YAŞAR UNIVERSITY

FACULTY OF ENGINEERING

DEPARTMENT OF COMPUTER ENGINEERING

COMP4920 Senior Design Project II, Spring 2019

Advisor: Asst. Prof. Dr. Hüseyin Hışıl

K4SIDH: A 4 way SIMD Implementation of SIDH Compatible Isogeny Evaluations on Kummer Surfaces

Final Report(Bachelor of Science Thesis)

14.05.2019

Project by

Mert Yassı, 14070001001

PLAGIARISM STATEMENT

This report was written by the group members and in our own words, except for quotations from published and unpublished sources which are clearly indicated and acknowledged as such. We are conscious that the incorporation of material from other works or a paraphrase of such material without acknowledgement will be treated as plagiarism according to the University Regulations. The source of any picture, graph, map or other illustration is also indicated, as is the source, published or unpublished, of any material not resulting from our own experimentation, observation or specimen collecting.

Group Members:

Name Surname	Student Number	Signature	Date

Project Supervisors:

Name Surname	Department	Signature	Date

ACKNOWLEDGEMENTS

I would like to denote my most profound and sincerest appreciation to my advisor Asst. Prof. Dr. Hüseyin HIŞIL. He helped me greatly with his deep expertise in the area of cryptography. With his intensive assistance, I was able to surpass every obstacle that I faced during the preparation phase of this project.

I would also like to thank my family for their continuous support and empathetic behaviour. They were very supportive and backed me up in every aspect from the beginning to the end of the project.

KEYWORDS

Supersingular Isogeny Diffie-Hellman, Isogeny evaluation, Montgomery curve, Kummer surface, Post-quantum cryptography, Single Instruction Multiple Data, 4 way parallelization, Magma Computational Algebra System.

ABSTRACT

New technological developments and possible existence of practical quantum computers in the future led to the requirement of building much stronger, quantum-secure, security systems. Based on this, in 2011, Supersingular Isogeny Diffie-Hellman (SIDH) key exchange was proposed by De Feo, Jao, and Plut. Current SIDH implementations employ Montgomery curves in their entirety. However, changing the underlying algebraic structure from Montgomery curves to Kummer surfaces can greatly simplify the work done. This project presents an implementation of the isogeny evaluation performed in the SIDH key exchange with utilizing Kummer surface arithmetic. With the current findings, the isogeny evaluation sequence is only suitable for Alice's public key generation and shared secret computation parts. We can go back and forward between supersingular Montgomery curves and corresponding supersingular Kummer surfaces with the help of various maps from the literature and operate on the Kummer surfaces. This allows 4 way parallelization of operations with the usage of 256-bit AVX2 SIMD instruction set. The utilization of SIMD instructions with Kummer surface arithmetic for the evaluation of (2,2) isogenies may result in significant decrease of operation counts in a fully optimized implementation.

ÖZ

Yeni teknolojik gelişmeler ve kuantum bilgisayarların gelecekte yaygın olarak kullanılabilecek olması ihtimali, çok daha güçlü, kuantum ataklara dayanıklı, güvenlik sistemlerini inşa etmenin gerekliliğine yol açmıştır. Buna dayanarak, 2011'de, De Feo, Jao ve Plut Süpertekil İzojeni Diffie-Hellman (SIDH) anahtar değişimi şemasını öne sürmüştür. Güncel SIDH uygulamaları baştan sona Montgomery eğrisini kullanmaktadırlar. Ancak, temelde yatan cebirsel yapı olarak Montgomery eğrisi yerine Kummer yüzeyini kullanmak toplamda yapılan işi büyük oranda azaltabilir. Bu proje SIDH anahtar değişiminde gerçekleştirilen izojeni değerlendirmelerinin Kummer yüzey aritmetiği kullanılarak uygulanışını ortaya koymaktadır. Şu an var olan bulgular ile Kummer yüzey aritmetiği sadece Alice'nin açık anahtar üretimi ve paylaşılan giz hesaplamasında kullanılabilir. Literatürde yer alan çeşitli haritalar yardımıyla, süpertekil Montgomery eğrileri ve bu eğrilere karşılık gelen süpertekil Kummer yüzeyleri arasında geçiş yapabilir ve Kummer yüzeyleri üzerinde çalışabiliriz. Bu durum, 256-bit AVX2 SIMD komut setinin kullanımıyla birlikte, işlemlerin 4 kanalda paralelleştirilmesini sağlamaktadır. (2,2) izojenilerin değerlendirilmesinde SIMD komutlarının Kummer yüzey aritmetiği ile birlikte kullanımı, tamamen optimize edilmiş bir uygulamada toplam işlem sayısının kayda değer derecede düşüşünü sağlayabilir.

Contents

List of Figures

List of Tables

List of Codes

LIST OF ACRONYMS/ABBREVIATIONS

AVX	Advanced Vector Extensions (128-bit)
AVX2	Advanced Vector Extensions (256-bit)
CPU	Central Processing Unit
DH	Diffie-Hellman
GNU	Gnu's Not Unix
GCC	GNU C Compiler
GNOME	GNU Network Object Model Environment
LTS	Long Term Support
SIDH	Supersingular Isogeny Diffie-Hellman
SIMD	Single Instruction Multiple Data

Chapter 1

INTRODUCTION

1.1 Description of the Problem

Continuous progress in technology has paved the way for possessing more and more processing power throughout the years. This situation poses danger to the existing cryptographic systems. Due to the developments in the area, preserving the security of these systems becomes gradually harder.

The difficulty of decrypting a cryptographic system lies on the complexity of the mathematical problem in the background. For instance, Diffie-Hellman key exchange [7] gains its security from the discrete logarithm problem. Classical computers can solve a problem in polynomial time. However, even with recent supercomputers, the discrete logarithm problem can only be solved in exponential time.

Researchs and developments about the creation of quantum computers have been emerged in recent years. It is assumed that a quantum computer will solve any exponential time problem with quantum attacks in a short period of time. The security of Diffie-Hellman key exchange may be affected from the rise of quantum computers in the long term. To be able to provide a secure communication, considering the existence of quantum computers, De Feo, Jao and Plut [11] proposed a scheme called Supersingular Isogeny Diffie-Hellman key exchange in 2011. The

scheme is considered as resistant to quantum attacks. The details about SIDH are stated in Appendix A: Requirement Specifications Document.

1.2 Project Goal

In the past, the implementations of SIDH key exchange were done using supersingular Montgomery curves from beginnning to end. In 2018, Costello [4] claimed that using Kummer surfaces in Alice's parts of the key exchange instead of Montgomery curves is possible with the use of back-and-forth maps he stated. The goal of the project is, using the maps given, to show that consecutive isogeny evaluations, placed in Alice's public key generation and shared secret computation phases, can be evaluated on Kummer surfaces. We benefit from the Kummer isogeny evaluation sequence written by Costello [4], in Magma [3] language. We utilize 4 way AVX2 instructions in Kummer surface arithmetic. Also, we compose all the maps given as Magma code [4] with other necessary maps in the literature to be able to transfer points and constants between supersingular Montgomery curves and supersingular Kummer surfaces.

1.3 Project Output

The outputs of the project are:

- (2,2) Isogeny Evaluations on Kummer Surfaces C Implementation
- Back-and-Forth Maps Magma Implementation
- Requirements Specification Document
- Design Specification Document
- Product Manual

1.4 Project Activities and Schedule

Throughout the creation of this project, several documents (and different versions of them) were created. The table below specifies any activity done for completing the project with the time needed for each activity. Note that most of the activities are done simultaneously with each other.

Activity	Time
Problem Definition	2 weeks
Project Form preparation	2 weeks
Requirements Specification Document v1.0 creation	4 weeks
Requirements Specification Document v2.0 creation	4 weeks
Design Specification Document v1.0 creation	4 weeks
Requirements Specification Document v2.1 creation	1 week
Project Poster creation	2 weeks
Design Specification Document v2.0 creation	4 weeks
Requirements Specification Document v3.0 creation	2 week
Design Specification Document v3.0 creation	5 weeks
Final Report writing	4 weeks
Project implementation	14 weeks
Project Summary Form & 2nd Poster creation	1 week
Project Presentation/Demo preparation	1 week

Table 1.1: Activity-Time Table

Chapter 2

DESIGN

2.1 High Level Design

For make the things clear, we first briefly explain how the evaluations of isogenies are done in SIDH key exchange. Note that all the information given in this report is based on the SIDH implementation of Microsoft Research [6] [17]. The parameters and prime of SIDH can be chosen differently from what we stated here.

SIDH key exchange consists of four stages which are

- Alice's public key generation
- Bob's public key generation
- Alice's shared secret computation
- Bob's shared secret computation.

In public key generation and shared secret computation phases of Alice, the bottleneck operations are the isogeny evaluations. Alice evaluates 4-isogenies on supersingular Montgomery curves. After evaluating an isogeny on a Montgomery curve, with the predetermined times of doubling of an isogeny point, Alice leaps towards to a new Montgomery

curve and starts to perform isogeny evaluation on that curve. This process continues until enough number of isogenies have been evaluated. The difference in the process for Bob is that, he evaluates 3-isogenies consecutively and he uses tripling of an isogeny point in order to get the new isogenous Montgomery curve.

K4SIDH forms the isogeny evaluations part of Alice's side of SIDH key exchange. The application computes a chain of Richelot isogenies [18] on Kummer surfaces, which we will call Kummer isogenies for the rest of this paper. The sequential execution of isogeny evaluations constitutes the most time consuming part of the Alice's public key generation and shared secret computation steps. In K4SIDH, Alice's 4-isogenies are replaced with (2,2) isogenies and Kummer surface arithmetic is used. Usage of Kummer surfaces leads to the introduction of 4 way AVX2 SIMD instruction set in arithmetic functions such as field multiplication, field squaring, modular reduction and Hadamard operation. Details of K4SIDH high level design is provided in Appendix B: Design Specification Document, Section B.2.1.

2.2 Detailed Design

K4SIDH has two outputs as the product: (2,2) Kummer isogeny evaluations implementation and back-and-forth maps between supersingular Montgomery curves and supersingular Kummer surfaces implementation.

The main function of the isogeny evaluations implementation consists of the comparison of the result that comes from the isogeny evaluations and the result of the Magma implementation [5]. Additionally, total cycle count of the implementation is measured after the validity check. Arithmetic functions and routines that the implementation has, are placed in **kummer_isogeny.c**. The maps between supersingular Montgomery curves and supersingular Kummer surfaces take part in **maps.mag**. These maps can be defined as the composition of different maps between different algebraic structures. We can reach the target supersingular Kummer surface only by visiting various algebraic structures.

The back-and-forth maps are given to prove that arriving to Kummer surfaces and coming back from them is possible. These two implementations complete each other. Due to the limited time, the maps were implemented in Magma language. However, with implementing

the maps in C language, placing the (2,2) Kummer isogeny evaluation sequence to the right place of a well-made SIDH implementation, like [17], and calling the maps before and after the evaluation sequence, one can actually perform the evaluations on Kummer surfaces, in a fully working key exchange implementation. What we try to achieve is to show that this whole process is feasible.

The details about the design of K4SIDH is provided in Appendix B: Design Specification Document, Section B.3.

2.3 Realistic Restrictions and Conditions in the Design

The only restriction in the design of K4SIDH is to have a working environment which has SIMD AVX2 instruction set support. Fast Kummer surface arithmetic functions are designed for operating with special AVX2 instructions. They are optimized for taking the most advantage of the structure of 4 way parallel instructions. The application cannot work on a system which is lack of AVX2 support.

AVX is the predecessor of AVX2 and it was proposed by Intel in March 2008 to be used for x86 architectures [13]. Intel Sandy Bridge codenamed processors supported AVX instruction set primarily. AVX allows performing 128-bit arithmetic. AVX2 (i.e. Haswell New Instructions) can be considered as the expansion pack of the AVX instructions and it supports arithmetic operations on 256-bit range. The set was first used in Intel Haswell processors in 2013 [16]. K4SIDH must be run on an Intel CPU whose architecture must be newer than Haswell series.

