

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

High-Assurance, High-Speed Post-Quantum Cryptography in Safe Rust

Leonardo Moura



Mestrado em Engenharia Informática e Computação

Supervisor: Prof. Dr. Rui Maranhão

Cosupervisor: Prof. Dr. Manuel Barbosa

July 31, 2022

High-Assurance, High-Speed Post-Quantum Cryptography in Safe Rust

Leonardo Moura

Mestrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

President: Prof. Dr. José Campos

Referee: Dr. Alexandre Perez

Referee: Prof. Dr. Rui Maranhão

Referee: Prof. Dr. Manuel Barbosa

July 31, 2022

Abstract

A quantum computer running Shor’s algorithm can break, in polynomial time, public-key cryptosystems that are widely used in modern communication. As we know, there have been promising advances in the development of quantum computers and even though their efficient application is still some years away, their threat must be addressed today. This is why the cryptographic community is working on devising new quantum-resistant algorithms and updating existing standards so that cryptographic algorithms are safe for years to come.

While these algorithms’ mathematical foundations and theoretical security are essential, the other side of the coin can often be overlooked. Flawed implementations of cryptographic software can have catastrophic effects and be vulnerable to attacks that the underlying algorithm is not. The Jasmin framework aims to tackle these challenges by allowing programmers to develop highly efficient verified cryptographic implementations.

This dissertation explores the connection of Jasmin to the Rust programming language, which is swiftly gaining popularity for systems programming due to its efficiency and safety concepts. We investigate how Jasmin implementations can be called from Rust while maintaining the security constraints of both languages, ultimately achieving a degree of efficiency, safety, and usability that none of the languages could achieve by itself.

We then apply this knowledge to the problem of post-quantum cryptography by extending an existing Jasmin implementation of the Kyber algorithm to Rust, providing a highly efficient Kyber library (RjKyber) that can be safely called from Rust programs.

Keywords: Post-quantum Cryptography. Formal Verification. Rust. Jasmin. Kyber.

Resumo

Um computador quântico consegue, executando o algoritmo de Shor, quebrar, em tempo polinomial, os sistemas criptográficos de chave pública que são amplamente usados em comunicações modernas. Como sabemos, têm havido bons progressos no desenvolvimento de computadores quânticos e, mesmo que a sua aplicação eficiente ainda esteja a uns anos de distância, a ameaça que representam tem de ser tomada em conta hoje. Por esta razão, a comunidade de criptografia está a trabalhar no desenvolvimento de novos algoritmos pós-quânticos e na atualização das convenções existentes para que os algoritmos criptográficos em utilização sejam seguros a longo prazo.

Enquanto que os fundamentos matemáticos destes algoritmos assim como a sua segurança teórica são essenciais, o outro lado da moeda pode por vezes subestimado. Implementações defeituosas de software criptográfico podem ter efeitos catastróficos e ser vulneráveis a ataques aos quais o algoritmo subjacente não é. A framework Jasmin visa enfrentar estes desafios permitindo a programadores o desenvolvimento de implementações criptográficas verificadas altamente eficientes.

Esta dissertação explora a ligação entre Jasmin e a linguagem de programação Rust, que está rapidamente a ganhar popularidade para programação de sistemas devido à sua eficiência e conceitos de segurança. Neste trabalho investigamos como implementações em Jasmin podem ser chamadas a partir de Rust mantendo as restrições de segurança de ambas as linguagens, alcançando um grau de eficiência, segurança e usabilidade que nenhuma das linguagens conseguiria atingir por si própria.

Finalmente, aplicamos este conhecimento ao problema da criptografia pós-quântica estendendo a Rust uma implementação existente em Jasmin do algoritmo Kyber, fornecendo uma biblioteca altamente eficiente, a que chamamos RjKyber, que pode ser chamada com segurança a partir de programas de Rust.

Keywords: Criptografia Pós-quantica. Verificação formal. Rust. Jasmin. Kyber.

Acknowledgements

This dissertation is the culmination of many years of challenges that shaped me into a better person. Challenges that I would have not been able to overcome without the help of many people that in some moment, and for some reason, believed in me.

Firstly I want to thank my supervisor Rui for believing in me when I proposed a project outside of both our comfort zones, for giving me the opportunity to work on something I was truly passionate about and for teaching me the ways of academic research.

I also want to thank my cosupervisor Manuel for giving me guidance when I was taking my first steps in the field of cryptography, which is something that I had no contact in University but had an immense curiosity for. I'm also very grateful for all his tireless help whenever I wrote big emails, full of questions and problems that I had no idea how to solve.

To both my supervisors, I am grateful for all the hard challenges they presented me which, at first left me overwhelmed, but pushed my limits and helped me make a much better work than I was initially expecting. Finally, I thank them for since day one, giving me the freedom to work on what I thought was more interesting to me and above all, for believing in me when even I doubted myself.

I thank thank my parents for all the sacrifices they made to ensure me and my brother had access to the best education possible and became the best persons we could possibly be. They also were my main pillar during these 5 years, making all they could to allow me to work on my passion while supporting me through the hard moments. They always told me that I could achieve anything if I set my mind on it and worked hard. These words carried me through many adversities and are one of the main reasons I was able to reach the moment I am writing this.

I thank my brother for challenging me many times, for being an embodiment of hard work and for all the discussions we had on how to be our best self.

I thank my grandparents for always immediately helping me whenever I asked them for, and for giving me the motivation of telling me that they wanted to see the day I graduate.

I thank all of my friends and colleagues for teaching me a lot during these 5 years and for being by side in many adventures that helped shape the person I am today.

These words are not enough to express my gratitude towards all of you and many others that I could not mention here. But please know that I am forever grateful to all of you and that I would not have been able to reach this moment without without every single person that for some moment believed in me.

Thank you.

Leonardo Moura

July 2022

*”Quem que passar além do Bojador
Tem que passar além da dor.
Deus ao mar o perigo e o abismo deu,
Mas nele é que espelhou o céu.”*

Fernando Pessoa in *Mensagem*

Contents

1	Introduction	1
1.1	Public-key cryptography	1
1.2	The threat of quantum computers	1
1.3	Implementing cryptography	2
1.4	Goals	3
1.5	Our contribution	3
1.6	Document structure	3
2	Post-quantum Cryptography	5
2.1	Quantum Computers	5
2.2	Quantum bits	6
2.3	Quantum gates	7
2.3.1	The Pauli-X or NOT gate	7
2.3.2	The Hadamard gate	8
2.4	RSA and the integer factoring problem	8
2.4.1	The RSA trapdoor permutation	8
2.4.2	The integer factoring problem	9
2.5	Shor’s algorithm	9
2.6	Hard problems for quantum computers	10
2.7	Post-quantum cryptographic algorithms	11
3	High-assurance Cryptography	13
3.1	Challenges of cryptographic implementations	13
3.2	The Jasmin framework	14
3.2.1	The Jasmin language	14
3.2.2	Jasmin verification	16
3.2.3	The Jasmin compiler	18
3.3	Calling Jasmin routines in a high-level language	18
4	The Rust Programming Language	20
4.1	Rust basics	20
4.2	Ownership	21
4.3	Unsafe Rust	23
4.4	Safe interfaces for unsafe Rust	24
4.5	Jasminify	24
4.6	Formal verification in Rust	26
4.6.1	Functional correctness	26
4.6.2	Verification of Unsafe code	29

5	Creating a Safe Post-quantum Cryptography Library in Rust	30
5.1	CRYSTALS - Kyber	30
5.1.1	Learning with errors	31
5.1.2	Kyber internals	31
5.1.3	Kyber parameters	32
5.2	RjKyber	33
5.2.1	First iteration: Jasminify	33
5.2.2	Second iteration: cc	36
5.2.3	Third iteration: adding a high-level interface	37
5.3	Safety Discussion	38
6	Testing and Benchmarking	40
6.1	Testing	40
6.2	Benchmarking	41
6.2.1	Benchmarking the high-level interface	42
6.2.2	Comparing with pqc_kyber	43
7	Conclusion	45
7.1	Future work	46
	References	47
A	Jasmin Dockerfile	51
B	Jasmin bindings build script	52
C	Full code of RjKyber	53
D	Test Code	61
E	Benchmark code	69
F	Additional Benchmarks	77
F.1	RjKyber benchmarks for Kyber768 AVX2	77
F.2	RjKyber benchmarks for Kyber768 Reference	79
F.3	Comparison between RjKyber and pqc_kyber using Kyber768 AVX2	81

List of Figures

6.1	Key generation functions comparison	42
(a)	Rust interface	42
(b)	Jasmin Binding	42
6.2	Encapsulation functions comparison	43
(a)	Rust interface	43
(b)	Jasmin Binding	43
6.3	Decapsulation functions comparison	43
(a)	Rust interface	43
(b)	Jasmin Binding	43
6.4	Keygen functions comparison	43
(a)	pqc kyber	43
(b)	RjKyber	43
6.5	Encapsulation functions comparison	44
(a)	pqc kyber	44
(b)	RjKyber	44
6.6	Keygen functions comparison	44
(a)	pqc kyber	44
(b)	RjKyber	44
F.1	High level interface functions	77
(a)	Keygen	77
(b)	Encapsulate	77
(c)	Decapsulation	77
F.2	High level interface functions with automatic random number generation	77
(a)	Keygen	77
(b)	Encapsulate	77
F.3	Rust bindings for Jasmin IND-CCA KEM functions	77
(a)	Keygen	77
(b)	Encapsulate	77
(c)	Decapsulate	77
F.4	Rust bindings for Jasmin IND-CPA PKE functions	78
(a)	Keygen	78
(b)	Encaryption	78
(c)	Decapryption	78
F.5	Comparison between RjKyber high-level keygen and the corresponding Jasmin binding	78
(a)	Rust interface	78
(b)	Jasmin binding	78

F.6	Comparison between RjKyber high-level encapsulation and the corresponding Jasmin binding	78
	(a) Rust interface	78
	(b) Jasmin binding	78
F.7	Comparison between RjKyber high-level decapsulation and the corresponding Jasmin binding	78
	(a) Rust interface	78
	(b) Jasmin binding	78
F.8	High level interface functions	79
	(a) Keygen	79
	(b) Encapsulate	79
	(c) Decapsulation	79
F.9	High level interface functions with automatic random number generation	79
	(a) Keygen	79
	(b) Encapsulate	79
F.10	Rust bindings for Jasmin IND-CCA KEM functions	79
	(a) Keygen	79
	(b) Encapsulate	79
	(c) Decapsulate	79
F.11	Rust bindings for Jasmin IND-CPA PKE functions	79
	(a) Keygen	79
	(b) Encryption	79
	(c) Decryption	79
F.12	Comparison between RjKyber high-level keygen and the corresponding Jasmin binding	80
	(a) Rust interface	80
	(b) Jasmin binding	80
F.13	Comparison between RjKyber high-level encapsulation and the corresponding Jasmin binding	80
	(a) Rust interface	80
	(b) Jasmin binding	80
F.14	Comparison between RjKyber high-level decapsulation and the corresponding Jasmin binding	80
	(a) Rust interface	80
	(b) Jasmin binding	80
F.15	Comparison between RjKyber key generation and the corresponding pqc_kyber function	81
	(a) RjKyber	81
	(b) pqc_kyber	81
F.16	Comparison between RjKyber encapsulation and the corresponding pqc_kyber function	81
	(a) RjKyber	81
	(b) pqc_kyber	81
F.17	Comparison between RjKyber decapsulation and the corresponding pqc_kyber function	81
	(a) RjKyber	81
	(b) pqc_kyber	81

List of Tables

2.1	Threats on modern cryptography	11
2.2	NIST competition finalists [36]	12
4.1	Overview of Rust verification tools	29
5.1	Parameters for different Kyber security levels	33
6.1	Mean execution times of the Rust interface and Jasmin bindings	42
6.2	Mean execution times of pqc kyber and RjKyber	44

List of Listings

1	Jasmin program adding two parameters	15
2	Exporting a Jasmin programm	15
3	Jasmin compilation	15
4	Assembly code generated by Jasmin	16
5	Sum with memory accesses	17
6	Safety checker output	17
7	Calling Jasmin functions from C	19
8	Compiling and calling the test code	19
9	Variable mutability example	21
10	Drop trait example	21
11	Ownership transfer example	22
12	Parameter copy example	22
13	Borrow example	22
14	Mutable borrow example	23
15	Calling a C function in Rust	24
16	Rust stub of the Jasmin funciton	25
17	Rust stub of the Jasmin funciton	25
18	Creating the static library containing the Jasmin function	25
19	Test code for the Jasmin function	26
20	Compilation and execution of the test code	26
21	Example of well typed Rust program that panics	27
22	Prusti annotations of an example one time pad function	28
23	Rust stub for <code>Kyber.CCAKEM.Keygen</code>	34
24	Rust stub for <code>Kyber.CCAKEM.Keygen</code>	34
25	Cargo build file	35
26	New library functions	36
27	Build script using <code>cc</code> and conditional compilation	37
28	High-level interface for <code>Kyber.CCAKEM.Keygen</code>	38
29	Example of a Key exchange using Kyber	38
30	Testing that a decapsulation with the wrong secret key results in the wrong shared secret	41
31	Testing the one-sided authenticated key exchange protocol	41
32	Dockerfile that can be used to create a jasmin image in any manchine	51
33	Command to build the image	51
34	Command to run the image linking <code>/jasmin_code</code> on the local machine to <code>/jasmin_code</code> inside the container	51
35	Jasmin bindings build script	52
36	RjKyber <code>src/lib.rs</code> file	55

37	RjKyber <code>src/kem.rs</code> file	56
38	RjKyber <code>src/indcpa.rs</code> file	58
39	RjKyber <code>src/params.rs</code> file	59
40	RjKyber <code>build.rs</code> file	60
41	Tests for high-level interface	64
42	Tests for the IND-CPA functions	68
43	Benchmark code for RjKyber	75
44	Benchmark code to compare RjKyber and <code>pqc_kyber</code>	76

List of Algorithms

1	Kyber.CCAKEM.KeyGen	32
2	Kyber.CCAKEM.Encaps	32
3	Kyber.CCAKEM.Decaps	32

Abbreviations

API	Application Programming Interface
IFP	Integer Factoring Problem
KAT	Known Answer Test
KDF	Key Derivation Function
KEM	Key Encapsulation Mechanism
LWE	Learning With Errors
NIST	National Institute of Standards and Technology
PDF	Probability Density Function
PKE	Public Key Encryption
PRNG	Pseudorandom Number Generator
QPU	Quantum Processing Unit
RSA	Rivest-Shamir-Aldeman

Chapter 1

Introduction

With the ever-increasing amount of information circulating in modern communication networks, the essential role of cryptography can often be overlooked. Cryptography has evolved from the initial role of ciphering classified messages so that unintended parties cannot read them to a broader spectrum of applications. From banking systems to cryptocurrencies, from military communications to everyday internet access, cryptography is present, ensuring that the transmitted information is private and reliable.

1.1 Public-key cryptography

The introduction of the concept of *public-key encryption* (PKE) in 1976 [23] marked a significant paradigm change in cryptography. One year later, the proposition of RSA [40] was such a breakthrough that the algorithm is still the standard for public-key schemes today, more than 40 years later, protecting most modern communications.

Public-key schemes allow the usage of two different cryptographic keys for encryption and decryption, unlike symmetric-key schemes which use the same key for both operations. In public-key cryptography, each party has one private key, which is kept secret, and a public one shared with other parties. A message encrypted with the private key can only be decrypted with the corresponding public key and vice versa. This means that if Alice wants to send a message to Bob that only Bob can read, she will encrypt that message with Bob's public key. Then Bob would receive the message and decrypt it with his private key.

1.2 The threat of quantum computers

Recent advances in the development of quantum computers impose a big threat to established public-key schemes. By running Shor's algorithm [41], a quantum computer is able to break a

public-key system in polynomial time, greatly reducing the effectiveness of currently deployed algorithms, and in many of situations, render them useless.

Even though the creation of reliable quantum computers is still some years away, now is the time to prepare our systems to face that threat. There is an ongoing effort of the scientific community to design algorithms that are resistant to attacks performed by quantum computers in the form of a competition held by NIST [36]. These algorithms are known as post-quantum cryptographic algorithms. The goal of this competition is to standardize algorithms that will protect information, even if quantum computers become available to attackers.

These efforts go way beyond algorithm design. Algorithm implementation is also a critical aspect of cryptography, and it is a challenging topic by itself.

1.3 Implementing cryptography

Implementing cryptography presents specific challenges when compared to other types of software. Cryptographic implementations must comply with a set of strict requirements that are often more relaxed when it comes to conventional software [13]. The need for highly optimized code that at the same time is robust against a wide range of attacks makes cryptographic software very hard to implement correctly.

This type of software's high degree of criticality makes it so that implementations have to be tuned to near perfection, which is incredibly hard to do without the aid of computerized tools.

To address these challenges, the field of *Computer-aided Cryptography* was born [13]. It consists of merging formal approaches with the computational power of modern machines to verify the security of cryptographic implementations. For example, it is possible to verify that an algorithm is implemented according to specification without introducing unexpected vulnerabilities.

One of these frameworks is the Jasmin framework [6]. Jasmin is designed to help implement high-performance cryptographic primitives, verified to behave according to specification at the assembly level. Jasmin also includes a safety checker that is able to check if the implementation complies with several security constraints.

Jasmin is a handy framework for safe cryptographic implementations, however, due to the nature of using its own compiler, cryptographic primitives that are implemented in Jasmin live inside the Jasmin ecosystem and can only be called by outside programs through their binaries. This is not convenient for the overall practical usefulness of Jasmin because most developers usually use cryptographic implementations available as libraries for the programming language they are using.

It is already possible to call Jasmin binaries from Rust [43], however this approach delegates several safety responsibilities to the developer, ultimately contradicting the philosophies of both languages.

1.4 Goals

With all of this in mind, we consider that providing a high-speed, high-assurance post-quantum cryptographic library for Rust that uses Jasmin under the hood will be a significant step in streamlining the usability of this framework to a wider audience of developers and organizations, helping them address the modern challenges in cryptography and ultimately providing verified and highly efficient cryptographic code to Rust programmers.

