R,\ \psi_R(B))$
    check $E_{BR} = \texttt{com}[i]_1$
if all checks succeed **then**
  return 1
else
  return 0

Outside of simply replacing Π'_{SIDH} procedure calls with $\texttt{SIDH}_\texttt{C}$ functions, the reader may notice additional differences between `Sign` and `Verify` and their Σ' counterparts. Namely, Yoo et al. have chosen to exclude the challenge bit *ch* in the $\texttt{SIDH}_\texttt{C}$ implementations of these functions, consequently excluding the conditional and **Swap** statement of lines 8 and 9 in Algorithm 4.

Chapter 3

Batching Operations for Isogenies

Our first contribution to the $\texttt{SIDH}_\texttt{C}$ codebase is the implementation and integration of a procedure for batching together many $\mathbb{F}_{p^2}$ element inversions. This contribution is discussed in detail in the following Chapter. The chapter is split into three sections: a high-level discussion of the procedure itself, the low-level details of its integration into $\texttt{SIDH}_\texttt{C}$, and finally, the resulting affects of this procedure on the performance of $\texttt{SIDH}_\texttt{C}$.

In the first Section of this Chapter we will detail the specifics of the partial batched inversion procedure. We will show how the procedure can be constructed by combining two techniques: a well known method for reducing a $\mathbb{F}_{p^2}$ inversion to several $\mathbb{F}_p$ operations, and an inversion batching technique outlined in [SB01].

As we then venture into the lower-level implementation details, we will explore how the procedure can be leveraged efficiently in the codebase. We will take a closer look at several of the aforementioned $\texttt{SIDH}_\texttt{C}$ functions as we illustrate some of the performance bottlenecks in the system. At this time, we will also discuss the design decisions made while implementing the partial batched inversion procedure as well as some of the function's lower-level minutiae.

We will end this Chapter by taking a detailed look at the performance gains offered by the inclusion of partial batched inversions in $\texttt{SIDH}_\texttt{C}$. More precisely, we will be examining the effects of the procedure on the Yoo et al. signature layer. We will contrast the measured performance of our implementation with an analytical calculation of the expected improvement, and discuss the possible origins of divergent behaviour.

3.1 Partial Batched Inversions

We will now outline the procedure that is central to our first contribution. The "partial batched inversion" procedure reduces arbitrarily many *unrelated*[1] $\mathbb{F}_{p^2}$ inversions to a sequence of $\mathbb{F}_p$ operations. The fact that the elements being inverted need not hold any relation will be significant to the applicability of this procedure. For brevity's sake, we will henceforth refer to this procedure as `pb_inv` in the $\texttt{SIDH}_\texttt{C}$ context, and **PartialBatchedInversion** in the more general mathematical context.

As mentioned above, `pb_inv` is constructed by combining two distinct techniques. Both of these techniques improve the efficiency of computing field element inversions:

[1]To clarify; the elements subject to these inversion must all be over the same field, but can otherwise be unrelated.

the first is specific to extension fields (in our case, $\mathbb{F}_{p^2}$ elements,) but the second is a technique applicable to field element inversions in a more general setting.

We will begin with a dissection of these two techniques, starting first with the "partial" inversion technique and then looking at batched inversions. The definitions we will give for these techniques below are given at the level of field arithmetic. When we proceed to sketch `pb_inv`, we will offer two definitions: one in this section given at the abstraction-level of field arithmetic, and one in the proceeding section given in terms of $\texttt{SIDH}_\texttt{C}$ syntax.

In the subsections to come, when we are working at the level of field arithmetic we will denote the first and second portions of an arbitrary $x \in \mathbb{F}_{p^2}$ as x_a and x_b respectively, where $x = x_a + x_b \cdot i$. Additionally, we may write x as (x_a, x_b), as this more closely reflects the structure of $\mathbb{F}_{p^2}$ elements in $\texttt{SIDH}_\texttt{C}$. Recall from Section 2.2.1 that both x_a and x_b are valid $\mathbb{F}_p$ elements.

We will express the time-complexity of the coming procedures in terms of the number of underlying field operations within them. We denote the computation time for base field arithmetic with bold letters (such as $\mathbf{a}$ for $\mathbb{F}_p$ **a**ddition), and we use bold letters accented with a "closure" bar for extension field arithmetic ($\bar{\mathbf{a}}$ for $\mathbb{F}_{p^2}$ **a**ddition). For example, the time-complexity of some procedure P, which we might write as C_P, may look like the following:

$$C_P = 2\bar{\mathbf{a}} + x\bar{\mathbf{i}} + y\mathbf{m} + \mathbf{s}$$

Which denotes that P is a procedure composed of 2 $\mathbb{F}_{p^2}$ additions, x-many $\mathbb{F}_{p^2}$ inversions, y-many $\mathbb{F}_p$ multiplications, and a single $\mathbb{F}_p$ squaring. We reserve uppercase bold letters for arithmetic over elliptic curve points (such as $\mathbf{A}$ to denote the point-wise addition operation).

3.1.1 $\mathbb{F}_{p^2}$ Inversions done in $\mathbb{F}_p$

There is a simple way in which we can perform one $\mathbb{F}_{p^2}$ inversion by means of doing several $\mathbb{F}_p$ operations. We will begin by considering multiplicative inverses of complex numbers. Fields of the form $\mathbb{F}_{q^2}$ for some prime q are, after all, quadratic extension fields; because of this $\mathbb{F}_{p^2}$ arithmetic is treated, for the most part, analogously to complex number arithmetic.

Consider the complex number $C = a + bi$. We have that $C^{-1} = 1/(a+bi)$, from which we can rationalize the denominator like so:

$$C^{-1} = \frac{1}{(a+bi)} \cdot \frac{(a-bi)}{(a-bi)}$$

$$C^{-1} = \frac{a-bi}{(a+bi)(a-bi)}$$

Here we note that $(a+bi)(a-bi)$ is equivalently (a^2+b^2) and so we can rewrite C^{-1} as the following:

$$C^{-1} = \frac{a-bi}{(a)^2-(bi)^2}$$

$$C^{-1} = \frac{a-bi}{a^2+b^2}.$$

Elements in the quadratic extension of a finite field are treated similarly, such that if we take some element $x = (x_a, x_b) \in \mathbb{F}_{p^2}$ for some prime p, we can equivalently represent

x as $x_a + x_b i$ and treat arithmetic on x exactly as we would for a complex number (modulo p, of course). From this we can see that x^{-1} can be defined as:

$$x^{-1} = (\frac{x_a}{x_a^2 + x_b^2}, \frac{-x_b}{x_a^2 + x_b^2})$$

Now it is clear that we can compute the multiplicative inverse of x by computing the inverse of $x_a^2 + x_b^2$ (an inversion in $\mathbb{F}_p$) and $-x_b$ (a relatively inexpensive operation, also in the base field). We formulate this technique in Algorithm 14, which we refer to as **PartialInv**.

Algorithm 12 – **PartialInv**($x \in \mathbb{F}_{p^2}$)

1: $den \leftarrow x_a^2 + x_b^2$
2: $den_{inv} \leftarrow den^{-1} \pmod p$
3: $a \leftarrow x_a \cdot den_{inv} \pmod p$
4: $b \leftarrow -(x_b) \cdot den_{inv} \pmod p$
5: $inv \leftarrow \{a, b\}$
6: **return** inv

Effectively, this procedure reduces one $\mathbb{F}_{p^2}$ inversion to the following operations:

- 2 $\mathbb{F}_p$ squarings – *line 1 of algorithm* 14
- 1 $\mathbb{F}_p$ addition – *line 1 of algorithm* 14
- 1 $\mathbb{F}_p$ inversion – *line 2 of algorithm* 14
- 3 $\mathbb{F}_p$ multiplications – *lines 3 & 4 of algorithm* 14

Let $C_{\textbf{PartialInv}}$ represent the time complexity of **PartialInv**, in the format outlined above. We have

$$C_{\textbf{PartialInv}} = 2\mathbf{s} + \mathbf{a} + \mathbf{i} + 3\mathbf{m}$$

In some contexts, computing squares can be done more efficiently than the multiplication of two arbitrary elements. A noteworthy example of this is would be binary fields ($\mathbb{F}_{2^k}$) where squaring a number is equivalent to simply performing a bit-shift. However, because we are working in the quadratic extension of some prime field $\mathbb{F}_p$ for a large prime p, we can assume that computing the square of some arbitrary element x is no more or less efficient than simply computing $x \cdot x$. With this in mind, we can further simplify A.

$$C_{\textbf{PartialInv}} = 5\mathbf{m} + \mathbf{a} + \mathbf{i}$$

We claim that this "reduces" the computation time of field element inversion, but it may not be immediately clear that this technique is computationally favourable over any other approach to computing an $\mathbb{F}_{p^2}$ inversion.

3.1.2 Batching Field Element Inversions

The second technique used in the composition of `pb_inv` reduces arbitrarily many (general) field element inversions to *one* inversion and a linearly scaling amount of multiplcations in the *same* field.

This technique was outlined by Shacham and Boneh in [SB01]. Shacham and Boneh provided several techniques for improving the performance of SSL handshakes, most of which built on the earlier efforts of Fiat in batching multiple RSA decryptions [Fia89]. While somewhat related, Fiat's work admittedly is only applicable to the RSA cryptosystem, and requires additional constraints on the elements being batched.

One improvement offered by Shacham and Boneh, however, is their proposed notion of batching together divisions from across multiple unrelated SSL instances.

Suppose we want to compute the inverses of three elements $x, y, z \in F$ where F is some arbitrary field. The batched division technique allows us to reduce these three inversions to one. The technique can be organized into three phases. In the first phase, all the elements of the batch are multiplied together into one product, yielding $a = xyz$. We refer to this first phase as "upward-percolation". Next, we compute the inverse of a: $a^{-1} = (xyz)^{-1}$, which we refer to as the inversion phase. In the final phase, "downward-percolation", we can compute each individual element's multiplicative inverse as follows:

$$x^{-1} = a^{-1} \cdot (yz)$$

$$y^{-1} = a^{-1} \cdot (xz)$$

$$z^{-1} = a^{-1} \cdot (xy)$$

Let us analyse these phases a little more closely while we generalize to n-many elements. In the upward-percolation phase, constructing a requires $n-1$ multiplications; and so has a complexity of $\mathcal{O}(n)$. The inversion phase requires one field element inversion, and so has complexity of $\mathcal{O}(1)$.

If we implement the downward-percolation phase directly as outlined in the three-element example above, computing every output requires n products each composed of $n-1$ multiplications. These n products are each also multiplied by a^{-1}. This multiplication by a^{-1} can be added to our $n-1$ inversion count resulting in n-many products composed of n multiplications; bringing the complixity of the downward-percolation phase to $\mathcal{O}(n^2)$.

We will refer to this roughly-sketched procedure as $\mathbf{BatchedInv}_0$. Let $C_{\mathbf{BatchedInv}_0}$ denote the performance of $\mathbf{BatchedInv}_0$ in the format outlined above. We have, then, that

$$C_{\mathbf{BatchedInv}_0} = n^2\bar{\mathbf{m}} + (n-1)\bar{\mathbf{m}} + \bar{\mathbf{i}}.$$

This batching proceedure can be thought of as analogous to traditional time-memory tradeoff algorithms. In a general time-memory tradeoff algorithm you can continue to make some linear or polynomial (or otherwise) sacrifice of memory in order to gain some increase in performance. In the batching procedure described above we are in some sense sacrificing some marginal amount of memory to gain an increase in performance, but it is not a tradeoff that we can adjust to our liking.

There is a way, much akin to this time-memory tradeoff strategy, that we can further reduce the execution time of $\mathbf{BatchedInv}_0$. In the upward-percolation phase, we currently store in a the product of elements $x_0 \cdot x_1 \cdot \ldots \cdot x_{n-1}$. Suppose instead that we store in a an *array* (size n) of elements, defined in the following way:

$$a_i = \begin{cases} x_0 & i = 0 \\ a_{i-1} \cdot x_i & \text{otherwise} \end{cases}$$

Equivalently, the elements of this array are

$$a_0 = x_0, \quad a_1 = x_0 \cdot x_1, \quad a_2 = x_0 \cdot x_1 \cdot x_2, \quad ...$$

and so on and so forth up to $n-1$. In the inversion phase we will compute $inv = a_{n-1}^{-1}$; we are still inverting the product of all the elements, but because we have stored the value of the product at every step of the way, we can save on a significant number of operations in the downward-percolation phase.

Going into the final stage of the procedure now, we can compute x_{n-1}^{-1} simply by computing $inv \cdot a_{n-2}$. Moving forward (or backwards, technically), we peel the previously used x_{n-1}^{-1} off of inv by computing $inv := inv \cdot x_{n-1}$ and, with our updated inv, we compute $x_{n-2}^{-1} = inv \cdot a_{n-3}$. We proceed in this fashion until we reach x_0^{-1}, which (if we've been updating inv every step of the way) is simply equal to inv.

We formalize this improvement in the form of a new procedure, **BatchedInv**, which we provide a concrete definition for in Algorithm 13. In this procedure lines 1–3 implement the upward-percolation phase. Line 4 carries out the second phase: the inversion of a_{n-1}. The third and final stage, downward-percolation, occurs from lines 5 to 7.