Chapter 3

IMPLEMENTATION, TESTS and TEST DISCUSSIONS

3.1 Implementation of the Product

K4SIDH is an application that possesses a (2,2) isogeny evaluation sequence, directly compatible to the Alice's public key generation and shared secret computation parts of the SIDH key exchange. Therefore, it works on a prime field of $p = 2^{372}3^{239} - 1$, which is a 751-bit prime. To be able to deal with the field arithmetic of p, we split a field element to 32 words. Each word is 24-bit, except the last word. For the last word, we have $751 - 24 \times 31 = 7$ bits. The reason we represent a field element with 32 words is that, because AVX2 instruction set does not have 64-bit multipliers. Unfortunately, this is a very big drawback for us. The multiplication, squaring and modular reduction operations become very expensive operations with these settings. Better setups for how a field element should be split into words can be found, as a future work. Also, if new AVX instructions support 64-bit multipliers in the future, the performance of this application will greatly increase.

In the implementation of K4SIDH, we strictly avoid the usage of branching or any

conditional structures. K4SIDH is a constant-time running application. Therefore, it is resistant to timing analysis attacks.

The constants and initial points are hardcoded in 4 way format, 32 words in K4SIDH. Also, for the subtraction operation in Hadamard, we precompute double of the prime p and use it to keep every word in positive. In Hadamard and modular reduction, we cleared the bits of every word of a result, kept every word in 24-bits, in order to avoid possible bit overflows. In addition, for printing a 4 way field element, we wrote various print functions and various conversion functions between the 4 way 256-bit integer type __m256i and four 64-bit unsigned long long integer. For more implementation details, see Appendix C: Product Manual, Section C.2.

3.2 Tests and Results of Tests

In the main function of K4SIDH, we perform two different tests which are for checking validation of the application and measuring the speed of the application. The validation checking process is done with the help of the Magma script version [5] of the chain of isogeny evaluations on Kummer surfaces.Both results from K4SIDH and the script are compared and the comparison results are printed to the screen. Note that, we made slight changes in the script in order to use it for the comparison process.

For the speed test, we use the function *cpucycles* that simply counts the total number of clock cycles for an operation with the use of system counter. We measured the speed of K4SIDH with this function. The speed test was done on a computer with an Intel® Core™ i7-7500U 2.70 GHz x 4 CPU. Also, the parameter -O3 (optimize most) is used when compiling K4SIDH. The total cycle count for the application is approximately 2.2 billion cycles. The total running time of the application is approximately 0.76 seconds.

The numbers given above are quite large mainly because of the reason given in Section 3.1. Also, the technique used for implementing the modular reduction (Barrett reduction [1]) is another reason for getting the numbers above. Changing the used technique for reduction and making further optimizations can decrease the cycle count excessively, as discussed in Section 4.4. Further information about the testing design can be found in [20] and Appendix C.

Chapter 4

CONCLUSION

4.1 Summary

At the beginning of this project, we decided to develop a 4 way SIMD SIDH application that uses Kummer surfaces in its every phase, without gain insight into the SIDH key exchange. As our knowledge about SIDH deepened, we observed that Kummer surface arithmetic is only applicable in Alice's side of the key exchange. This is because of (2,2) Kummer isogenies just correspond to Alice's 4-isogenies, not Bob's 3-isogenies. Knowing that, we started to implement necessary maps between supersingular Montgomery curves and supersingular Kummer surfaces with Magma Computer Algebra Software System [3]. After the implementation of back-and-forth maps was done, we started to implement the isogeny evaluation sequence on Kummer surfaces, in C language. At the implementation stage, we came across to the problem of having only 32-bit multipliers in AVX2 SIMD instruction set. For this reason, we had to represent a field element in 32 24-bit words. This caused getting expensive field multiplication and squaring operations. Also, due to the limited time, we used Barrett reduction method [1] for modular reduction after multiplication, squaring and Hadamard operations. At the end of the implementation phase, we had a 4 way SIMD supported, but slow C implementation of chain of (2,2) isogeny evaluations on Kummer surfaces. Making further optimizations

in the evaluation implementation, converting the maps implementation from Magma to C language, and combining maps and the isogeny sequence in a functioning SIDH key exchange implementation are left as future works.

4.2 Cost Analysis

This project is a one man project. Also, no software or hardware was purchased for this specific project. For this reason, performing a cost analysis for this project is meaningless. We only give a table that indicates the amount of time spent for the project.

The table below shows the total manpower spent month by month in terms of work hours. Note that the values given for total work hours are approximate values.

Month	Total Work Hours
September 2018	40
October 2018	50
November 2018	70
December 2018	100
January 2019	100
February 2019	100
March 2019	120
April 2019	130
May 2019	70

Table 4.1: Month-Total Work Hours Table

4.3 Benefits of the Project

The main aim of the project is to provide fast Kummer surface arithmetic in isogeny evaluations instead of using Montgomery arithmetic. The nature of Kummer surface functions is suitable for the usage of SIMD instructions. The utilization of SIMD instructions and making correct optimizations in the implementation may come with the decrease of total cycle count. Using 4 way instructions instead of traditional arithmetic instructions can speed up the work done. This is beneficial because cryptographic protocols are used around the world every second (e.g. TLS uses DH) and accelerations in this area can affect greatly to the quality of communications.

4.4 Future Work

Although the utilization of 4 way AVX2 SIMD instructions in Kummer surface arithmetic functions, K4SIDH is a really slow implementation. This is because the AVX2 instruction set only allows 32-bit multipliers. Thus, we had to store a field element in 32 24-bit words. This makes the multiplication and squaring functions very expensive. Also, due to the lack of time we had, we used Barrett reduction for the modular reduction part of prime field arithmetic. Usage of Montgomery reduction will decrease the cycle count greatly. Making further optimizations, such as using Karatsuba or Toom-Cook algorithm instead of Comba for multiplication and squaring, may speed up the implementation.

In addition, instead of traversing through all the nodes of the isogeny tree, an optimal tree strategy can be applied to the isogeny evaluation sequence, as in [6].

As mentioned briefly in the detailed design section, Kummer isogeny evaluations implementation can be composed with back-and-forth maps and can be placed in a fully working SIDH key exchange implementation. With correct optimization choices and time, the power of SIMD and Kummer surfaces can be exposed.

Appendix A

REQUIREMENTS SPECIFICATION DOCUMENT

14.05.2019

Revision 3.0

Revision History

Revision	Date	Explanation
1.0	26.11.2018	Initial requirements
2.0	19.12.2018	The initial requirements were modified and new requirements were added. Requirements Model was provided.
2.1	16.01.2019	Section explanations were added and small changes were made in the diagrams as suggested by the advisor.
3.0	14.05.2019	Aim of the project is shifted from producing a 4 way SIDH implementation on Kummer surfaces to producing a 4 way Kummer isogeny evaluation sequence compatible to SIDH key exchange. Also, some requirements were modified and new requirements were added.

A.1 Introduction

This section explains Supersingular Isogeny Diffie-Hellman key exchange and its relation with Kummer Surfaces briefly.

A.1.1 What is SIDH

Whitfield Diffie and Martin Hellman published a scheme in 1976 [7] in order to achieve a safe communication between two parties which named as Diffie-Hellman (DH) key exchange protocol. This protocol removed the necessity of having a secure communication channel to ensure a reliable and safe flow of information. The protocol has been used in the past and is still in use in different areas. However, with the emergence of the newly developed algorithms and researches about the quantum cryptography, the throne of Diffie-Hellman is in danger now. Even though we do not have any existing practical quantum computer at the moment, there are various researches on how to defend ourselves from the possible future quantum attacks. These researches are collected under a title called Post Quantum Cryptography.

SIDH can be identified as a quantum-resistant key exchange algorithm which is resistant to cryptanalytic attacks that made by a quantum computer. It was proposed by De Feo, Jao, and Plut [11] in 2011. The algorithm gets its resistance from the isogenies between supersingular elliptic curves of smooth order and it uses the properties of these curves and supersingular isogeny graphs in order to create a key exchange protocol similar to DH. It is apparent that the traditional DH and elliptic curve DH which uses elliptic curves for generating key pairs will be replaced by the SIDH in the future.

A.1.2 Kummer Surface for SIDH

In 2018, Costello [4] proposed a method that allows simpler computations for the work done in SIDH. Instead of performing the arithmetic on the supersingular Montgomery curve, it computes a set of supersingular Kummer surface parameters from the corresponding Montgomery curve and does the computations of key pairs on that Kummer surface. This mapping reveals the path for a 4 way parallel SIMD implementation of SIDH thanks to the structure of Kummer surfaces.

In this project, the inner workings of SIDH is studied and at the software side, a 4 way parallel SIMD implementation of (2,2) isogeny evaluations on Kummer surfaces and a Magma implementation of necessary maps between supersingular Montgomery curves and supersingular Kummer surfaces are delivered.

A.2 Requirements

Requirements for the project with various diagrams and figures are given in detail in this section.

A.2.1 To perform a fully working Isogeny Evaluation Sequence

To have a sequence of isogeny evaluations compatible to SIDH key exchange

Chain of isogeny evaluations must be done on Kummer surfaces and it also must be suitable with SIDH key exchange. The evaluation sequences form the bottleneck of SIDH's key generation and shared secret computation steps. These steps must have an order as in the traditional DH key exchange scheme. To be able to see the similarities and differences between DH and SIDH, use case and sequence diagrams of both key exchange schemes are provided. The diagrams for DH key exchange are as follows.

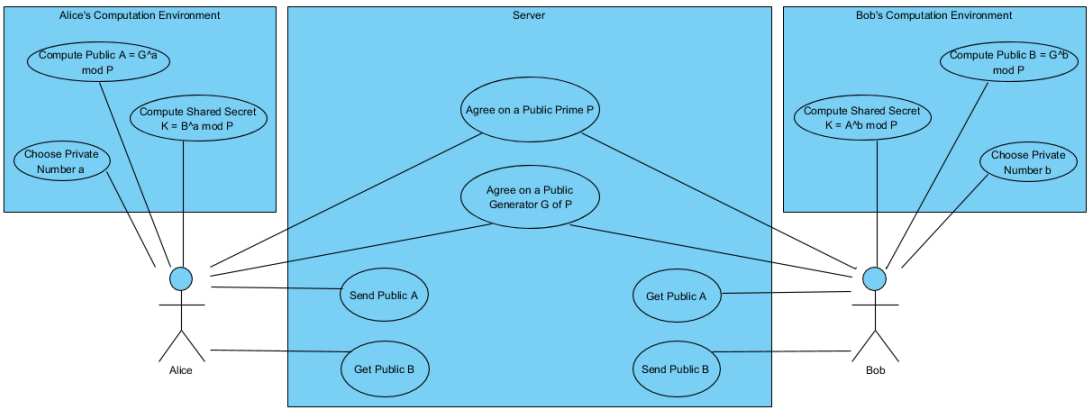

Figure A.1: DH Key Exchange Use Case Diagram

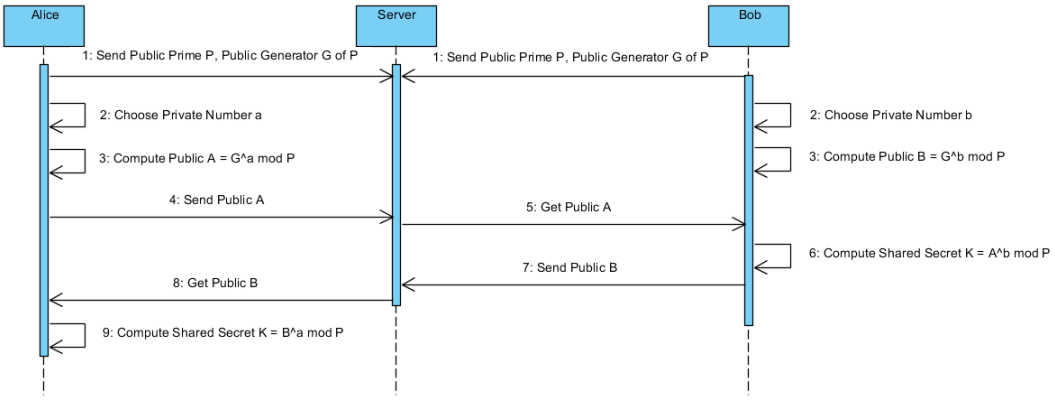

Figure A.2: DH Key Exchange Sequence Diagram

The structure of SIDH is the same with DH. However, it requires more steps for key generations and shared secret computations. The use-case diagram for SIDH key exchange is given below.