We believe that the high-level constructs Rust offers to ease application building as well as the unique safety features of this language, in conjunction with the safety and speed that characterize Jasmin, make a good combination, mixing speed, safety and ease of use to achieve something that none of these tools could achieve by itself.

We also aim to explore the safety constraints of both languages and what each requires of the other in order to maintain its safety properties. We consider this to be a significant step forward when it comes to linking Rust and Jasmin, laying out the foundations for more safe integrations of Jasmin code in Rust applications.

So, this thesis aims to extend the safety of Jasmin to a higher-level programming language, namely Rust, and use that knowledge to implement a post-quantum cryptographic library, thus contributing to the preparation of society for the advent of quantum computers.

1.5 Our contribution

We propose RjKyber, a Rust cryptographic library that encapsulates a Jasmin implementation of the CRYSTALS - Kyber post-quantum key exchange mechanism and extends its use to Rust applications.

RjKyber can be seamlessly integrated into Rust programs using the cargo dependency manager to automatically compile and link the Jasmin code, completely black boxing the use of Jasmin to outside users. RjKyber also provides a high-level *application programming interface* (API) to abstract data structures and interaction with Jasmin functions, enabling access to arrays with complex specific lengths as Rust types and structs.

We also discuss the constraints both languages have to obey in order to maintain their safety when interacting with each other and we apply these principles to justify the safety of our library.

Finally, we intensively test and benchmark RjKyber to ensure its correct functioning and good performance, comparing it with another Rust implementation of the same algorithm and ultimately empirically demonstrate the advantages of using Jasmin for critical cryptographic routines.

1.6 Document structure

Chapter 2 further introduces the concept of post-quantum cryptography, explaining how quantum computers work and how they can break public-key cryptography. In order to do this, we also

introduce the basics of RSA. Finally, we present the current solutions for post-quantum cryptographic algorithms.

Chapter 3 addresses modern cryptographic implementations and why they are challenging. It also details the Jasmin framework and its inner workings, as well as how Jasmin programs can be called from a high-level language.

Chapter 4 introduces the Rust programming language and its main safety concepts, reasoning on why it is a good choice for a high-assurance post-quantum cryptographic library. It also presents Jasminify, a tool to ease the process of calling Jasmin functions from Rust. Finally, it discusses the current state of formal verification in Rust and its implications in cryptography.

Chapter 5 discusses how we implemented RjKyber, starting with an introduction to our post-quantum algorithm of choice, Kyber, and then going through every iteration of the library, justifying the design decisions behind each one as well as what we learned that motivated us to opt for next iteration as a better solution. Finally, it discusses the safety of this library, arguing about why can consider it safe.

Chapter 6 presents our approach to testing and benchmarking RjKyber and discusses the results we obtained as well as their implications.

Chapter 7 sums up this dissertation presenting our main conclusions and findings as well as future work that is to come.

Chapter 2

Post-quantum Cryptography

Quantum computers are a significant breakthrough in computation. It is believed that they are able to, in a reasonable amount of time, find solutions for problems that are known to be very hard for classical computers¹. One of these problems, is the basis of one of the most used cryptographic algorithms for public-key cryptosystems.

Even if the realization of efficient and usable quantum computers is still some years ahead of us, cryptographers around the world are already preparing and updating existing algorithms so that they are resistant to this technology. These algorithms are known as post-quantum cryptographic algorithms.

This chapter outlines the foundations of quantum computers and explains how they break current public-key cryptography. It also analyses state-of-the-art solutions for this problem that act as the basis for post-quantum algorithms. Finally, it presents some post-quantum cryptographic algorithms and outlines the current status of the standardization process for quantum-resistant algorithms.

2.1 Quantum Computers

Quantum computers introduce a new computation paradigm based on quantum mechanics. Quantum computers are based on *quantum superposition*. A particle is considered to be in a superposition if it assumes several states at the same time until its state is measured. In this moment, the particle assumes one of the several possible states and the superposition collapses. This is the foundation of the concept of *quantum bits*.

A common misconception regarding quantum computers is that they are much faster computers than the ones we use every day, capable of solving new problems otherwise impossible. This

¹This is known as *quantum supremacy*.

is however not true. Quantum computers will not replace classical computers or render them obsolete. They instead allow us to more efficiently compute solutions for certain problems that are very hard for classical computers.

This is why quantum computers should be seen as *QPUs* (Quantum Processing Units) rather than fully-fledged computation machines. They allow us to execute some instructions and algorithms more efficiently than regular computers and should be used as auxiliary computation units for these specific cases, much like how *GPUs* (Graphical Processing Units) work [27].

2.2 Quantum bits

Unlike classical bits, which exclusively assume either the values of 0 or 1, quantum bits (also known as *qubits*) assume superposition states before they are measured. This means that when a qubit is measured it will yield either a value of $|0\rangle$ or $|1\rangle$ ², but before that measurement, it is in a superposition state.

Qubits are defined by specifying this superposition state. Each qubit is characterized by two *amplitude* values: complex numbers which are used to calculate the probability of the qubit being in a given state upon a measurement. If we call these amplitude values α and β , then we can represent a qubit using the following notation:

$$\Psi = \alpha|0\rangle + \beta|1\rangle \quad (2.1)$$

In order to calculate the probability of a qubit being in a given state upon measurement, we calculate the square of the modulus of the corresponding amplitude. For example, if we consider the following qubit:

$$\Psi = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle \quad (2.2)$$

We can calculate it has an equal probability of 1/2 of it being measured in either state $|0\rangle$ or $|1\rangle$:

$$|\alpha|^2 = |\beta|^2 = \left(\sqrt{\frac{1}{\sqrt{2}^2}} \right)^2 = \frac{1}{2} \quad (2.3)$$

Amplitudes are an important concept because state probabilities are not enough to completely define a qubit. Due to the nature of complex numbers, qubits with different amplitudes can be observed in the same way, even if they have different properties.

The concept of *quantum entanglement* is what allows the existence of groups of qubits. Two particles are entangled when a measurement of the state of one, also reveals information on the state of the other, no matter the distance that separates them.

²This notation is known as *Dirac Notation* or *Bra-Ket Notation* and is used to represent quantum states. It is the standard notation to represent qubits

Groups of qubits are constructed by entangling all of them. This way, a measurement fully describes the state of the entire group. For example a quantum byte would have 256 possible states, which means that to define it we would need 256 amplitudes.

Generalizing, a group of n qubits is defined by 2^n amplitudes, such that the sum of all state probabilities is equal to one. This is why it is difficult to simulate quantum computers with classical computers: we need an exponential amount of memory in order to encode all possible states [10].

2.3 Quantum gates

Much like logic gates in classical computing, *quantum gates* apply operations to qubits in a quantum computer. Quantum algorithms consist of circuits of these quantum gates to execute operations on groups of qubits before measuring their final value. This is why quantum algorithms are often referred to as *quantum circuits*.

What greatly distinguishes quantum computers from their classical counterparts is the fact that n qubits can preform operations on 2^n numbers. This is the reason for the exponential speedup for some problems.

Unlike logic gates, quantum gates can be seen as matrices that are multiplied by a vector defining the amplitudes of the qubits. This means that quantum computers can be modeled by classical computers when using this matrix model: a classical computer would preform matrix multiplications instead of applying physical quantum gates. The problem with this approach is that the size of the matrices and vectors grows exponentially with the number of qubits considered, which means that as the number of qubits needed for an algorithm grows, a classical computer would need a very large amount of resources to preform the simulation.

We now present some simple examples of quantum gates.

2.3.1 The Pauli-X or NOT gate

The Pauli-X gate or NOT gate, is the quantum equivalent to the classical NOT gate: it flips qubits. It can however, preform this operation on a superposition state. Its matrix representation is the following:

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \quad (2.4)$$

For example, when applied to a $|0\rangle$ qubit:

$$X|0\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} = |1\rangle \quad (2.5)$$

2.3.2 The Hadamard gate

The *Hadamard* gate, denoted H , produces uniform superpositions on a qubit and unlike NOT, it has no classical equivalent. Its matrix representation is the following:

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \quad (2.6)$$

When applied to the $|0\rangle$ qubit:

$$H|0\rangle = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \end{bmatrix} = |\Psi\rangle \quad (2.7)$$

We obtain $|\Psi\rangle$, which as we have seen, has the same probability of being $|0\rangle$ or $|1\rangle$ when measured. We can now apply it back to Ψ to obtain the starting $|0\rangle$:

$$H|\Psi\rangle = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} = |0\rangle \quad (2.8)$$

As we can see, the Hadamard gate can be used to go from a deterministic state to a uniform superposition and when applied to that same superposition, back to the deterministic state. It is often used in quantum algorithms for this exact purpose: going from a deterministic state to a uniformly random one, without the loss of information.

2.4 RSA and the integer factoring problem

In order to understand how quantum computers break public-key cryptography, we need to understand the foundations of this cryptosystem and its most popular algorithm.

2.4.1 The RSA trapdoor permutation

RSA [40] is based on modular arithmetic. In order to encrypt a number x using RSA we perform the following operation:

$$y = x^e \mod n \quad (2.9)$$

Where e is the *public exponent*, n is modulus of the operation and y is the cyphered number. We can then decrypt y in the following way:

$$y^d \mod n = (x^e)^d \mod n = x^{ed} \mod n = x \quad (2.10)$$

All of this is based on the fact of finding d such that:

$$ed = 1 \mod \phi(n) \quad (2.11)$$

Where $\varphi(n)$ is *Euler's totient function*, in which if n is the product of prime numbers $p_1, \dots, p_m$:

$$\varphi(n) = (p_1 - 1) \times \dots \times (p_m - 1) \quad (2.12)$$

In the case of RSA we use two large prime numbers p and q to calculate n such that $pq = n$, which means that:

$$\varphi(n) = (p - 1) \cdot (q - 1) \quad (2.13)$$

Considering this, we can set e as a random prime lower than $\varphi(n)$ and d as the *greatest common divisor* of e and $\varphi(n)$.

So, in RSA, the public key consists of the public exponent e and the modulus, n while the private key consists of d . Numbers p and q as well as $\varphi(n)$ also need to be kept secret, so they are often considered part of the private key.

2.4.2 The integer factoring problem

From equation 2.13, and the way the private key is calculated, we can see that if an attacker is able to compute p and q from n , then he would be easily able to find the private key, thus breaking the cipher. This means that RSA's security relies on the fact that for a large number, it is very hard to find two prime numbers whose product equals said number. This is known as the *integer factoring problem* (IFP).

The most efficient known non-quantum algorithm to solve this problem is the *general number field sieve* [19], with a complexity of $\exp(1.91n^{1/3}(\log n^{2/3}))$, which is still exponential.

In practical terms, RSA is considered secure against attacks with modern classical computers when used with modulus of 2048 to 4096-bits [11].

2.5 Shor's algorithm

Shor's algorithm, proposed in 1994 [41], achieves an exponential speedup when solving the IFP.

The algorithm works by finding a period ω such that $f(x + \omega) = f(x)$. This solves a more general class of problems than the IFP, but can be applied to this specific problem.

In order to apply Shor's algorithm to the IFP, factoring N , we consider the following with a randomly selected value of a :

$$f(x) = a^x \mod N \quad (2.14)$$

This means that the algorithm will return ω such that:

$$a^x \mod N = a^{x+\omega} \mod N \quad (2.15)$$

Which in turn means that:

$$a^\omega \mod N = 1 \quad (2.16)$$

From equation 2.16, we can conclude that $a^\omega - 1$ is a multiple of N . By factoring this last expression we obtain:

$$a^\omega - 1 = (a^{\omega/2} - 1)(a^{\omega/2} + 1) \quad (2.17)$$

Finally, we can compute a factor p of N :

$$p = \gcd(a^{\omega/2} - 1, N) \quad (2.18)$$

If p is a nontrivial divisor (divisor other than 1) of N , then we have solved the problem. Otherwise we can repeat the algorithm with a different value of a .

It is important to note that only the step of finding ω is executed on a QPU. A classical CPU then retrieves this value and performs the other computations, namely the calculation of 2.18, since the Euclidean Algorithm can perform the computation in polynomial time [41].

2.6 Hard problems for quantum computers

As seen in section 2.4.2, the application of Shor's algorithm to the IFP, can be devastating to public-key cryptography, so it is necessary to design algorithms that rely on other problems that cannot be easily solved by either classical or quantum algorithms. This section briefly presents the main problems that are used to design post-quantum cryptographic schemes.

Hash functions. Quantum computers are not known to be able to efficiently find collisions for hash functions. This is why signature schemes can rely on the difficulty of finding hash collisions to achieve security. Hash based signature schemes use hash functions to generate signatures based on a secret key. These signatures can then be verified using public information. These systems, however, often produce very large signatures.

Systems of multivariate equations. Large systems of equations with multiple unknowns are known to be hard to solve, even for quantum computers, so they are also a possible basis for post-quantum algorithms. In this type of system, a message signed using a set of nonlinear equation systems. This signature can then be verified by solving another set of equations in which the public key consists. The main advantage of these systems is that they are fast and produce short signatures.

Error correcting codes. Error correcting codes are normally used when transmitting information over a noisy channel, however they can be applied to cryptography. The main idea is to introduce errors in a message, such that they require a special key to correct. An attacker would

not have sufficient information to correct the errors and would thus be unable to decrypt the message. This principle is the basis of the McEliece algorithm [34], developed in 1978.

Lattices. Lattices are mathematical constructs that consist on a set of points with a periodic structure in an n -dimensional space. Finding the closest vector in the lattice to an arbitrary point³ as a linear combination of a set of basis vectors is believed to be a hard problem both for classical and quantum computers. Lattice based schemes are constructed such that breaking them is equivalent to solving this problem.

Quantum computers are mainly a threat to current public-key systems, which rely on problems that can be broken using Shor's algorithm. However, Groover's quantum search algorithm [26] can be used to achieve a quadratic speedup in search problems. This speedup, despite not representing a threat against any particular set of algorithms, reduces the time complexity of searching for secret keys, which arises then need for bigger key lengths even for algorithms that are not directly affected by Shor's algorithm, which is the case of symmetric schemes like AES.

2.7 Post-quantum cryptographic algorithms

As described in previews sections, quantum computers pose a big threat to modern cryptographic ciphers and communication protocols. Table 2.1 summarises the main threats quantum computers impose in modern cryptography.

Table 2.1: Threats on modern cryptography

Algorithm type	Algorithm examples	Impact
Symmetric encryption	AES	Larger key lengths needed
Hash functions	SHA-2; SHA-3	Larger hash outputs needed
Public-key encryption	RSA; Elliptic Curves	No longer secure

Post-quantum cryptographic algorithms aim to solve the insecurity of public-key encryption when facing quantum computers. They leverage problems like the ones described in section 2.6 to achieve security in an environment where an attacker has access to an efficient quantum computer. This means that these algorithms must not be efficiently broken either by quantum or classical computers.

Recent advances in the development of quantum computers created the need for a transition to post-quantum cryptographic algorithms in modern applications and communication protocols, which forced NIST to start in 2016 a process for standardization of post-quantum cryptographic

³This point does not necessarily need to belong to the lattice.

algorithms [20]. This process is currently in the third and final round with standardization predicted for 2022 [36]⁴. Table 2.2 presents the finalist algorithms both for public-key encryption as well as digital signature schemes.

Table 2.2: NIST competition finalists [36]

Public-key Encryption		Digital Signatures	
Name	Type	Name	Type
KRYSTALS-KYBER	<i>Lattice-based</i>	KRYSTALS-DILITHIUM	<i>Lattice-based</i>
NTRU	<i>Lattice-based</i>	Falcon	<i>Lattice-based</i>
SABER	<i>Lattice-based</i>	Rainbow	<i>Multivariate</i>
Classic McEliece	<i>Code-based</i>		

These algorithms are currently facing great scrutiny by the cryptographic community in the form of studies involving cryptanalysis, performance benchmarking (namely hardware implementations), and side-channel resistant implementations, so that the most promising candidates can be standardised.

Due to the fact that most algorithms for both categories are lattice-based, at most one of these types of algorithms will be standardised for each category, despite the fact that algorithms based on structured lattices look like good candidates for widely used post-quantum algorithms. This decision protects cryptographic protocols against significant advances in cryptanalysis that may lead to an algorithm which, just as Shor's, might break systems based on a specific set of related problems.

⁴The competition advanced to the fourth round with the release of the status report for the third round [35] in the day of the submission of this document. This report includes the algorithms that are going to be standardized, which include our study case algorithm, Kyber.

Chapter 3

High-assurance Cryptography

Cryptographic primitives are often at the core of modern software, sometimes even being unnoticed by users. They cipher communications, authenticate communicating parties and ensure message integrity, among other critical uses. This is why cryptographic software is considered critical, to the point that a failure in this layer of a program can be catastrophic.

For this reason, it is very important to carefully design and implement cryptographic algorithms addressing specific challenges that can often be overlooked when implementing conventional applications. High-assurance cryptography uses formal methods and computer verification techniques to address the high security ceiling that modern cryptography entails.

This chapter introduces the main challenges of modern cryptographic implementations and presents the *Jasmin* framework [6], designed to address such challenges.

3.1 Challenges of cryptographic implementations

There are some concerns cryptographic software implementations needs to address. The first one is *efficiency*: cryptographic implementations must be extremely optimized in order to cause minimum overhead to the program that calls them [13]. Maximum performance is often a central focus of cryptographic code.

This performance focus often leads to heavily optimized code which can end up being quite different from the original algorithm. This is why it is important to ensure the *functional correctness* of these implementations, namely, ensure that the implementation behaves exactly like the algorithm specification without introducing any unexpected errors or vulnerabilities.

Another consequence of the great degree of optimization cryptographic code is subjected to, is on the topic of *memory safety*. The usage of low level languages for implementations, which give the programmer a larger responsibility over memory management, can lead to the memory related vulnerabilities that are often related to the use of these languages. So, it becomes important to enforce the implementation's memory safety through other tools.

Side-channel attacks are another big threat to cryptographic implementations: they enable an attacker to leak secrets by observing the behaviour of the implementation and extracting private information from that behaviour. For example, cache timing attacks [15] observe timings on the execution of the algorithm to extract keys. This vulnerability happens when the execution time of some part of the algorithm depends on some secret, allowing an attacker to deduce it by observing timings in the execution for some controlled input. These vulnerabilities arise concerns of *constant time* implementations: ensuring that the logical execution time of the program does not depend on secret information.