Algorithm 13 – **BatchedInv**($\{x_0, x_1, ..., x_n - 1\} \in \mathbb{F}_{p^2}^n$)

1: $a_0 \leftarrow x_0$
2: **for** `i = 1..(n-1)` **do**
3: $\quad a_i \leftarrow a_{i-1} \cdot x_i$
4: $inv \leftarrow a_{n-1}^{-1}$
5: **for** `i = (n-1)..1` **do**
6: $\quad x_i^{-1} \leftarrow a_{i-1} \cdot inv$
7: $\quad inv \leftarrow inv \cdot x_i$
8: $x_0^{-1} = inv$
9: **return** $\{x_0^{-1}, x_1^{-1}, ..., x_{n-1}^{-1}\}$

BatchedInv can be used to reduce n-many $\mathbb{F}_{p^2}$ inversions to the following operations:

- $n-1$ $\mathbb{F}_{p^2}$ multiplications – *line 2-3 of algorithm* 13
- 1 $\mathbb{F}_{p^2}$ inversion – *line 4 of algorithm* 13
- $2(n-1)$ $\mathbb{F}_{p^2}$ multiplications – *line 5-7 of algorithm* 13

Let $C_{\textbf{BatchedInv}}$ denote the performance of **BatchedInv**.

$$\begin{aligned} C_{\textbf{BatchedInv}} &= 2(n-1)\bar{\mathbf{m}} + (n-1)\bar{\mathbf{m}} + \bar{\mathbf{i}} \\ &= 3(n-1)\bar{\mathbf{m}} + \bar{\mathbf{i}} \end{aligned}$$

In comparing the performances of **BatchedInv** and **BatchedInv**$_0$, we see that $C_{\textbf{BatchedInv}} < C_{\textbf{BatchedInv}_0}$ holds when the following holds:

$$2(n-1)\bar{\mathbf{m}} + (n-1)\bar{\mathbf{m}} + \bar{\mathbf{i}} < n^2\bar{\mathbf{m}} + (n-1)\bar{\mathbf{m}} + \bar{\mathbf{i}}$$

$$2(n-1)\bar{\mathbf{m}} < n^2\bar{\mathbf{m}}$$
$$2(n-1) < n^2$$

And so, because n^2 is always larger than $2(n-1)$ for all $n \in \mathbb{R}$, **BatchedInv** outperforms **BatchedInv**$_0$ for every possible batch size. This can be checked, if one so wishes, by setting $n^2 = 2(n-1)$, simplifying to $n^2 - 2n + 2 = 0$, and noting that the discriminant $(2^2 - 4 \cdot 2)$ is negative.

3.1.3 Partial Batched Inversions

We have now outlined the following: **PartialInv** as a technique for computing $\mathbb{F}_{p^2}$ inversions by means of $\mathbb{F}_p$ arithmetic, and **BatchedInv** as a technique for batching together arbitrarily many inversion operations. We will now combine these procedures to achieve the partial batched inversion algorithm.

At first glance, an attempt to meld these two techniques together might be made in the same fashion as Algorithm 14. We denote this approach **PartialBatchedInv**$_0$.

Algorithm 14 – **PartialBatchedInv**$_0(\{x_0, x_1, ..., x_{n-1}\})$

1: $a \leftarrow$ upward-percolation of elements $\{x_0, x_1, ..., x_{n-1}\}$
2: $a^{-1} \leftarrow$ **PartialInv**(a)
3: $\{x_0^{-1}, x_1^{-1}, ..., x_{n-1}^{-1}\} \leftarrow$ downward-percolation of a^{-1}
4: **return** $\{x_0^{-1}, x_1^{-1}, ..., x_{n-1}^{-1}\}$

If we sum the operations in **PartialBatchedInv**$_0$, we have the following:

- n $\mathbb{F}_{p^2}$ multiplications – *upward-percolation phase*
- 2 $\mathbb{F}_p$ squarings, 1 $\mathbb{F}_p$ addition, 1 $\mathbb{F}_p$ inversion, and 3 $\mathbb{F}_p$ multiplications – *call to* **PartialInv**(a)
- $2n$ $\mathbb{F}_{p^2}$ multiplications – *downward-percolation phase*

To measure the complixity in terms of field operations, denoted C_0, we can surmize the the total operation count as:

$$C_0 = (n\bar{\mathbf{m}}) + (2\mathbf{s} + \mathbf{a} + \mathbf{i} + 3\mathbf{m}) + (2n\bar{\mathbf{m}})$$

$$C_0 = 3n\bar{\mathbf{m}} + 2\mathbf{s} + \mathbf{a} + \mathbf{i} + 3\mathbf{m}$$

Below we provide an alternative approach to building **PartialBatchedInv** that relies on only $\mathbb{F}_p$ operations. Afterward, we show by simple analysis why this approach yields the better performance. This procedure is formalized in a mathematical setting in Algorithm 15. We give a precise C function definition in Section 3.2.

In Algorithm 15, a is a simple auxillary set we use to hold the inverted $\mathbb{F}_p$ elements. After these are all computed via the for-loop on line 8, we can reconstruct $\mathbb{F}_p$.

More specifically, the procedure takes us from n $\mathbb{F}_{p^2}$ inversions to:

Algorithm 15 – **PartialBatchedInversion**($\mathbb{F}_{p^2}$ $\{x_0, x_1, ..., x_n - 1\}$)

for i = 0..(n-1) **do**
    $den_i \leftarrow (x_i)_a^2 + (x_i)_b^2 \pmod p$
$a_0 \leftarrow den_0$
for i = 1..(n-1) **do**
    $a_i \leftarrow a_{i-1} \cdot den_i \pmod p$
$inv \leftarrow a_{n-1}^{-1} \pmod p$
for i = n-1..1 **do**
    $a_i \leftarrow inv \cdot dest_{i-1} \pmod p$
    $inv \leftarrow inv \cdot den_i \pmod p$
$a_0 \leftarrow a_{inv}$
for i = 0..(n-1) **do**
    $(xinv_i)_a \leftarrow a_i \cdot (x_i)_a \pmod p$
    $(xinv_i)_b \leftarrow a_i \cdot -(x_i)_b \pmod p$
    $x_i^{-1} \leftarrow \{(xinv_i)_a, (xinv_i)_b\}$
return $\{x_0^{-1}, x_1^{-1}, ..., x_{n-1}^{-1}\}$

- $2n$ $\mathbb{F}_p$ squarings
- n $\mathbb{F}_p$ additions
- 1 $\mathbb{F}_p$ inversion
- $3(n$-1) $\mathbb{F}_p$ multiplications
- $2n$ $\mathbb{F}_p$ multiplications

And so, with C measuring the performance of **PartialBatchedInversion**, we have

$$C = 2n\mathbf{s} + n\mathbf{a} + \mathbf{i} + 3(n-1)\mathbf{m} + 2n\mathbf{m}$$

We can further simplify C if we presume that the execution time of squaring is roughly the same as multiplication. Additionally, we can simplify $3(n-1)$ to $3n$ in the spirit of complexity theory. With these simplifications we arrive at

$$C \approx 7n\mathbf{m} + n\mathbf{a} + \mathbf{i}$$

Applying the same simplifying assumptions to C_0, we arrive at

$$C_0 \approx 3n\bar{\mathbf{m}} + 5\mathbf{m} + \mathbf{a} + \mathbf{i}$$

We note here that an $\mathbb{F}_{p^2}$ multiplication ($\bar{\mathbf{m}}$) is performed simply by means of 4 $\mathbb{F}_p$ multiplications (again, recall the multiplcation of complex numbers). So we have $\bar{\mathbf{m}} = 4\mathbf{m}$, and can further simplify C_0:

$$C_0 \approx (12n + 5)\mathbf{m} + \mathbf{a} + \mathbf{i}$$

Finally we've simplified C and C_0 to forms that are more easily compared. Lets us turn our attention to the proposition that C runs in fewer operations than C_0:

$$C < C_0$$

$$7n\mathbf{m} + n\mathbf{a} + \mathbf{i} < (12n + 5)\mathbf{m} + \mathbf{a} + \mathbf{i}$$

Simplifying slightly, we need now to resolve

$$7n\mathbf{m} + n\mathbf{a} < (12n + 5)\mathbf{m} + \mathbf{a}$$

$$n\mathbf{a} - \mathbf{a} < (12n + 5)\mathbf{m} - 7n\mathbf{m}$$

$$n\mathbf{a} - \mathbf{a} < 5n\mathbf{m} + 5\mathbf{m}$$

$$(n - 1)\mathbf{a} < (5n + 1)\mathbf{m}$$

It appears now that in order for **PartialBatchedInv**$_0$ to be computationally favourable over **PartialBatchedInv**, the execution time for one $\mathbb{F}_p$ addition would need to be larger than at least 5 times that of one $\mathbb{F}_p$ multiplication.

Though it seems trivially true, we can verify this by measuring and comparing the execution times of the `SIDH`$_\mathtt{C}$ addition and multiplcation functions we will be using for our implementation.

When doing so (using the arithmetic test cases included in `arith_tests.c` by Microsoft Research) we arrive at the measurements outlined in table ??.

Operation	`SIDH`$_\mathtt{C}$ function	performance in clock cycles
$\mathbb{F}_p$ addition	`fpadd751`	206
$\mathbb{F}_p$ multiplication	`fpmult751_mont`	1,009

If we query for the performance of other operations (including $\mathbb{F}_{p^2}$ arithmetic) we can estimate to what degree roughly **PartialBatchedInv** outperforms **PartialBatchedInv**$_0$. We can also measure to what degree we can expect that it will outperform an unbatched implementation of n-many inversions.

Operation	`SIDH`$_\mathtt{C}$ function	performance in clock cycles
$\mathbb{F}_p$ inversion	`fpinv751_mont`	826,228
$\mathbb{F}_{p^2}$ addition	`fp2add751`	172
$\mathbb{F}_{p^2}$ multiplication	`fp2mult751_mont`	2,793
$\mathbb{F}_{p^2}$ inversion	`fp2inv751_mont`	829,786

All of these results are computed as the average over 100 distinct applications. Furthermore, because they are measured in clock cycles, they are independent of any CPU clock rate. Because of this they are indicative of the complexity of each operation (or rather, the complexity of these implementations,) opposed to the performance of these operations on any one particular machine.

We conclude this section by using these results, along with the operation counts of each procedure, to compare the expected performances of **PartialBatchedInv**, **PartialBatchedInv**$_0$, and unbatched inversion. These results are shown in Table ??. For these estimations we have set the number of elements (n) equal to 248. This closely reflects the setting in which **PartialBatchedInv** will be implemented in `SIDH`$_\mathtt{C}$, as will be discussed in the following section.

PartialBatchedInv$_0$:

If we substitute the performance variables in C_0 with the corresponding results from the tables above, we have:

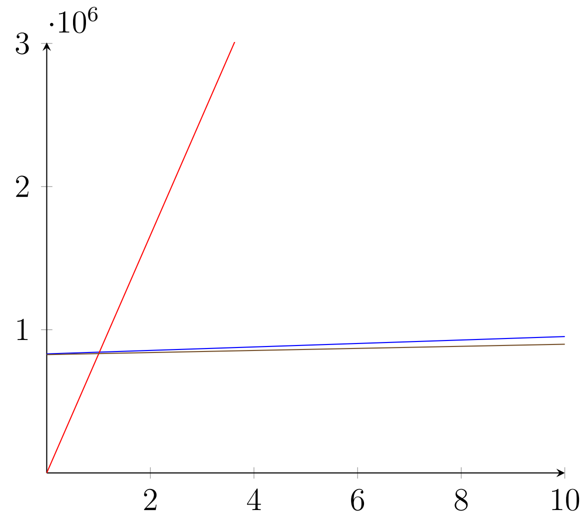
$$C_0 \approx (12n + 5)\mathbf{m} + \mathbf{a} + \mathbf{i}$$
$$C_0 \approx (12n + 5)1,009 + 206 + 826,228$$
$$C_0 \approx 12,108n + 831,479$$

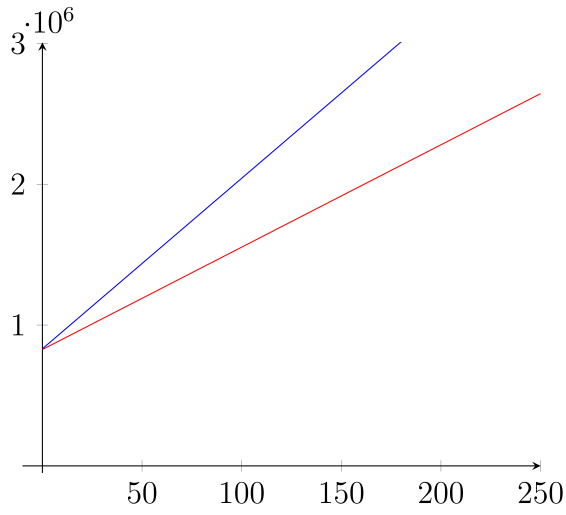
unbatched $\mathbb{F}_{p^2}$ inversions:

The performance of n-many unbatched $\mathbb{F}_{p^2}$ inversions can be modelled plainly by $n\bar{\mathbf{i}}$. The cost of n unbatched inversions is therefore $829,786n$.

PartialBatchedInv:

$$C \approx 7n\mathbf{m} + n\mathbf{a} + \mathbf{i}$$
$$C \approx 7,269n + 826,228$$





Procedure	operation count	expected performance in clock cycles
PartialBatchedInv	`fpinv751_mont`	826,228
PartialBatchedInv$_0$	`fp2add751`	172
248 unbatched $\mathbb{F}_{p^2}$ inversions	`fp2inv751_mont`	829,786

3.2 Implementation Details

We will now take the work of the previous subsection and explain in detail how it can be applied to the Yoo et al. signature layer of `SIDH_C`. We will begin with an examination of the lower-level details of our procedures implementation. In this first subsection, we transcribe **PartialBatchedInversion** to its C variant, `pb_inv`, which is defined almost entirely by means of the `SIDH_C` API. We will discuss some design specifics of `pb_inv`, and look breifly at the security of the function with respect to the signature scheme.