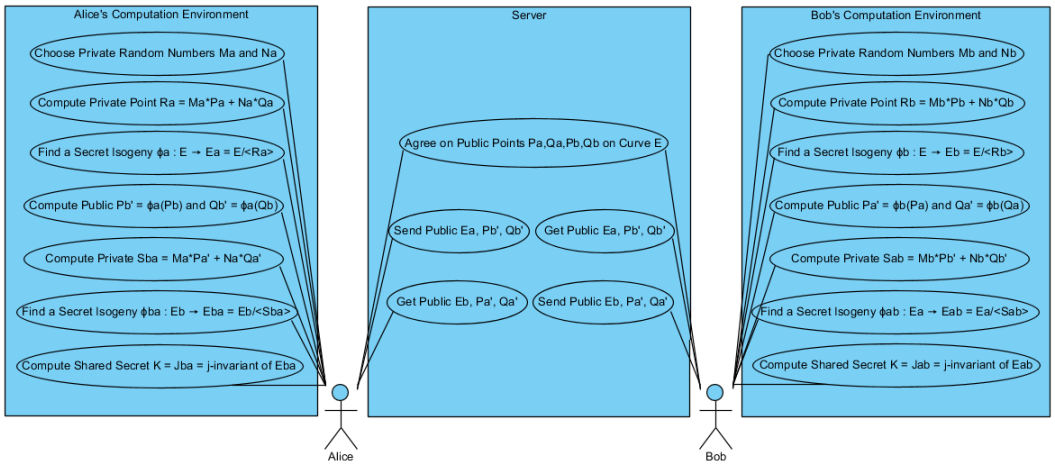

Figure A.3: SIDH Key Exchange Use Case Diagram

The order of the procedures of SIDH is crucial and they are needed to be demonstrated for clarifying the overall work. The sequence diagram for SIDH can be seen below.

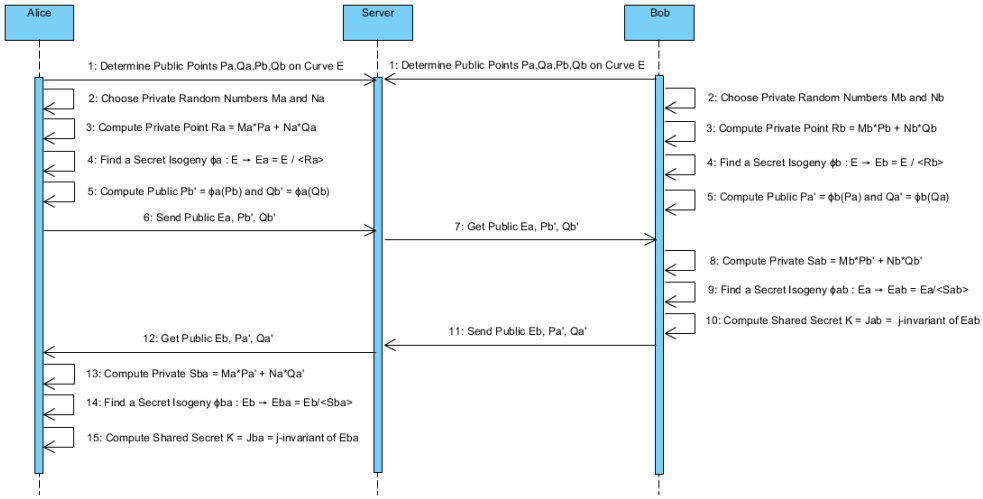

Figure A.4: SIDH Key Exchange Sequence Diagram

A.2.2 To provide useful functions for fast Kummer surface arithmetic

To perform Hadamard operation

A function that has the capability of performing a 4 way parallel SIMD friendly Hadamard transform operation on a tuple of four projective field elements $(x : y : z : t)$ must be provided. For an input point $(x : y : z : t)$, the result after performing an Hadamard operation must be $(x + y + z + t : x + y - z - t : x - y + z - t : x - y - z + t)$.

To perform field multiplication

A function that can compute the 4 way coordinate-wise multiplication of two different tuples of four projective field elements $(x_1 : y_1 : z_1 : t_1)$ and $(x_2 : y_2 : z_2 : t_2)$ must be provided. The output of the function must be $(x_1x_2 : y_1y_2 : z_1z_2 : t_1t_2)$.

To perform field squaring

A function must be provided which performs a 4 way parallel SIMD friendly coordinate-wise squaring operation on a tuple of four projective field elements $(x : y : z : t)$. The input must be $(x : y : z : t)$ and the function must return $(x^2 : y^2 : z^2 : t^2)$.

To perform inversion operation

A function that can compute the inverse of a tuple of four projective field elements $(x : y : z : t)$ in projective 3 space must be provided. The input must be $(x : y : z : t)$ and the function must return $(1/x : 1/y : 1/z : 1/t)$.

To perform the doubling routine

A doubling routine must be provided which can be capable of computing the double of a Kummer point in projective 3 space that is represented as a tuple of four projective field elements $(x : y : z : t)$. The routine consists of various sub-operations such as Hadamard, squaring, multiplication and inversion. All of these operations are 4 way parallel SIMD friendly.

A.2.3 To gather present open source codes related to the project

To get the open source codes for isogeny evaluations scheme on Kummer surfaces

An existing Magma [3] implementation of (2,2) isogeny evaluations written by Costello [5] is used as a guide when creating K4SIDH.

To get the open source codes for back-and-forth maps

In order to provide full maps between Montgomery curve and Kummer surface, we benefit from Costello's implementation [5] written in Magma algebraic language [3] for the implementation of the back-and-forth maps.

A.2.4 Non-functional Requirements

To have a test scenario for the application

In the end of the isogeny evaluations, a test scheme must be provided in order to test the validity of the implementation.

To ensure the maximum performance as possible in the application

The application should avoid unnecessary computations. The Kummer surface arithmetic functions should be constructed in a way that they can get the maximum profit from 4 way

parallel AVX2 instructions.

To have a constant-time running application

The application must run always in constant-time. Any instruction that can violate the constant-time running (e.g. branching) must not be used in the implementation.

To provide a clock cycle counter in the application

A clock cycle counter must be used to detect how many clock cycles were needed to complete the isogeny evaluations. The overall performance of the application can be measured with this clock cycle counter.

A.3 Glossary

A glossary is given for clarifying the domain specific terms used in the document. The terms and their descriptions are listed in the following table.

Term	Description
SIDH	Abbreviation for Supersingular Isogeny Diffie-Hellman. SIDH is a new type of key exchange procedure which is accepted that it can resist quantum attacks.
SIMD	Abbreviation for Single Instruction Multiple Data. SIMD allows effecting numerous data with using a single special instruction and enables parallelization in the implementation.
Clock Cycle	A measure for a CPU's speed. A clock cycle is the total time needed for a single pulse that is made by the oscillator of the CPU.
Magma	A computational algebra system. Magma is a strong and effective algebraic tool which is distributed by the Computational Algebra Group at the University of Sydney.
AVX2	256-bit Advanced Vector Extensions. AVX2 stands for the 256-bit extensions for SIMD instructions. It was first proposed by Intel.

Appendix B

DESIGN SPECIFICATION DOCUMENT

14.05.2019

Revision 3.0

Revision History

Revision	Date	Explanation
1.0	16.01.2019	Initial high level design.
2.0	09.04.2019	Detailed design.
3.0	14.05.2019	Major changes and additions in the high level and detailed design.

B.1 Introduction

The purpose of the project is to provide an implementation of chain of isogeny evaluations on Kummer surfaces that fits perfectly to the Supersingular Isogeny Diffie-Hellman key exchange scheme. These evaluations take part in Alice's public key generation and shared secret computation phases. The implementation made use of the SIMD AVX2 instructions in the Kummer Surface arithmetic functions. 4 way field multiplication and squaring codes were generated with the help of a Comba multiplication based generator **comba_generator.c**. In addition, with the usage of [4], a Magma code of the maps between supersingular Montgomery curves and supersingular Kummer surfaces was written.

The implementation was done in a LINUX operating system environment, Ubuntu 16.04 LTS, with the C language, benefiting from Intel AVX2 SIMD instructions.

The design is based on K4SIDH Requirements Specification Document, Revision 3.0, in file Yassi-K4SIDH-RSD-2019-05-14-Rev-3.0.pdf [21]. The overall high-level software system structure and the working environment of K4SIDH are given in Section 2 and design details of all implementation functions are given in Section 3.

B.2 K4SIDH High Level Design

In this section, the high level design of the K4SIDH software system is given in terms of the general system structure and the development and working environment.

B.2.1 K4SIDH Software System Structure

K4SIDH forms the bottleneck part of the Alice's public key generation and shared secret computation phases of SIDH key exchange. In order to clarify the overall work done in the SIDH and clearly see where K4SIDH falls within the structure of SIDH, first we give an high level overview of SIDH key exchange.

SIDH consists of four major functions which are Alice's public key generation, Bob's public key generation, Alice's shared secret computation and Bob's shared secret computation. The structure of SIDH is shown in Figure B.1.

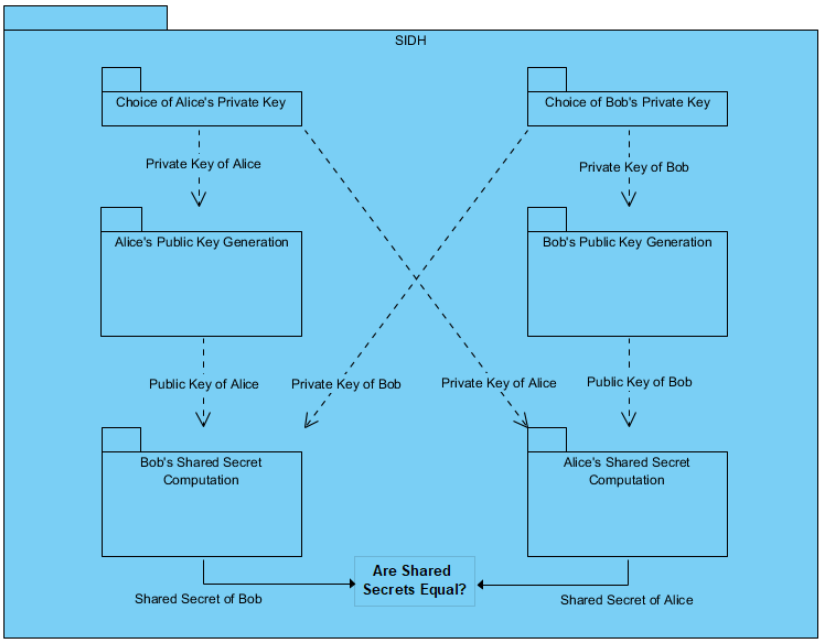

Figure B.1: SIDH High Level Design Package Diagram

Before starting the public key generation phases for Alice and Bob, both of them choose their private keys. The private key of Alice is chosen as a random even number between 1 and $2^{372} - 1$, and the private key of Bob is a random number which is multiple of 3, between 1 and $3^{239} - 1$. Public key generation function of Alice/Bob takes the private key of Alice/Bob and uses them in the isogeny evaluations. Alice evaluates consecutive 4-isogenies, while Bob evaluates consecutive 3-isogenies. After evaluating the isogenies for predetermined number of times, a normalization is performed thanks to an inversion function. This normalization follows getting the public key of Alice/Bob. Finally, the created public key of Alice/Bob is sent to the shared secret computation function.

From this point, we will describe only Alice's side of the key exchange because of its relation with K4SIDH. The process for Bob is very similar to Alice's.

Shared secret for Alice is computed with the private key of Alice and the public key of Bob. First, the keys are sent to a ladder of operations which accepts three affine points. These three points come from Bob's public key. After computing the corresponding curve constant A with the help of the input keys, the ladder is performed with a sequence of double-add routines.

When the ladder is completed, similar to the operations in the public key generation phases, 4-isogenies are computed and evaluated. Note that, these 4-isogenies can be replaced with (2,2) Kummer isogenies in order to reduce the time spent on the bottleneck operations. In the end, the j-invariant value of the last curve we get after the sequence of 4-isogenies is computed and this value corresponds to the shared secret of Alice. If the results which come from Alice and Bob's shared secret computation functions are the same, then this shows that the key exchange is completed successfully.