More recently, *spectre attacks* [31] exploit speculative execution in modern processors to leak secret information. This also introduces the need for *speculative constant time* [14] implementations that do not leak secrets through speculative execution channels.

3.2 The Jasmin framework

Jasmin [6] is a framework designed for high-assurance, high-efficiency cryptographic software. It uses formal verification techniques to ensure the functional correctness of the output code as well as memory safety and constant time security. The Jasmin language is designed with the *Assembly in the head* paradigm in mind, which allows the programmer to write code while predicting the structure of the assembly code that will be generated by the compiler, while providing some high-level constructs. This allows Jasmin to generate very efficient assembly while providing some ease in the writing and verification of Jasmin code.

The Jasmin compiler does not natively run some Linux distributions like Arch Linux due to some differences in libc. For these cases, and due to the fact that Jasmin's compiler compilation can be complicated, we preset in appendix A, a Dockerfile that can be used to ease the process by running Jasmin in a container.

3.2.1 The Jasmin language

The Jasmin programming language is designed to ease the implementation and verification of highly efficient cryptographic code while ensuring predictability and correctness of the generated assembly code.

Jasmin supports two types of idioms: one very close to assembly in which each instruction is directly mapped to a processor instruction, allowing for the manual tuning of certain sections of the implementation. Another idiom resembles high-level languages, supporting loops and function calls to allow for an easier implementation of less critical parts of the code as well as code modularity (in the case of function calls) which also helps with verification.

In order to ensure predictability and ease verification the programmer must explicitly declare all stack and register variables, including flags, and handle them specifically. The compiler then checks this behaviour in the verification step, making it easier to enforce security constraints.

Listing 1 illustrates a basic Jasmin program. In this example the function `add` is defined as `inline`, which means that the whole assembly code is inserted in place of the function during

```
1 inline
2 fn add(reg u64 a, reg u64 b) -> reg u64 {
3     reg u64 x;
4     reg bool cf;
5     x = a;
6     cf, x += b;
7     return x;
8 }
```

Listing 1: Jasmin program adding two parameters

compilation. This optimizes code execution as there are no instructions wasted on function calling protocols. However, this approach has the drawback of generating larger binaries. This is not a big concern for Jasmin programs as they are small, critical pieces of code and not entire applications.

The example function takes two 64-bit registers as parameters and returns another 64-bit register. It then defines another boolean register that will contain the carry flag resulting from the sum. The `x` variable is defined due to register allocation: the result is returned in register `rax`, so we need to move one of the parameters (which are allocated in `rdi` and `rsi` respectively) to this register. Creating the `x` variable will do exactly this. The function then uses the `sum` instruction to place the result of the sum of the two registers in register `x` which is returned.

Inlined functions only exist inside the Jasmin environment, so in order for this function to be used by an application, it needs to be exported. Listing 2 illustrates a way to do this.

```
1 export
2 fn add_export(reg u64 a, reg u64 b) -> reg u64 {
3     reg u64 c;
4     c = add(a,b);
5     return c;
6 }
```

Listing 2: Exporting a Jasmin program

In this example we declare the function with the `export` keyword indicating the function will be called by an external application. Like in listing 1, we create a new variable for the return value and set it to the result of the `add` function. We can compile Jasmin and store the output assembly code in an object file as shown in listing 3.

```
1 $ jasminc -o example.o example.jaz
```

Listing 3: Jasmin compilation

Listing 4 contains the result of the Jasmin compilation of this small program. As we can see, the entire function `add_export` corresponds to the code of `add`, which is inlined. The compiler automatically allocates registers in an optimal way, so the output assembly simply moves the value in `rdi` to `rax` and then preforms the sum. Besides the simple instructions used in this example, Jasmin also more sophisticated constructs such as loops, arrays and stack variables.

```

1      .att_syntax
2      .text
3      .p2align      5
4      .globl      _add_export
5      .globl      add_export
6  _add_export:
7  add_export:
8      movq      %rdi, %rax
9      addq      %rsi, %rax
10     ret

```

Listing 4: Assembly code generated by Jasmin

The semantics of the Jasmin language are deterministic, ensuring that for every initial state there is at most one final state. This is essential in ensuring the predictability of Jasmin programs. These semantics are formalized in the Coq proof assistant [1].

3.2.2 Jasmin verification

The Jasmin verification infrastructure consists of translating Jasmin programs to two annotated Dafny programs: one that encodes safety concepts and one that encodes constant time security assuming the safety of the first.

The first safety verification enforces the termination of the program without runtime errors. Things like invalid memory accesses and divisions by zero are all verified during this stage. To ease this process, Jasmin supports an annotation language to specify some properties of the program such as loop invariants. These annotations will then be passed to Dafny in the embedding of the program to the Dafny language.

For the verification of constant time security, Jasmin programs are annotated on control flow conditions and memory accesses, specifying that they must not depend on secrets. For this purpose, the special `Public` annotation is used which in some cases must be specified by the programmer. Conditional assignments are an exception to this rule, as they are compiled to constant-time instructions, thus not requiring any verification.

Memory safety analysis of the example in section 3.2.1 is trivial since the program only preforms operations on the values of registers and no memory accesses are preformed. Listing 5 presents a similar example which sums the values of two values contained in memory. In Jasmin, the syntax `[p+n]` accesses the value pointed by `p` with offset `n`.

```

1 export
2 fn add_mem(reg u64 a, reg u64 b) -> reg u64 {
3     reg u64 x,y,c;
4     x = [a+0];
5     c = x + [b+0];
6     return c;
7 }

```

Listing 5: Sum with memory accesses

Listing 6 shows the output of running the Jasmin safety checker on this example. The checker starts by stating that the example is memory safe, however this statement assumes the preconditions that follow it are met. In this case, both variables `a` and `b` must point to valid allocated memory with a minimum length of 8 bytes and must be aligned to 64 bits. The `Rel` section of the output reinforces these statements. If the program performed reads on memory locations based on a length passed as a parameter, the conditional length of the allocated memory in function of that parameter would be specified in this section.

```

1 $ jasminc -checksafety example2.jazz
2
3 Default checker parameters.
4
5 Analyzing function add_mem
6
7 *** No Safety Violation
8
9 Memory ranges:
10     mem_a: [0; 8]
11     mem_b: [0; 8]
12
13 * Rel:
14 {mem_a ≥ 0, inv_a ≥ 0, inv_b ≥ 0, mem_b ≥ 0, mem_b ≤ 8,
15     mem_a ≤ 8, inv_a ≤ 18446744073709551615,
16     inv_b ≤ 18446744073709551615}
17 mem_a ∈ [0; 8]
18 mem_b ∈ [0; 8]
19
20 * Alignment: a 64; b 64;

```

Listing 6: Safety checker output

3.2.3 The Jasmin compiler

The Jasmin compiler, formally verified in Coq, is designed to ensure that program written in the Jasmin programming language is accurately translated to assembly, maintaining all the security guarantees described in 3.2.2. Formally, the compiler ensures that upon a successful compilation, an execution of the compiled program corresponds to an execution of the source, which implies that if the source program is safe when a certain precondition set is met, then the compiled program will also be safe for those same preconditions. A similar reasoning is used to prove constant time security.

Because the compiler is mainly written in Coq with a small portion written in OCaml, then the Coq code is easily formalized within the theorem prover itself. Whenever some compiler pass calls the untrusted OCaml code, the result produced by this OCaml code is verified by a trusted Coq routine.

3.3 Calling Jasmin routines in a high-level language

The Jasmin compiler outputs assembly files which are assembled into binaries containing the exported functions. The reasoning behind this architecture relies on the fact that Jasmin is purposefully made for implementation of safety and performance critical functions, such as cryptographic algorithms, and not entire applications. Thus, the need to call Jasmin functions from other higher-level languages (which are more suitable for application development) arises. This allows broader pieces of software to benefit from Jasmin's security, verification and efficiency in critical cryptographic routines, while using conventional high-level constructs for the rest of the application which consists of non-critical code.

Jasmin's exported functions use the System V AMD64 calling convention [33], which means that parameters are passed via the `rdi`, `rsi`, `rdx`, `rcx`, `r8` and `r9` registers and additional parameters are passed via the stack¹. The `rax` and `rdx` registers is used for return values. Jasmin also supports AVX/AVX2 instructions, so the `xmmn` and `ymm` registers can also be used.

The most common language used to interface with Jasmin is C due to its relative ease when interacting with assembly code. Listing 7 presets a way for a C program to interact with example listing 2.

GCC is able to automatically link the assembly function and the C code, so compiling and running this code snippet is trivial. This is shown in listing 8.

The issue with using external functions in any language is the fact that the programmer must specify the interface of the function to be called. This must be done because due to the nature of assembly code, the compiler can not infer the interface of the function or its internal properties. This means that the programmer must ensure that the safety of the assembly code and that it is called with the correct interface. If low level languages are error-prone when it comes to memory management, assembly routines should be implemented with extreme care.

¹Jasmin does not currently support function arguments via the stack for exported functions

```
1  #include <stdio.h>
2
3  extern int add_export(int a, int b);
4
5  int main() {
6      int a = 2;
7      int b = 5;
8      int c = add_export(a,b);
9      printf("Result: %d\n",c);
10     return 0;
11 }
```

Listing 7: Calling Jasmin functions from C

```
1  $ gcc test.c example.s -o test.out
2
3  $ ./test.out
4  Result: 7
```

Listing 8: Compiling and calling the test code

In the case of Jasmin, the compiler ensures the safety of the output code, so in the case of C, the programmer must only ensure that the routine is called with the appropriate interface. For languages with larger safety constraints like Rust, there are more details that must be considered. Section 4.5 further details how this can be done.

Chapter 4

The Rust Programming Language

Rust is a programming language designed to merge the programming ease of high-level constructs with the higher power and control that characterizes low level languages, while assuring the necessary safety needed to build reliable code [30].

Usually, the power of being able to manually manipulate memory comes with the added responsibility of carefully ensuring the memory safety of that manipulation. This approach is obviously error-prone and memory corruption bugs can haunt even the most cautious low-level implementations. This is why there is usually a trade-off between a higher degree of control and efficiency in a low-level language and a greater ease of programming and memory safety in a high-level language. Rust aims to break this paradox by providing efficiency and power, while ensuring memory and thread safety together with some high-level abstractions. In order to achieve this, Rust introduces some unique programming concepts.

This chapter presents an overview of the Rust programming language, showing some of the unique concepts it provides and how they help in achieving greater security guarantees while providing fast execution times and some memory control. It also discusses unsafe rust and its implications. Finally, it presents the Jasminify tool [43], which can be used to call Jasmin code from Rust and explains how this is achieved. Finally, it discusses the topic of Formal verification in Rust, presenting existing methods as well as their strengths and gaps.

4.1 Rust basics

Rust is a *strong typed* language, which means that the type of every variable must be known at compile time. Some types can be inferred by the compiler, but most of the times they are declared explicitly by the programmer. Rust distinguishes from native data types, such as integers or arrays, whose size is known at compile time, from *reference* types which reference data allocated on the heap, like vectors.

One of the most important Rust concepts is the one of *mutability*. Variables are declared as immutable unless otherwise specified. This specific statement of variable mutability helps prevent many bugs where a value that's not supposed to be changed is mistakenly altered.

For example, in listing 9, only variable `y` can be changed.

```
1 let x = 6; //immutable variable
2 let mut y = 10; // mutable variable
```

Listing 9: Variable mutability example

4.2 Ownership

In Rust, we say that each variable *owns* a value and the language enforces that each value only has one owner at a time. Tracking the owner of each value allows the compiler to call the destructor of that value (implemented via the `Drop` trait) when the owner drops out of scope, relieving the programmer from the burden of memory management without the need for inefficient garbage collectors. Listing 10 illustrates this behaviour.

```
1 fn main() {
2     let x = String::from("hello");
3 }
4 // the destructor of x is called when it drops out of scope
```

Listing 10: Drop trait example

In Rust, function parameter passing for heap allocated structures is *shallow* (a pointer to the value is passed instead of copying the value to a new memory location) to increase efficiency. This means that by default, when a value is passed to a function, its ownership is moved to that function. So, considering ownership rules, the value is dropped when the function returns. Deep copies of these values can still be created using the `clone` trait. Like variables, function arguments are immutable by default. Listing 11 illustrates this.

For stack allocated types, such as integers, their size is known at compile time and they are automatically deleted from the stack when a function returns due to calling conventions. So, for these cases, the value that is used inside the function is a copy of the one that was passed, so this value is owned by the parameter variable. In practice, these values can be used after they are passed to a function. Listing 12 illustrates the this.

In order to prevent the behaviour in listing 11 Rust allows passing arguments using *references* (not to be confused with pointers). A reference (symbol `&`) states that the variable does not own the underlying value, but is instead referencing the original owner. When we pass a reference to

```
1 fn main() {
2     let x = String::from("hello");
3     print(x); //ownership of x moved to print
4     // x is no longer valid because it was dropped
5 }
6
7 fn print(s: String) {
8     println!("{}", s);
9 }
10 // s is dropped
```

Listing 11: Ownership transfer example

```
1 fn main() {
2     let x: u32 = 3;
3     print(x); // x is copied into print
4     // x can still be used
5 }
6
7 fn print(n: u32) {
8     println!("{}", n);
9 }
10 // n is dropped
```

Listing 12: Parameter copy example

a function, we say that this function *borrow*s the value. So, because ownership is not moved, we can still use the value after the borrow.

Listing 13 presents the same example as listing 11 using the reference concept.

```
1 fn main() {
2     let x = String::from("hello");
3     print(&x); //print borrows x
4     // x can be used normally
5 }
6
7 // &String represents a reference to a string
8 fn print(s: &String) {
9     println!("{}", s);
10 }
11 // s is not dropped
```

Listing 13: Borrow example

Just like variables, references can be declared as mutable too, but this can lead to some problems. A lot of *race conditions* are caused by the fact that two threads access the same data at the same time. Rust prevents this at compile time by enforcing that only one mutable reference to a value exists at a time. Furthermore, when a mutable reference to a value exists, no immutable references to that same value can exist as they do not expect that value to change while they are borrowing it. Multiple mutable references to the same value can however exist as none expects the value to change. Listing 14 shows the same example, using a mutable reference.

```
1 fn main() {  
2     let x = String::from("hello");  
3     push(&mut x);  
4     // x now has the value of "hello world"  
5     // x was modified and can be used  
6 }  
7  
8 // &mut String represents a mutable reference  
9 fn push(s: &mut String) {  
10     s.push_str(" world"); // s is modified  
11 }  
12 // s is not dropped
```

Listing 14: Mutable borrow example

4.3 Unsafe Rust

All of the previous concepts are known as *safe Rust*. However, programmers can instruct the Rust compiler to relax some of the constraints it asserts. For this purpose, the `unsafe` keyword is used.

Some algorithms benefit from a less conservative memory management, such as dereferencing a raw pointer, without being inherently unsafe upon careful implementation. Unsafe Rust allows the programmer to take responsibility from the code he writes while knowing the sections of code where this happens. This eases the process of debugging because the programmer will know which sections are error prone.

Unsafe Rust allows things like dereferencing pointers or calling functions written in other languages. Dereferencing a pointer can allow the existence of several mutable references to the same value, for example, while calling external functions is considered unsafe because the compiler cannot guarantee the safety of these functions, since other languages do not enforce Rust's guarantees and there is no way for the compiler to check if other code follows Rust's safety rules.

Listing 15 illustrates how unsafe Rust can be used to call C code.

```
1 extern "C" {  
2     fn some_c_function(n: i32) -> i32;  
3 }  
4  
5 fn main() {  
6     let n = 2;  
7     unsafe {  
8         let i = some_c_function(n);  
9     }  
10 }
```

Listing 15: Calling a C function in Rust

4.4 Safe interfaces for unsafe Rust

The use of the `unsafe` keyword does not necessarily mean that code is unsafe and will always lead to inconsistent states. In fact, the keyword is used all throughout the Rust standard library in implementations used regularly by Rust programmers [8]. This happens because, as stated previously, there are algorithms in which a more direct memory management is necessary to increase efficiency.

These implementations often use runtime assertions to ensure the program is in a consistent state when entering an unsafe block in order to prevent undefined behaviour. They are actually safe and can be trusted despite the fact the compiler is not able to ensure the safety of some blocks they use.

Whenever we use `Mutex`, which is implemented using unsafe code, we don't need to use the `unsafe` keyword every time we either lock or unlock it, despite it being a mutable reference shared by several threads. This happens because `Mutex` provides a safe programming interface despite the unsafe code running under the hood. In fact, the `Mutex` class was formally proved in [28] to be safe according to Rust's constraints.

So, when these implementations are used through their respective APIS, the end user can assume their safety, and the compiler can guarantee their safe use, despite the fact that internally they contain `unsafe` blocks.

4.5 Jasminify

Jasminify [43] is a tool that eases and automates some a part of the process of calling Jasmin routines from Rust without using the `unsafe` keyword. It automatically swaps the object file generated by the compilation of a Rust library with the binary that results from the assembly of the code compiled by Jasmin, containing the desired implementation. Lets consider the example in listing 2 and see the process *Jasminify* preforms under the hood so that it can be called from

Rust. Listing 16 shows the first step of this process which is to create an empty stub of the Jasmin function in Rust.

```
1  #[no_mangle]
2  pub fn add_export(a: u64, b: u64) -> u64 {
3      return 0;
4  }
```

Listing 16: Rust stub of the Jasmin function

The `#[no_mangle]` annotation must be specified to prevent the Rust compiler from *mangling*¹ the name of the function. Now, both the stub and the Jasmin assembly can be compiled. This is shown in listing 17.

```
1  $ rustc --crate-type=rlib add_export.rs
2
3  $ gcc -c example.s -o example.o
```

Listing 17: Rust stub of the Jasmin function

Now we can extract the `rlib` file and pack its metadata with the Jasmin object file. Listing 18 shows how this done.

```
1  $ ar -x libadd_export.rlib && rm add_export.add_export.*
2
3  $ rm libadd_export.rlib
4
5  $ ar -crs libadd_export.rlib example.o lib.rmeta
```

Listing 18: Creating the static library containing the Jasmin function

Now we can call the Jasmin code from a Rust application. In order to do this, we need to import the static library as an `extern crate`. Listing 19 shows an example test code using the Jasmin function we just compiled. It is important to notice that `add_export` is called without using the `unsafe` keyword.