After outlining the specifics of our C implementation, we will move onto a high-level overview of the signature layer architecture. This mapping will allow efficient highlighting of execution paths in the codebase where batching inversions could offer a performance increase. Additionally, we will discuss properties of the signature scheme that can be leveraged to optimize the performance increases offered by `pb_inv`.

3.2.1 Implementation & Design Decisions

With Figure **??** we provide an explicit C definition for the function `pb_inv`. For descriptions of the functions called in this procedure, the reader can refer to section 2.6.3. For explicit definitions of some of these functions, the reader can refer to Appendix A.

`pb_inv`. The `pb_inv` function can be divided into six sections: local variable declaration, conversion to the base field, the upward-percolation phase, the inversion phase, the downward-percolation phase, and finally conversion back to the extension field.

In converting to the base field (beginning at line 9) we are peforming line 1 of Algorithm 14 (as outlined in Subsection 3.1.1) for all elements in the batch. This constructs the "denominator" for each element x_i as if we were going to compute each inverse individually by means of $x_i^{-1} = \{\frac{(x_i)_a}{(x_i)_a^2+(x_i)_b^2}, \frac{-(x_i)_b}{(x_i)_a^2+(x_i)_b^2}\}$. The memory cost for this portion of the function is $2n$ `felm_t`'s. We save memory by using `den` temporarily to store ${(x_i)_b}^2$, then summing both powers into memory at `den`.

The succeeding sections of the function require the use of the temporary buffer `a`, adding an additional n `felm_t`'s to local memory usage.

Security Considerations. Recall the notion of a general side-channel attack: A side-channel attack is performed when an unauthorized individual is able to acquire information by measuring properties of the physical implementation of the system at hand. This can be done by analyzing the power consumption, timing properties, or electromagnetic leaks of a CPU while it operates on (or generates) confidential information.

In the context of information security, algorithms for performing operations over mathematical objects can be said to fall under one of two categories: *constant time* and *non-constant time* algorithms. Constant time algorithms are designed to protect confidential information from side-channel attacks, but come at the cost of computational efficiency.

In the $\texttt{SIDH}_\texttt{C}$ library, there are two distinct functions for computing field element inversions: `fp2inv751_mont` and `fp2inv751_mont_bingcd`. `fp2inv751_mont_bingcd` performs inversion by means of the binary GCD (greatest common denominator) algorithm, and is a *non-constant time* implementation. `fp2inv751_mont` is a *constant time* implementation, and as such runs slower than `fp2inv751_mont_bingcd` in nearly all cases, but protects against timing based side-channel attacks. They perform comparatively as such:

Procedure	Performance in clock cycles
`fp2inv751_mont`	68,881,331
`fp2inv751_mont_bingcd`	15,744,477,032

Take for example some private data c being manipulated or operated on by some algorithm $\mathbf{A}$. In order to be entirely certain that c in $\mathbf{A}(c)$ is not vulnerable to *any* imagineable side-channel attack it must be the case that the structure of $\mathbf{A}$ does not in anyway depend on the information stored in c.

```c
void pb_inv (const f2elm_t* vec, f2elm_t* dest, const int n) {
    felm_t t0[n];       //a portion of vec elements
    felm_t t1[n];       //b portion of vec elements
    felm_t den[n];      //denominator of vec elements
    felm_t a[n];

    // conversion to base field ------------------------------------//

    for (int i = 0; i < n; i++) {
        fpsqr751_mont((vec[i])[0], t0[i]);
        fpsqr751_mont((vec[i])[1], t1[i]);
        fpadd751(t0[i], t1[i], den[i]);
    }

    // upward-percolation phase ------------------------------------//

    fpcopy751(den[0], a[0]);
    for (int i = 1; i < n; i++) {
        fpmul751_mont(a[i-1], den[i], a[i]);
    }

    // inversion phase ---------------------------------------------//

    felm_t a_inv;
    fpcopy751(a[n-1], a_inv);
    fpinv751_mont_bingcd(a_inv);

    // downward-percolation phase ----------------------------------//

    for (int i = n-1; i >= 1; i--) {
        fpmul751_mont(a[i-1], a_inv, a[i]);
        fpmul751_mont(a_inv, den[i], a_inv);
    }

    // conversion back to extension field --------------------------//

    fpcopy751(a_inv, a[0]);

    for (int i = 0; i < n; i++) {
        fpmul751_mont(a[i],vec[i][0],dest[i][0]);
        fpneg751((vec[i])[1]);
        fpmul751_mont(a[i],vec[i][1],dest[i][1]);
    }
}
```

Figure 3.1: `pb_inv`

As will be illuminated in the following subsection, there are two settings in our implementation where `pb_inv` us called. In the first case, the elements passed to `pb_inv` are the constituents of Randall's public key as derived in `KeyGeneration_A`. Because Randall's public key values appear as public information in the signature (as commitment E_0) they needn't consider for protection from side-channel analysis.

In the second case, the inputs to `pb_inv` are the j-invariant representations of Bob and Randall's shared secret, as derived in `SecretAgreement_A` and `SecretAgreement_B`. When one of these secret agreement functions are used in the context of SIDH key exchange, the same j-invariant is used as the shared secret between party members A and B, and so would need to be protected against side-channel attacks. This is not the case in the context of signatures, however, because every signature includes the commitments E_1 which are precisely the shared secrets between the signer and Randall. And so this second case is also free from concerns of side-channel analysis.

Because our deployments of `pb_inv` are only concerned with public data, we are able to opt for `fp2inv751_mont` in the definition of our function and significantly save on execution cost. While there are no occurances of `pb_inv` in our implementation that require protection from side-channel analysis, there are scenarios in isogeny-based cryptography where `pb_inv` could be deployed over confidential information. In these cases, changes to the definition of `pb_inv` would need to made. Such scenarios are explored in Section 5.2.1.

3.2.2 Embedding Partial Batched Inversions

Recall Figure 2.6 which details the abstraction levels of the SIDH protocols as they relate to the modules of $\texttt{SIDH}_\texttt{C}$. We can further expand on this figure to illustrate how the Yoo et al. signature layer interoperates with the original $\texttt{SIDH}_\texttt{C}$ codebase. See Figure 3.2 - "SIDH_signature.c" signifies the C module added by Yoo et al., which implements Σ' :**KeyGen**, Σ' :**Sign**, and Σ' :**Verify** as they are outlined in Section 2.5. For the remainder of this section we will refer to these higher-level procedures as simply **KeyGen**, **Sign**, and **Verify**.

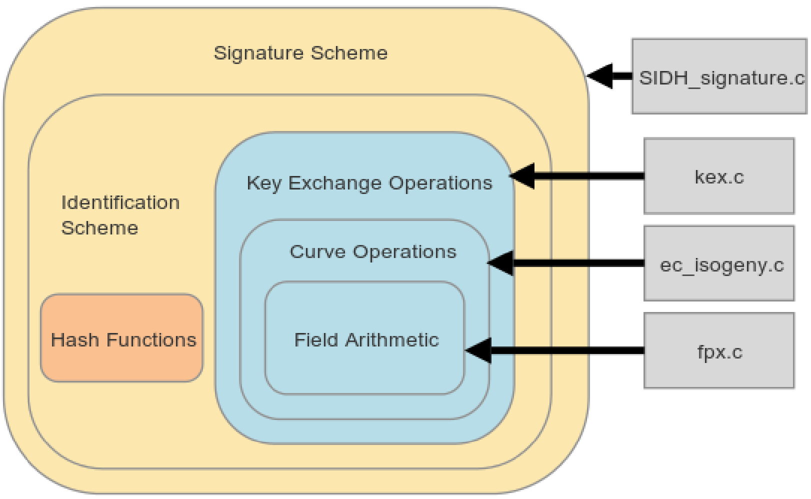


Figure 3.2: Relationship between SIDH based signatures & the Yoo et al. fork of the SIDH C library

Parallelizing Signatures. Recall now the construction of **Sign** and **Verify** from Section 2.5. The sign procedure requires running 2λ distinct instances of the underlying key exchange protocol, after which these instances are reproduced in **Verify** to check for their validity. It is clear that, because every 2λ iteration of **Sign** and **Verify** are entirely independent of each other, these procedures present themselves as embarrassingly parallel.[2]

[2]in the field of high performance computing, a problem that is trivially parallizable is often referred

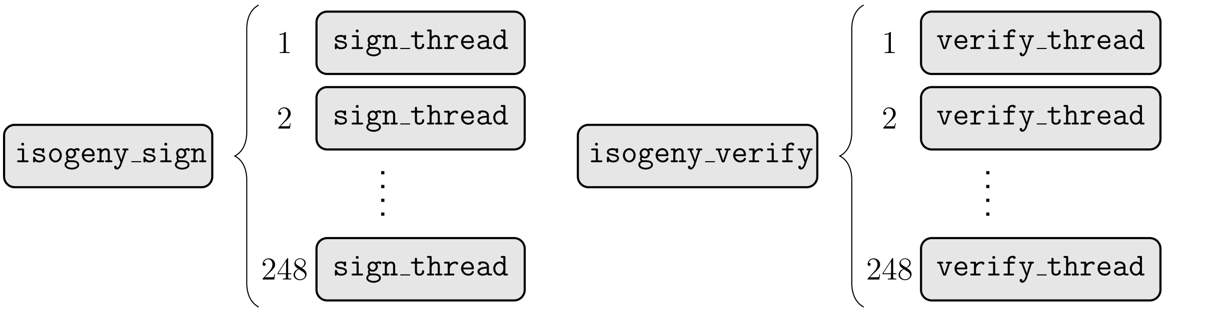


Figure 3.3: The implementations of **Sign** and **Verify**, divided into serial segments `isogeny_sign` and `isogeny_verify` and then parallel segments `sign_thread` and `verify_thread`.

This parallelization approach was exactly the one taken by Yoo et al. in their C implementation. Refer again to the `SIDH_signature.c` functions outlined in Figure **??**: `isogey_sign` acts as the entry point for **Sign** and spawns a POSIX thread for every instance of the procedure's for-loop. So now, in parallel, every thread spawned by `isogeny_sign` makes a call to `sign_thread`, which in turn performs Bob's interaction with Randall. This is outlined below in Figure 3.3. Verification proceeds analogously; `isogeny_verify` is executed and spawns POSIX threads executing `verify_thread` until all 2λ iterations are complete. λ here denotes the security level in bits (128 by default in SIDH), and so 248 threads are spawned in both `sign_thread` and `verify_thread`. Refer to the Figure directly below, roughly illustrating a parallel call-graph for `isogeny_sign` and `isogeny_verify`.

And so, there are two settings in which the same sequence of operations will be carried out 248 times in parallel. This means that we need only one occurance of an $\mathbb{F}_{p^2}$ inversion in either `sign_thread` or `verify_thread` to be able to fill a element batch of size 248, suitable for partial batched inversion.

Costella et al. have concisely outlined many of the $\mathtt{SIDH}_\mathtt{C}$ isogeny and point-wise functions in Table 1 of [CLN16]. Examinig this Figure, we note that there are only three candidate functions containing element inversion calls: `j_inv`, `inv_4_way`, and `get_A`. The fact that so few functions require inversions is, again, thanks to the design decisions outlined in Section 2.6.2.

`j_inv` is a function returning the j-invariant of a curve which, as the reader may recall, is used in the computation of the shared secret. If we refer back to our definitions of `Sign` and `Verify` (Algorithms 9 and 10, respectively) we note that `Sign` contains a call to `SecretAgreement_B` in every iteration of its for-loop. Similarly, `Verify` contains a call to `SercretAgreement_A` in roughly half of the iterations of its for-loop, and a call to `SecretAgreement_B` in the remaining iterations. This totals to 248 secret agreement computations in both signature signing and verifying procedures. This means that somewhere in the exeuction flow of `isogney_sign` and `isogeny_verify` there are calls to these secret agreement functions, illustrating the presence of 1 `j_inv` function call (and by extension, 1 extension field inversion,) in every signing and verification thread.

`inv_4_way` is a function which takes 4 $\mathbb{F}_{p^2}$ elements and returns each elements inversion by means of calculating only one inversion (via the same method outlined by **Batched-**

to as *embarrassingly* parallizable.

Inversion). This function is used in the key generation to process to invert the Z-values of the public key curve elements; $\phi(P)$, $\phi(Q)$, and $\phi(P-Q)$), so that they can be converted from projective to affine representation. Because every `sign_thread` execution represents Bob's key exchange with a distinct and random Randall, `KeyGeneration_A` must be called in each thread to generate Randall's public and private keys. This results in another candidate batch of size 248 for batched partial inversion.

`get_A`, while containing an extension field inversion, does not arise in the execution flow of the signature scheme. The potential candidacy of this function for partial batched inversion processing is discussed further in Section 4.

In Figure 3.4 we illustrate a heavily simplified call-graph for the `sign_thread` and `verify_thread`, demonstrating where in the execution pipeline `j_inv` and `inv_4_way` occur. The reader may suspect that, in `sign_thread` for example, the inversions in `SecretAgreement_B` and `KeyGeneration_A` could be batched together to form a batch of 512 elements and to reduce the total number of inversions in `isogeny_sign` to one. This is not possible, however, because the valid execution of `SecretAgreement_B` relies on information returned by `KeyGeneration_A`, and so these inversions must occur sequentially.