Now the general structure of SIDH is clear, we can represent the usage of K4SIDH in the key exchange. At the beginning of Alice's public key generation or shared secret computation, we can use the chain of maps given in the function *MonToKum* for travelling from the supersingular Montgomery curve we are currently on to the corresponding supersingular Kummer surface. After deriving necessary Kummer surface parameters and with the usage of a Kummer point, we can start evaluating (2,2) isogenies on Kummer surfaces as in the function *kummer_ISOG* given below.

```
void kummer_ISOG(vec *R) {  // the main isogeny evaluation loop
  vec P4[32], P2[32], mus[32], pis[32], tP[32], tK3[32], tK4[32];
  int i;

  for(i = 0; i < 32; i++) tP[i] = P[i];
  for(i = 0; i < 32; i++) tK3[i] = K3[i];
  for(i = 0; i < 32; i++) tK4[i] = K4[i];
  for(i = 371; i >= 3; i--) {
    kummer_eDBLs(P4,tP,tK3,tK4,i-2);
    kummer_DBL(P2,P4,tK3,tK4);
    IsogenousSquaredThetasFromTorsionKernel(mus,pis,P2,P4);
    IsogenyKummerPoint(tP,pis);
    KummerConstantsFromSquaredThetas(tK3,tK4,mus);
  }
  for(i = 0; i < 32; i++) R[i] = tP[i];
}
```

Code B.1: Main Isogeny Evaluation Loop

When the evaluations are done, Kummer surface parameters and the Kummer point obtained latest can be used for returning to corresponding supersingular Montgomery curve from the last supersingular Kummer surface we currently operate on. The chain of maps for these operations are given in the function *KumToMon* in the file **maps.mag**. By reaching to a Montgomery curve after evaluating isogenies on Kummer surfaces, Alice's public key generation or shared secret computation phase can be terminated as usual. The rest of the process proceeds the same as before.

B.2.2 K4SIDH Environment

The implementation must be executed on a computer which has a 64-bit Intel processor. The Intel processor must support AVX2 instructions to be able to run the application. 4 way parallel SIMD AVX2 instructions take part in the Kummer surface arithmetic functions of the implementation.

As mentioned in the introduction, the implementation was done with C language, in the Eclipse Integrated Development Environment. GCC (GNU C Compiler) version 5.5.0 was used to compile the machine code.

In addition, to be able to run the script of maps, Magma Computational Algebra System [3] must be installed. Alternatively, online Magma calculator [15] can be used for running the script.

B.3 K4SIDH Detailed Design

In this section, the main function, the back-and-forth maps between supersingular Montgomery curves and supersingular Kummer surfaces and the bottleneck functions used in the Kummer surface arithmetic are explained in detail.

B.3.1 K4SIDH Main Function

The main function of K4SIDH consists of three parts: printing the last Kummer point, checking the validity of the application, measuring the speed. The validity checking is performed by calling the Magma script version of the (2,2) isogeny evaluation sequence [5], and checking whether the returned points from K4SIDH and the Magma script are equal to each other or not. The last Kummer points reached by isogeny evaluations in K4SIDH and the Magma script are used for the comparison operation. Note that, the file paths demonstrated in the function must be changed when running the application in different working environments. The paths are given for the purpose of demonstration. For measuring the total cycle count of the application, benchmarking, system counter is used by calling the function *cpucycles*. This function is called before and after a run of K4SIDH, and difference of the returned numbers are cumulatively added. The sum indicates the total number of clock cycles needed for running

K4SIDH. The function is given below.

```
int main() {
  vec RES[32];
#ifdef PRINT
  kummer_ISOG(RES); // run Kummer isogeny evaluations sequence
  printf("\n\nThe last Kummer point is:\n");
  print_kpt4x751(RES); // print the last Kummer point to screen
#endif
#ifdef VALIDITY
  FILE *f, *s, *t;
  char ch;
  s = fopen("../K4SIDH/Kummer.mag", "r"); // read Magma script version of the isogeny loop
  t = fopen("../K4SIDH/Test.mag", "w"); // make changes in new file
  while((ch = fgetc(s)) != EOF) fputc(ch, t); // copy script to new file
  fclose(s); fclose(t);
  f = fopen("../K4SIDH/Test.mag", "a"); // open new file to append results
  printf("\n\nRunning K4SIDH Validation Test.......\n\n\n");
  kummer_ISOG(RES); // run Kummer isogeny evaluations sequence
  fprint_kpt4x751(f,RES); // print results to opened file
  fprintf(f, "P1 eq P[1];\n"); // check if 1st channel of the result is true
  fprintf(f, "P2 eq P[2];\n"); // check if 2nd channel of the result is true
  fprintf(f, "P3 eq P[3];\n"); // check if 3rd channel of the result is true
  fprintf(f, "P4 eq P[4];\n"); // check if 4th channel of the result is true
  fprintf(f, "exit;\n"); // append exit statement for closing Magma after checking is done
  fclose(f);
  int ret = system("cd ~ && ./magma \"K4SIDH/Test.mag\""); // run new Magma script and make the comparison
  if(ret != 0) return -1;
#endif
#ifdef SPEED
  unsigned long long cycles, cycles1, cycles2, n;
  clock_t src, trg;
  double time;
  cycles = 0;
  int LOOP = 10;
  printf("\n\nRunning K4SIDH Speed Test.......\n\n\n");
    src = clock(); // get starting time
  for (n = 0; n < LOOP; n++){ // start running isogeny sequence LOOP times
    cycles1 = cpucycles(); // measure speed of the application
    kummer_ISOG(RES); // run Kummer isogeny evaluations sequence
    cycles2 = cpucycles(); // measure speed of the application
    cycles = cycles+(cycles2-cycles1); // measure speed of the application
  }
    trg = clock(); // get finish time
    time = ((double) (trg - src)) / CLOCKS_PER_SEC; // calculate running time
  printf("K4SIDH runs in:\n%10lld cycles, %f seconds.\n", cycles/LOOP, time/LOOP);
  printf("\n");
#endif
  return 0;
}
```

Code B.2: Main Function of K4SIDH

B.3.2 K4SIDH Back-and-forth Maps

To be able to see where the back-and-forth maps between Montgomery curves and Kummer surfaces should be located and which parameters and points should be associated with the maps, an existing SIDH implementation of Microsoft Research [17] was used and all the assumptions were made based on that implementation. The implementation makes use of a large prime field of the form $p = 2^{372}3^{239} - 1 = 3 \pmod 4$. It starts operating on a Montgomery curve $By^2 = x^3 + Ax^2 + x$ over $\mathbb{F}_{p^2}$ with constants $A = 0$ and $B = 1$. The scheme continues on supersingular Montgomery curves until the end of the shared secret computations. To be able to evaluate the isogenies and complete the Alice's public key generation and shared

secret computation on Kummer surfaces, one must carry the points and constants used in evaluation from supersingular Montgomery curves to Kummer surfaces. Costello [4] introduced the necessary back-and-forth maps between supersingular Montgomery curves and Kummer surfaces. With the usage of these maps and three other maps which will be declared at the later part of this section, transfer of points and constants can be accomplished.

Chain of Maps from Montgomery Curve to Kummer Surface

In the beginning of the Alice's public key generation and shared secret computation, we are on a supersingular Montgomery curve and all we need to do is, deriving necessary constants for constructing algebraic structures isogenous to supersingular Montgomery curves. Instead of expressing a supersingular Montgomery curve in the form of $By^2 = x^3 + Ax^2 + x$, we can define it with two rational 2-torsion points $(\alpha, 0), (1/\alpha, 0)$ as the roots of the curve and declare it as

$$E_\alpha/\mathbb{F}_{p^2} : y^2 = x(x - \alpha)(x - 1/\alpha).$$

The coordinates x, y are representatives for a point on E_α and can be acquired from computing R_x/R_z and the isogenous 2-torsion point $(\alpha, 0)$ on the curve E_α can be computed as

$$\alpha = (\sqrt{(A/C)^2 - 4} - A/C)/2.$$

With using α, three roots r_1, r_2, r_3 can be calculated for composing the supersingular Montgomery curve $\tilde{E}_\alpha$. The roots and equation of the curve are

$$r_1 = (\alpha - 1/\alpha)^{p-1}, \quad r_2 = (\alpha)^{p-1}, \quad r_3 = (1/\alpha)^{p-1},$$

$$\tilde{E}_\alpha/\mathbb{F}_{p^2} : y^2 = (x - r_1)(x - r_2)(x - r_3).$$

Therefore, a map ψ between E_α and $\tilde{E}_\alpha$ can be defined as

$$\psi : E_\alpha/\mathbb{F}_{p^2} \to \tilde{E}_\alpha/\mathbb{F}_{p^2}, \quad (x, y) \mapsto \Big((\hat{\beta}/\beta)^2 x + r_1, (\hat{\beta}/\beta)^3 y\Big)$$

with $\beta = \sqrt{\alpha^2 - 1/\alpha}$ and $\hat{\beta} = \sqrt{r_3 - r_2}$.

Next, we need to construct an hyperelliptic curve C_α over $\mathbb{F}_p$ and by taking its jacobian J_{C_α}, a map between $\tilde{E}_\alpha$ and J_{C_α} can be built. We can get C_α by fixing the roots of the sextic $z_1, z_2, \ldots, z_6$ and write it as

$$C_\alpha/\mathbb{F}_p : y^2 = (x - z_1)(x - z_2)(x - z_3)(x - z_4)(x - z_5)(x - z_6).$$

The roots can be deduced from β and γ. We know how to compute β from the construction of $\tilde{E}_\alpha$. So, defining $\gamma = \sqrt{\alpha}$ and getting the coefficients in $\mathbb{F}_p$, one can get $\beta_0, \beta_1, \gamma_0, \gamma_1$ from $\beta = \beta_0 + \beta_1 i$ and $\gamma = \gamma_0 + \gamma_1 i$.

Hereby, the roots of the sextic are shaped as

$$z_1 = \beta_0/\beta_1, \quad z_2 = \gamma_0/\gamma_1, \quad z_3 = -\gamma_0/\gamma_1, \quad z_4 = -\beta_1/\beta_0, \quad z_5 = -\gamma_1/\gamma_0, \quad z_6 = \gamma_1/\gamma_0$$

and C_α is defined. Now, if we get the J_{C_α}, a map ρ between $\tilde{E}_\alpha$ and J_{C_α} can be composed as

$$\rho : \tilde{E}_\alpha/\mathbb{F}_{p^2} \to J_{C_\alpha}/\mathbb{F}_{p^2}, \quad (\tilde{x}, \tilde{y}) \mapsto (x^2 + u_1 x + u_0, v_1 + v_0),$$

with $w = r_3(1 - r_1)(r_2 - 1)^2$ where

$$u_1 = 2i\frac{\tilde{x} + 1}{\tilde{x} - 1}, \quad u_0 = -1, \quad v_1 = -4i\frac{\tilde{y}(\tilde{x} + 3)}{w(\tilde{x} - 1)^2}, \quad v_0 = \frac{4\tilde{y}}{w(\tilde{x} - 1)}.$$

The map above is only suitable for jacobian of C_α over $\mathbb{F}_{p^2}$. Use of a trace map $\mathcal{T}$ can solve this situation and map $J_{C_\alpha}/\mathbb{F}_{p^2}$ to $J_{C_\alpha}/\mathbb{F}_p$ by

$$\mathcal{T} : J_{C_\alpha}/\mathbb{F}_{p^2} \to J_{C_\alpha}/\mathbb{F}_p, (x^2+u_1x+u_0, v_1x+v_0) \mapsto (x^2+u_1x+u_0, v_1x+v_0)+_J(x^p+u_1^px+u_0^p, v_1^px+v_0^p)$$

where $+_J$ stands for the addition law in $J_{C_\alpha}/\mathbb{F}_{p^2}$.