Listing 20 shows how to compile the test code and run it. The `-L` flag must be used during compilation to inform the Rust compiler of where to look for the external library.

Jasminify automates this process. A programmer can annotate a Rust function with the special `//Jasmin` annotation and call `jasminify generate` on that code. The tool will automatically generate a stub of Jasmin code for that function with consistent parameters to the ones used by the

¹Mangling consists of changing the function name during compilation so that it contains useful information for the compiler.

```
1 extern crate add_export;
2 use add_export::add_export;
3
4 fn main() {
5     let a: u64 = 5;
6     let b: u64 = 2;
7     let c = add_export(a,b);
8     println!("Result: {}",c);
9 }
```

Listing 19: Test code for the Jasmin function

```
1 $ rustc test.rs -L .
2
3 $ ./test
4 Result: 7
```

Listing 20: Compilation and execution of the test code

Rust counterpart. After the desired function is implemented in Jasmin, `jasminify build` calls the Jasmin compiler and preforms the process specified previously so that the programmer only needs to preform the step in listing 20.

However, there are some gaps in security guarantees when using Jasminify, which will be detailed later in this document. The root of this problem is the fact that there is no formal verification and enforcement of the security properties of both languages before the compilation process. [ref]

4.6 Formal verification in Rust

As we have seen, Rust provides many features that can be used to write more reliable code. This is why it is being adopted for many projects with a focus on safety and reliability like operating systems development [5, 17]. Likewise, there is rising interest in using Rust to write formally verified software, which is often used in critical systems. We can divide Rust formalization projects into two main groups: *functional correctness* and *verification of unsafe code*.

4.6.1 Functional correctness

The Rust type system ensures that well typed Rust programs do not contain memory safety bugs, however these conditions are not sufficient to guarantee that a program will not exhibit undefined behaviour. Panics and crashes can still occur as well violations of runtime assertions. The example in listing 21 illustrates such code.

It may seem obvious to a human that this program will crash as it attempts a division by zero. However the Rust compiler is not able to infer that information. This program indeed compiles and

```
1 fn div(a: u32, b:u32) -> u32 {  
2     a/b  
3 }  
4  
5 fn main() {  
6     div(2,0);  
7 }
```

Listing 21: Example of well typed Rust program that panics

only panics at runtime. For critical software these runtime panics can have serious consequences so it is important for programmers to detect them statically.

Usually, formal verification of software uses computerized tools to logically prove that a piece of code is functionally correct and has no unexpected bugs in its implementation. In practice these tools specify functions as Hoare triples: when a set of preconditions hold, the function guarantees termination ensuring that a set of postconditions are true. These specifications of conditions are also known *function contracts*.

The programmer specifies these preconditions and postconditions by annotating his code and the tool ensures that when that function is called, all preconditions hold. Furthermore, using SMT solvers and theorem provers, it also proves that upon a correct execution of the function, the postconditions also hold. In order to aid these poofs, programmers might also have to specify loop variants and invariants.

Because Rust is a relatively recent language and formalizing borrows is a very challenging topic (such that different tools have different ways to do it), there is no de facto standard verification tools. Creusot [22] and Prusti [9] work using the contract process described above.

Creusot works by translating Rust code to why3 [24] as well as function contracts, written by programmers as annotations. Then a programmer can run why3 on the generated file and see if it is able to prove its correctness. Creusot uses a novel approach to borrows, which leads to shorter verification times.

Prusti translates Rust code to Viper [37] and, like Creusot, proves program correctness using contracts specified by the programmer. Prusti works by proving that no `panic!()` instructions are called, which in practice leads to a smoother verification process for a programmer. The standard way of using Prusti is via its VSCode extension, which provides inline code diagnostics upon verification, easing the process of finding errors in a program. Whereas in Creusot, we need to attempt verification in why3 and check if it is able to prove correctness of the code, often not even identifying the part of the code that leads to the error. Actually when we introduce the code in listing 21 in Prusti, it automatically detects a possible division by zero in the `div` function which can lead to a panic. In order to solve this, we need to annotate it with `#[requires(b != 0)]` which in turn will detect the precondition violation when we call the function in line 6.

These tools can be applied to cryptographic code to enforce preconditions of critical routines.

For example we can consider the simple *one time pad* (OTP) cipher, which consists of xoring message m with key k and placing it buffer d . From the definition of this cipher it follows that m and k must have the same length. Equations 4.1 and 4.2 respectively present the pre and postconditions of this function.

$$\text{requires: } \text{len}(m) = \text{len}(k) = \text{len}(d) \quad (4.1)$$

$$\text{ensures: } \forall n \in \mathbb{N} (0 \leq i < \text{len}(m)) \implies (d_i = m_i \oplus k_i) \quad (4.2)$$

From these equations it is obvious to verify that in order for postcondition to hold, it is imperative that the precondition must also hold. If, for example, the length of k is lower than the length of m , the program will have an invalid memory access. If the implementation uses heap allocated structures like `Vec<>`, there is no way for the compiler to enforce correct parameter lengths since these are not known at compile time. These verification tools allow us to statically enforce program correctness for cases such as these.

Listing 22 presents a simple implementation of this example with Prusti annotations. The `body_invariant!` macro is used to specify the loop invariant.

```

1  extern crate creusot_contracts;
2  use creusot_contracts::*;
3
4  #[requires(inpt.len() == key.len() && inpt.len() == dest.len())]
5  #[ensures(forall (|i: usize | ( 0<= && i<inpt.len()) ==>
6      (dest[i] == inpt[i] ^ key[i])))]
7  fn otp(inpt: &Vec<u8>, key: &Vec<u8>, dest: &mut Vec<u8>) {
8      let mut i = 0;
9
10     while i < inpt.len() {
11         body_invariant!(forall( |n:usize| (0<=n && n<=i) ==>
12             (dest[b] == inpt[b] ^ key[b])));
13         dest[i] = inpt[i] + key[i];
14         i += 1;
15     }
16 }
```

Listing 22: Prusti annotations of an example one time pad function

The problem with listing 22 is that currently Prusti does not support verification using neither bitwise operators (such as `xor`) or vectors, so it cannot verify the code. Likewise, Cresusot also does not support the `xor` bitwise operator and vector verification is limited to functions. It cannot for example verify that the length of an allocated vector will be invalid when the function is called.

These limitations are serious for cryptography verification and will be discussed ahead. [ref]

4.6.2 Verification of Unsafe code

The methods described in section 4.6.1 are only able to verify limited subsets of safe Rust and are thus unable to deal with unsafe code. The verification of unsafe code is an important topic in Rust verification because, as stated in 4.3, unsafe code is used extensively in many Rust implementations, including the standard library.

However, verifying unsafe code presents many challenges and there is currently no tool like the ones described previously which can verify unsafe code. Nevertheless, there are projects which aim to contribute towards the safety verification of these implementations.

The RustBelt project [28] formalized a subset of the Rust language, which was called λ_{Rust} . It is closer to the Rust’s *Mid-level Representation* (MIR) than to Rust itself, removing a lot of syntax sugar introduced by Rust. This subset of the language was then modeled in the Iris theorem prover [29] and used to prove the safety of some unsafe implementations in Rust’s standard library like `Mutex` and `Cell`. Both these implementations use unsafe code to be able to create several mutable references to the same value, which something clearly forbidden by the compiler. Despite this unsafe usage, both implementations were proved to be safe when used through their respective interfaces.

CRUST [42] presets a more practical approach to verifying unsafe code. It is defined as a bounded verifier for Rust that aims to verify the memory safety of unsafe library code. It works by collecting exported API functions and generating a set of drivers for them. These drivers are essentially are tests covering the space of valid inputs for a function, testing memory safety violations using assertions. They are then translated to C by invoking the Rust compiler and collecting the *Abstract Syntax Tree* (AST) of the program. This C code is then verified by the CBMC [21] bounded model checker to verify that there is no possible input that leads to an assertion violation, thus proving the memory safety of a function.

CRUST was evaluated by checking a version of the vector library that was known to contain a bug and it was able to successfully find it. It was also used to test several other libraries, taking several hours to verify larger ones.

To conclude this section, we now preset in table 4.1 an overview of all projects discussed.

Table 4.1: Overview of Rust verification tools

Project	Purpose	Downsides
Prusti	functional correctness	no support for vectors or bitwise instructions
Creusot	functional correctness	no support for <code>xor</code> and limited vector support
Rustbelt	unsafe implementation formalization	no automated verification method
CRUST	memory safety verification	only verifies memory safety

Chapter 5

Creating a Safe Post-quantum Cryptography Library in Rust

Our goal is to create a verified Rust post-quantum cryptographic library that leverages Rust’s safety and high-level constructs as well as Jasmin’s efficiency and verification features to achieve a new degree of usability and safety for cryptographic implementations. This presents some challenges that are outlined in this chapter, namely the interaction between Rust and Jasmin while maintaining the safety of both languages.

In order to achieve this, we have chosen the CRYSTALS - Kyber post-quantum algorithm, which already has an in development implementation in Jasmin [7].

This chapter starts by introducing Kyber, so that we can understand some design decisions down the line as well as the overall functioning of the library. Then, we describe how the Jasmin implementation can be encapsulated in Rust, starting from a simpler method and building on it to achieve the final higher level library that fully abstracts the usage of the algorithm. Finally, we discuss the safety of this library and the details this safety involves.

5.1 CRYSTALS - Kyber

CRYSTALS - Kyber is a post-quantum cryptographic *Key Encapsulation Mechanism* (KEM) which is IND-CCA2 secure ¹. As seen in section 2.7, it was just announced that it will be standardized in NIST standardization effort for post-quantum cryptography. It will be used as our case study for the creation of a post-quantum cryptographic library in Rust, so it is briefly introduced in this section.

¹It provides indistinguishably under chosen-ciphertext attacks.

5.1.1 Learning with errors

Let's consider an integer $n \geq 1$, a prime p , vector $\mathbf{s}^2 \in \mathbb{Z}_p^n$, and a list of i equations with errors χ , where $\mathbf{a}_i$ are chosen uniformly from $\mathbb{Z}_p^n$ and $b_i \in \mathbb{Z}_p$:

$$\langle \mathbf{s}, \mathbf{a}_i \rangle \approx_{\chi} b_i \pmod{p} \quad (5.1)$$

If we consider that χ follows a probability distribution $\chi : \mathbb{Z}_p \rightarrow \mathbb{R}^+$ on $\mathbb{Z}_p$, then we can independently choose $e_i \in \mathbb{Z}_p$ according to χ and write the equations in 5.1 as:

$$b_i = \langle \mathbf{s}, \mathbf{a}_i \rangle + e_i \quad (5.2)$$

The problem of recovering $\mathbf{s}$ from these equations is known as $\text{LWE}_{p,\chi}$ (*Learning With Errors*) and, as demonstrated by Regev [39], there are no known efficient quantum or classical algorithms to solve it.

Kyber is based on a version of LWE, known as Module-LWE [32] which is in turn based in Ring-LWE. This specific instance of the problem was chosen because it generates an efficient cryptosystem with a somewhat reduced exploitable structure.

5.1.2 Kyber internals

As seen in 1.1, a PKE is defined by the set of three algorithms (KeyGen , Enc , Dec) which encrypt and decrypt a message using the keys that the key generation algorithm outputs.

Since its original version, Kyber [18] defines the `Kyber.CPAPKE` suite of PKE functions, which offer IND-CPA security³. These functions are parameterized by integers $n, k, q, \eta_1, \eta_2, d_u$ and d_v , which are detailed in 5.1.3, and work on messages with a fixed length of 32 bytes. Their current formulation can be consulted in the most recent specification of Kyber [12].

A KEM is defined by three probabilistic algorithms: KeyGen is the key generation algorithm which returns a pair (pk, sk) consisting of the public and secret key respectively. Encaps is the encapsulation algorithm which takes a public key pk as input and outputs a cipher text c as well as key K . Finally, the decapsulation algorithm, Decaps , takes the secret key sk and the same ciphertext c to output K .

The most basic protocol in which a KEM can be used is a simple *key exchange protocol*: Bob creates a pair of keys using Keygen and Alice uses Encaps with Bob's public key to generate a ciphertext and a shared secret. The, bob can use that same ciphertext with his secret key to decapsulate the same shared secret. Then Alice and Bob can communicate using a symmetric encryption scheme with a key derived from the shared secret. KEMs can also be used in more complex authenticated key exchange protocols, both authenticating a single party as well as both parties.

²We denote the inner product of vectors $\mathbf{a}$ and $\mathbf{b}$ as $\langle \mathbf{a}, \mathbf{b} \rangle$.

³It provides indistinguishability under chosen plaintext attacks

Kyber’s `Kyber.CCAKEM` functions are constructed by extending `Kyber.CPAPKE` using a Fujisaki–Okamoto transform [25]. Algorithms 1, 2 and 3 present Kyber’s KEM functions according to their most recent specification [12]. They use hash functions H and G and the KDF *Key Derivation Function*, which are defined in 5.1.3.

Algorithm 1 `Kyber.CCAKEM.KeyGen`

```

1: procedure KEYGEN
2:    $z \leftarrow B^{32}$ 
3:    $(pk, sk') := \text{Kyber.CPAPKE.KeyGen}()$ 
4:    $sk := (sk' || pk || H(pk) || z)$ 
5:   return  $(pk, sk)$ 
6: end procedure

```

Algorithm 2 `Kyber.CCAKEM.Encaps`

```

1: procedure ENCAPS( $pk$ )
2:    $m \leftarrow B^{32}$ 
3:    $m \leftarrow H(m)$ 
4:    $(\bar{K}, r) := G(m || H(pk))$ 
5:    $c := \text{Kyber.CPAPKE.Enc}(pk, m, r)$ 
6:    $K := \text{KDF}(\bar{K} || H(c))$ 
7:   return  $(c, K)$ 
8: end procedure

```

Algorithm 3 `Kyber.CCAKEM.Decaps`

```

1: procedure DECAPS( $c, sk$ )
2:    $pk := sk + 12 \cdot k \cdot n / 8$ 
3:    $h := sk + 24 \cdot k \cdot n / 8 + 32 \in B^{32}$ 
4:    $z := sk + 24 \cdot k \cdot n / 8 + 64$ 
5:    $m' := \text{Kyber.CPAPKE.Dec}(sk, c)$ 
6:    $(\bar{K}', r') := G(m' || h)$ 
7:    $c' := \text{Kyber.CPAPKE.Enc}(pk, m', r')$ 
8:   if  $c = c'$  then
9:     return  $K := \text{KDF}(\bar{K}' || H(c))$ 
10:  else
11:    return  $K := \text{KDF}(z || H(c))$ 
12:  end if
13: end procedure

```

5.1.3 Kyber parameters

Kyber can be parameterized with different values, defining different security settings. These are shown in table 5.1, where δ is the probability of failure in the decapsulation of valid ciphertext by `Kyber.CCAKEM`.

Table 5.1: Parameters for different Kyber security levels

Algorithm	n	k	q	η_1	η_2	(d_u, d_v)	δ	NIST Sec. Level
Kyber512	256	2	3329	3	2	(10,4)	2^{-139}	1
Kyber768	256	3	3329	2	2	(10,4)	2^{-164}	3
Kyber1024	256	4	3329	2	2	(11,5)	2^{-174}	5

The value of n represents the dimension of the ring Kyber operates on and reflects itself in the entropy of Kyber’s keys. In practice, this means that `Kyber.CPAPKE.Enc` will encapsulate messages with sizes of 32 bytes. The value of q is simply a small prime satisfying $n|(q-1)$ and is used internally by the algorithm. Parameter k defines the lattice dimension as multiple of n by factor k and is the parameter that primarily sets the security level of the algorithm. The remaining parameters are chosen to balance security and ciphertext size while keeping δ low. The H hash function is instantiated with SHA3-256 and G is instantiated with SHA3-512. The KDF used is SHAKE-256.

The default Kyber version is considered to be `Kyber768`, balancing security and speed. `Kyber512` offers faster execution times, but is considered to offer minimal security. `Kyber1024` however, is the version that offers the greatest security but has the drawback of lower execution times.

5.2 RjKyber

RjKyber is the name of library we created to encapsulate a Jasmin Kyber implementation [7] in Rust. This section goes through the different iterations of this process explaining the reasoning behind them as well as the drawbacks that motivated changes in design decisions.

5.2.1 First iteration: Jasminify

The first approach to creating a library for Kyber’s high-level functions ⁴ is to apply `Jasminify` to our case. However, `Jasminify`’s workflow is designed to translate Rust functions to Jasmin, assuming there is something resembling a Rust implementation of the code to be called from Jasmin. This approach is not inherently wrong, but is simply not useful for our case, which is the inverse: we have a Jasmin implementation that we want to create Rust bindings for. The Jasmin code exists before the Rust code and not the other way around like is assumed by `Jasminify`. We can nevertheless take inspiration from `Jasminify` and perform the steps explained in section 4.5 to create Jasmin bindings for our Jasmin library.

This means that we cannot directly apply `jasminify generate` and `jasminify build` to build our library. What we need to do instead is to manually run the commands described in section 4.5.

⁴We consider `Kyber.CCAKEM` as Kyber’s high-level functions as these are the ones that are supposed to be used by protocols and end users.

In order to do this, we firstly need to create the Rust functions stubs for the Jasmin counterparts. The stub for `Kyber.CCAKEM.Keygen` is shown in listing 23, where module `params` contains the same parameters used by the Jasmin implementation, encoded in Rust as `pub const PARAM: usize`. In order to simplify this first phase, all parameters are declared as mutable references, but this will be changed later.

```

1 mod params;
2
3 #[no_mangle]
4 pub fn crypto_kem_keypair_jazz(
5     pk: &mut [u8; params::KYBER_PUBLICKEYBYTES],
6     sk: &mut [u8; params::KYBER_SECRETKEYBYTES],
7     random_bytes: &mut [u8; 2*params::KYBER_SYMBYTES]
8 ) {}

```

Listing 23: Rust stub for `Kyber.CCAKEM.Keygen`

After performing the steps in section 4.5 using this stub, we are already able to call the Jasmin Kyber implementation from a Rust program. However, in modern projects, Rust libraries and dependencies are managed by `cargo` and it is not practical for a developer to manually compile and link the library whenever he wants to use it a project. So, the next step is to automate this process.