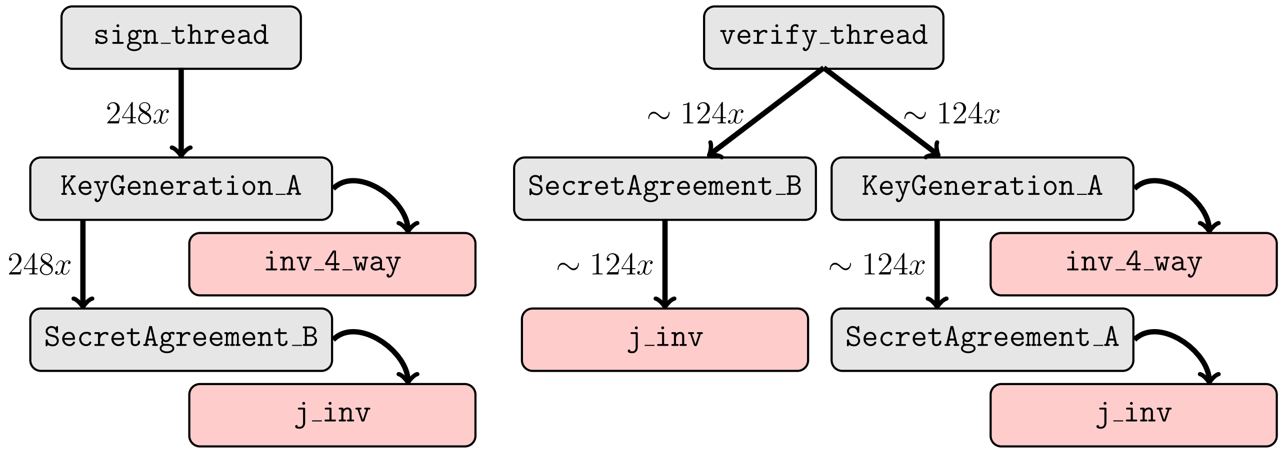


Figure 3.4: The execution flow of `sign_thread` and `verify_thread` as originally implemented by Yoo et al.

To enable batching across execution instances of `j_inv` and `inv_4_way`, we've supplied new functions `j_inv_batch` and `inv_4_way_batch`. These functions, upon reaching what were originally $\mathbb{F}_{p^2}$ inversions (calls to `fp2inv751_mont`), add their elements that are awaiting inversion to a buffer. Once the buffer of elements has reached its predefined capacity, the final thread to add its element executes `pb_inv` on the buffer. Each thread thereafter, having kept track of where in the buffer they entered their element, retrieves their now inverted element from the buffer returned by `pb_inv`.

To properly implement `pb_inv` in these functions, we modify every function along the call stack leading up to `j_inv` and `inv_4_way`: `SecretAgreement_A`, `SecretAgreement_B`, and `KeyGeneration_A`. Our modifications allow these functions to optionally pass a C struct we've defined which holds all of the information necessary for a successfull execution of `pb_inv`. We refer to this structure as `batch_struct`, and it holds the following: an integer `batchSize` denoting the number of elements in the batch, an integer `cntr` which

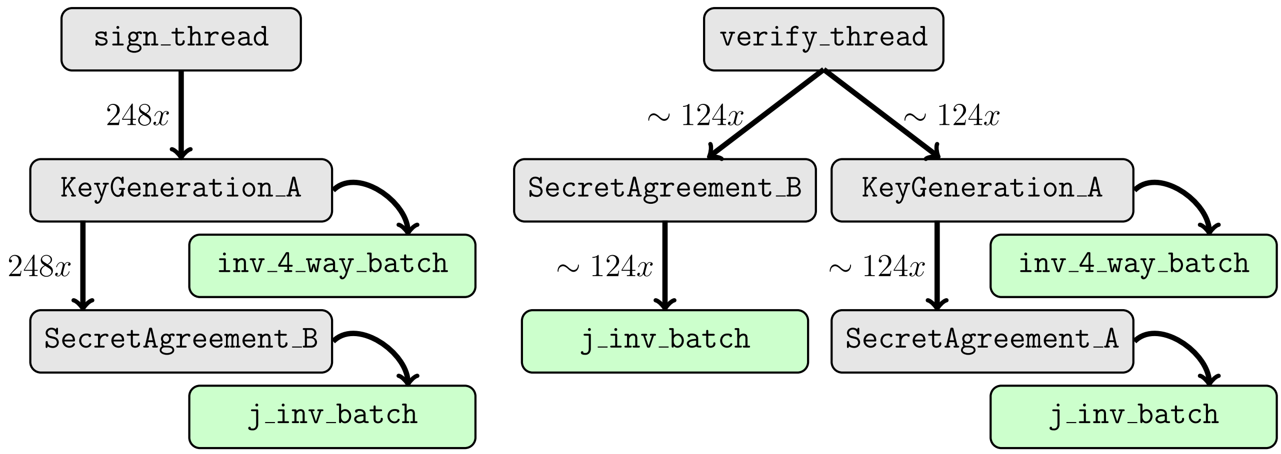


Figure 3.5: The execution flow of `sign_thread` and `verify_thread` when run with inversion batching enabled

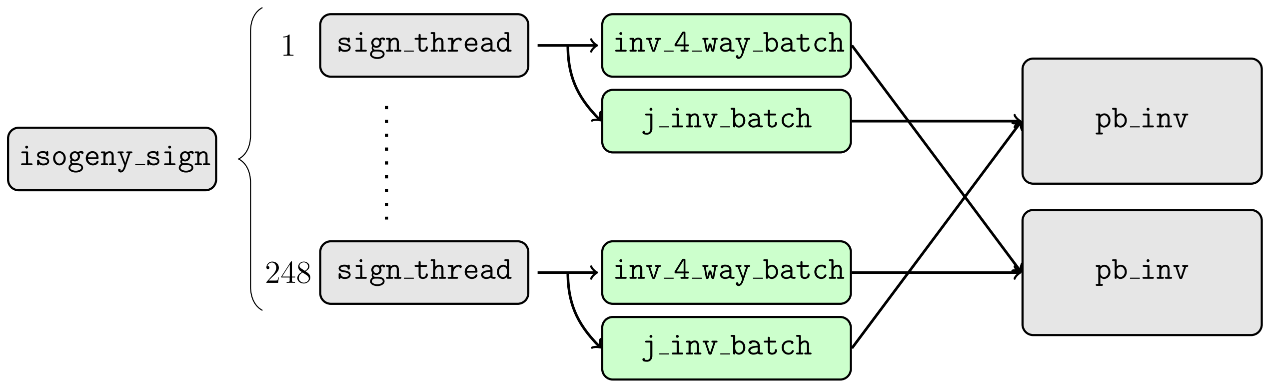
tracks how many elements are currently in the batch (and is invariably less than or equal to `batchSize`), an `f2elm_t` buffer `invArray` for storing the elements to be inverted, and an `f2elm_t` buffer `invDest` for storing the inversion results.

Once one of the aforementioned `kex.c` functions reaches its call to either `j_inv` or `inv_4_way`, the function checks whether the `batch_struct` it has been passed is `NULL`. If the `batch_struct` is defined, the call to `j_inv` or `inv_4_way` is replaced with a call to `j_inv_batch` or `inv_4_way_batch`, respectively.

A mutex lock can also be found in the `batch_struct`, allowing `j_inv` and `inv_4_way` to increment the size of the batch safely across threads. Each thread performs the following as it approaches the inversion call:

1. acquire the mutex lock
2. add element to be inverted to `invArray`
3. store the current value of `cntr` locally
4. increment `cntr`
5. release the lock

A semaphore has also been included in `batch_struct`, the function of which is to ensure that each thread knows to wait until the batch has been filled (248 elements in the signing case, 128 in the verification cases) before it attempts to access its inverted element. If the locally stored `cntr` is less than `batchSize`, the current thread waits on the semaphore. If the locally stored `cntr` is equal to `batchSize`, this implies the current thread is the last to add its element - this thread then carries out execution of `pb_inv` and upon completion posts the sempahore. After the semaphore has been posted, all other threads are able to resume execution and retrieve their now inverted elements.



C code for all of these functions (with comparable differences highlighted) can be found in Appendix A.

3.3 Results

Our results come in several forms. First, there are the execution-time results of `pb_inv` measured in clock cycles. Measurements of this first type are gathered from two environments, a general $\mathbb{F}_{p^2}$ environment constructed using the NTL C library, and the $\texttt{SIDH}_\texttt{C}$ exeuction environment using the API provided by Costello et al. The first environment allows us to meaure how the performance of `pb_inv` compares with an unbatched approach for arbitrarily sized moduli. Measurements in $\texttt{SIDH}_\texttt{C}$ are limited to the parameters of the library, but more closely reflect the performance gains in that particular system.

Following that, we measure the improvement in the performance of signature signing and verifying procedures offered by the inclusion of the batched partial inversion mechanism. These results are gathered both in a multi-core setting and a single-core environment, also measured in clock-cycles.

Modulus Size	Regular Batch	Partial Batched Inversion	Unbatched
32 Inversion	0.033685	0.00080204	
64 Inversion	0.00033685	0.00080204	
128 Inversion	0.00033685	0.00080204	
256 Inversion	0.00033685	0.00080204	
512 Inversion	0.00033685	0.00080204	
1024 Inversion	0.00033685	0.00080204	
2048 Inversion	0.00033685	0.00080204	

Modulus Size	Regular Batch	Partial Batched Inversion	Unbatched
32	0.00033685	0.00080204	
64	0.00033685	0.00080204	
128	0.00033685	0.00080204	
256	0.00033685	0.00080204	
512	0.00033685	0.00080204	
1024	0.00033685	0.00080204	
2048	0.00033685	0.00080204	

Modulus Size	Regular Batch	Partial Batched Inversion	Unbatched
32	0.00033685	0.00080204	
64	0.00033685	0.00080204	
128	0.00033685	0.00080204	
256	0.00033685	0.00080204	
512	0.00033685	0.00080204	
1024	0.00033685	0.00080204	
2048	0.00033685	0.00080204	

The two figures below provide benchmarks for KeyGen, Sign, and Verify procedures with both batched partial inversion implemented (in the previously mentioned locations) and not implemented. All benchmarks are averages computed from 100 randomized sample runs. All results are measured in clock cycles.

Two different machines were used for benchmarking. System A denotes a single-core, 1.70 GHz Intel Celeron CPU. System B denotes a quad-core, 3.1 GHz AMD A8-7600.

Procedure	System A Without Batching	System A With Batching
KeyGen	68,881,331	68,881,331
Signature Signing	15,744,477,032	15,565,738,003
Signature Verification	11,183,112,648	10,800,158,871

Procedure	System B Without Batching	System B With Batching
KeyGen	84,499,270	84,499,270
Signature Sign	10,227,466,210	10,134,441,024
Signature Verify	7,268,804,442	7,106,663,106

System A: With inversion batching turned on we notice a 1.1 % performance increase for key signing and a 3.5 % performance increase for key verification.

System B: With inversion batching turned on we a observe a 0.9 % performance increase for key signing and a 2.3 % performance increase for key verification.

3.3.1 Analysis

It should first be noted that, because our benchmarks are measured in terms of clock cycles, the difference between our two system clock speeds should be essentially ineffective.

In the following table, "Batched Inversion" signifies running `pb_inv` on 248 $\mathbb{F}_{p^2}$ elements.

Procedure	Performance
Batched Inversion	1721718
$\mathbb{F}_{p^2}$ Montgomery Inversion	874178

The following are all measured in clock cycles, as the computed average of 1000 distinct executions:

Modulus Size	Multiplication	Inversion Time
32	0.00033685	0.00080204
64	0.00033685	0.00080204
128	0.00033685	0.00080204
256	0.00033685	0.00080204
512	0.00033685	0.00080204
1024	0.00033685	0.00080204
2048	0.00033685	0.00080204

Chapter 4

Compressing Signatures

Our second contribution, also in the form of an addition to the $\texttt{SIDH}_{\texttt{C}}$ signature extension, is a mechanism for compressing signatures. This chapter will cover the compression technique used. This chapter, much like the last, will be split into three sections: a brief coverage of the employed compression technique, the details of our implementation and integration of this technique into $\texttt{SIDH}_{\texttt{C}}$, and finally an analysis of the results of this contribution.

In the first section of this chapter, we discuss the SIDH public key compression technique resulting from combined efforts of Azerderakhsh et al. [AJK+16] and Costello et al. [CJL+17]. We attempt to provide a sufficient overview of the technique while only covering in detail the components that are of significant relevance to our implementation. Those who seek to better understand the ins and outs of this technique should direct themselves to the original papers.

The second section covers in detail how we apply this public key compression to Yoo et al. signatures. We make use of the functions offered by Costello et al. which implement the previously mentioned technique. This code was first made available in the second installment of Microsoft's SIDH library [**?**].

Finally, we round off the Chapter with an analysis of the memory improvement offered by this technique. We contrast this spatial improvement with the computational cost of compressing points, and discuss the practicality of employing this technique.

4.1 SIDH Key Compression Background

In this section we will briefly cover the literature surrounding the compression technique that we employ. This technique was first outlined by Azerderakhsh et al. [AJK+16] and later improved upon by Costello et al. [CJL+17]. Here we investigate the details of these works that are relevant to our implementation.

First, recall from Section 2.3.1 the structure of an SIDH public key, denoted pk;

$$pk = (E, P, Q)$$

Where E is a supersingular elliptic curve and P and Q are elliptic curve points such that $P, Q \in E$. Recall that E can be sufficiently represented by one $\mathbb{F}_p$ element which denotes A from the following definition of E:

$$E : y^2 = x^3 + Ax + B.$$

A sufficiently represents E in this context because in $\texttt{SIDH}_\texttt{C}$ we are concerned only with curves where $B = 0$.