Before reaching the Kummer surface, we need to visit only one more algebraic structure which is the jacobian of a hyperelliptic curve over $\mathbb{F}_p$ that must be in Rosenhain Form and be isomorphic to C_α. This curve is $C_{\lambda,\mu,\nu}$ and its equation is

$$C_{\lambda,\mu,\nu}/\mathbb{F}_p : y^2 = x(x - 1)(x - \lambda)(x - \mu)(x - \nu).$$

The constants used in the equation can be derived with the help of $\beta_0, \beta_1, \gamma_0, \gamma_1$ as

$$\lambda = -\frac{(\beta_0\gamma_1 + \beta_1\gamma_0)(\beta_0\gamma_0 + \beta_1\gamma_1)}{(\beta_0\gamma_0 - \beta_1\gamma_1)(\beta_0\gamma_1 - \beta_1\gamma_0)}, \quad \mu = -\frac{(\beta_0\gamma_0 + \beta_1\gamma_1)(\beta_0\gamma_0 - \beta_1\gamma_1)}{(\beta_0\gamma_1 + \beta_1\gamma_0)(\beta_0\gamma_1 - \beta_1\gamma_0)}, \quad \nu = -\frac{(\beta_0\gamma_0 + \beta_1\gamma_1)^2}{(\beta_0\gamma_1 - \beta_1\gamma_0)^2}.$$

Taking the jacobian of $C_{\lambda,\mu,\nu}$, now we can produce a map ζ between J_{C_α} and $J_{C_{\lambda,\mu,\nu}}$ which corresponds to

$$\zeta : J_{C_\alpha}/\mathbb{F}_p \to J_{C_{\lambda,\mu,\nu}}/\mathbb{F}_p, \quad (x^2 + u_1x + u_0, v_1x + v_0) \mapsto (x^2 + u_1'x + u_0', v_1'x + v_0')$$

where

$$u_1' = \ell_1^{-1}((ad + bc)u_1 - 2acu_0 - 2bc), \quad u_0' = \ell_1^{-1}(a^2u_0 - abu_1 + b^2),$$

$$v_1' = e(\ell_1^2\ell_2)^{-1}(c^2(cu_1 - 3d)(u_0v_1 - u_1v_0) + cv_0(c^2u_0 - 3d^2) + d^3v_1),$$

$$v_0' = -e(\ell_1^2\ell_2)^{-1}(ac^2(u_0u_1v_1 - u_1^2v_0 + u_0v_0) - c(2ad + bc)(u_0v_1 - u_1v_0) - d(ad + 2bc)v_0 + bd^2v_1).$$

Undefined constants in the above formulas can be found by using z_1, z_2, z_4 from the roots of the sextic in C_α as

$$a = z_2 - z_4, \quad b = -az_1, \quad c = z_2 - z_1, \quad d = -cz_4, \quad e = \sqrt{ac(a - c)(a - \nu c)(a - \mu c)(a - \lambda c)},$$

$$\ell_1 = c^2u_0 - cdu_1 + d^2, \quad \ell_2 = ad - bc.$$

Finally, defining a map Ψ from $J_{C_{\lambda,\mu,\nu}}$ to $\mathcal{K}$ [9], will complete the chain of maps from the Montgomery curve and corresponding Kummer surface. However, we need to construct a special squared Kummer surface with the relevant parameters (i.e., λ, γ_0, γ_1) arose from the previously defined curves. Four constants $\mu_1, \mu_2, \mu_3, \mu_4$ can be constituted as

$$\mu_1 = \left(\frac{\gamma_0^2 - \gamma_1^2}{\gamma_0^2 + \gamma_1^2}\right)\sqrt{\lambda}, \quad \mu_2 = \left(\frac{\gamma_0^2 - \gamma_1^2}{\gamma_0^2 + \gamma_1^2}\right)/\sqrt{\lambda}, \quad \mu_3 = 1, \quad \mu_4 = 1.$$

Now, equation of the squared Kummer surface can be given as

$$\mathcal{K}^{Sqr} : FX_1X_2X_3X_4 = (X_1^2 + X_2^2 + X_3^2 + X_4^2 - G(X_1 + X_2)(X_3 + X_4) - H(X_1X_2 + X_3X_4))^2$$

where

$$F = 4\mu_1\mu_2\frac{(\mu_1+\mu_2+2)^2(\mu_1+\mu_2-2)^2}{(\mu_1\mu_2-1)^2}, \quad G = \mu_1+\mu_2, \quad H = \frac{\mu_1^2+\mu_2^2-2}{\mu_1\mu_2-1}.$$

This leads to the map

$$\Psi : J_{C_{\lambda,\mu,\nu}} \to \mathcal{K}^{Sqr}, \quad (x^2+u_1'x+u_0', v_1'x+v_0') \mapsto (X:Y:Z:T)$$

where

$$X = \mu_1(u_0'(\mu-u_0')(\lambda+u_1'+\nu)-v_0'^2), \quad Y = \mu_2(u_0'(\nu\lambda-u_0')(1+u_1'+\mu)-v_0'^2),$$
$$Z = \mu_3(u_0'(\nu-u_0')(\lambda+u_1'+\mu)-v_0'^2), \quad T = \mu_4(u_0'(\mu\lambda-u_0')(1+u_1'+\nu)-v_0'^2).$$

At last, we successfully carry the points and parameters of a supersingular Montgomery curve to a squared Kummer surface. We can denote the composition of the maps used with a map η as

$$\eta : E_\alpha/\mathbb{F}_{p^2} \to \mathcal{K}^{Sqr}, \quad P \mapsto (\Psi \circ \zeta \circ \mathcal{T} \circ \rho \circ \psi)(P).$$

Chain of Maps from Kummer Surface to Montgomery Curve

After evaluating the isogenies on Kummer Surface and exiting the main loops in Alice's public key generation and shared secret computation, we need to return to the supersingular Montgomery curve which we came from. Now, we have a Kummer point $(X:Y:Z:T)$ and corresponding squared Kummer constants μ_1,μ_2,μ_3,μ_4 and we can use them in the map from $\mathcal{K}^{Sqr}$ to $J_{C_{\lambda,\mu,\nu}}$, given in [8], [10].

First, we call $\vartheta_1^2,\vartheta_2^2,\vartheta_3^2,\vartheta_4^2$(fundamental squared Theta constants) to μ_1,μ_2,μ_3,μ_4 and $\Theta_1^2,\Theta_2^2,\Theta_3^2,\Theta_4^2$(fundamental squared Theta functions) to X,Y,Z,T for the sake of terminology, respectively. From Frobenius identities [14], the squared Theta constants $\vartheta_5^2,\vartheta_6^2$ and ϑ_7^2 can be constituted with the equations below as

$$\vartheta_5^4+\vartheta_6^4 = \vartheta_1^4-\vartheta_2^4-\vartheta_3^4+\vartheta_4^4, \quad \vartheta_5^2\vartheta_6^2 = \vartheta_1^2\vartheta_4^2-\vartheta_2^2\vartheta_3^2, \quad \vartheta_7^4 = \vartheta_3^4-\vartheta_4^4+\vartheta_5^4.$$

It can be seen from the equations that ϑ_5^2 has four, ϑ_7^2 has two solutions. In this way, with calculating

$$\vartheta_8^2 = (\vartheta_1^2\vartheta_5^2 - \vartheta_4^2\vartheta_6^2)/\vartheta_7^2, \quad \vartheta_9^2 = (\vartheta_1^2\vartheta_3^2 - \vartheta_2^2\vartheta_4^2)/\vartheta_7^2, \quad \vartheta_{10}^2 = (\vartheta_2^2\vartheta_5^2 - \vartheta_3^2\vartheta_6^2)/\vartheta_7^2,$$

all squared Theta constants needed are acquired.

Now, using $\vartheta_1, \ldots, \vartheta_{10}$ and $\Theta_1^2, \Theta_2^2, \Theta_3^2, \Theta_4^2$, with respect to [8], squares of non-fundamental Theta functions which will be used in the computation of coefficients u_0, u_1, v_0, v_1 can be found as

$$\Theta_7^2 = \frac{\Theta_4^2\vartheta_6^2\vartheta_8^2 + \Theta_2^2\vartheta_5^2\vartheta_{10}^2 - \Theta_3^2\vartheta_6^2\vartheta_{10}^2 - \Theta_1^2\vartheta_5^2\vartheta_8^2}{\vartheta_6^4 - \vartheta_5^4},$$

$$\Theta_9^2 = \frac{\Theta_3^2\vartheta_5^2\vartheta_8^2 + \Theta_1^2\vartheta_6^2\vartheta_{10}^2 - \Theta_4^2\vartheta_5^2\vartheta_{10}^2 - \Theta_2^2\vartheta_6^2\vartheta_8^2}{\vartheta_8^4 - \vartheta_{10}^4},$$

$$\Theta_{11}^2 = \frac{\Theta_3^2\vartheta_5^2\vartheta_{10}^2 + \Theta_1^2\vartheta_6^2\vartheta_8^2 - \Theta_4^2\vartheta_5^2\vartheta_8^2 - \Theta_2^2\vartheta_6^2\vartheta_{10}^2}{\vartheta_6^4 - \vartheta_5^4},$$

$$\Theta_{12}^2 = \frac{\Theta_4^2\vartheta_6^2\vartheta_{10}^2 + \Theta_2^2\vartheta_5^2\vartheta_8^2 - \Theta_3^2\vartheta_6^2\vartheta_8^2 - \Theta_1^2\vartheta_5^2\vartheta_{10}^2}{\vartheta_8^4 - \vartheta_{10}^4},$$

$$\Theta_{13}^2 = \frac{\Theta_3^2\vartheta_7^2\vartheta_8^2 + \Theta_2^2\vartheta_9^2\vartheta_{10}^2 - \Theta_4^2\vartheta_7^2\vartheta_{10}^2 - \Theta_1^2\vartheta_8^2\vartheta_9^2}{\vartheta_7^4 - \vartheta_9^4},$$

$$\Theta_{14}^2 = \frac{\Theta_4^2\vartheta_6^2\vartheta_9^2 + \Theta_3^2\vartheta_5^2\vartheta_7^2 - \Theta_2^2\vartheta_6^2\vartheta_7^2 - \Theta_1^2\vartheta_5^2\vartheta_9^2}{\vartheta_7^4 - \vartheta_9^4},$$

$$\Theta_{16}^2 = \frac{\Theta_4^2\vartheta_5^2\vartheta_7^2 + \Theta_3^2\vartheta_6^2\vartheta_9^2 - \Theta_2^2\vartheta_5^2\vartheta_9^2 - \Theta_1^2\vartheta_6^2\vartheta_7^2}{\vartheta_7^4 - \vartheta_9^4}.$$

Consequently, after the calculations of constants A, B, C, D, E, F, G, H as

$$A = \vartheta_1^2+\vartheta_2^2+\vartheta_3^2+\vartheta_4^2, \quad B = \vartheta_1^2+\vartheta_2^2-\vartheta_3^2-\vartheta_4^2, \quad C = \vartheta_1^2-\vartheta_2^2+\vartheta_3^2-\vartheta_4^2, \quad D = \vartheta_1^2-\vartheta_2^2-\vartheta_3^2+\vartheta_4^2,$$

$$E = ABCD/(\vartheta_1^2\vartheta_4^2 - \vartheta_2^2\vartheta_3^2)/(\vartheta_1^2\vartheta_3^2 - \vartheta_2^2\vartheta_4^2)/(\vartheta_1^2\vartheta_2^2 - \vartheta_3^2\vartheta_4^2),$$

$$F = (\vartheta_1^4 - \vartheta_2^4 - \vartheta_3^4 + \vartheta_4^4)/(\vartheta_1^2\vartheta_4^2 - \vartheta_2^2\vartheta_3^2), \quad G = (\vartheta_1^4 - \vartheta_2^4 + \vartheta_3^4 - \vartheta_4^4)/(\vartheta_1^2\vartheta_3^2 - \vartheta_2^2\vartheta_4^2),$$

$$H = (\vartheta_1^4 + \vartheta_2^4 - \vartheta_3^4 - \vartheta_4^4)/(\vartheta_1^2\vartheta_2^2 - \vartheta_3^2\vartheta_4^2)$$

and with the use of λ, μ and ν, the coefficients appear as

$$u_0' = \lambda\vartheta_8^2\Theta_{14}^2/(\vartheta_{10}^2\Theta_{16}^2), \quad u_1' = (\lambda-1)\vartheta_5^2\Theta_{13}^2/(\vartheta_{10}^2\Theta_{16}^2) - u_0' - 1,$$

$$v_0'^2 = -(\Theta_{12}^2\Theta_7^2\vartheta_2^2\vartheta_3^2\vartheta_9^4+\Theta_{11}^2\Theta_9^2\vartheta_1^2\vartheta_4^2\vartheta_7^4+2\vartheta_1^2\vartheta_2^2\vartheta_3^2\vartheta_4^2(\Theta_1^2\Theta_3^2+\Theta_2^2\Theta_4^2)+(\Theta_1^4+\Theta_2^4+\Theta_3^4+\Theta_4^4-F(\Theta_1^2\Theta_4^2 +\Theta_2^2+\Theta_3^2)-G(\Theta_1^2+\Theta_3^2+\Theta_2^2+\Theta_4^2)-H(\Theta_1^2\Theta_2^2+\Theta_3^2\Theta_4^2))(\vartheta_1^2\vartheta_3^2+\vartheta_2^2\vartheta_4^2)/E)\vartheta_8^2\vartheta_3^4\vartheta_1^4\Theta_{14}^2/(\Theta_{16}^6\vartheta_2^6\vartheta_4^6\vartheta_{10}^6),$$

$$v_1' = (1/2)(u_0'^3-u_0'^2(u_1'^2+u_1'(1+\lambda+\mu+\nu)+\lambda+\mu+\nu+\lambda\mu+\nu\lambda+\mu+\nu)+u_0'\lambda\mu\nu+u_1'v_0'^2)/(v_0'u_0').$$