We start by creating a Rust library crate and use the Jasmin bindings created above as an `extern crate` dependency to that library. So, the library itself is responsible for compiling and linking the Jasmin code and a user only has to add the library as a dependency.

To to this, we include the assembly code resulting from the Jasmin compilation of Kyber's KEM functions (both reference and AVX2 versions) in the `src/libjkem/asm` directory. Then, we create the library itself, simply calling the Jasmin bindings. Listing 24 shows the corresponding library function to listing 23 in the `src/lib.rs` file.

```

1 extern crate jkem;
2 use jkem::crypto_kem_keypair_jazz;
3 pub mod libjkem;
4
5 pub fn keypair (
6     pk: &mut [u8; libjkem::params::KYBER_PUBLICKEYBYTES],
7     sk: &mut [u8; libjkem::params::KYBER_SECRETKEYBYTES],
8     random_bytes: &mut [u8; 2*libjkem::params::KYBER_SYMBYTES]
9 ) {
10     crypto_kem_keypair_jazz(pk, sk, random_bytes);
11 }

```

Listing 24: Rust stub for `Kyber.CCAKEM.Keygen`

The `jkem` crate is the static library containing the Jasmin bindings compiled by the `Jasminify` process. The `libjkem` module simply contains the parameters for Kyber. They are defined inside this module because this way we only need to define them once in the project (the same file is used both for the library and external users as well as the stubs). To do this, we need to place both the file containing the stubs (`jkem.rs`) and the parameters file inside the `src/libjem` directory. The need for this structure is another drawback of using this process.

Finally, we need to tell cargo how to compile the Jasmin bindings and link them. This is done using the special `build.rs` file, shown in listing 25.

```

1 use std::process::Command;
2 use std::env;
3
4 fn main() {
5     let out_dir = env::var("OUT_DIR").unwrap(); //get output dir
6     Command::new("sh")
7         .args(&["./src/libjkem/build.sh", "ref"]) //build script
8         .arg(&format!("{}", out_dir)) // output dir as argument
9         .output().unwrap();
10
11     println!("cargo:rustc-link-search={}", out_dir); //link location
12 }
```

Listing 25: Cargo build file

This build script simply calls the `src/libjkem/build.sh` script passing as parameter the location where the static library file should be placed, which is generated by cargo. Then we give the information to `rustc` to link external static libraries from that directory. The `build.sh` script is shown in appendix B and performs the `Jasminify` compilation of the Jasmin bindings, moving them to the directory specified in the second argument. The first argument specifies the version of the Kyber assembly file to compile, so that we can change between different Kyber versions simply by changing the cargo build script. This is done by changing the name of the assembly file in line 7 from `ref` to `avx2`, for example.

This approach has some drawbacks when we consider the way Rust expects dependencies to be compiled and linked. Firstly, the compilation process generates a confusing structure for the project, which limits its extensibility. And another important issue arises from the compilation script: it is written for Linux and assumes the necessary packages are installed to run the appropriate commands. It would not work in Windows, for example, limiting the amount of platforms in which the library can be used⁵.

⁵We could use `rustc`'s target feature to write a compilation script specifically for every platform, but this is not a very extensible approach

5.2.2 Second iteration: cc

We can solve the problems described in 5.2.1 by changing the compilation method. Cargo is able to automatically compile and link C and assembly code using the `cc` crate. This crate is able to, platform independently call the appropriate C compiler. Using this method, we can greatly simplify the structure of the library and its compilation process.

In order apply this new process, we use the external function calling method described in listing 15. This requires the use of the `unsafe` keyword, which as we have seen, does not impact in any way the actual safety of the code. The tradeoff of having greater extensibility and a more streamlined compilation process makes it worth the use of the keyword which has a purely syntactic impact. Listing 26 shows the new way of calling Jasmin functions in the library. Now there is no need to have function stubs, simplifying the overall organization of the code. This approach also allows us to easily add `Kyber.CPAPKE` functions.

```

1  extern "C" {
2      fn crypto_kem_keypair_jazz(
3          pk: &mut [u8; KYBER_PUBLICKEYBYTES],
4          sk: &mut [u8; KYBER_SECRETKEYBYTES],
5          rand: &mut [u8; 2*KYBER_SYMBYTES]
6      );
7  }
8
9  pub fn keygen(pk: &mut [u8; KYBER_PUBLICKEYBYTES],
10             sk: &mut [u8; KYBER_SECRETKEYBYTES],
11             random_bytes: &mut [u8; 2*KYBER_SYMBYTES]
12         ) {
13     unsafe {
14         crypto_kem_keypair_jazz(pk, sk, random_bytes);
15     }
16 }

```

Listing 26: New library functions

We can also use the concept of *features* to introduce conditional compilation to the project. This allows us to specify that a function is compiled when a certain feature flag is passed to the compiler so that a developer can change the Kyber version used without making any change to his code. Listing 27 shows the new, much simpler build script functions for the default Kyber768.

The main function simply calls the `compiler_asm` function with the appropriate Kyber version. According to the presence of the feature `ref`, the correct version of the function (using either reference or AVX2 implementations) will be compiled. The `KYBER_VERSION` variable is also defined according to the specified features. If no Kyber version is specified, Kyber768 AVX2 is used by default. This step gives us the foundations for the easy extension of RjKyber to Kyber512 and Kyber1024 which is something we intend to do.

```

1  #[cfg(not(any(feature = "kyber512", feature = "kyber1024")))]
2  const KYBER_VERSION: &str = "kyber768";
3
4  #[cfg(not(feature = "ref"))]
5  fn compile_asm(version: &str) {
6      cc::Build::new()
7          .file(&format!("src/asm/{}/jkem_avx2.s", version))
8          .file(&format!("src/asm/{}/jindcpa_avx2.s", version))
9          .compile("jkem");
10 }
11
12 fn main() {
13     compile_asm(KYBER_VERSION);
14 }

```

Listing 27: Build script using `cc` and conditional compilation

5.2.3 Third iteration: adding a high-level interface

To further improve the library we can create a high-level interface that adds a layer of abstraction to the algorithm. This simplifies the usage of Kyber by automating the creation the several arrays used by the algorithm and defining them as Rust types to simplify the code. It is much more practical to define a secret key array as `SecretKey` than `[u8; KYBER_SECRETKEYBYTES]`. Listing 28 presents the relevant code for the `Kyber.CCAKEM.KeyGen` function and appendix C contains the full code of the library. In this stage we also remove unnecessary mutable references using the process described in section 5.3.

As we can see, in order to generate a keypair, we only need to call the `keygen_from` function, passing it the `Randomness` and obtaining a `Keypair` structure that organizes the generated keys. In order to access the public key we only need to access the `public_key` attribute. Furthermore, the library includes an even simpler `keygen` function, which also abstracts the creation of the random bytes. This uses the `generate_randomness` function, which is also public, to generate an appropriate array of `random_bytes`. It uses the `ChaChaRng` *pseudorandom number generator* (PRNG) from the Rust `rand` crate [4], which is an implementation of the ChaCha20 cipher [16]. The PRNG is seeded randomly from entropy by `rand` and was chosen to provide a fast and cryptographically secure method for random number generation.

Listing 29 presents an example on how to use the high-level abstractions to obtain a shared secret between two parties (`p1` and `p2`). As we can see, the usage of Kyber is greatly simplified.

```

1 pub type SecretKey = [u8; KYBER_SECRETKEYBYTES];
2 pub type PublicKey = [u8; KYBER_PUBLICKEYBYTES];
3
4 pub type Randomness = [u8; 2 * KYBER_SYMBYTES];
5
6 pub struct Keypair {
7     pub secret_key: SecretKey,
8     pub public_key: PublicKey,
9 }
10
11 pub fn keygen_from(random_bytes: &Randomness) -> Keypair {
12     let mut pk = [0u8; KYBER_PUBLICKEYBYTES];
13     let mut sk = [0u8; KYBER_SECRETKEYBYTES];
14
15     kem::keygen(&mut pk, &mut sk, random_bytes);
16
17     Keypair {
18         secret_key: sk,
19         public_key: pk
20     }
21 }

```

Listing 28: High-level interface for `Kyber.CCAKEM.Keygen`

```

1 #[test]
2 fn kem_kyber_ke() {
3     let p1_keypair = rjkyber::keygen();
4     let p2_ecaps = rjkyber::encaps(&p1_keypair.public_key);
5     let p1_shared_secret = rjkyber::decaps(&p1_keypair.secret_key,
6     &p2_ecaps.ciphertext);
7     assert_eq!(p1_shared_secret, p2_ecaps.shared_secret);
8 }

```

Listing 29: Example of a Key exchange using Kyber

5.3 Safety Discussion

It is not trivial to argue about the safety of our approach. The interoperation of Rust and Jasmin can introduce safety concerns if this topic is not carefully addressed. There is no way for either the Rust or the Jasmin compilers to enforce consistency of their interaction. For example, we can allocate an array of size n in Rust pass that array as a parameter to a Jasmin function. There is nothing preventing Jasmin from accessing memory regions beyond length n , which can lead to memory safety problems. So, need to carefully analyse the safety of our approach.

Due to the limitations of Rust’s verification tools mentioned in section 4.6 (which do not

allow us to automatically verify the safety of our approach), we need to argue about the safety of the library by understanding the concept of safety in both Rust and Jasmin as well as how those concepts are transposed from one language to another.

Rust ensures the absence of race conditions and memory safety bugs through the ownership mechanism. This is done by enforcing that every memory region is valid and that every mutable reference has no other aliases [38, 42]. Jasmin ensures the safety of the code through its safety checker, but it assumes that all memory pointers it receives point to valid memory.

In our library, communication between the two languages is done via pointers passed via processor registers. Rust calls the Jasmin assembly code by passing it 64-bit pointers to the memory locations where the arrays are allocated in the stack. So, the first constraint for the safety of our library is to ensure that Jasmin is called with valid and initialized memory. Fortunately, Rust ensures this because it does not allow the declaration of uninitialized memory. Furthermore, because the lengths of all arrays are known at compile time, Rust enforces that the arrays passed to each function have the exact correct length used in the algorithm. It is also important that Jasmin makes no invalid memory accesses beyond these memory regions, which is something that neither Rust nor Jasmin are able to verify, so an EasyCrypt proof was constructed in [7] to prove that no invalid memory accesses are performed by the Kyber implementation.

Several mutable references are passed as parameters to the Jasmin Kyber implementation, but there are also immutable ones, so it is important to ensure that memory Rust assumes will not be changed is in fact not altered. An automatic way to enforce this would require an exploration of Jasmin's internal memory model and probably an extension to the compiler which is out of the scope of this dissertation. The same proof in [7] also addresses this issue. It proves that for example the public key in the encapsulation function is not altered, which allows us to pass it to Jasmin as an immutable reference.

Finally, because Jasmin implementations are single threaded, there is no way a mutable reference is accessed in multiple threads unless Rust spawns those threads, which is something that it does not allow thanks to the ownership mechanism. So, we can state that there are no other aliases for a mutable reference, respecting Rust's thread safety principles, which ensure the absence of race conditions.

Chapter 6

Testing and Benchmarking

It is important that our library is extensively tested and benchmarked. This will allow us to assess its correct functioning and that it does not introduce any bugs to underlying implementation. We also evaluate its performance, comparing it to other implementations.

This chapter describes how we tested RjKyber to make sure it introduces no bugs and presents the most relevant results we obtained when benchmarking it. It also compares these results with another Kyber implementation in Rust, discussing the impact of the results obtained.

6.1 Testing

Using cargo, Rust supports a test framework outside of the box, which is what we used to evaluate the behaviour of RjKyber. We test both the high-level interface (implicitly testing the IND-CCA KEM functions) and the IND-CPA PKE functions. This is done by creating functions annotated with `#[test]` that use assertions to enforce the correct behaviour of the code. If a panic occurs (specially in the case of an assertion violation), then the test fails, otherwise it passes. The tests can be found in the `tests` directory and can be executed using `cargo test`. We now present some examples in the different ways we test the library. Appendix D contains the full test code.

We test structural algorithm behaviour like the fact that the result of the encapsulation using two different public keys, produces a different ciphertext and shared secret, even when using the same random bytes or that decapsulating a ciphertext with the wrong private key yields the wrong shared secret. Listing 30 illustrates this exact case.

We also test the key exchange protocols the algorithm is supposed to be used in, to ensure that the behaviour of the library is correct for its intended use cases. For example, listing 31 tests the behaviour of the library in a one-sided authenticated key exchange protocol where one party has long term static keys with its public key known by the other party.

```

1  #[test]
2  fn kem_decaps_wrong_secret_key() {
3      let kp = rjkyber::keygen();
4      let mut wrong_sk = kp.secret_key.clone();
5      wrong_sk[0] = !wrong_sk[0]; //flip first byte
6
7      let enc = rjkyber::encaps(&kp.public_key);
8
9      let ss1 = rjkyber::decaps(&kp.secret_key, &enc.ciphertext);
10     let ss2 = rjkyber::decaps(&wrong_sk, &enc.ciphertext);
11
12     assert_ne!(ss1, ss2);
13 }

```

Listing 30: Testing that a decapsulation with the wrong secret key results in the wrong shared secret

```

1  #[test]
2  fn kem_kyber_uake() {
3      // static p2 keys
4      let p2_keypair = rjkyber::keygen();
5
6      // p1 generates keys and encapsulates with pk2
7      let p1_keypair = rjkyber::keygen();
8      let p1_encaps = rjkyber::encaps(&p2_keypair.public_key);
9
10     // p2 encapsulates with pk1 and decapsulates sk2 and c1
11     let p2_encaps = rjkyber::encaps(&p1_keypair.public_key);
12     let p2_shared_secret = rjkyber::decaps(&p2_keypair.secret_key,
13     &p1_encaps.ciphertext);
14
15     // p1 decaps with sk1 and c2
16     let p1_shared_secret = rjkyber::decaps(&p1_keypair.secret_key,
17     &p2_encaps.ciphertext);
18
19     assert_eq!(p1_shared_secret, p2_encaps.shared_secret);
20     assert_eq!(p1_encaps.shared_secret, p2_shared_secret);
21 }

```

Listing 31: Testing the one-sided authenticated key exchange protocol

6.2 Benchmarking

In order to benchmark the library we use the `Criterion Rust` crate [2], an automatic statistics driven benchmarking library which generates statistical reports about the different benchmarks ex-

ecuted. These benchmarks were run in a machine featuring an AMD Ryzen 7 5800X processor at 3.8GHz with Arch Linux x86_64 installed running the 5.18.8-arch1-1 kernel. Rust was compiled using the `rustc 1.60.0` compiler and criterion was used in version 0.3. Every benchmark collected a population of 100 samples to calculate the several statistics. The complete benchmark code used can be consulted in appendix E and additional benchmarks results, not presented in this section, can be consulted in appendix F.

6.2.1 Benchmarking the high-level interface

These benchmarks compare execution times for the high-level interface KEM functions and the corresponding Jasmin bindings. This serves to assess the impact the high-level interface has in the execution times of the algorithm.

These benchmarks use the AVX2 Kyber768 implementation and use the same random bytes for both functions, ensuring the execution of both functions is exactly the same. Figures 6.1, 6.2 and 6.3 present the *probability density functions* (PDFs) for key generation, encapsulation and decapsulation functions for both the Rust interfaces and the Jasmin bindings. Table 6.1 compares the mean execution time.

As we can see, the impact of adding the high-level interface layer is minimal, impacting the performance of the functions in less than 5%. In a lot of cases, this impact is insignificant because the interface mainly creates the necessary data structures before calling the bindings, which is something that would have to be done anyway if no interface existed. There are however applications that cannot afford this performance penalty. It is precisely for those cases that we also provide direct access to the Jasmin bindings via the `rkjkyber::kem` module.

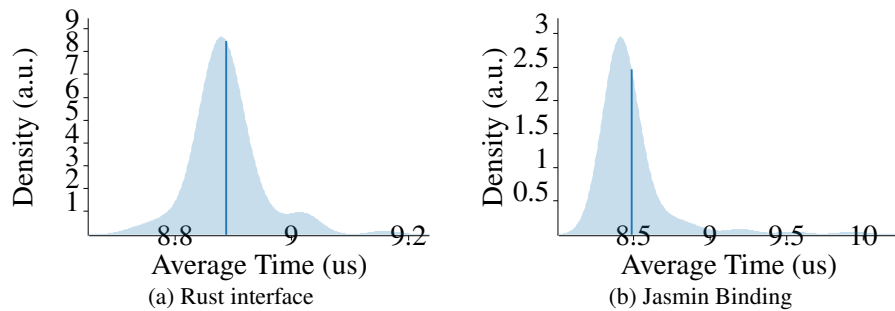


Figure 6.1: Key generation functions comparison

Table 6.1: Mean execution times of the Rust interface and Jasmin bindings

Function	Rust Interface (us)	Jasmin Binding (us)	Performance difference (%)
Keygen	8.8879	8.4868	4.726
Encapsulation	11.660	11.121	4.847
Decapsulation	9.1280	9.0079	1.333

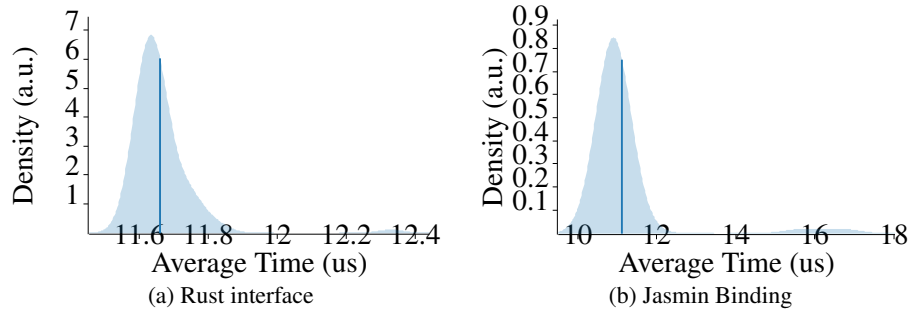


Figure 6.2: Encapsulation functions comparison

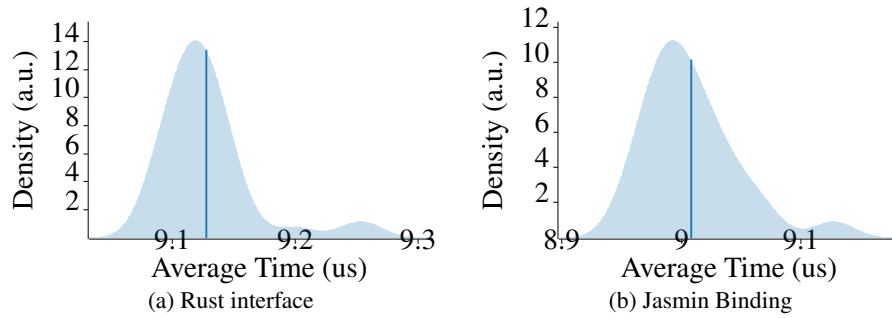


Figure 6.3: Decapsulation functions comparison

6.2.2 Comparing with `pqc_kyber`

There is an existing Kyber library for Rust called `pqc_kyber` [3], which ports Kyber’s reference C implementation to safe Rust. It also supports the optimized AVX2 version, although it uses unsafe Rust in the form of direct calls to assembly code.