P and Q, on the other hand, can each be represented by their x-coordinate (two $\mathbb{F}_p$ elements) and a single bit determining the correct y-coordinate. Therefore, without more sophisticated compression, an $\texttt{SIDH}_\texttt{C}$ public key can be represented with $6 \log p$ bits.

4.1.1 Compressing SIDH Public Keys

Recall the discrete logarithm problem in the context of elliptic curves: given an elliptic curve group $E(K)$ and points $P, Q \in E(K)$, find n such that $P = nQ$. The two-dimensional discrete log problem is then the following: given an elliptic curve group $E(K)$, two points $\{R_1, R_2\}$ generating a subgroup H of $E(K)$, and an element $P_H \in H$, compute α and β such that:

$$P_H = \alpha R_1 + \beta R_2$$

The Pohlig-Hellman algorithm can be applied to solve the discrete logarithm problem in groups whose order is a smooth integer [?], and there is a variation of this algorithm which solves this two-dimensional discrete log problem with time complexity $O(\sqrt[q]{[}\log p])$, where q is the largest prime dividing $|H|$ [?].

Azerderakhsh et al. show that an SIDH public key can be compressed in the following way. Taking Alice's SIDH key pair, for example, we have her public key $pk_A = (E_A, \phi_A(P_B), \phi_A(Q_B))$ and her private key $sk_A = m_A$ such that $ker(\phi_A) = \langle P_A + [m_A]Q_A \rangle$. Because $\{P_B, Q_B\}$ generates the torsion subgroup $E_A[\ell_B^{e_B}]$, we have that $\phi_A(P_B) \in E[\ell_B^{e_B}]$ and $\phi_A(Q_B) \in E[\ell_B^{e_B}]$. Thus, the Pohlig-Hellman algorithm can be used to resolve $\phi_A(P_B) = [\alpha_P]R_1 + [\beta_P]R_2$ and $\phi_A(Q_B) = \alpha_Q R_1 + \beta_Q R_2$ where $\{R_1, R_2\}$ is a basis for $E_A[\ell_B^{e_B}]$ [AJK+16].

Then, instead of sending Bob $(E_A, \phi_A(P_B), \phi_A(Q_B))$, Alice can send $(E_A, \alpha_P, \beta_P, \alpha_Q, \beta_Q)$.[1] And so, as long as Alice and Bob can seperately generate the same $\{R_1, R_2\}$, they can both sufficiently represent one anothers public keys with only $4 \log p$ bits.

Constructing the Basis. Constructing R_1 and R_2 can be done with a relatively simple yet time consuming process. We will continue to use the compression of Alice's public key, pk_A, as our example.

1. Choose a random point $P \leftarrow_\$ E(\mathbb{F}_{p^2})$.
2. Multiply P by $\ell_B^{e_B} \cdot f$ to obtain P', the order of which will divide $\ell_B^{e_B}$.
3. Check the order of P' by multiplying it by powers of ℓ_A until the identity is given.
4. If the order is $\ell_B^{e_B}$, set $R_1 = P'$, otherwise return to step one.
5. Repeat the same process for a new random point Q until Q' of order $\ell_B^{e_B}$ is found.
6. Check that Q' is independent of R_1 by computing their Weil pairing: $e(R_1, Q')$.

[1] The approach outlined by Azerderakhsh et al. involves sending the j-invariant of E, which can be represented with the same amount of space as one $\mathbb{F}_{p^2}$ element. Because we are working in $\texttt{SIDH}_\texttt{C}$ where we can already represent curves with one $\mathbb{F}_{p^2}$ element, we omit this detail.

7. If the pairing results in anything other than 1, set $R_2 = Q'$, otherwise return to step 5.

The same (R_1, R_2) pair will be derived by both Alice and Bob if they use a psuedorandom number generator for generating P and Q AND they run their PRNGs with identical seeds [**?**].

The literature, to our knowledge, has thus far neglected the details of generating and transmitting this common seed necessary for basis generation. We note that the seed can be generated by the signer (using any PRF of their liking) and transmitted

Decompressing Public Keys. Decompression for this technique varies depending on the setting, but for our purpose we are concerned only with how decompression is done for SIDH key exchange. Bob computes the basis $\{R_1, R_2\}$ by seeding his PRNG with the same value as Alice. Bob then uses $\alpha_P, \beta_P, \alpha_Q$ and β_Q to recompute $\phi_A(P_B)$ and $\phi_A(Q_B)$. Then Bob computes the isogeny $\phi'_B : E_A \to E_{AB}$ with $ker(\phi'_B) =$ [AJK+16].

Alice then acts identically on Bob's now compressed public key, pk_B, and the two arrive at the same shared secret, the j-invariant of E_{AB}, just as in the original SIDH key exchange.[2]

4.1.2 Improvements to SIDH Key Compression

The work of Costello et al. further developed this approach to achieve public key sizes of $\frac{7}{2}\log p$ [CJL+17]. In addition to this, Costello et al. also outline several algorithmic improvements which decrease the runtime of this compression mechanism.

Many of the algorithms offered by Costello et al. can be treated as black-boxes in our setting, and so finer grain details of their work on efficient compression are ommitted.

Improved Compression. Take Alice's public key pk_A, compressed via the Azerderakhsh et al. technique, to be $(E_A, \alpha_P, \beta_P, \alpha_Q, \beta_Q)$. Therefore we have

$$P = \alpha_P R_1 + \beta_P R_2,$$

$$Q = \alpha_Q R_1 + \beta_Q R_2;$$

Where $\{R_1, R_2\}$ forms a basis of $E_A[\ell_A^{e_A}]$, and P and Q are exactly the elliptic curve point components of Alice's original, uncompressed public key.

From here, Costello et al. note the following: The end goal of the key exchange (in our running example) is for Bob to compute $\langle P + m_B Q\rangle$, where m_B is Bob's secretly generated value. Given that P has order $n = \ell_A^{e_A}$, we have that either $\alpha_P \in \mathbb{Z}_n^*$ or $\beta_P \in \mathbb{Z}_n^*$, and so it follows that

$$\langle P + m_B Q\rangle = \begin{cases} \langle \alpha_P^{-1} P + \alpha^{-1} m_B Q\rangle & \text{if } \ \alpha_P \in \mathbb{Z}_n^* \\ \langle \beta_P^{-1} P + \beta^{-1} m_B Q\rangle & \text{if } \ \beta_P \in \mathbb{Z}_n^* \end{cases}$$

And so, computing $\langle P + m_B Q\rangle$ to arrive at the shared secret does not require recomputing P and Q. Instead, the scalar factors of P and Q with respect to the generated basis can be normalized to yield

$$(\alpha_P^{-1} P, \alpha_P^{-1} Q) = \begin{cases} (R_1 + \alpha_P^{-1}\beta_P R_2, \alpha_P^{-1}\alpha_Q R_1 + \alpha_P^{-1}\beta_Q R_2 & \text{if } \ \alpha_P \in \mathbb{Z}_n^* \\ (R_1 + \beta_P^{-1}\alpha_P R_2, \beta_P^{-1}\alpha_Q R_1 + \beta_P^{-1}\beta_Q R_2 & \text{if } \ \beta_P \in \mathbb{Z}_n^* \end{cases}$$

[2] We have omitted details from this method of compression that are concerned with potential *twists* of Alice and Bob's curves when compressing public keys, as this does not play a role in our implementation.

And, thus, Alice has reduced the information she needs to send over the wire from $(E_A, \phi_A(P_B), \phi_A(Q_B))$ to the following:

$$pk_A = \begin{cases} (E_A, 0, \alpha_P^{-1}\beta_P, \alpha_P^{-1}\alpha_Q, \alpha_P^{-1}\beta_Q) & \text{if } \ \alpha_P \in \mathbb{Z}_n^* \\ (E_A, 1, \beta_P^{-1}\alpha_P, \beta_P^{-1}\alpha_Q, \beta_P^{-1}\beta_Q) & \text{if } \ \beta_P \in \mathbb{Z}_n^* \end{cases}$$

Or, alternatively we write

$$pk_A = \begin{cases} (E_A, 0, \zeta_P, \alpha'_Q, \beta'_Q) & \text{if } \ \alpha_P \in \mathbb{Z}_n^* \\ (E_A, 1, \zeta'_P, \alpha'_Q, \beta'_Q) & \text{if } \beta_P \in \mathbb{Z}_n^* \end{cases}$$

For readability. This reduction from 4 $\mathbb{Z}_n^*$ elements to 3 takes Alice's compressed public key from $4 \log p$ bits to $\frac{7}{2} \log p$ bits.

Alternative Decompression. Due to the loss of information in either α_P or β_P, an alternative route to secret agreement is required. For a compressed public key $(E, b, \zeta_P, \alpha_Q, \beta_Q)$ there exists a $\gamma \in \mathbb{Z}_n^*$ such that

$$(\gamma^{-1}P, \gamma^{-1}Q) = \begin{cases} (R_1 + \zeta_P R_2, \alpha_Q R_1 + \beta_Q R_2) & \text{if } \ b = 0 \\ (\zeta_P R_1 + R_2, \alpha_Q R_1 + \beta_Q R_2) & \text{if } \ b = 1 \end{cases}$$

The verifier could reconstruct $\{R_1, R_2\}$ to produce $\langle P + mQ \rangle$ by computing P and Q from $\zeta_P, \alpha_Q, \beta_Q$ with R_1 and R_2, and then multiplying Q by their private key m. This would require a 1-dimensional and a 2-dimensional scalar multiplication of points on the curve E. Costello et al. note instead that

$$\langle P + mQ \rangle = \begin{cases} \langle (1 + m\alpha_Q R_1 + (\zeta_P + m\beta_Q) R_2 \rangle & \text{if } \ b = 0 \\ \langle (\zeta_P + m\alpha_Q) R_1 + (1 + m\beta_Q) R_2 \rangle & \text{if } \ b = 1 \end{cases}$$

And since $n = l^e$ we have $(1 + m\alpha_Q), (1 + m\beta_Q) \in \mathbb{Z}_n^*$, giving

$$\langle P + mQ \rangle = \begin{cases} \langle R_1 + (1 + m\alpha_Q)(\zeta_P + m\beta_Q) R_2 \rangle & \text{if } \ b = 0 \\ \langle (1 + m\beta_Q)(\zeta_P + m\alpha_Q) R_1 + R_2 \rangle & \text{if } \ b = 1 \end{cases}$$

Reducing decompression to a single 1-dimensional scalar multiplication of a point on E along with a handful of $\mathbb{F}_{p^2}$ operations.

4.2 Implementation Details

In this section we demonstrate how the previously detailed public key compression technique can be used to compress Yoo et al.'s isogeny-based signatures. Again, we will turn to the `SIDH_C` library and reference portions of C code (some contributed by Patrick Longa [?], some by Yoo and his associates [YAJ+17], and some by us,) to investigate details of our implementations performance.

Recall from 2.5 the structure of a Yoo et al. signature, σ:

$$\sigma = (com, ch, h, resp)$$

Where

- *com* is a list of 2λ pairs of supersingular elliptic curves: $\{(E_{1,1}, E_{2,1}), (E_{1,2}, E_{2,2}), ..., (E_{1,2\lambda}, E_{2,2\lambda})\}$,
- *ch* is a list of 2λ randomly chosen bits,
- *resp* is a list of size 2λ where each element is either a single elliptic curve point $\psi_R(S)$, where ψ_R is Randall's isogeny and S is the signers secretly generated point, or the pair of points $(R, \phi(R))$, where R is Randall's secretly generated point, and ϕ is the signers isogeny.
- h is a list of 2λ queries to a random oracle $\mathbf{G}$, such that $h_i = \mathbf{G}(resp_i)$

Recall also from Subsection 2.6.1 the following definitions:

- an `felm_t` denotes a $\mathbb{F}_p$ element, e.g. a 751-bit element in $\texttt{SIDH}_\texttt{C}$,
- an `f2elm_t` denotes an $\mathbb{F}_{p^2}$ element, or two `felm_ts`,
- a `point_affine` denotes an elliptic curve point represented in affine space (two `f2elm_ts`), and
- a `point_proj` denotes an elliptic curve point represented in projective space (two `f2elm_ts`).

The representation of σ in $\texttt{SIDH}_\texttt{C}$ (as implemented by Yoo et al.) has a few noteworthy differences. Signatures in this setting are defined via a C `struct` in the following way:

```
struct Signature {
    f2elm_t *Commitments1[NUM_ROUNDS];
    f2elm_t *Commitments2[NUM_ROUNDS];
    unsigned char *HashResp;
    felm_t *Randoms[NUM_ROUNDS];
    point_proj *psiS[NUM_ROUNDS];
};
```

With `NUM_ROUNDS` equal to 2λ. The bit-level security of a given signature, then, can be computed as `NUM_ROUNDS`/2.

`Commitments1` is an array containing the first entry from every pair in *resp*, e.g. $\{E_{1,1}, E_{1,2}, ..., E_{1,2\lambda}\}$, and `Commitments2` holds the second entry from each pair.

`HashResp` contains the elements of h. In practice the Keccak function[3] is used in place of the random oracle $\mathbf{G}$. 32-byte hash digests are computed using Keccak such that `HashResp[i]` = `Randoms[i/2]` if the challenge bit *ch* is 0, and `HashResp[i]` = `psiS[i/2]` if *ch* is 1.

`Randoms` is an array of λ `felm_t`'s. The element at index i of `Randoms` holds m_{Ri}, and represents Randall's secretly generated $\mathbb{F}_p$ value for iteration i of the signing algorithm. These values sufficiently represent the elements of *resp* which take the form $(R, \phi(R))$ for two reasons:

1. R can be reconstructed using the torsion subgroup generating points that correspond to Randall ($R = P_A + m_{Ri}Q_A$ if Bob is signing, and $R = P_B + m_{Ri}Q_B$ if Alice is signing)

[3] Keccak is a cryptographic hash function from which the newly standardized SHA-3 is based.