These coefficients result in a map $\hat{\Psi}$ that is

$$\hat{\Psi} : \mathcal{K}^{Sqr} \to J_{C_{\lambda,\mu,\nu}}, \quad (X:Y:Z:T) \mapsto (x^2+u_1'x+u_0', v_1'x+v_0').$$

At the next step, we should jump back to J_{C_α} from $J_{C_{\lambda,\mu,\nu}}$. Inverting the formulae of the map ζ and using the constants a, b, c, d, e derived from C_α, following coefficients are obtained.

$$u_0 = (u_1'bd + u_0'd^2 + b^2)/(a^2 + u_1'ac + u_0'c^2),$$

$$u_1 = (adu_1' + 2ab + bcu_1' + 2dcu_0')/(a^2 + u_1'ac + u_0'c^2),$$

$v_0 = (a^4v_0'd^3+a^4v_1'd^2b-2a^3d^3u_0'cv_1'+2a^3v_0'd^3cu_1'-2a^3b^2v_1'cd-3a^2v_0'c^2d^2bu_1'-a^2c^2u_1'u_0'd^3v_1'-a^2 v_0'd^3c^2u_0'-3a^2dc^2v_0'b^2+a^2v_0'd^3c^2u_1'^2+a^2b^3c^2v_1'+3a^2v_1'd^2bu_0'c^2+2abd^2u_0'c^3v_1'u_1'+2av_0'c^3d^2bu_0'+2a c^3v_0'b^3-2av_0'c^3d^2bu_1'^2+b^2dc^4u_1'^2v_0'-b^3u_0'c^4v_1'+b^3c^4u_1'v_0'-b^2dc^4u_0'v_0'-b^2du_0'c^4v_1'u_1')/((a^4+2ca^3u_1' +2c^2a^2u_0' + c^2a^2u_1'^2 + 2c^3au_1'u_0' + u_0'^2c^4)e),$

$v_1 = (v_1'd^2a^5+3a^4d^2cv_0'-2a^4v_1'bcd-3a^3d^2v_1'u_0'c^2+3a^3d^2c^2u_1'v_0'-6a^3c^2bv_0'd+a^3c^2v_1'b^2+a^2d^2c^3u_1'^2 v_0'-a^2d^2u_0'c^3v_1'u_1'-a^2d^2c^3u_0'v_0'+6a^2du_0'c^3v_1'b-6a^2dc^3u_1'bv_0'+3a^2c^3b^2v_0'+2adu_0'c^4v_1'bu_1'-2adc^4u_1'^2 v_0'b+2adc^4u_0'v_0'b-3au_0'c^4v_1'b^2+3ac^4u_1'b^2v_0'-u_0'c^5v_1'b^2u_1'+c^5u_1'^2v_0'b^2-c^5u_0'v_0'b^2)/((a^4+2ca^3u_1'+2c^2 a^2u_0' + c^2a^2u_1'^2 + 2c^3au_1'u_0' + u_0'^2c^4)e).$

The map ζ^{-1} is now formed as

$$\zeta^{-1} : J_{C_{\lambda,\mu,\nu}}/\mathbb{F}_p \to J_{C_\alpha}/\mathbb{F}_p, \quad (x^2+u_1'x+u_0', v_1'x+v_0') \mapsto (x^2+u_1x+u_0, v_1x+v_0).$$

From this point, the route we will follow is directly taken from [4, §3].

First, we define a map ϕ^{-1} from $C_\alpha/\mathbb{F}_{p^2}$ to $\tilde{C}_\alpha/\mathbb{F}_{p^2}$ as

$$\phi^{-1} : C_\alpha/\mathbb{F}_{p^2} \to \tilde{C}_\alpha/\mathbb{F}_{p^2}, \quad (x, y) \mapsto \Big(-\frac{x-i}{x+i}, -i\frac{yw}{(x+i)^3}\Big).$$

Defining the *pullback* map ω [19],

$$\omega : \tilde{C}_\alpha/\mathbb{F}_{p^2} \to \tilde{E}_\alpha/\mathbb{F}_{p^2}, \quad (x, y) \mapsto (x^2, y)$$

and and extending the map ϕ^{-1} linearly to Divisor class of C_α $(Div_{\bar{\mathbb{F}}_p}(C_\alpha))$, we can compose a map $\hat{\rho}$ in the way that

$$\hat{\rho} : J_{C_\alpha}/\mathbb{F}_p \to \tilde{E}_\alpha/\mathbb{F}_{p^2} \times \tilde{E}_\alpha/\mathbb{F}_{p^2}, \quad P \mapsto ((\omega \circ \phi^{-1})(x_1, y_1), (\omega \circ \phi^{-1})(x_2, y_2)),$$

where $P \in J_{C_\alpha}/\mathbb{F}_p$.

Performing the addition of the points according to the addition law on $\tilde{E}_\alpha/\mathbb{F}_{p^2}$ as

$$+_{\tilde{E}_\alpha} : \tilde{E}_\alpha/\mathbb{F}_{p^2} \times \tilde{E}_\alpha/\mathbb{F}_{p^2} \to \tilde{E}_\alpha/\mathbb{F}_{p^2},$$

and using the inverse of the map ψ as

$$\psi^{-1} : \tilde{E}_\alpha/\mathbb{F}_{p^2} \to E_\alpha/\mathbb{F}_{p^2}, \quad (x, y) \mapsto \Big((\beta/\hat{\beta})^2(x - r_1), (\beta/\hat{\beta})^3 y\Big),$$

we accomplished carrying the points from a squared Kummer surface to a supersingular Montgomery curve.

The following is a composition map $\hat{\eta}$ that is constructed with all of the maps used in the path between squared Kummer surface and supersingular Montgomery curve.

$$\hat{\eta} : \mathcal{K}^{Sqr} \to E_\alpha/\mathbb{F}_{p^2}, \quad P \mapsto (\psi^{-1} \circ +_{\tilde{E}_\alpha} \circ \hat{\rho} \circ \zeta^{-1} \circ \hat{\Psi})(P).$$

B.3.3 K4SIDH Functions

Crucial Kummer surface arithmetic functions are given in this section. Figure of the doubling routine and a brief explanation of the figure are given below for the purpose of illustration. The figure is taken from [2].

Hadamard Function

In the Hadamard function, from the input point $(x : y : z : t)$, we try to derive the point $(x + y + z + t : x + y - z - t : x - y + z - t : x - y - z + t)$ as the output. In the end of the Hadamard operation, modular reduction function *kummer_RDC* is called to avoid exceeding 751-bit limit. With using the AVX2 instruction set, this procedure can be done as the following.

```
// Function for Hadamard operation. Input: [x,y,z,t], Output: [x+y+z+t,x+y-z-t,x-y+z-t,x-y-z+t]
void kummer_HDM(vec *RES, vec *P) {
  vec P2P1P4P3[32], P3P4P1P2[32], P4P3P2P1[32], P2P1P4P1[32], P2P1P1P1[32], P3P3P2P2[32], P4P4P1P2[32],
    P4P4P4P3[32], tempvec1[32], tempvec2[32], tempvec3[32], tempvec4[32], tempvec5[32], tempvec6[63], tmp;
  int i;

  for(i = 0; i < 32; i++) {
    tempvec3[i] = vmask0; tempvec4[i] = vmask0; // tempvec3 = 0, tempvec4 = 0
  }

  for(i = 0; i < 32; i++) {
    P3P4P1P2[i] = PER4x(P[i], _MM_SHUFFLE(1,0,3,2)); // P3P4P1P2 = [z,t,x,y]
    P4P3P2P1[i] = SHF4x(P3P4P1P2[i], _MM_SHUFFLE(1,0,3,2)); // P4P3P2P1 = [t,z,y,x]
    P2P1P4P3[i] = SHF4x(P[i], _MM_SHUFFLE(1,0,3,2)); // P2P1P4P3 = [y,x,t,z]
    P2P1P4P1[i] = BLD4x(P2P1P4P3[i], P4P3P2P1[i], _MM_SHUFFLE(3,0,0,0)); // P2P1P4P1 = [y,x,t,x]
    P2P1P1P1[i] = BLD4x(P2P1P4P1[i], P3P4P1P2[i], _MM_SHUFFLE(0,3,0,0)); // P2P1P1P1 = [y,x,x,x]
    tempvec1[i] = ADD4x(P[i], P2P1P1P1[i]);  // tempvec1 = [x+y,y+x,z+x,t+x]
    P3P3P2P2[i] = BLD4x(P3P4P1P2[i], P4P3P2P1[i], _MM_SHUFFLE(0,3,3,0)); // P3P3P2P2 = [z,z,y,y]
    P4P4P1P2[i] = BLD4x(P3P4P1P2[i], P4P3P2P1[i], _MM_SHUFFLE(0,0,0,3)); // P4P4P1P2 = [t,t,x,y]
    P4P4P4P3[i] = BLD4x(P4P4P1P2[i], P2P1P4P3[i], _MM_SHUFFLE(3,3,0,0)); // P4P4P4P3 = [t,t,t,z]
    tempvec2[i] = ADD4x(P3P3P2P2[i], P4P4P4P3[i]); // tempvec2 = [z+t,z+t,y+t,y+z]
    tempvec3[i] = BLD4x(tempvec2[i], tempvec3[i], _MM_SHUFFLE(0,0,0,3)); // tempvec3 = [0,z+t,y+t,y+z]
    tempvec4[i] = BLD4x(tempvec2[i], tempvec4[i], _MM_SHUFFLE(3,3,3,0)); // tempvec4 = [z+t,0,0,0]
    tempvec4[i] = ADD4x(tempvec1[i], tempvec4[i]); // tempvec4 = [x+y+z+t,y+x,z+x,t+x]
  }

  for (i = 0; i < 31; i++) {
    tmp = SHR4x(tempvec4[i], 24);
    tempvec4[i] = AND4x(tempvec4[i], vmask24);
    tempvec4[i+1] = ADD4x(tempvec4[i+1], tmp); // clear tempvec4 to 24-bit representation
    tmp = SHR4x(tempvec3[i], 24);
    tempvec3[i] = AND4x(tempvec3[i], vmask24);
    tempvec3[i+1] = ADD4x(tempvec3[i+1], tmp); // clear tempvec3 to 24-bit representation
  }

  for (i = 0; i < 32; i++) {
    tempvec5[i] = SUB4x(tempvec4[i], tempvec3[i]); // tempvec5 = [x+y+z+t,y+x-z-t,z+x-y-t,t+x-y-z]
    tempvec5[i] = ADD4x(tempvec5[i], MM[i]); // tempvec5 = [x+y+z+t+2p,y+x-z-t+2p,z+x-y-t+2p,t+x-y-z+2p]
  }

  for (i = 0; i < 31; i++) { // adjust tempvec5 to make every word positive
    tempvec5[i] = ADD4x(tempvec5[i], vmaskB);
    tempvec5[i+1] = SUB4x(tempvec5[i+1], vmask1);
  }

  for(i = 00; i < 32; i++) tempvec6[i] = tempvec5[i]; // copy tempvec5 to tempvec6
  for(i = 32; i < 63; i++) tempvec6[i] = vmask0; // clear upper words of tempvec6

  kummer_RDC(RES,tempvec6); // reduce the result to fit in 751-bit
}
```

Code B.3: Hadamard Function

Note that for performing the subtraction, we first subtract the subtrahend from a precomputed number. This number is created by taking the double of every word of the prime $2^{372}3^{239}-1$. Then we add the difference to the minuend. Thus, we make sure that every limb of the result stays positive after the subtraction.