It is then interesting to compare benchmarks for both implementations to see the impact Jasmin has in a cryptographic library. Since `pqc_kyber` also uses a high-level interface to interact with algorithm functions, we compare both implementations through their high-level APIs. Both libraries are running Kyber768 in its reference implementation using the hash functions and KDFs defined in the current Kyber specification. Again, figures 6.4, 6.5 and 6.6 contain the PDFs of the samples collected while table 6.2 contains the mean execution times of both implementations.

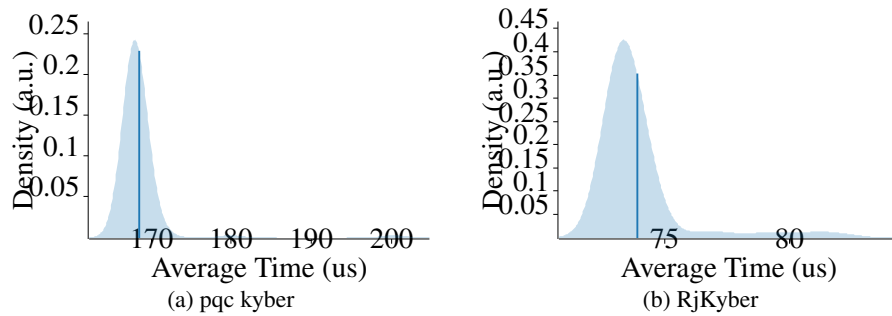


Figure 6.4: Keygen functions comparison

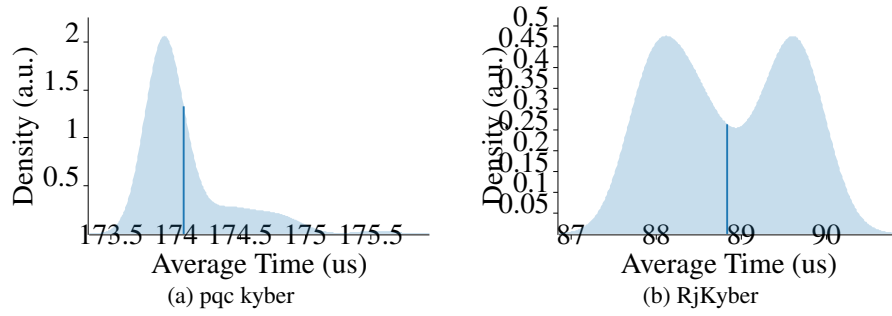


Figure 6.5: Encapsulation functions comparison

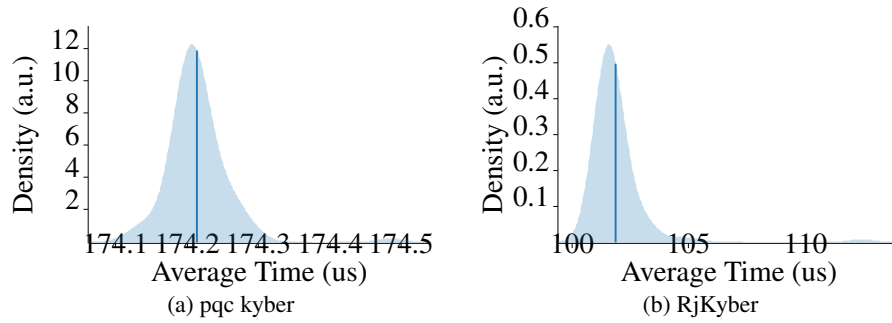


Figure 6.6: Keygen functions comparison

Table 6.2: Mean execution times of pqc kyber and RjKyber

Function	pqc kyber (us)	RjKyber (us)
Keygen	168.51	73.882
Encapsulation	174.06	88.832
Decapsulation	174.21	101.90

Running these same benchmarks with both of the most optimized versions of each library yields an interesting result: both implementations have similar execution times. This is most likely due to the fact that pqc kyber also uses very efficient assembly code in its AVX2 implementation, closing the gap between the implementations. These results can be observed in appendix F.3.

Besides performance, RjKyber presents some other advantages when compared with pqc kyber. Firstly, the Jasmin implementation is provably safe and as discussed in section 5.3, that safety is maintained and even extended to some degree in the Rust layer while `pqc_kyber` uses unsafe Rust in its AVX2 implementation, requiring a fuzzer to test the safety of the implementation. Also, RjKyber enforces that the arrays passed as parameters have the correct length for that specific input of the function called, which is something `pqc_kyber` does not. In `pqc_kyber`, we can pass arrays with the wrong length to a function, resulting in an error.

Chapter 7

Conclusion

We have presented RjKyber, a novel approach to developing a Rust library that uses a Jasmin implementation of the CRYSTALS - Kyber post-quantum cryptographic algorithm to provide a high-speed, high assurance alternative for post-quantum cryptography to Rust developers. RjKyber provides Jasmin’s advantages while abstracting all the interaction with the framework, working outside of the box when imported by cargo without any complications.

We explored the field of formal verification in Rust, concluding it has potential, but the several tools available are not still in a sufficiently mature stage to be effectively used in critical cryptographic code. It would be interesting however to further explore them to be able to annotate and verify the usage of cryptographic code in Rust.

We described the development process of RjKyber and the several iterations the library went through, concluding that despite having the discomfort using the `unsafe` keyword to call verified code (which we know is not unsafe), this brings advantages in the interaction of the library with the Rust ecosystem. We also discussed the safety of RjKyber, concluding that the interaction between Rust and Jasmin can be done in a way that preserves the safety properties of both languages, ultimately achieving a higher degree of safety and efficiency that none of the languages could achieve by itself. This, however assumes that the Kyber implementation we use is indeed safe and correct. Our tests empirically indicate it is, but the formal proofs of safety are still the subject of ongoing work. We have also intensively benchmarked RjKyber and verified the performance advantages of using Jasmin for cryptographic routines.

Our conclusions when arguing the safety of the interaction between Rust and Jasmin present an important step forward in the usage of both of these languages that we hope will provide a basis for future libraries that will be faster and safer, without relying in actually unsafe Rust code.

Overall, we consider that RjKyber provides a practical and easy to use post-quantum cryptography library for Rust developers to seamlessly deploy in their code, but probably more importantly, it provides foundational work in the safety of the interaction between Rust and Jasmin.

7.1 Future work

As noted in section 5.2.2, we currently do not support Kyber512 and Kyber1024. We plan to support them in a future version of RjKyber, as the foundations for their integration have already been implemented. Furthermore, it would also be interesting to rerun our benchmarks in these versions of the algorithm. We would also like to increase the testing surface of RjKyber using *known answer tests* (KATs) to ensure perfect execution of the algorithm, according to specification.

RjKyber currently only supports x86_64 architectures because we currently only include assembly code for this architecture. Rustc is able to detect the architecture it is compiling for, so we can use conditional compilation annotations to also support RjKyber in ARM for example, which Jasmin also supports.

We also intend to publish RjKyber as Rust crate in `crates.io` so that it can indeed be used by Rust programmers for post-quantum cryptography. This cannot be done yet because the Kyber implementation we use is not public and its formalization is still in progress.

Finally, this dissertation provides a good starting point in the interaction between Jasmin and Rust for cryptographic code. This means that RjKyber could integrate a wider library providing access to Jasmin implementations of the main cryptographic algorithms used in modern protocols. In the future, we could integrate other exiting algorithms written in Jasmin to create a much more powerful and versatile library.

References

- [1] The Coq Reference Manual. URL: <https://coq.inria.fr/distrib/current/refman/>.
- [2] criterion - crates.io: Rust Package Registry. URL: <https://crates.io/crates/criterion>.
- [3] pqc_kyber - crates.io: Rust Package Registry. URL: https://crates.io/crates/pqc_kyber.
- [4] rand - crates.io: Rust Package Registry. URL: <https://crates.io/crates/rand>.
- [5] Redox Operating System. URL: <https://doc.redox-os.org/book/>.
- [6] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. Jasmin: High-Assurance and High-Speed Cryptography. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1807–1823, Dallas Texas USA, October 2017. ACM. URL: <https://dl.acm.org/doi/10.1145/3133956.3134078>, doi:10.1145/3133956.3134078.
- [7] José Bacelar Almeida, Manuel Barbosa, Benjamin Grégoire, Vincent Laporte, Miguel Quaresma, Antoine Seré, Pierre-Yves Strub, and Peter Schwabe. Formally Verifying Kyber. Manuscript.
- [8] Vytautas Astrauskas, Christoph Matheja, Federico Poli, Peter Müller, and Alexander J. Summers. How do programmers use unsafe rust? *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–27, November 2020. URL: <https://dl.acm.org/doi/10.1145/3428204>, doi:10.1145/3428204.
- [9] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. Leveraging rust types for modular specification and verification. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–30, October 2019. URL: <https://dl.acm.org/doi/10.1145/3360573>, doi:10.1145/3360573.
- [10] Jean-Philippe Aumasson. Quantum and post-quantum. In *Serious Cryptography: A Practical Introduction to Modern Encryption*, pages 251–269. No Starch Press, 2018.
- [11] Jean-Philippe Aumasson. RSA. In *Serious Cryptography: A Practical Introduction to Modern Encryption*, pages 181–199. No Starch Press, 2018.
- [12] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS - Kyber: Algorithm Specifications And Supporting Documentation. URL: <https://>

- pq-crystals.org/kyber/data/kyber-specification-round3-20210131.pdf.
- [13] Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. SoK: Computer-Aided Cryptography. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 777–795, San Francisco, CA, USA, May 2021. IEEE. URL: <https://ieeexplore.ieee.org/document/9519449/>, doi: 10.1109/SP40001.2021.00008.
 - [14] Gilles Barthe, Sunjay Cauligi, Benjamin Gregoire, Adrien Koutsos, Kevin Liao, Tiago Oliveira, Swarn Priya, Tamara Rezk, and Peter Schwabe. High-Assurance Cryptography in the Spectre Era. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1884–1901, San Francisco, CA, USA, May 2021. IEEE. URL: <https://ieeexplore.ieee.org/document/9519434/>, doi:10.1109/SP40001.2021.00046.
 - [15] Daniel J Bernstein. Cache-timing attacks on AES. URL: <https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.140.2835&rep=rep1&type=pdf>.
 - [16] Daniel J Bernstein. ChaCha, a variant of Salsa20. URL: <https://cr.yp.to/chacha/chacha-20080120.pdf>.
 - [17] Kevin Boos, Namitha Liyanage, Ramla Ijaz, and Lin Zhong. Theseus: an Experiment in Operating System Structure and State Management. URL: <https://www.usenix.org/system/files/osdi20-boos.pdf>.
 - [18] Joppe Bos, Leo Ducas, Eike Kiltz, T Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehle. CRYSTALS - Kyber: A CCA-Secure Module-Lattice-Based KEM. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 353–367, London, April 2018. IEEE. URL: <https://ieeexplore.ieee.org/document/8406610/>, doi:10.1109/EuroSP.2018.00032.
 - [19] J. P. Buhler, H. W. Lenstra, and Carl Pomerance. Factoring integers with the number field sieve. In Arjen K. Lenstra and Hendrik W. Lenstra, editors, *The development of the number field sieve*, volume 1554, pages 50–94. Springer Berlin Heidelberg, Berlin, Heidelberg, 1993. Series Title: Lecture Notes in Mathematics. URL: <http://link.springer.com/10.1007/BFb0091539>, doi:10.1007/BFb0091539.
 - [20] Lily Chen, Stephen Jordan, Yi-Kai Liu, Dustin Moody, Rene Peralta, Ray Perlner, and Daniel Smith-Tone. Report on Post-Quantum Cryptography. Technical Report NIST IR 8105, National Institute of Standards and Technology, April 2016. URL: <https://nvlpubs.nist.gov/nistpubs/ir/2016/NIST.IR.8105.pdf>, doi:10.6028/NIST.IR.8105.
 - [21] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A Tool for Checking ANSI-C Programs. In Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, Kurt Jensen, and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988, pages 168–176. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. Series Title: Lecture Notes in Computer Science. URL: http://link.springer.com/10.1007/978-3-540-24730-2_15, doi:10.1007/978-3-540-24730-2_15.
 - [22] Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. The CREUSOT Environment for the Deductive Verification of Rust Programs. URL: <https://hal.inria.fr/hal-03526634v2/document>.

- [23] Whitfield Diffie and Martin E Hellman. New Directions in Cryptography. URL: <https://ieeexplore.ieee.org/document/1055638?reload=true>.
- [24] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — Where Programs Meet Provers. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Matthias Felleisen, and Philippa Gardner, editors, *Programming Languages and Systems*, volume 7792, pages 125–128. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. Series Title: Lecture Notes in Computer Science. URL: http://link.springer.com/10.1007/978-3-642-37036-6_8, doi:10.1007/978-3-642-37036-6_8.
- [25] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure Integration of Asymmetric and Symmetric Encryption Schemes. *Journal of Cryptology*, 26(1):80–101, January 2013. URL: <http://link.springer.com/10.1007/s00145-011-9114-1>, doi:10.1007/s00145-011-9114-1.
- [26] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing - STOC '96*, pages 212–219, Philadelphia, Pennsylvania, United States, 1996. ACM Press. URL: <http://portal.acm.org/citation.cfm?doid=237814.237866>, doi:10.1145/237814.237866.
- [27] Eric R Johnston. *Programming Quantum Computers*.
- [28] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: securing the foundations of the Rust programming language. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–34, January 2018. URL: <https://dl.acm.org/doi/10.1145/3158154>, doi:10.1145/3158154.
- [29] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28:e20, 2018. URL: https://www.cambridge.org/core/product/identifier/S0956796818000151/type/journal_article, doi:10.1017/S0956796818000151.
- [30] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. No Starch Press, 2018.
- [31] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19, May 2019. ISSN: 2375-1207. doi:10.1109/SP.2019.00002.
- [32] Adeline Langlois and Damien Stehlé. Worst-case to average-case reductions for module lattices. *Designs, Codes and Cryptography*, 75(3):565–599, June 2015. URL: <http://link.springer.com/10.1007/s10623-014-9938-4>, doi:10.1007/s10623-014-9938-4.
- [33] H.J. Lu, Michael Matz, Milind Girkar, Jan Hubicka, Andreas Jaeger, and Mark Mitchell. System V Application Binary Interface - AMD64 Architecture Processor Supplement. Technical report, 2021. URL: <https://gitlab.com/x86-psABIs/x86-64-ABI>.

- [34] Robert J. McEliece. A public-key cryptosystem based on algebraic coding theory. 1978. URL: <https://ntrs.nasa.gov/api/citations/19780016269/downloads/19780016269.pdf#page=123>.
- [35] Dustin Moody. Status Report on the Third Round of the NIST Post-Quantum Cryptography Standardization Process. Technical Report NIST IR 8413, National Institute of Standards and Technology, Gaithersburg, MD, 2022. URL: <https://nvlpubs.nist.gov/nistpubs/ir/2022/NIST.IR.8413.pdf>, doi:10.6028/NIST.IR.8413.
- [36] Dustin Moody, Gorjan Alagic, Daniel C Apon, David A Cooper, Quynh H Dang, John M Kelsey, Yi-Kai Liu, Carl A Miller, Rene C Peralta, Ray A Perlner, Angela Y Robinson, Daniel C Smith-Tone, and Jacob Alperin-Sheriff. Status report on the second round of the NIST post-quantum cryptography standardization process. Technical Report NIST IR 8309, National Institute of Standards and Technology, Gaithersburg, MD, July 2020. URL: <https://nvlpubs.nist.gov/nistpubs/ir/2020/NIST.IR.8309.pdf>, doi:10.6028/NIST.IR.8309.
- [37] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A Verification Infrastructure for Permission-Based Reasoning. In Barbara Jobstmann and K. Rustan M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 9583, pages 41–62. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016. Series Title: Lecture Notes in Computer Science. URL: http://link.springer.com/10.1007/978-3-662-49122-5_2, doi:10.1007/978-3-662-49122-5_2.
- [38] Eric Reed. Patina: A Formalization of the Rust Programming Language. URL: <https://dada.cs.washington.edu/research/tr/2015/03/UW-CSE-15-03-02.pdf>.
- [39] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM*, 56(6):1–40, September 2009. URL: <https://dl.acm.org/doi/10.1145/1568318.1568324>, doi:10.1145/1568318.1568324.
- [40] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978. URL: <https://dl.acm.org/doi/10.1145/359340.359342>, doi:10.1145/359340.359342.
- [41] Peter W. Shor. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *SIAM Journal on Computing*, 26(5):1484–1509, October 1997. arXiv: quant-ph/9508027. URL: <http://arxiv.org/abs/quant-ph/9508027>, doi:10.1137/S0097539795293172.
- [42] John Toman, Stuart Pernsteiner, and Emina Torlak. Crust: A Bounded Verifier for Rust (N). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 75–80, November 2015. doi:10.1109/ASE.2015.77.
- [43] Juriaan van Drunen. Introducing an approach to safely interoperate between the Rust and Jasmin programming languages. Master’s thesis, Eindhoven University of Technology, 2021. URL: https://pure.tue.nl/ws/portalfiles/portal/199766729/Drunen_Jl.pdf.