2. Because isogenies are (structure preserving) morphisms, it holds that $R = P + [m_R]Q \Rightarrow \phi(R) = \phi(P) + [m_R]\phi(Q)$. Thus, because $\phi(P)$ and $\phi(Q)$ are members of the signers public key, m_R is sufficient for reconstructing $\phi(R)$.

Lastly, `psiS` denotes the elements of *resp* which take the form of $\psi_R(S)$. These points cannot be represented by a single `felm_t` (as in `Randoms`) because doing so would leak the signers private information.

And so, the total size of an uncompressed $\texttt{SIDH}_\texttt{C}$ signature, in bytes, is $(\texttt{sizeof(felm\_t)} + \texttt{sizeof(f2elm\_t)} + \texttt{sizeof(f2elm\_t)} + \texttt{sizeof(point\_proj)} + 32) \cdot 2\lambda + (\texttt{sizeof(felm\_t)} + \texttt{sizeof(point\_proj)}) \cdot \lambda =$.

4.2.1 $\psi(S)$ Compression

For the following two subsections we will assume the signer to be Bob (e.g. using B values) and the verifier to be Alice (e.g. using A vaues). This is done only for simplicities sake - to reverse the roles one need only to swap all blue variables with their red counterparts, and vice-versa. Additionally, function names ending in `_B` would then need to be replaced with their `_A` alternatives, and vice-versa.

Our contribution uses the compression technique covered in the previous section to compress every element of `psiS` individually. Consider the following.

Each element of `psiS` has the form $\psi_R(S) = \psi_R(P_B) + [m_B]\psi_R(Q_B)$. (P_B, Q_B) generates the torsion subgroup $E[\ell_B^{e_B}]$ so we know that S has order $\ell_B^{e_B}$ (e.g. $[\ell_B^{e_B}]P = \mathcal{O}$) and, because isogenies preserve the identity we know $[\ell_B^{e_B}]\psi_R(S) = \mathcal{O}$ and $\psi_R(S) \in E_R[\ell_B^{e_B}]$. Therefore, we can be certain that the compression technique of the previous subsection can be applied to all elements of `psiS` if we chose our basis $\{R_1, R_2\}$ such that it generates $E_R[\ell_B^{e_B}]$.

Recall that Bob has private key $sk_B = m_B$ from which we can generate $S = P_B + [m_B]Q_B$ and $\phi_B : E \to E/\langle S \rangle$, and public key $pk_B = (E_B, \phi_B(P_A), \phi_B(Q_A))$. Recall also from previous sections that the general procedure for signature signing begins with Bob calling the `isogeny_sign` function, which in turn spawns 2λ threads, each executing `sign_thread`. Each of these threads has an identifier `r`, and performs the following via `sign_thread`:

1. Makes a call to `KeyGeneration_A` to generate Randall's keypair (pk_R, sk_R)
   - $sk_R = m_R$
   - $pk_R = (E_R, \psi_R(P_B), \psi_R(Q_B))$ where $\psi_R : E \to E_R$
2. Sets `Randoms[r]` $\leftarrow sk_R$
3. Sets `Commitments1[r]` $\leftarrow E_R$
4. Performs `SecretAgreement_B` with sk_B and pk_R to generate $(E_{BR}, \psi_R(S))$.
5. Sets `Commitments2[r]` $\leftarrow E_{BR}$
6. Sets `psiS[r]` $\leftarrow \psi_R(S)$

And so, if we wish to apply point compression to the elements of `psiS`, we must invoke our compression function within every `sign_thread` instance, after the execution of `SecretAgreement_B`. We provide a function `CompressPsiS`, based on the original point compression function of Costello et al. [CJL+17]. This modified program path for `sign_thread` is outlined in Figure 4.1.

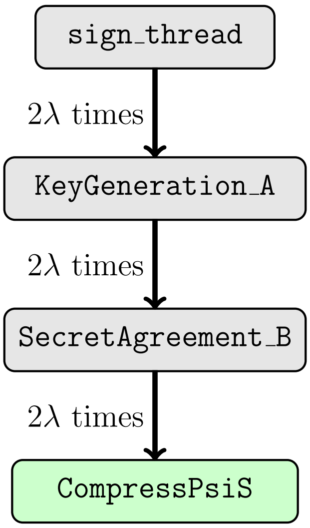


Figure 4.1: The general execution flow of `sign_thread` with the addition of $\psi(S)$ compression

The `CompressPsiS` Function. On round `r` of signature signing, our compression function takes the following as parameters:

- the `point_proj psiS[r]`,
- an `f2elm_t A`, denoting E_R (equivalently `Commitments1[r]`), and
- the set of curve parameters `CurveIsogeny`, which we set equal to `SIDHp751`.

And the output of `CompressPsiS` includes:

- A number `CompressedPsiS` $\in E_R[\ell_B^{e_B}]$, and
- the bit `compBit`.

Our compression algorithm then follows closely the technique of Azerderakhsh et al., making use of the efficient algorithms provided by Costello and company. An abstracted and generalized version of this function can be seen in Figure 16 as **CompressPsiS**. For our concrete C definition see Appendix **??**.

`CompressPsiS` returns an element of $E_R[\ell_B^{e_B}]$ (denoted by γ in Figure 16) which becomes `compPsiS[r]`. This element can be represented with fewer bytes than an element of $\mathbb{F}_{p^2}$ because $|E_R[\ell_B^{e_B}]| = \ell_B^{e_B}$. The `Signature` structure used to construct σ is necessarily modified as follows:

```
struct Signature {
    f2elm_t *Commitments1[NUM_ROUNDS];
    f2elm_t *Commitments2[NUM_ROUNDS];
```

Algorithm 16 – **CompressPsiS**($\psi_R(S)_{\mathtt{r}}$, E_R, $User$)

```
if User = Alice then
    l^e ← ℓ_A^{e_A}
if User = Bob then
    l^e ← ℓ_B^{e_B}
Check that ψ_R(S_r) has order l^e
Generate(R_1, R_2) as the basis for E_R[l^e]
Compute α, β such that ψ_R(S)_r = αR_1 + βR_2
if α mod l ≠ 0 then
    b ← 0
    γ ← α^{-1}β
else
    b ← 1
    γ ← β^{-1}α
return (γ, b)
```

```
    unsigned char *HashResp;
    felm_t *Randoms[NUM_ROUNDS];
    point_proj *psiS[NUM_ROUNDS];
    digit_t compPsiS[NUM_ROUNDS][NWORDS_ORDER];
    int compBit[NUM_ROUNDS];
    int compressed;
};
```

Note the additional bit value `compressed` in the `Signature` struct. It is important that this bit is packaged as part of the final signature so that the verifier knows whether or not they need to perform decompression. We have also included an array of bits, 2λ in size, such that `compPsiS[r] =` $\alpha^{-1}\beta$ if `compBit[r] = 0` and `compPsiS[r] =` $\beta^{-1}\alpha$ otherwise (b in Figure 16).

4.2.2 Verifying A Compressed Signature

Decompression can be embedded into `verify_thread` rather simply. On the code path where $ch = 1$, the verifier (Alice in our case) simply needs to run decompression on `compPsiS[r]` and `compBit[r]` before she runs `SecretAgreement_B`. Figure 4.2 reflects this modified code path at a high level.

However, if we look back to the SIDH public key decompression mechanism described in Subsection 4.1.2, we note again that the aim is *not* to reconstruct the originally compressed values. Instead, an instance of compressed SIDH key exchange is able to arrive at the shared secret $j(E_{AB})$ between Alice and Bob without reconstructing the original points, by absorbing the constants into the shared secret value.

This means that in transmitting `compPsiS[r]` in its normalized form we lose the ability to reconstruct $\psi_R(S)$ exactly. We are only able to construct the point $S^0 = R_1$`compPsiS[r]`R_2, or $S^0 =$ `compPsiS[r]`$R_1 + R_2$ depending on `compBit[r]`. Looking back to the definition of the **Verify** procedure (Figure **??**), we note that the verification path where $ch = 1$ requires that Alice checks 1. that $\psi_R(S)$ has order $\ell_B^{e_B}$, and 2. that $\psi_R(S)$ generates the kernel of the isogeny $\psi_R' : E_R \to E_{BR}$. Thus, we needn't return to the original $\psi_R(S)$ value to successfully verify because

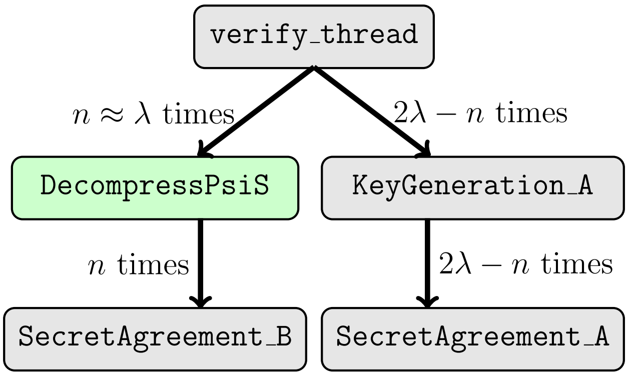


Figure 4.2: The general execution flow of `verify_thread` with the addition of $\psi(S)$ decompression

1. if R_1 and R_2 have order $\ell_B^{e_B}$ (which by definition they do) and `compPsiS[r]` is a multiple of ℓ_B, then S^0 is guaranteed to have order $\ell_B^{e_B}$, and
2. because S^0 and $\psi_R(S)$ having equivalent order implies that they generate the same kernel.

And so, `DecompressPsiS` can be defined to generate the same basis $\{R_1, R_2\}$ as in `CompressPsiS` and then compute $S^0 = R_1 + [$`compPsiS[r]`$]R_2$ if `compBit` $= 0$ and $S^0 = [$`compPsiS[r]`$]R_1 + R_2$ otherwise. The resulting S^0 can then be passes to `SecretAgreement_B` just as in uncompressed verification and the verification will run successfully. Figure 17 outlines the general functioning of `DecompressPsiS` at a high level, for our specific C implementation see Appendix A.

Algorithm 17 – **DecompressPsiS**(γ, b, E_R, $User$)

if $User = Alice$ **then**
    $l^e \leftarrow \ell_B^{e_B}$
if $User = Bob$ **then**
    $l^e \leftarrow \ell_A^{e_A}$
Check that $\psi_R(S_r)$ has order l^e
Generate(R_1, R_2) as the basis for $E_R[l^e]$
if $b = 0$ **then**
    $b \leftarrow 0$
    $S^0 \leftarrow R_1 + [\gamma]R_2$
else
    $b \leftarrow 1$
    $S^0 \leftarrow [\gamma]R_1 + R_2$
return S^0

4.2.3 Combining Batching & Compression

Looking more closely at the `CompressPsiS` function, we highlight the following segment:

```
struct Signature {
```

```
    f2elm_t *Commitments1[NUM_ROUNDS];
    f2elm_t *Commitments2[NUM_ROUNDS];
    unsigned char *HashResp;
    felm_t *Randoms[NUM_ROUNDS];
    point_proj *psiS[NUM_ROUNDS];
    digit_t compPsiS[NUM_ROUNDS][NWORDS_ORDER];
    int compBit[NUM_ROUNDS];
    int compressed;
};
```

Including compression in the signature codebase introduces yet another cross-thread $\mathbb{F}_{p^2}$ inversion.

4.3 Results

Let σ denote an umcompressed Yoo et al. isogeny-based signature. The size of σ can be computed as the sum of the sizes of its constituent parts. To reiterate, σ is composed of:

- 4λ $\mathbb{F}_{p^2}$ elements (the commitments), totaling 384λ bytes,
- 2λ Keccak hash-function digests, totaling 64λ bytes,
- $\sim\lambda$ elements of $\mathbb{Z}/\ell_A^{e_A}\mathbb{Z}$ (Randall's secret key value), totaling $\sim 48\lambda$ bytes, and
- $\sim\lambda$ elliptic curve points ($\psi_R(S)$ points), totaling $\sim 92\lambda$ bytes

Therefore, in the case where the challenge bits are equally divided between 1 and 0, $|\sigma| = 688\lambda$bytes.

Let $\sigma_{\text{compressed}}$ denote a compressed Yoo et al. signature using the techniques outlined by Azerderakhs, Costello, and company. $\sigma_{\text{compressed}}$ swaps the buffer of $\sim\lambda$ elliptic curve points for one of $\sim\lambda$ $\mathbb{Z}/\ell_B^{e_B}\mathbb{Z}$ elements. This subtracts $\sim 192\lambda$ bytes from the size of the signature and adds $\sim 48\lambda$ bytes - reducing the signature size by $\sim 144\lambda$ bytes for a final size of 544λ bytes.

Looking at the storage requirements for the following. Figure **??** outlines the space requirements for different data structures in $\texttt{SIDH}_\texttt{C}$. From this information we can build up the data requirements based on

Our technique can reduce the size of $\texttt{SIDH}_\texttt{C}$ signature compression from ___ bits to ___ bits.

4.3.1 Performance of Signature Compression

regular
keygen: 40,702,754 sign: 4,696,942,672 verify: 3,302,858,925

keygen: 40,831,316 sign: 11,667,991,571 verify: 5,245,975,977

Because of how the signature extension of $\texttt{SIDH}_\texttt{C}$ was originally written, our benchmarks reflect the time it takes for the signer to compress `psiS[r]` for ALL `r` (all 2λ iterations). In a finalized implementation of this system, significant time can be saved if the signer computes the challenge for an interaction before compressing `psiS`; this will allow them to avoid running the compression procedure in cases where `psiS` is not part of the final signature, effectively reducing the time spent compressing points by $\sim$50%.