Multiplication Function

Because of the choice of number of limbs for an element of a point, unfortunately, the multiplication sequence is quite long in K4SIDH. We perform a 32×32 word Comba method based multiplication. The code is generated by the function *mulMxN_generator* from **comba_generator.c**. This function simply prints the 4 way 256-bit multiplication and addition instructions in a multiplication optimization technique called Comba. It is a generic function and it also has the capability of multiplying two numbers which can have different number of limbs (MxN). The code can be seen below.

```
1  void mulMxN_generator(int DIGITSM, int DIGITSN) {
2    int i, k, l, ct = 0, r = 0, arr[2][2], tmp;
3    if(DIGITSM < DIGITSN){
4      tmp = DIGITSN; DIGITSN = DIGITSM; DIGITSM = tmp;
5    }
6    int DIFF = DIGITSM-DIGITSN;
7    printf("\n%d x %d COMBA based 4-way multiplication:\n\n",DIGITSM,DIGITSN);
8    printf("R[0] = MUL4x(P[0],Q[0]);\n\n");
9    for(i = 1; i < DIGITSN; i++){
10     for(k = 0; k <= i; k++){
11       printf("P%dQ%d = MUL4x(P[%d],Q[%d]);\n",i-k,k,i-k,k);
12       if(k == 1){
13         arr[0][0] = i-k+1;  arr[0][1] = k-1;
14         arr[1][0] = i-k;  arr[1][1] = k;
15       }
16     }
17     if(i == 1){
18       printf("R[1] = ADD4x(P%dQ%d,P%dQ%d);\n",arr[0][0],arr[0][1],arr[1][0],arr[1][1]);
19     }else{
20       printf("tmp%d = ADD4x(P%dQ%d,P%dQ%d);\n",ct,arr[0][0],arr[0][1],arr[1][0],arr[1][1]); ct++;
21     }
22     for(k = 2; k <= i; k++){
23       if(k == i){
24         printf("R[%d] = ADD4x(tmp%d,P%dQ%d);\n",r+2,ct-1,i-k,k); r++;
25       }else{
26         printf("tmp%d = ADD4x(tmp%d,P%dQ%d);\n",ct,ct-1,i-k,k);
27       }
28       ct++;
29     }
30     ct = 0; printf("\n");
31   }
32   for(i = DIGITSN; i < DIGITSM; i++){
33     for(k = 0; k <= DIGITSN-1; k++){
34       printf("P%dQ%d = MUL4x(P[%d],Q[%d]);\n",i-k,k,i-k,k);
35       if(k == 1){
36         arr[0][0] = i-k+1;  arr[0][1] = k-1;
37         arr[1][0] = i-k;  arr[1][1] = k;
38       }
39     }
40     if(i == 1){
41       printf("R[1] = ADD4x(P%dQ%d,P%dQ%d);\n",arr[0][0],arr[0][1],arr[1][0],arr[1][1]);
42     }else{
43       printf("tmp%d = ADD4x(P%dQ%d,P%dQ%d);\n",ct,arr[0][0],arr[0][1],arr[1][0],arr[1][1]); ct++;
44     }
45     for(k = 2; k <= DIGITSN-1; k++){
46       if(k == DIGITSN-1){
47         printf("R[%d] = ADD4x(tmp%d,P%dQ%d);\n",r+2,ct-1,i-k,k); r++;
```

```
48        }else{
49          printf("tmp%d = ADD4x(tmp%d,P%dQ%d);\n",ct,ct-1,i-k,k);
50        }
51        ct++;
52      }
53      ct = 0; printf("\n");
54    }
55    for(i = 1; i < DIGITSN-1; i++){
56      for(k = DIGITSM-1, l = i; k >= DIFF+i; k--, l++){
57        printf("P%dQ%d = MUL4x(P[%d],Q[%d]);\n",k,l,k,l);
58        if(k == DIGITSM-2){
59          arr[0][0] = k+1;  arr[0][1] = l-1;
60          arr[1][0] = k;    arr[1][1] = l;
61        }
62      }
63      if(i == DIGITSN-2){
64        printf("R[%d] = ADD4x(P%dQ%d,P%dQ%d);\n",r+2,arr[0][0],arr[0][1],arr[1][0],arr[1][1]); r++;
65      }else{
66        printf("tmp%d = ADD4x(P%dQ%d,P%dQ%d);\n",ct,arr[0][0],arr[0][1],arr[1][0],arr[1][1]); ct++;
67      }
68      for(k = DIGITSM-3, l = i; k >= DIFF+i; k--, l++){
69        if(k == DIFF+i){
70          printf("R[%d] = ADD4x(tmp%d,P%dQ%d);\n",r+2,ct-1,k,l+2); r++;
71        }else{
72          printf("tmp%d = ADD4x(tmp%d,P%dQ%d);\n",ct,ct-1,k,l+2);
73        }
74        ct++;
75      }
76      ct = 0; printf("\n");
77    }
78    printf("R[%d] = MUL4x(P[%d],Q[%d]);\n",r+2,DIGITSM-1,DIGITSN-1);
79  }
```

Code B.4: Comba Based 4 way Generic Multiplication Generator

Squaring Function

Squaring is also done with the Comba multiplication technique in K4SIDH. The code of squaring is generated by the function *squ_generator* from **comba_generator.c**. Again, this function prints the 4 way 256-bit multiplication and addition instructions in Comba technique. However, it is optimized and is faster than multiplication because of the nature of the squaring. 32×32 word squaring is used in K4SIDH. The code is given below.

```
1   void squ_generator(int DIGITS) {
2     int i, k, l, ct = 0, r = 0, flag = 0, arr[2][2];
3     printf("\n%d x %d COMBA based 4-way squaring:\n\n",DIGITS,DIGITS);
4
5     for(k = 0; k < DIGITS-1; k++) {
6       printf("P%d_2 = ADD4x(P[%d],P[%d]);\n",k,k,k);
7     }
8     printf("\n");
9     printf("R[0] = MUL4x(P[0],P[0]);\n\n");
10    printf("R[1] = MUL4x(P[1],P0_2);\n\n");
11    for(i = 2; i < DIGITS; i++) {
12      for(k = 0; k <= i/2; k++) {
13        if(i-k == k){
14          printf("P%dP%d = MUL4x(P[%d],P[%d]);\n",i-k,k,i-k,k);
15        }else{
16          printf("P%dP%d = MUL4x(P[%d],P%d_2);\n",i-k,k,i-k,k);
17        }
18        if(k == 1){
19          arr[0][0] = i-k+1;  arr[0][1] = k-1;
20          arr[1][0] = i-k;  arr[1][1] = k;
21        }
22      }
23      if(i == 2){
24        printf("R[2] = ADD4x(P%dP%d,P%dP%d);\n",arr[0][0],arr[0][1],arr[1][0],arr[1][1]);
25      }else if(i == 3){
26        printf("R[3] = ADD4x(P%dP%d,P%dP%d);\n",arr[0][0],arr[0][1],arr[1][0],arr[1][1]);
27      }else{
```

```
28      printf("tmp%d = ADD4x(P%dP%d,P%dP%d);\n",ct,arr[0][0],arr[0][1],arr[1][0],arr[1][1]); ct++;
29    }for(k = 2; k <= i/2; k++) {
30      if(k == i/2){
31        printf("R[%d] = ADD4x(tmp%d,P%dP%d);\n",r+4,ct-1,i-k,k); r++;
32      }else{
33        printf("tmp%d = ADD4x(tmp%d,P%dP%d);\n",ct,ct-1,i-k,k);
34      }
35      ct++;
36    }
37    ct = 0; printf("\n");
38  }
39  for(i = 1; i < DIGITS-2; i++) {
40    for(k = DIGITS-1, l = i; k >= i; k--, l++){
41      if(k == DIGITS-2){
42        arr[0][0] = k+1;  arr[0][1] = l-1;
43        arr[1][0] = k;    arr[1][1] = l;
44      }
45      if(k == l){
46        printf("P%dP%d = MUL4x(P[%d],P[%d]);\n",k,l,k,l);
47        break;
48      }
49      else if(k == l+1){
50        printf("P%dP%d = MUL4x(P[%d],P%d_2);\n",k,l,k,l);
51        break;
52      }
53      else{
54        printf("P%dP%d = MUL4x(P[%d],P%d_2);\n",k,l,k,l);
55
56      }
57    }
58    if((i == DIGITS-3) | (i == DIGITS-4)){
59      printf("R[%d] = ADD4x(P%dP%d,P%dP%d);\n",r+4,arr[0][0],arr[0][1],arr[1][0],arr[1][1]); r++;
60      flag = 1;
61    }else{
62      printf("tmp%d = ADD4x(P%dP%d,P%dP%d);\n",ct,arr[0][0],arr[0][1],arr[1][0],arr[1][1]); ct++;
63    }
64    for(k = DIGITS-3, l = i; k >= i; k--, l++){
65      if(flag == 1){
66        break;
67      }else if((k == l+2) | (k == l+3)){
68        printf("R[%d] = ADD4x(tmp%d,P%dP%d);\n",r+4,ct-1,k,l+2); r++;
69        break;
70      }else{
71        printf("tmp%d = ADD4x(tmp%d,P%dP%d);\n",ct,ct-1,k,l+2);
72      }
73      ct++;
74    }
75    flag = 0; ct = 0; printf("\n");
76  }
77  printf("R[%d] = MUL4x(P[%d],P%d_2);\n\n",r+4,DIGITS-1,DIGITS-2); r++;
78  printf("R[%d] = MUL4x(P[%d],P[%d]);\n",r+4,DIGITS-1,DIGITS-1);
79 }
```

Code B.5: Comba Based 4 way Generic Squaring Generator

Modular Reduction Function

In field arithmetic, every element must be in the bound of the chosen prime. In K4SIDH, this prime is $p = 2^{372}3^{239} - 1$. The result of multiplication, squaring or Hadamard functions can exceed the 751-bit boundary ($\log_2 p = 750.81$). Therefore after these functions, the result must be reduced in order to meet the required qualification. For the reduction of a number, Barrett reduction [1] is used. Usage of Montgomery reduction [12] instead of Barrett reduction could decrease the cycle count greatly, yet we were lack of time for implementing Montgomery reduction.

Barrett reduction avoids division for the reduction and uses multiplication and shift

operations. We first choose a number $N = \lceil \log_2 p \rceil$ and use it to precompute a factor $C = 2^{2N}/p$. This precomputed factor is a number that fits to 32 24-bit words. Next, we multiply C with the number that we want to reduce, say it AB. This multiplication $(AB \times C)$ is a 63×32 word multiplication, which costs more than our usual 32×32 word multiplication. After the multiplication, we have the result in 94 24-bit words. We get rid of the first 62 words of the result. We need to extract a 751-bit number from the result. So, we cut 14-bit from the remaining 32 words of the result. Now, we have the 751-bit number we wanted to get (t) and we can multiply it with p. At last, if we subtract the result of the multiplication from AB, we get $R = AB - tp = AB \pmod p$. The modular reduction ends with making necessary adjustments on R. The result R is the 751-bit reduced version of the input AB. The code for the function is given below.

```
// Modular reduction for the SIDH prime 2^372*3^239-1. Barrett reduction technique is used in the function.
void kummer_RDC(vec *U, vec *R) {
  vec T[94], Z[63], W[63], tmp, ttt;
  int i;

  for(i = 0; i < 62; i++){ // clear R to 24-bit representation
    tmp = SHR4x(R[i], 24);
    R[i] = AND4x(R[i], vmask24);
    R[i+1] = ADD4x(R[i+1], tmp);
  }

  combaMUL63x32(T,R,C); // T = R*C

  for(i = 0; i < 93; i++){ // clear T to 24-bit representation
    tmp = SHR4x(T[i], 24);
    T[i] = AND4x(T[i], vmask24);
    T[i+1] = ADD4x(T[i+1], tmp);
  }

  T[62] = SHR4x(T[62],14); //2*751=1502, 24*62=1488, 1502-1488=14.

  for(i = 62; i < 93; i++){ // Cut 14-bit from the upper 32 words of T (t)
    ttt = SHR4x(T[i+1],14);
    tmp = AND4x(T[i+1],vmask14);
    tmp = SHL4x(tmp,10);
    T[i] = ADD4x(T[i],tmp); T[i+1] = ttt;
  }

  combaMUL32x32(Z,T+62,M); // Z = t*p

  for(i = 0; i < 62; i++){
    tmp = SHR4x(Z[i], 24);
    Z[i] = AND4x(Z[i], vmask24);
    Z[i+1] = ADD4x(Z[i+1], tmp); // clear Z to 24-bit representation
  }

  for(i = 0; i < 63; i++) W[i] = SUB4x(R[i], Z[i]); // W = R - t*p

  tmp = MUL4x(W[32],vmaskB); // if W32 == 1
  W[31] = ADD4x(W[31],tmp); // then W31 = W31 + 2^24

  for(i = 0; i < 31; i++){
    W[i] = ADD4x(W[i], vmaskB);
    W[i+1] = SUB4x(W[i+1],vmask1); // adjust W to make every word positive
    tmp = SHR4x(W[i], 24);
    W[i] = AND4x(W[i], vmask24);
    W[i+1] = ADD4x(W[i+1], tmp); // clear W to 24-bit representation
  }

  for(i = 0; i < 32; i++) U[i] = W[i]; // return W
}
```