Appendix A

Jasmin Dockerfile

```
1 FROM nixos/nix
2
3 RUN nix-channel --update
4
5 COPY compiler /jasmin
6
7 WORKDIR /jasmin
8
9 RUN nix-env -i -f default.nix
```

Listing 32: Dockerfile that can be used to create a jasmin image in any machine

```
1 docker build . -t jasmin
```

Listing 33: Command to build the image

```
1 docker run -it -v ~/jasmin_code:/jasmin_code jasmin
```

Listing 34: Command to run the image linking /jasmin_code on the local machine to /jasmin_code inside the container

Appendix B

Jasmin bindings build script

```
1  #!/bin/sh
2
3  # Compile the assembly to object file
4  cd ./src/libjkem
5  clang asm/jkem_$1.s -c -o jkem.o
6
7  # Compile the rust code to rlib file
8  rustc --crate-type=rlib jkem.rs
9
10 # Substitute the rlib for the jasmin object
11 ar -x libjkem.rlib
12 rm jkem.jkem.*
13 rm libjkem.rlib
14 ar -crs libjkem.rlib jkem.o lib.rmeta
15
16 # Clean up
17 rm jkem.o lib.rmeta
18
19 # Move to OUT
20 mv libjkem.rlib $2
```

Listing 35: Jasmin bindings build script

Appendix C

Full code of RjKyber

```
1  // exports
2  pub mod params; // parameters
3  pub mod kem;    // ind-cca kem
4  pub mod indcpa; // ind-cpa pke
5
6
7  use crate::params::*;
8  use rand::{prng::ChaChaRng, FromEntropy, RngCore};
9
10 pub type SecretKey = [u8; KYBER_SECRETKEYBYTES];
11 pub type PublicKey = [u8; KYBER_PUBLICKEYBYTES];
12
13 pub type Ciphertext = [u8; KYBER_CIPHERTEXTBYTES];
14 pub type SharedSecret = [u8; KYBER_SSBYTES];
15
16 pub type Randomness = [u8; 2*KYBER_SYMBYTES];
17
18 pub struct Keypair {
19     pub secret_key: SecretKey,
20     pub public_key: PublicKey,
21 }
22
23 pub struct Encapsulation {
24     pub ciphertext: Ciphertext,
25     pub shared_secret: SharedSecret
26 }
27
```

```
28 pub fn generate_randomness() -> Randomness {
29     let mut rand = [0u8; 2*KYBER_SYMBYTES];
30     let mut rng = ChaChaRng::from_entropy();
31     rng.fill_bytes(&mut rand);
32
33     rand
34 }
35
36 pub fn keygen_from(random_bytes: &Randomness) -> Keypair {
37     let mut pk = [0u8; KYBER_PUBLICKEYBYTES];
38     let mut sk = [0u8; KYBER_SECRETKEYBYTES];
39
40     kem::keygen(&mut pk, &mut sk, random_bytes);
41
42     Keypair {
43         secret_key: sk,
44         public_key: pk
45     }
46 }
47
48 pub fn keygen() -> Keypair {
49     let rand = generate_randomness();
50     keygen_from(&rand)
51 }
52
53
54
55 pub fn encaps_from(
56     public_key: &PublicKey,
57     random_bytes: &Randomness
58 ) -> Encapsulation {
59     let mut ct = [0u8; KYBER_CIPHERTEXTBYTES];
60     let mut ss = [0u8; KYBER_SSBYTES];
61
62     kem::enc(&mut ct, &mut ss, public_key, random_bytes);
63
64     Encapsulation {
65         ciphertext: ct,
66         shared_secret: ss
67     }
68 }
```

```

68 }
69
70 pub fn encaps(public_key: &PublicKey) -> Encapsulation {
71     let rand = generate_randomness();
72     encaps_from(public_key, &rand)
73 }
74
75
76
77 pub fn decaps(
78     secret_key: &SecretKey,
79     ciphertext: &Ciphertext
80 ) -> SharedSecret {
81     let mut ss = [0u8; KYBER_SSBYTES];
82     kem::dec(&mut ss, ciphertext, secret_key);
83     ss
84 }

```

Listing 36: RjKyber src/lib.rs file

```

1 use crate::params::*;
2
3 extern {
4     // kem functions
5     fn crypto_kem_keypair_jazz(
6         pk: &mut [u8; KYBER_PUBLICKEYBYTES],
7         sk: &mut [u8; KYBER_SECRETKEYBYTES],
8         rand: &[u8; 2*KYBER_SYMBYTES]
9     );
10
11     fn crypto_kem_enc_jazz(
12         ct: &mut [u8; KYBER_CIPHERTEXTBYTES],
13         shk: &mut [u8; KYBER_SSBYTES],
14         pk: &[u8; KYBER_PUBLICKEYBYTES],
15         rand: &[u8; 2*KYBER_SYMBYTES]
16     );
17
18     fn crypto_kem_dec_jazz(
19         shk: &mut [u8; KYBER_SSBYTES],
20         ct: &[u8; KYBER_CIPHERTEXTBYTES],

```

```

21         sk: &[u8; KYBER_SECRETKEYBYTES]
22     );
23 }
24
25
26 // define KEM functions
27
28 pub fn keygen(
29     pk: &mut [u8; KYBER_PUBLICKEYBYTES],
30     sk: &mut [u8; KYBER_SECRETKEYBYTES],
31     rand: &[u8; 2 * KYBER_SYMBYTES]
32 ) {
33     unsafe {
34         crypto_kem_keypair_jazz(pk, sk, rand);
35     }
36 }
37
38 pub fn encaps(
39     ct: &mut [u8; KYBER_CIPHERTEXTBYTES],
40     shk: &mut [u8; KYBER_SSBYTES],
41     pk: &[u8; KYBER_PUBLICKEYBYTES],
42     rand: &[u8; 2 * KYBER_SYMBYTES]
43 ) {
44     unsafe {
45         crypto_kem_enc_jazz(ct, shk, pk, rand);
46     }
47 }
48
49
50 pub fn decaps(
51     shk: &mut [u8; KYBER_SSBYTES],
52     ct: &[u8; KYBER_CIPHERTEXTBYTES],
53     sk: &[u8; KYBER_SECRETKEYBYTES]
54 ) {
55     unsafe {
56         crypto_kem_dec_jazz(shk, ct, sk);
57     }
58 }

```

Listing 37: RjKyber src/kem.rs file

```

1 use crate::params::*;
2
3 extern "C" {
4     // pke functions
5     fn indcpa_keypair_jazz(
6         pk: &mut [u8; KYBER_INDCPA_PUBLICKEYBYTES],
7         sk: &mut [u8; KYBER_INDCPA_SECRETKEYBYTES],
8         rand: &[u8; KYBER_SYMBYTES]
9     );
10
11     fn indcpa_enc_jazz(
12         ct: &mut [u8; KYBER_INDCPA_BYTES],
13         msg: &[u8; KYBER_INDCPA_MSGBYTES],
14         pk: &[u8; KYBER_INDCPA_PUBLICKEYBYTES],
15         rand: &[u8; KYBER_SYMBYTES]
16     );
17
18     fn indcpa_dec_jazz(
19         msg: &mut [u8; KYBER_INDCPA_MSGBYTES],
20         ct: &[u8; KYBER_INDCPA_BYTES],
21         sk: &[u8; KYBER_INDCPA_SECRETKEYBYTES]
22     );
23 }
24
25 pub fn keygen(
26     pk: &mut [u8; KYBER_INDCPA_PUBLICKEYBYTES],
27     sk: &mut [u8; KYBER_INDCPA_SECRETKEYBYTES],
28     rand: &[u8; KYBER_SYMBYTES]
29 ) {
30     unsafe {
31         indcpa_keypair_jazz(pk, sk, rand);
32     }
33 }
34
35
36 pub fn enc(
37     ct: &mut [u8; KYBER_INDCPA_BYTES],
38     msg: &[u8; KYBER_INDCPA_MSGBYTES],

```

```

39     pk: &[u8; KYBER_INDCPA_PUBLICKEYBYTES],
40     rand: &[u8; KYBER_SYMBYTES]
41 ) {
42     unsafe{
43         indcpa_enc_jazz(ct, msg, pk, rand);
44     }
45 }
46
47
48 pub fn dec(
49     msg: &mut [u8; KYBER_INDCPA_MSGBYTES],
50     ct: &[u8; KYBER_INDCPA_BYTES],
51     sk: &[u8; KYBER_INDCPA_SECRETKEYBYTES]
52 ) {
53     unsafe {
54         indcpa_dec_jazz(msg, ct, sk);
55     }
56 }

```

Listing 38: RjKyber src/indcpa.rs file

```

1  // define params for different kyber versions
2
3  #[cfg(feature = "kyber512")]
4  pub const KYBER_K: usize = 2;
5
6  #[cfg(not(any(feature = "kyber512", feature = "kyber1024")))]
7  pub const KYBER_K: usize = 3;
8
9  #[cfg(feature = "kyber1024")]
10 pub const KYBER_K: usize = 4;
11
12
13 pub const KYBER_SYMBYTES: usize = 32;
14 pub const KYBER_SSBYTES: usize = 32;
15
16 pub const KYBER_POLYBYTES: usize = 384;
17 pub const KYBER_POLYVECBYTES: usize = KYBER_K * KYBER_POLYBYTES;
18
19

```

```

20  #[cfg(feature = "kyber512")]
21  pub const KYBER_POLYCOMPRESSEDBYTES: usize = 96;
22  #[cfg(feature = "kyber512")]
23  pub const KYBER_POLYVECCOMPRESSEDBYTES: usize = KYBER_K * 320;
24
25  #[cfg(not(any(feature = "kyber512", feature = "kyber1024")))]
26  pub const KYBER_POLYCOMPRESSEDBYTES: usize = 128;
27  #[cfg(not(any(feature = "kyber512", feature = "kyber1024")))]
28  pub const KYBER_POLYVECCOMPRESSEDBYTES: usize = KYBER_K * 320;
29
30  #[cfg(feature = "kyber1024")]
31  pub const KYBER_POLYCOMPRESSEDBYTES: usize = 160;
32  #[cfg(feature = "kyber1024")]
33  pub const KYBER_POLYVECCOMPRESSEDBYTES: usize = KYBER_K * 352;
34
35
36  pub const KYBER_INDCPA_BYTES: usize = KYBER_POLYVECCOMPRESSEDBYTES
37      + KYBER_POLYCOMPRESSEDBYTES;
38  pub const KYBER_INDCPA_PUBLICKEYBYTES: usize = KYBER_POLYVECBYTES
39      + KYBER_SYMBYTES;
40  pub const KYBER_INDCPA_SECRETKEYBYTES: usize = KYBER_POLYVECBYTES;
41  pub const KYBER_INDCPA_MSGBYTES: usize = KYBER_SYMBYTES;
42
43  pub const KYBER_PUBLICKEYBYTES: usize = KYBER_INDCPA_PUBLICKEYBYTES;
44  pub const KYBER_SECRETKEYBYTES: usize = KYBER_INDCPA_SECRETKEYBYTES
45      + KYBER_INDCPA_PUBLICKEYBYTES + 2*KYBER_SYMBYTES;
46
47  pub const KYBER_CIPHertextBYTES: usize = KYBER_INDCPA_BYTES;

```

Listing 39: RjKyber src/params.rs file

```

1  use cc;
2
3  // define different kyber versions
4
5  #[cfg(feature = "kyber512")]
6  const KYBER_VERSION: &str = "kyber512";
7
8  #[cfg(not(any(feature = "kyber512", feature = "kyber1024")))]
9  const KYBER_VERSION: &str = "kyber768";

```

```

10
11
12 #[cfg(feature = "kyber1024")]
13 const KYBER_VERSION: &str = "kyber1024";
14
15 // =====
16 // define compilation functions
17
18 #[cfg(not(feature = "ref"))]
19 fn compile_asm(version: &str) {
20     cc::Build::new()
21         .file(&format!("src/asm/{}/jkem_avx2.s", version))
22         .file(&format!("src/asm/{}/jindcpa_avx2.s", version))
23         .compile("jkem");
24 }
25
26 #[cfg(feature = "ref")]
27 fn compile_asm(version: &str) {
28     cc::Build::new()
29         .file(&format!("src/asm/{}/jkem_ref.s", version))
30         .file(&format!("src/asm/{}/jindcpa_ref.s", version))
31         .compile("jkem");
32 }
33
34 // =====
35
36 fn main() {
37     compile_asm(KYBER_VERSION);
38 }
39

```

Listing 40: RjKyber build.rs file

Appendix D

Test Code

```
1 use rjkyber;
2
3 // high level interface also tests jasmin kem functions
4
5 #[test]
6 fn kem_different_keypairs() {
7     let kp1 = rjkyber::keygen();
8     let kp2 = rjkyber::keygen();
9
10    assert_ne!(kp1.secret_key, kp2.secret_key);
11    assert_ne!(kp1.public_key, kp2.public_key);
12 }
13
14 #[test]
15 fn kem_same_key() {
16     let rand = rjkyber::generate_randomness();
17     let kp1 = rjkyber::keygen_from(&rand);
18     let kp2 = rjkyber::keygen_from(&rand);
19
20    assert_eq!(kp1.secret_key, kp2.secret_key);
21    assert_eq!(kp1.public_key, kp2.public_key);
22 }
23
24 #[test]
25 fn kem_encaps_diff_keys() {
26     let kp = rjkyber::keygen();
27     let mut wrong_pk = kp.public_key.clone();
```

```
28     wrong_pk[0] = !wrong_pk[0]; //flip first byte
29
30     let rand = rjkyber::generate_randomness();
31
32     let e1 = rjkyber::encaps_from(&kp.public_key, &rand);
33     let e2 = rjkyber::encaps_from(&wrong_pk, &rand);
34
35     assert_ne!(e1.ciphertext, e2.ciphertext);
36     assert_ne!(e1.shared_secret, e2.shared_secret);
37 }
38
39 #[test]
40 fn kem_decaps_wrong_secret_key() {
41     let kp = rjkyber::keygen();
42     let mut wrong_sk = kp.secret_key.clone();
43     wrong_sk[0] = !wrong_sk[0]; //flip first byte
44
45     let enc = rjkyber::encaps(&kp.public_key);
46
47     let ss1 = rjkyber::decaps(&kp.secret_key, &enc.ciphertext);
48     let ss2 = rjkyber::decaps(&wrong_sk, &enc.ciphertext);
49
50     assert_ne!(ss1, ss2);
51 }
52
53 #[test]
54 fn kem_decaps_wrong_ciphertext() {
55     let kp = rjkyber::keygen();
56     let enc = rjkyber::encaps(&kp.public_key);
57
58     let mut wrong_c = enc.ciphertext.clone();
59     wrong_c[0] = !wrong_c[0]; //flip first byte
60
61     let ss1 = rjkyber::decaps(&kp.secret_key, &enc.ciphertext);
62     let ss2 = rjkyber::decaps(&kp.secret_key, &wrong_c);
63
64     assert_ne!(ss1, ss2);
65 }
66
67 #[test]
```

```
68 fn kem_kyber_ke() {
69     let p1_keypair = rjkyber::keygen();
70     let p2_ecaps = rjkyber::encaps(&p1_keypair.public_key);
71     let p1_shared_secret = rjkyber::decaps(&p1_keypair.secret_key,
72         &p2_ecaps.ciphertext);
73     assert_eq!(p1_shared_secret, p2_ecaps.shared_secret);
74 }
75
76 #[test]
77 fn kem_kyber_uake() {
78     // static p2 keys
79     let p2_keypair = rjkyber::keygen();
80
81     // p1 generates keys and encapsulates with pk2
82     let p1_keypair = rjkyber::keygen();
83     let p1_encaps = rjkyber::encaps(&p2_keypair.public_key);
84
85     // p2 encapsulates with pk1 and decapsulates sk2 and c1
86     let p2_encaps = rjkyber::encaps(&p1_keypair.public_key);
87     let p2_shared_secret = rjkyber::decaps(&p2_keypair.secret_key,
88         &p1_encaps.ciphertext);
89
90     // p1 decaps with sk1 and c2
91     let p1_shared_secret = rjkyber::decaps(&p1_keypair.secret_key,
92         &p2_encaps.ciphertext);
93
94     assert_eq!(p1_shared_secret, p2_encaps.shared_secret);
95     assert_eq!(p1_encaps.shared_secret, p2_shared_secret);
96 }
97
98 #[test]
99 fn kem_test_ake() {
100     // static keys
101     let p1_static_keypair = rjkyber::keygen();
102     let p2_static_keypair = rjkyber::keygen();
103
104     // p1 generates keys and encaps pk2_static
105     let p1_keypair = rjkyber::keygen();
106     let p1_encaps = rjkyber::encaps(&p2_static_keypair.public_key);
107
```

```

108 // p2 encaps pk1 and pk1_static and decaps (sk2_static,c1)
109 let p2_encaps1 = rjkyber::encaps(&p1_keypair.public_key);
110 let p2_encaps2 = rjkyber::encaps(&p1_static_keypair.public_key);
111 let p2_shared_secret = rjkyber::decaps(
112     &p2_static_keypair.secret_key,
113     &p1_encaps.ciphertext
114 );
115
116 // p1 decaps(sk1,c21) and (sk1_static,c22)
117 let p1_shared_secret1 = rjkyber::decaps(&p1_keypair.secret_key,
118     &p2_encaps1.ciphertext);
119 let p1_shared_secret2 = rjkyber::decaps(
120     &p1_static_keypair.secret_key,
121     &p2_encaps2.ciphertext
122 );
123
124 assert_eq!(p1_shared_secret1,p2_encaps1.shared_secret);
125 assert_eq!(p1_shared_secret2,p2_encaps2.shared_secret);
126 assert_eq!(p1_encaps.shared_secret,p2_shared_secret);
127 }