Chapter 5

Conclusion

In this chapter we provide our final set of metrics for the performance of the original isogeny-based signature scheme, our batched inversion implementation of the protocol, and our implementation feauturing $\psi(S)$ compression. We also offer measurements for how the compressed version of the protocol performs when combined with batched inversion. We outline also how the compressed protocol offers an additional route for batching inversions, yielding further performance improvements.

Following the debriefing of our results, we offer one final section wherein we discuss the ramifications of our work in a general context. In this section we also discuss some possible future work to further progress the practicality of isogeny-based cryptography.

5.1 Performance Results

In this section we compile performance metrics for the original Yoo et al. signature scheme, our batched-inversion signature scheme, our compressed signature scheme, and also our combined compression with batched inversions implementation. For each of these implementations we show the average cycle time for **Sign** and **Verify** as well as the standard deviation. These measurements are outlined in 5.1 (where "C+B" denotes the combined compression with batching scheme). These averages are derived from 100 subsequent runs of each implementation.

We include graphical representations of our collected data, these can be found in 5.2, 5.3, **??**, and **??**. The collected measurements for the unedited signature depicts several

	Average Cycles	Standard Deviation
Original Sign	4,611,010,573.608	300,643,097.882
Original Verify	3,221,544,625.173	263,674,018.528
Batched Sign	4,552,062,482.520	18,113,276.904
Batched Verify	3,173,340,239.461	68,672,478.339
Compressed Sign	10224610996	b
Compressed Verify	4472444449	b
C+B Sign	10105688822	b
C+B Verify	4327225903	b

Figure 5.1: Average performance and standard deviation in clock cycles for all versions of the Yoo et al. signature scheme.

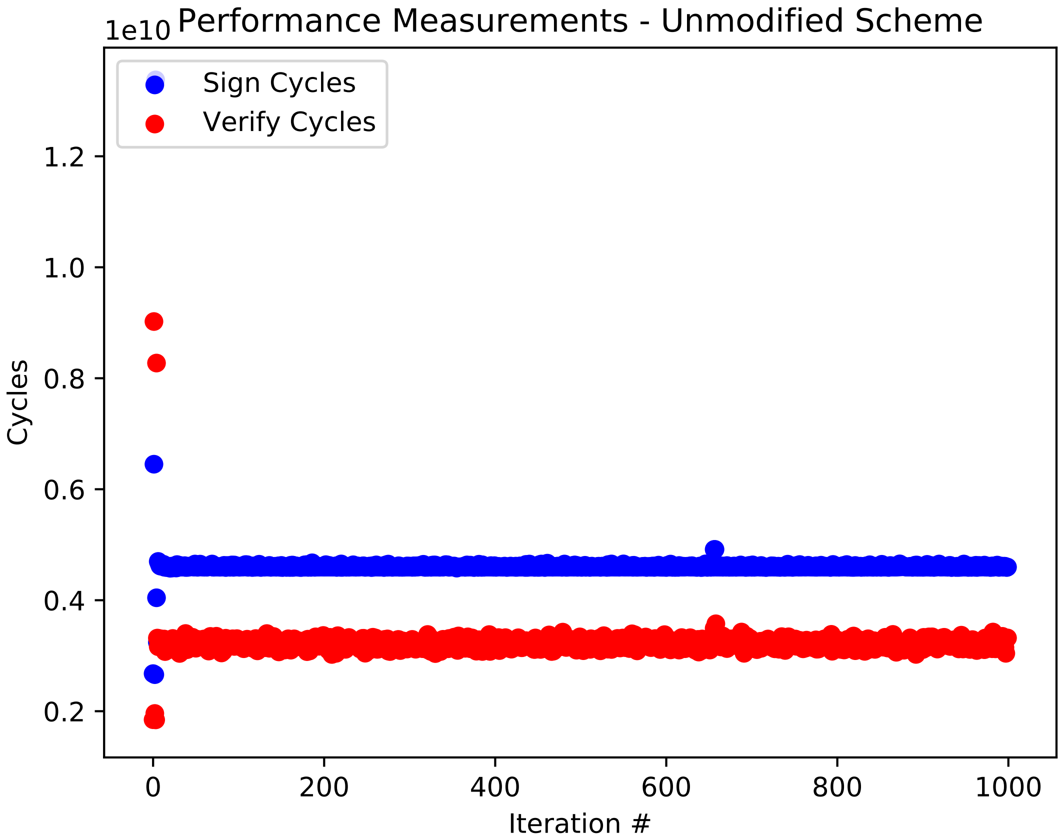


Figure 5.2: Cycle times for 1,000 unedited Yoo et al. signatures.

outliers taken at the beginning of data collection: this is quite likely attributed to time taken loading program data into RAM. The measurements that follow these outliers reflect the cycle time for the procedure once the code has been fetched and stored in memory.

The reader might note that the the performance metrics of this protocol all yield a considerably high standard deviation. This can be attributed to a few factors. The first and perhaps most influential factor is the size of the private key value m. The larger m is, the longer basic $\mathbb{F}_{p^2}$ arithmetic can take. This can be attributed to

On that same point, the reader will also note increased variance in the compressed implemetations. Part of this variance can be attributed to the fact that basis generation is a probabilistic process running in non-constant time - if favourable starting points are chosen, this process is completed significantly faster.

We return again to the comparison charts employed in Section 1.1.1 to compare the tomporal and spatial performance of these isogeny-based signatures to other post-quantum and classical alternatives. This time, we use the metrics resulting from our modified implementations as the point of comparison. These comparisons can be found in Figure 5.4 (comparing subroutine performances) and Figure 5.5 (compairing key and signature sizes). These metrics are all taken, yet again, at the 128-bit post quantum securty level (or classical security level, in the case of RSA and ECDSA).

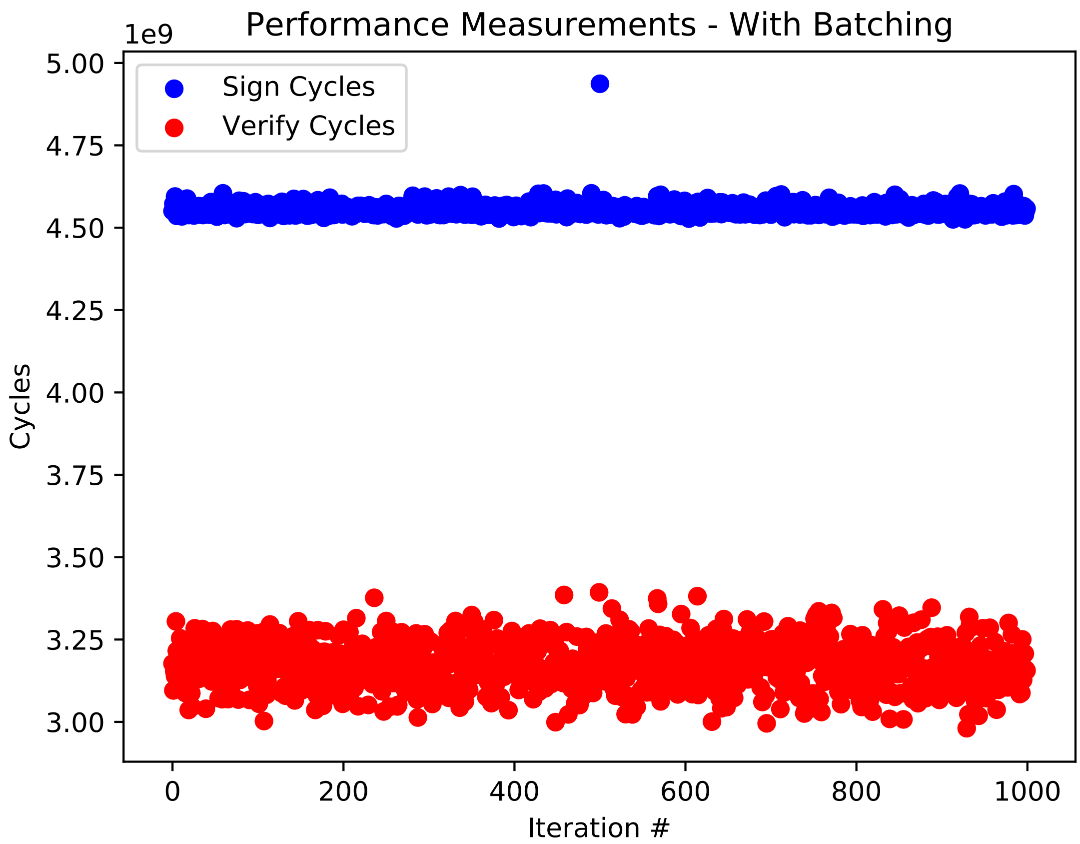


Figure 5.3: Cycle times for 1,000 signature with batched inversions.

	Key Gen	Sign	Verify
SIDH	a	b	c
SIDH Batched	a	b	c
SIDH Compressed	a	b	c
SIDH C+B	a	b	c
Sphincs	17,535,886.94	653,013,784	27,732,049
qTESLA	1,059,388	460,592	66,491
Picnic	13,272	9,560,749	6,701,701
RSA	a	1,113,600	32400
ECDSA	1,470,000	128,928	140,869

Figure 5.4: Performance in clock cycles for our improved isogeny-based signatures in comparison with other post-quantum and classical alternatives.

	Public Key	Private Key	Signature
SIDH	768	48	141,312
SIDH Compressed	768	b	141,312
Sphincs	32	64	8,080 - 16,976
Rainbow	152,097 - 192,241	100,209 - 114,308	64 - 104
qTESLA	4,128	2,112	3,104
Picnic	33	49	34,004 - 53,933
RSA	384	b	384
ECDSA	32	b	32

Figure 5.5: Key and signature sizes for our compressed isogeny-based signatures in comparison with other post-quantum and classical alternatives.

5.2 Discussion & Concluding Remarks

In this final section, we finish off the dissertation with some concluding remarks on the applicability of SIDH and isogeny-based cryptography, the importance of post-quantum cryptography, and the possible avenues for future work in this specific area.

If one assumes security of the original SIDH key exchange protocol, then the Yoo et al. signature scheme is provably secure and requires no additional underlying assumptions. Given that $\texttt{SIDH}_\texttt{C}$ keys are non-ephemeral, and so could pose a promising candidate in the context of TLS certificate signing. Certificate authority signatures, not requiring constant transfer over the wire (as they are offen packaged with Internet browsers or sperating systems) are less desperate for small signature sizes.

An unfortunate drawback to the "proof of knowledge" method for constructing signatures (how the Yoo et. al signatures are derived) is that the execution time of these schemes scale poorly with an increase in security.

5.2.1 Future Work

The next stage for this line of work is to finish applying inversion batching to the remaining cross-thread $\mathbb{F}_{p^2}$ inversions made throughout the signature scheme. There are n inversion calls in **Sign** and m in **Verify** that have yet to be , and from which further (comparable) performance improvements can be made.

There are several other areas of the code-base where relatively simple changes could be made to gain marginal performance improvements. Take for example functions which previously ran on private information but have now been adopted to run on public information, such as the key-exchange functions used in the verification process. These functions are designed to employ constant time algorithms for performing arithmetic (such as the Montgomery ladder) but could now be modified to support non-constant time implementations. Changes here could save time at several points of the verification process.

Following further efforts to improve performance, the code-base should be heavily tested in terms of correctness and application security, and after continued scrutiny (and improvements to code design,) a pull request can be made to the Microsoft SIDH repository [?] to merge both the Yoo et al. signature scheme and our improved implementations into their code-base.

Parallel to this line of work would be the continued efforts in developing alternative designs for isogeny-based signature schemes, and alternative isogeny-based schemes offering solutions to other information security goals.

There is another setting in which the inversion batching technique of Chapter 3 could be used to assist the performance of isogeny-based schemes.

Implementations of cryptographic primitives do have a lot to gain from intelligent design and implementation when it comes to performance metrics. Classical cryptographic algorithms have been . We believe that as the underlying foundations of post-quantum protocols gain traction and wide-spread confidence, more developers will begin to experiment with these protocols and the number of alternative implementation mechanisms and techniques will flourish, offering variety in terms of time-space tradeoffs and efficient, system-specific implementations.

With that said, however, some families of post-quantum protocols are still considerably young - their mathematical foundations and underlying primitives still have . As mathematical and developmental research both continue to provide more efficient and secure implementations of post-quantum protocols, we can continually approach a cryptographically secure and in the face of a rapidly developing landscape of threat to our privacy.