Code B.6: Modular Reduction Function

Doubling Function

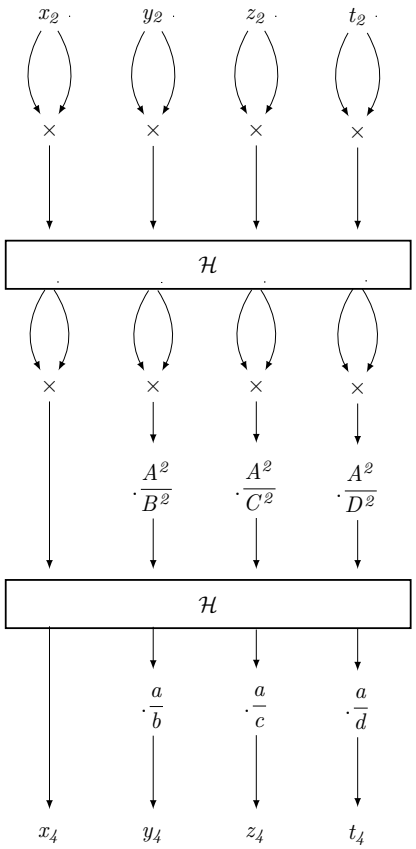

Figure B.2: Doubling Routine

In the doubling routine, first, squaring of a given point in projective 3 space which is represented as a tuple of four projective field elements $(x_2 : y_2 : z_2 : t_2)$ is performed. When the squaring is done, Hadamard (H) and another squaring operation are performed to the tuple. Next, the last three coordinates of the result $(H({x_2}^2 : {y_2}^2 : {z_2}^2 : {t_2}^2))^2$ are multiplied by Kummer surface constants $A^2/B^2, A^2/C^2$ and A^2/D^2, respectively. After performing another Hadamard sequence, again the last three coordinates of the result are multiplied by different Kummer surface constants $a/b, a/c$ and a/d, respectively. The whole procedure and the resulting tuple at the end of the doubling routine $(x_4 : y_4 : z_4 : t_4)$ can be seen in Figure B.2.

B.4 Testing Design

The accuracy of the implementation is verified with the help of a Magma script, written by Costello [5]. We slightly changed the original Magma script in order to make it compatible with a C language implementation. The comparison operation of the results taken from both K4SIDH and the script is done in main function.

When creating the back-and-forth maps Magma script, Magma built-in functions are used for checking whether a map works correctly or not. We also check that if a point used in the maps script, lies on the right algebraic structure. The validity of every map used when going between supersingular Montgomery curve and supersingular Kummer surface is checked separately.

Additionally, all of the fast Kummer surface arithmetic functions (Hadamard, inversion, multiplication, squaring) were tested separately before using them in the application. For testing the accuracy of multiplication and squaring (with modular reduction), separate Magma scripts were written. Multiplication is tested in **mul751.mag** and squaring is tested in **squ751.mag**. In both of the scripts, first, the operation is done with using the largest possible numbers, and then random numbers are used. After completing an operation, the result is compared with the result of Magma. All the results are chained. Only one erroneous result can make the last result false.

Appendix C

PRODUCT MANUAL

14.05.2019

Revision 1.0

Revision History

Revision	Date	Explanation
1.0	14.05.2019	Initial Product Manual

C.1 Introduction

The purpose of this product manual is to document the implementation, testing, installation and operation of K4SIDH as a software product. The application is implemented and tested as it is described in K4SIDH Design Specification Document, Revision 3.0 [20], satisfying the requirements in K4SIDH Requirements Specification Document, Revision 3.0 [21].

Implementation, testing and operation details are given in the following sections of this document.

C.2 Implementation

C.2.1 Source Code and Executable Organization

At first, we started the implementation with gathering and producing necessary codes of back-and-forth maps in Magma [3] environment. The chain of maps between a Montgomery curve and corresponding Kummer surface can be found in **maps.mag**. In the organization of this file, we first call the function *MonToKum*, then with the help of the returned variables from this function, we call the function *KumToMon*. This travel is equivalent evaluating an isogeny. After these operations, we return to the double of the point at the beginning.

The C implementation was written benefiting from [5]. For the multiplication, squaring and their modular reduction operations, first we wrote Magma scripts and made sure that these operations work error-free. Next, we set up the organization of the implementation. We have four different files that form K4SIDH which are:

- kummer_api.h
- kummer_consts.h
- kummer_isogeny.c
- main.c.

The declaration of the functions, library, variable and macro definitions are placed in **kummer_api.h**. The Kummer surface constants and precomputed points that are needed

for initiating the application take part in **kummer_consts.h**. Main function is placed in **main.c**. All fast Kummer surface arithmetic functions, type conversion functions, printing functions, auxiliary functions that are used in isogeny evaluation sequence and the main (2,2) isogeny evaluation loop can be found in **kummer_isogeny.c**.

C.2.2 Software Tools

For the creation of **maps.mag**, Magma Computational Algebra System was used. We also benefited from the Online Magma [15] occasionally. When writing Magma codes locally, GNOME text editor, Gedit, was used.

All C codes were written with the usage of Eclipse Integrated Development Environment. For the compiler, GCC version 5.5.0 and for the assembler, GNU assembler version 2.26.1 was used.

C.2.3 Hardware/Software Platform

All the project was done on a personal computer, in LINUX operating system, Ubuntu 16.04 LTS distribution environment.

The used CPU in the development was Intel® Core™ i7-7500U CPU with 2.70 GHz x 4. The CPU's codename is KabyLake and it has SIMD AVX2 instruction set support.

C.3 Testing

As mentioned in [20], testing in K4SIDH is done in many ways. Testing of 4 way Kummer surface arithmetic functions are performed separately, after each of the functions implementation phase is done. Three different scripts are used for testing purposes which are:

- kummer.mag
- mul751.mag
- squ751.mag.

kummer.mag is used in main function and is originally taken from [5]. Also, in main function, a speed test is performed with counting the total cycle count.

In addition, testing of **maps.mag** was made part by part, for every part of a map used in the construction of chain of maps that are used when going from a supersingular Montgomery curve to a supersingular Kummer surface, and vice versa.

C.4 Application Installation, Configuration, Operation

The script **maps.mag** can be run online or locally. For running the script online, Online Magma [15] can be used. One should simply paste all the script to the area in the site and click to "Submit" button. The result will appear at the below part of the site.

For running the script in local, Magma Computational Algebra System [3] must be installed. Owning the latest version of Magma may come in handy because some of the functions used in the script may not exist in the older versions of the Magma. After starting the Magma, the script can be run with the command below.

```
load "maps.mag";
```

In addition, the Magma scripts for testing 751-bit field multiplication and squaring, **mul751.mag** and **squ751.mag** can be executed with the same way as

```
load "mul751.mag";
load "squ751.mag";
```

The C application can be run with using the GCC compiler. The executable can run in any UNIX based system. From terminal, one can use the command below to compile the K4SIDH source code. The executable can be run directly with the second command.

```
gcc -m64 -mavx2 -Wall -O3 main.c kummer_isogeny.c -o K4SIDH.out
./K4SIDH.out
```

Appendix D

SOURCE CODE / SCRIPTS / EXECUTABLES IN CD / DVD

Bibliography

[1] P. Barrett. Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor. In *Advances in Cryptology — CRYPTO' 86*, pages 311–323. Springer Berlin Heidelberg, 1987.

[2] D. J. Bernstein, C. Chuengsatiansup, T. Lange, and P. Schwabe. Kummer Strikes Back: New DH Speed Records. In *Advances in Cryptology – ASIACRYPT 2014*, pages 317–337, 2014.

[3] W. Bosma, J. Cannon, and C. Playoust. The Magma algebra system. I. The user language. *J. Symbolic Comput.*, 24(3-4):235–265, 1997. Computational algebra and number theory (London, 1993).

[4] C. Costello. Computing Supersingular Isogenies on Kummer Surfaces. In *Advances in Cryptology – ASIACRYPT 2018*, pages 428–456. Springer International Publishing, 2018.

[5] C. Costello. Computing supersingular isogenies on kummer surfaces. `https://www.microsoft.com/en-us/download/details.aspx?id=57309`, Sep 2018, accessed 14 May, 2019.

[6] C. Costello, P. Longa, and M. Naehrig. Efficient Algorithms for Supersingular Isogeny Diffie-Hellman. In *Proceedings, Part I, of the 36th Annual International Cryptology Conference on Advances in Cryptology — CRYPTO 2016 - Volume 9814*, pages 572–601. Springer-Verlag, 2016.

[7] W. Diffie and M. Hellman. New Directions in Cryptography. *IEEE Trans. Inf. Theor.*, 22(6):644–654, Nov 1976.

[8] P. Gaudry. Fast genus 2 arithmetic based on Theta functions. *Journal of Mathematical Cryptology*, 1(3):243–265, 2007.

[9] H. Hisil and C. Costello. Jacobian Coordinates on Genus 2 Curves. In *Advances in Cryptology – ASIACRYPT 2014*, pages 338–357. Springer Berlin Heidelberg, 2014.

[10] H. Hisil and C. Costello. Jacobian Coordinates on Genus 2 Curves. `http://research.microsoft.com/en-us/downloads/37730278-3e37-47eb-91d1-cf889373677a`, May 2014, accessed 14 May, 2019.

[11] D. Jao and L. D. Feo. Towards Quantum-Resistant Cryptosystems from Supersingular Elliptic Curve Isogenies. In B. Yang, editor, *Post-Quantum Cryptography - 4th International Workshop, PQCrypto 2011, Taipei, Taiwan, November 29 - December 2, 2011. Proceedings*, volume 7071 of *Lecture Notes in Computer Science*, pages 19–34. Springer, 2011.

[12] P. L. Montgomery. Modular Multiplication Without Trial Division. *Mathematics of Computation - Math. Comput.*, 44:519–519, 04 1985.

[13] C. Lomont. Introduction to Intel advanced vector extensions. `https://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions`, May, 2011.

[14] D. Mumford. Tata lectures on theta II. volume 43 of Progr. Math. Birkhauser, 1984.

[15] Computational Algebra Group, University of Sydney. The Magma computational algebra system. `http://magma.maths.usyd.edu.au/calc`, accessed 14 May, 2019.

[16] Intel Corporation. Haswell new instructions. `https://software.intel.com/en-us/blogs/2011/06/13/haswell-new-instruction-descriptions-now-available`, accessed 14 May, 2019.

[17] Microsoft Research. SIDH library. `https://github.com/Microsoft/PQCrypto-SIDH`, Apr 2017, accessed 14 May, 2019.

[18] F. Richelot. Essai sur une methode generale pour determiner la valuer des integrales ultra-elliptiques,fondee sur des transformations remarquables des ce transcendantes. *CR Acad. Sc. Paris*, 2:622–627, 1836.

[19] J. Scholten. Weil restriction of an elliptic curve over a quadratic extension, 2003.

[20] M. Yassi. K4SIDH: A 4 way SIMD Implementation of SIDH Compatible Isogeny Evaluations on Kummer Surfaces DSD,2019-05-14-rev-3.0.pdf, 2019.

[21] M. Yassi. K4SIDH: A 4 way SIMD Implementation of SIDH Compatible Isogeny Evaluations on Kummer Surfaces RSD,2019-05-14-rev-3.0.pdf, 2019.