Appendices

Appendix A

SIDH$_C$ Functions

A.1 $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$ Functions

A.2 Isogeny and Point-wise Functions

A.2.1 j_inv

```
void j_inv(const f2elm_t A, const f2elm_t C, f2elm_t jinv) {
    f2elm_t t0, t1;
    fp2sqr751_mont(A, jinv);                    // jinv = A^2
    fp2sqr751_mont(C, t1);                      // t1 = C^2
    fp2add751(t1, t1, t0);                      // t0 = t1+t1
    fp2sub751(jinv, t0, t0);                    // t0 = jinv-t0
    fp2sub751(t0, t1, t0);                      // t0 = t0-t1
    fp2sub751(t0, t1, jinv);                    // jinv = t0-t1
    fp2sqr751_mont(t1, t1);                     // t1 = t1^2
    fp2mul751_mont(jinv, t1, jinv);             // jinv = jinv*t1
    fp2add751(t0, t0, t0);                      // t0 = t0+t0
    fp2add751(t0, t0, t0);                      // t0 = t0+t0
    fp2sqr751_mont(t0, t1);                     // t1 = t0^2
    fp2mul751_mont(t0, t1, t0);                 // t0 = t0*t1
    fp2add751(t0, t0, t0);                      // t0 = t0+t0
    fp2add751(t0, t0, t0);                      // t0 = t0+t0
    fp2inv751_mont(jinv);                       // jinv = 1/jinv
    fp2mul751_mont(jinv, t0, jinv);             // jinv = t0*jinv
}
```

A.2.2 j_inv_batch

```
void j_inv_batch(f2elm_t A, f2elm_t C, f2elm_t jinv, invBatch* batch) {
    f2elm_t t0, t1;
    int tempCnt;
    fp2sqr751_mont(A, jinv);                    // jinv = A^2
    fp2sqr751_mont(C, t1);                      // t1 = C^2
    fp2add751(t1, t1, t0);                      // t0 = t1+t1
    fp2sub751(jinv, t0, t0);                    // t0 = jinv-t0
    fp2sub751(t0, t1, t0);                      // t0 = t0-t1
    fp2sub751(t0, t1, jinv);                    // jinv = t0-t1
    fp2sqr751_mont(t1, t1);                     // t1 = t1^2
    fp2mul751_mont(jinv, t1, jinv);             // jinv = jinv*t1
    fp2add751(t0, t0, t0);                      // t0 = t0+t0
    fp2add751(t0, t0, t0);                      // t0 = t0+t0
```

```
    fp2sqr751_mont(t0, t1);                         // t1 = t0^2
    fp2mul751_mont(t0, t1, t0);                     // t0 = t0*t1
    fp2add751(t0, t0, t0);                          // t0 = t0+t0
    fp2add751(t0, t0, t0);                          // t0 = t0+t0
    pthread_mutex_lock(&batch->arrayLock);
    fp2copy751(jinv, batch->invArray[batch->cntr]);
    tempCnt = batch->cntr;
    batch->cntr++;
    pthread_mutex_unlock(&batch->arrayLock);

    int i;
    if (tempCnt+1 == batch->batchSize) {
        partial_batched_inv(batch->invArray, batch->invDest, batch->batchSize);
        for (i = 0; i < batch->batchSize - 1; i++) {
            sem_post(&batch->sign_sem);
        }
    } else {
        sem_wait(&batch->sign_sem);
    }
    fp2copy751(batch->invDest[tempCnt], jinv);
    batch->cntr = 0;
    fp2mul751_mont(jinv, t0, jinv);                 // jinv = t0*jinv
}
```

A.2.3 inv_4_way

```
void inv_4_way(f2elm_t z1, f2elm_t z2, f2elm_t z3, f2elm_t z4) {
    f2elm_t t0, t1, t2;
    int tempCnt;

    fp2mul751_mont(z1, z2, t0);                   // t0 = z1*z2
    fp2mul751_mont(z3, z4, t1);                   // t1 = z3*z4
    fp2mul751_mont(t0, t1, t2);                   // t2 = z1*z2*z3*z4
    fp2inv751_mont(t2);                           // t2 = 1/(z1*z2*z3*z4)
    fp2mul751_mont(t0, t2, t0);                   // t0 = 1/(z3*z4)
    fp2mul751_mont(t1, t2, t1);                   // t1 = 1/(z1*z2)
    fp2mul751_mont(t0, z3, t2);                   // t2 = 1/z4
    fp2mul751_mont(t0, z4, z3);                   // z3 = 1/z3
    fp2copy751(t2, z4);                           // z4 = 1/z4
    fp2mul751_mont(z1, t1, t2);                   // t2 = 1/z2
    fp2mul751_mont(z2, t1, z1);                   // z1 = 1/z1
    fp2copy751(t2, z2);                           // z2 = 1/z2
}
```

A.2.4 inv_4_way_batch

```
void inv_4_way_batch(f2elm_t z1, f2elm_t z2, f2elm_t z3, f2elm_t z4, invBatch* batch) {
    f2elm_t t0, t1, t2;
    int tempCnt;

    fp2mul751_mont(z1, z2, t0);                   // t0 = z1*z2
    fp2mul751_mont(z3, z4, t1);                   // t1 = z3*z4
    fp2mul751_mont(t0, t1, t2);                   // t2 = z1*z2*z3*z4
    pthread_mutex_lock(&batch->arrayLock);
    fp2copy751(t2, batch->invArray[batch->cntr]);
    tempCnt = batch->cntr;
    batch->cntr++;
    pthread_mutex_unlock(&batch->arrayLock);
```

```
    int i;
    if (tempCnt+1 == batch->batchSize) {
        partial_batched_inv(batch->invArray, batch->invDest, batch->batchSize);
        for (i = 0; i < batch->batchSize; i++) {
            sem_post(&batch->sign_sem);
        }
    } else {
        sem_wait(&batch->sign_sem);
    }
    fp2copy751(batch->invDest[tempCnt], t2);
    batch->cntr = 0;
    fp2mul751_mont(t0, t2, t0);                     // t0 = 1/(z3*z4)
    fp2mul751_mont(t1, t2, t1);                     // t1 = 1/(z1*z2)
    fp2mul751_mont(t0, z3, t2);                     // t2 = 1/z4
    fp2mul751_mont(t0, z4, z3);                     // z3 = 1/z3
    fp2copy751(t2, z4);                             // z4 = 1/z4
    fp2mul751_mont(z1, t1, t2);                     // t2 = 1/z2
    fp2mul751_mont(z2, t1, z1);                     // z1 = 1/z1
    fp2copy751(t2, z2);                             // z2 = 1/z2
}
```

A.3 Key Exchange Functions

```
CRYPTO_STATUS KeyGeneration_A(unsigned char* pPrivateKeyA,
                              unsigned char* pPublicKeyA,
                              PCurveIsogenyStruct CurveIsogeny,
                              bool GenerateRandom, batch_struct* batch) {
    unsigned int owords = NBITS_TO_NWORDS(CurveIsogeny->owordbits);
    unsigned int pwords = NBITS_TO_NWORDS(CurveIsogeny->pwordbits);
    point_basefield_t P;
    point_proj_t R, phiP = {0}, phiQ = {0}, phiD = {0}, pts[MAX_INT_POINTS_ALICE];
    publickey_t* PublicKeyA = (publickey_t*)pPublicKeyA;
    unsigned int i, row, m, index = 0, pts_index[MAX_INT_POINTS_ALICE], npts = 0;
    f2elm_t coeff[5], A = {0}, C = {0}, Aout, Cout;
    CRYPTO_STATUS Status = CRYPTO_ERROR_UNKNOWN;

    if (pPrivateKeyA == NULL || pPublicKeyA == NULL || is_CurveIsogenyStruct_null(Curve
        return CRYPTO_ERROR_INVALID_PARAMETER;
    }

    if (GenerateRandom) {
        Status = random_mod_order((digit_t*)pPrivateKeyA, ALICE, CurveIsogeny);
        if (Status != CRYPTO_SUCCESS) {
            clear_words((void*)pPrivateKeyA, owords);
            return Status;
        }
    }

    to_mont((digit_t*)CurveIsogeny->PA, (digit_t*)P);
    to_mont(((digit_t*)CurveIsogeny->PA)+NWORDS_FIELD, ((digit_t*)P)+NWORDS_FIELD);

    Status = secret_pt(P, (digit_t*)pPrivateKeyA, ALICE, R, CurveIsogeny);
    if (Status != CRYPTO_SUCCESS) {
        clear_words((void*)pPrivateKeyA, owords);
        return Status;
    }

    copy_words((digit_t*)CurveIsogeny->PB, (digit_t*)phiP, pwords);
```

```
    fpcopy751((digit_t*)CurveIsogeny->Montgomery_one, (digit_t*)phiP->Z);
    to_mont((digit_t*)phiP, (digit_t*)phiP);
    copy_words((digit_t*)phiP, (digit_t*)phiQ, pwords);
    fpneg751(phiQ->X[0]);
    fpcopy751((digit_t*)CurveIsogeny->Montgomery_one, (digit_t*)phiQ->Z);
    distort_and_diff(phiP->X[0], phiD, CurveIsogeny);

    fpcopy751(CurveIsogeny->A, A[0]);
    fpcopy751(CurveIsogeny->C, C[0]);
    to_mont(A[0], A[0]);
    to_mont(C[0], C[0]);

    first_4_isog(phiP, A, Aout, Cout, CurveIsogeny);
    first_4_isog(phiQ, A, Aout, Cout, CurveIsogeny);
    first_4_isog(phiD, A, Aout, Cout, CurveIsogeny);
    first_4_isog(R, A, A, C, CurveIsogeny);

    index = 0;
    for (row = 1; row < MAX_Alice; row++) {
        while (index < MAX_Alice-row) {
            fp2copy751(R->X, pts[npts]->X);
            fp2copy751(R->Z, pts[npts]->Z);
            pts_index[npts] = index;
            npts += 1;
            m = splits_Alice[MAX_Alice-index-row];
            xDBLe(R, R, A, C, (int)(2*m));
            index += m;
        }
        get_4_isog(R, A, C, coeff);

        for (i = 0; i < npts; i++) {
            eval_4_isog(pts[i], coeff);
        }
        eval_4_isog(phiP, coeff);
        eval_4_isog(phiQ, coeff);
        eval_4_isog(phiD, coeff);

        fp2copy751(pts[npts-1]->X, R->X);
        fp2copy751(pts[npts-1]->Z, R->Z);
        index = pts_index[npts-1];
        npts -= 1;
    }

    get_4_isog(R, A, C, coeff);
    eval_4_isog(phiP, coeff);
    eval_4_isog(phiQ, coeff);
    eval_4_isog(phiD, coeff);

    if(batch != NULL) {
        inv_4_way_batch(C, phiP->Z, phiQ->Z, phiD->Z, batch);
    } else {
        inv_4_way(C, phiP->Z, phiQ->Z, phiD->Z);
    }

    fp2mul751_mont(A, C, A);
    fp2mul751_mont(phiP->X, phiP->Z, phiP->X);
    fp2mul751_mont(phiQ->X, phiQ->Z, phiQ->X);
    fp2mul751_mont(phiD->X, phiD->Z, phiD->X);
```

```

    from_fp2mont(A, ((f2elm_t*)PublicKeyA)[0]);
    from_fp2mont(phiP->X, ((f2elm_t*)PublicKeyA)[1]);
    from_fp2mont(phiQ->X, ((f2elm_t*)PublicKeyA)[2]);
    from_fp2mont(phiD->X, ((f2elm_t*)PublicKeyA)[3]);

    clear_words((void*)R, 2*2*pwords);
    clear_words((void*)phiP, 2*2*pwords);
    clear_words((void*)phiQ, 2*2*pwords);
    clear_words((void*)phiD, 2*2*pwords);
    clear_words((void*)pts, MAX_INT_POINTS_ALICE*2*2*pwords);
    clear_words((void*)A, 2*pwords);
    clear_words((void*)C, 2*pwords);
    clear_words((void*)coeff, 5*2*pwords);

    return Status;
}
```

A.4 Signature Layer Functions

References

[AJK+16] Reza Azerderakhsh, David Jao, Kassem Kalach, Brian Koziel, and Christopher Leonardi. Key compression for isogeny-based cryptosystems. 2016.

[ARU14] Andris Ambainis, Ansis Rosmanis, and Dominique Unruh. Quantum attacks on classical proof systems (the hardness of quantum rewinding). pages 474–483, 2014.

[CJL+17] Craig Costello, David Jao, Patrick Longa, Michael Naehrig, Joost Renes, and David Urbanik. Efficient compression of sidh public keys. 2017.

[CLN16] Craig Costello, Patrick Longa, and Michael Naehrig. Efficient algorithms for supersingular isogeny diffie-hellman. IACR-CRYPTO-2016, 2016.

[Cos] Craig Costello. Pairings for beginners.

[ea18] Daniel J. Bernstein et al. libpqcrypto.org, 2018. Available online at `https://libpqcrypto.org`.

[Fia89] Amos Fiat. Batch rsa. 1989.

[FJP12] Luca De Feo, David Jao, and Jerome Plut. Towards quantum-resistant cryptosystems from supersingular elliptic curvce isogenies. 2012.

[JS14] David Jao and Vladimir Soukharev. Isogeny-based quantum resistant undeniable signatures. 2014.

[Kat10] Jonathan Katz. *Digital Signatures*. Springer, 2010.

[Mon85] Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 1985.

[PS96] David Pointcheval and Jacques Stern. Security proofs for signature schemes. 1996.

[SB01] Hovav Shacham and Dan Boneh. Improving ssl handshake performance via batching. 2001.

[SC16] M. Seshadri Srinath and V. Chandrasekaren. Isogeny-based quantum-resistant undeniable blind signature scheme. 2016.

[Sil] Joseph H. Silverman. The arithmetic of elliptic curves.

[SM18] Douglas Stebila and Michele Mosca. Open quantum safe, 2018. Available online at `https://openquantumsafe.org/`.

[STW12] Xi Sun, Haibo Tian, and Yumin Wang. Toward quantum-resistant strong designated verifier signature from isogenies. 2012.

[Unr14] Dominique Unruh. Non-interactive zero-knowledge proofs in the quantum random oracle model. 2014.

[YAJ+12] Youngho Yoo, Reza Azarderakhsh, Amir Jalali, David Jao, and Vladimir Soukharev. A post-quantum digital signature scheme based on supersingular isogenies. 2012.

[YAJ+17] Youngho Yoo, Reza Azarderakhsh, Amir Jalali, David Jao, and Vladimir Soukharev. Isogeny based signatures from sidh library, 2017. Available online at `https://github.com/yhyoo93/isogenysignature`.