```

Listing 41: Tests for high-level interface

```

1 use rjkyber::params::*;
2 use rjkyber::*;
3 use rand::{prng::ChaChaRng, FromEntropy, RngCore};
4
5 #[test]
6 fn indcpa_diff_keypairs() {
7     let mut sk0 = [0u8; KYBER_INDCPA_SECRETKEYBYTES];
8     let mut pk0 = [0u8; KYBER_INDCPA_PUBLICKEYBYTES];
9
10    let mut sk1 = [0u8; KYBER_INDCPA_SECRETKEYBYTES];
11    let mut pk1 = [0u8; KYBER_INDCPA_PUBLICKEYBYTES];
12
13    let mut rand0 = [0u8; KYBER_SYMBYTES];
14    let mut rand1 = [0u8; KYBER_SYMBYTES];
15
16    let mut rng = ChaChaRng::from_entropy();
17    rng.fill_bytes(&mut rand0);

```

```
18     rng.fill_bytes(&mut rand1);
19
20
21     indcpa::keygen(&mut pk0, &mut sk0, &rand0);
22     indcpa::keygen(&mut pk1, &mut sk1, &rand1);
23
24     assert_ne!(pk0, pk1);
25     assert_ne!(sk0, sk1);
26 }
27
28 #[test]
29 fn indcpa_same_key() {
30     let mut sk0 = [0u8; KYBER_INDCPA_SECRETKEYBYTES];
31     let mut pk0 = [0u8; KYBER_INDCPA_PUBLICKEYBYTES];
32
33     let mut sk1 = [0u8; KYBER_INDCPA_SECRETKEYBYTES];
34     let mut pk1 = [0u8; KYBER_INDCPA_PUBLICKEYBYTES];
35
36     let mut rand = [0u8; KYBER_SYMBYTES];
37     let mut rng = ChaChaRng::from_entropy();
38     rng.fill_bytes(&mut rand);
39
40
41     indcpa::keygen(&mut pk0, &mut sk0, &rand);
42     indcpa::keygen(&mut pk1, &mut sk1, &rand);
43
44     assert_eq!(pk0, pk1);
45     assert_eq!(sk0, sk1);
46 }
47
48 #[test]
49 fn indcpa_enc_diff_keys() {
50     let mut ctp0 = [0u8; KYBER_CIPHERTEXTBYTES];
51     let mut msg = [0u8; KYBER_INDCPA_MSGBYTES];
52
53     let mut ctp1 = [0u8; KYBER_CIPHERTEXTBYTES];
54
55     let mut sk = [0u8; KYBER_INDCPA_SECRETKEYBYTES];
56     let mut pk = [0u8; KYBER_INDCPA_PUBLICKEYBYTES];
57
```

```

58     let mut rand = [0u8; KYBER_SYMBYTES];
59     let mut rng = ChaChaRng::from_entropy();
60     rng.fill_bytes(&mut rand);
61
62     // gen keys
63     indcpa::keygen(&mut pk, &mut sk, &rand);
64
65     let mut pk_wrong = pk.clone();
66     pk_wrong[0] = !pk_wrong[0]; // flip first byte public key
67
68     //init msg with random
69     rng.fill_bytes(&mut msg);
70
71     indcpa::enc(&mut ctp0, &msg, &pk, &rand);
72     indcpa::enc(&mut ctp1, &msg, &pk_wrong, &rand);
73
74     assert_ne!(ctp0, ctp1);
75 }
76
77 #[test]
78 fn indcpa_enc_diff_msg() {
79     let mut ctp0 = [0u8; KYBER_CIPHERTEXTBYTES];
80     let mut msg = [0u8; KYBER_INDCPA_MSGBYTES];
81
82     let mut ctp1 = [0u8; KYBER_CIPHERTEXTBYTES];
83
84     let mut sk = [0u8; KYBER_INDCPA_SECRETKEYBYTES];
85     let mut pk = [0u8; KYBER_INDCPA_PUBLICKEYBYTES];
86
87     let mut rand = [0u8; KYBER_SYMBYTES];
88     let mut rng = ChaChaRng::from_entropy();
89     rng.fill_bytes(&mut rand);
90
91     // gen keys
92     indcpa::keygen(&mut pk, &mut sk, &rand);
93
94     //init msg with random
95     rng.fill_bytes(&mut msg);
96
97     let mut wrong_msg = msg.clone();

```

```

98     wrong_msg[0] = !wrong_msg[0]; // flip first byte public key
99
100     indcpa::enc(&mut ctp0, &msg ,&pk,&rand);
101     indcpa::enc(&mut ctp1, &wrong_msg ,&pk,&rand);
102
103     assert_ne!(ctp0,ctp1);
104 }
105
106 #[test]
107 fn indcpa_dec_wrong_secret_key() {
108     let mut ctp = [0u8;KYBER_CIPHERTEXTBYTES];
109     let mut msg = [0u8;KYBER_INDCPA_MSGBYTES];
110     let mut msg_dec0 = [0u8;KYBER_INDCPA_MSGBYTES];
111     let mut msg_dec1 = [0u8;KYBER_INDCPA_MSGBYTES];
112
113     let mut sk = [0u8;KYBER_INDCPA_SECRETKEYBYTES];
114     let mut pk = [0u8;KYBER_INDCPA_PUBLICKEYBYTES];
115
116     let mut rand = [0u8;KYBER_SYMBYTES];
117     let mut rng = ChaChaRng::from_entropy();
118     rng.fill_bytes(&mut rand);
119
120     //init msg with random
121     rng.fill_bytes(&mut msg);
122
123     // gen keys and encrypt
124     indcpa::keygen(&mut pk, &mut sk,&rand);
125     indcpa::enc(&mut ctp, &mut msg ,&pk,&rand);
126
127     let mut wrong_sk = sk.clone();
128     wrong_sk[0] = !wrong_sk[0]; // flip first byte
129
130     indcpa::dec(&mut msg_dec0, &ctp,&sk);
131     indcpa::dec(&mut msg_dec1, &ctp,&wrong_sk);
132
133     assert_ne!(msg_dec0,msg_dec1);
134 }
135
136 #[test]
137 fn indcpa_test_pke() {

```

```
138     let mut ctp = [0u8; KYBER_CIPHERTEXTBYTES];
139     let mut msg = [0u8; KYBER_INDCPA_MSGBYTES];
140     let mut msg_dec = [0u8; KYBER_INDCPA_MSGBYTES];
141
142     let mut sk = [0u8; KYBER_INDCPA_SECRETKEYBYTES];
143     let mut pk = [0u8; KYBER_INDCPA_PUBLICKEYBYTES];
144
145     let mut rand = [0u8; KYBER_SYMBYTES];
146     let mut rng = ChaChaRng::from_entropy();
147     rng.fill_bytes(&mut rand);
148
149     //init msg with random
150     rng.fill_bytes(&mut msg);
151
152     // gen keys and encrypt
153     indcpa::keygen(&mut pk, &mut sk, &rand);
154     indcpa::enc(&mut ctp, &msg, &pk, &rand);
155
156     indcpa::dec(&mut msg_dec, &ctp, &sk);
157
158     assert_eq!(msg, msg_dec);
159 }
```

Listing 42: Tests for the IND-CPA functions

Appendix E

Benchmark code

```
1 use criterion::{criterion_group, criterion_main, Criterion};
2 use rjkyber::params::*;
3 use rjkyber::*;
4 use rand::{prng::ChaChaRng, RngCore};
5
6 // generate random bytes using always the same seed
7 fn gen_random(v: &mut[u8]) {
8     let mut rng = ChaChaRng::new_unseeded();
9     rng.fill_bytes(v);
10 }
11
12 // keygen functions
13
14 fn bench_kem_keypair(c: &mut Criterion) {
15     let mut sk0 = [0u8; KYBER_SECRETKEYBYTES];
16     let mut pk0 = [0u8; KYBER_PUBLICKEYBYTES];
17     let mut rand = [0u8; 2*KYBER_SYMBYTES];
18     gen_random(&mut rand);
19
20     c.bench_function("IND-CCA KEM Keypair",
21         |b| b.iter(|| kem::keygen(&mut pk0, &mut sk0, &rand))
22     );
23 }
24
25 fn bench_indcpa_keypair(c: &mut Criterion) {
26     let mut sk0 = [0u8; KYBER_INDCPA_SECRETKEYBYTES];
27     let mut pk0 = [0u8; KYBER_INDCPA_PUBLICKEYBYTES];
```

```

28     let mut rand = [0u8; KYBER_SYMBYTES];
29     gen_random(&mut rand);
30
31     c.bench_function("IND-CPA PKE Keypair",
32         |b| b.iter(|| indcpa::keygen(&mut pk0, &mut sk0, &rand))
33     );
34 }
35
36 fn bench_default_keygen_from(c: &mut Criterion) {
37     let mut rand = [0u8; 2*KYBER_SYMBYTES];
38     gen_random(&mut rand);
39
40     c.bench_function("Default Keygen",
41         |b| b.iter(|| keygen_from(&rand))
42     );
43 }
44
45 fn bench_default_keygen(c: &mut Criterion) {
46     c.bench_function("Default Keygen with prng",
47         |b| b.iter(|| keygen())
48     );
49 }
50
51 fn bench_keygen_compare(c: &mut Criterion) {
52     let mut group = c.benchmark_group("Keygen compare");
53
54     // generate random
55     let mut rand = [0u8; 2*KYBER_SYMBYTES];
56     gen_random(&mut rand);
57
58     let mut pk = [0u8; KYBER_PUBLICKEYBYTES];
59     let mut sk = [0u8; KYBER_SECRETKEYBYTES];
60
61
62     group.bench_function("Jasmin Binding",
63         |b| b.iter(|| kem::keygen(&mut pk, &mut sk, &rand))
64     );
65
66     group.bench_function("High level Rust",
67         |b| b.iter(|| keygen_from(&rand))

```

```

68     );
69 }
70
71 // enc functions
72
73 fn bench_kem_encaps(c: &mut Criterion) {
74     let mut ctp = [0u8; KYBER_CIPHERTEXTBYTES];
75     let mut shkp = [0u8; KYBER_SSBYTES];
76     let mut rand = [0u8; 2*KYBER_SYMBYTES];
77     gen_random(&mut rand);
78
79     let keypair = keygen_from(&rand);
80
81     c.bench_function("IND-CCA KEM Encaps",
82         |b| b.iter(|| kem::enc(
83             &mut ctp, &mut shkp, &keypair.public_key, &rand)
84         )
85     );
86 }
87
88 fn bench_indcpa_enc(c: &mut Criterion) {
89     let mut ctp = [0u8; KYBER_CIPHERTEXTBYTES];
90     let mut msg = [0u8; KYBER_INDCPA_MSGBYTES];
91
92     // generate keys
93     let mut sk0 = [0u8; KYBER_INDCPA_SECRETKEYBYTES];
94     let mut pk0 = [0u8; KYBER_INDCPA_PUBLICKEYBYTES];
95     let mut rand = [0u8; KYBER_SYMBYTES];
96     gen_random(&mut rand);
97     indcpa::keygen(&mut pk0, &mut sk0, &rand);
98
99
100     c.bench_function("IND-CPA PKE Enc",
101         |b| b.iter(|| indcpa::enc(
102             &mut ctp, &mut msg, &pk0, &rand)
103         )
104     );
105 }
106
107 fn bench_default_encaps_from(c: &mut Criterion) {

```

```

108     let mut rand = [0u8; 2*KYBER_SYMBYTES];
109     gen_random(&mut rand);
110     let keypair = keygen_from(&rand);
111
112     c.bench_function("Default Encaps", |b| b.iter(
113         || encaps_from(&keypair.public_key, &rand)
114     ));
115 }
116
117 fn bench_default_encaps(c: &mut Criterion) {
118     let mut rand = [0u8; 2*KYBER_SYMBYTES];
119     gen_random(&mut rand);
120     let keypair = keygen_from(&rand);
121
122     c.bench_function("Default Encaps with prng",
123         |b| b.iter(|| encaps(&keypair.public_key))
124     );
125 }
126
127 fn bench_encaps_compare(c: &mut Criterion) {
128     let mut group = c.benchmark_group("Encaps compare");
129
130     // generate random
131     let mut rand = [0u8; 2*KYBER_SYMBYTES];
132     gen_random(&mut rand);
133
134     let k = keygen_from(&rand);
135
136     let mut ctp = [0u8; KYBER_CIPHERTEXTBYTES];
137     let mut shkp = [0u8; KYBER_SSBYTES];
138
139     group.bench_function("Jasmin Binding",
140         |b| b.iter(|| kem::enc(
141             &mut ctp, &mut shkp, &k.public_key, &rand)
142         ));
143
144     group.bench_function("High level Rust",
145         |b| b.iter(|| encaps_from(&k.public_key, &rand))
146     );
147 }

```

```

148 }
149
150 // dec functions
151
152 fn bench_kem_decaps(c: &mut Criterion) {
153     let mut shkp = [0u8; KYBER_SSBYTES];
154     let mut rand = [0u8; 2*KYBER_SYMBYTES];
155     gen_random(&mut rand);
156
157     let k = keygen_from(&rand);
158     let e = encaps_from(&k.public_key, &rand);
159
160     c.bench_function("IND-CCA KEM Decaps",
161         |b| b.iter(|| kem::dec(
162             &mut shkp, &e.ciphertext, &k.secret_key)
163         ));
164 }
165
166
167 fn bench_indcapa_dec(c: &mut Criterion) {
168     let mut ctp = [0u8; KYBER_CIPHERTEXTBYTES];
169     let mut msg = [0u8; KYBER_INDCPA_MSGBYTES];
170     let mut msg_dec = [0u8; KYBER_INDCPA_MSGBYTES];
171
172     // generate keys
173     let mut sk = [0u8; KYBER_INDCPA_SECRETKEYBYTES];
174     let mut pk = [0u8; KYBER_INDCPA_PUBLICKEYBYTES];
175     let mut rand = [0u8; KYBER_SYMBYTES];
176     gen_random(&mut rand);
177
178     // gen keys and encrypt
179     indcpa::keygen(&mut pk, &mut sk, &rand);
180     indcpa::enc(&mut ctp, &mut msg, &pk, &rand);
181
182
183     c.bench_function("IND-CPA PKE Dec", |b|
184         b.iter(|| indcpa::dec(&mut msg_dec, &ctp, &sk))
185     );
186 }
187

```

```

188 fn bench_default_decaps(c: &mut Criterion) {
189     let mut rand = [0u8; 2*KYBER_SYMBYTES];
190     gen_random(&mut rand);
191     let k = keygen_from(&rand);
192     let e = encaps_from(&k.public_key, &rand);
193
194     c.bench_function("Default Decaps",
195         |b| b.iter(|| decaps(&k.secret_key, &e.ciphertext))
196     );
197 }
198
199 fn bench_decaps_compare(c: &mut Criterion) {
200     let mut group = c.benchmark_group("Decaps compare");
201
202     // generate random
203     let mut rand = [0u8; 2*KYBER_SYMBYTES];
204     gen_random(&mut rand);
205
206     let k = keygen_from(&rand);
207     let e = encaps_from(&k.public_key, &rand);
208
209     let mut shkp = [0u8; KYBER_SSBYTES];
210
211     group.bench_function("Jasmin Binding",
212         |b| b.iter(|| kem::dec(
213             &mut shkp, &e.ciphertext, &k.secret_key)
214         )
215     );
216
217     group.bench_function("High level Rust",
218         |b| b.iter(|| decaps(&k.secret_key, &e.ciphertext))
219     );
220 }
221
222
223 criterion_group!(benches,
224     bench_kem_keypair, bench_indcpa_keypair,
225     bench_default_keygen_from, bench_default_keygen,
226     bench_keygen_compare, bench_kem_encaps,
227     bench_indcpa_enc, bench_default_encaps_from,

```

```

228     bench_default_encaps, bench_encaps_compare,
229     bench_kem_decaps, bench_indcapa_dec,
230     bench_default_decaps, bench_decaps_compare
231 );
232 criterion_main!(benches);

```

Listing 43: Benchmark code for RjKyber

```

1  use criterion::{criterion_group, criterion_main, Criterion};
2  use pqc_kyber;
3  use rjkyber;
4  use rjkyber::params::*;
5
6
7  fn bench_pqc_keygen(c: &mut Criterion) {
8      let mut group = c.benchmark_group("Keygen compare pqc_kyber");
9
10     let mut rand = [0u8; 2*KYBER_SYMBYTES];
11     let mut rng = rand::thread_rng();
12     rng.fill_bytes(&mut rand);
13
14
15     group.bench_function("pqc_kyber",
16         |b| b.iter(|| pqc_kyber::keypair(&mut rng))
17     );
18     group.bench_function("rjkyber",
19         |b| b.iter(|| rjkyber::keygen_from(&mut rand))
20     );
21 }
22
23 fn bench_pqc_encaps(c: &mut Criterion) {
24     let mut group = c.benchmark_group("Encaps compare pqc_kyber");
25
26     let mut rand = [0u8; 2*KYBER_SYMBYTES];
27     let mut rng = rand::thread_rng();
28     rng.fill_bytes(&mut rand);
29
30     let keys = rjkyber::keygen_from(&rand);
31
32     group.bench_function("pqc_kyber",

```

```

33         |b| b.iter(|| pqc_kyber::encapsulate(
34             &keys.public_key, &mut rng)
35         )
36     );
37     group.bench_function("rjkyber",
38         |b| b.iter(|| rjkyber::encaps_from(
39             &keys.public_key, &rand)
40         )
41     );
42 }
43
44 fn bench_pqc_decaps(c: &mut Criterion) {
45     let mut group = c.benchmark_group("Decaps compare pqc_kyber");
46
47     let mut rand = [0u8; 2*KYBER_SYMBYTES];
48     let mut rng = rand::thread_rng();
49     rng.fill_bytes(&mut rand);
50
51     let keys = rjkyber::keygen_from(&rand);
52     let enc = rjkyber::encaps_from(&keys.public_key, &rand);
53
54     group.bench_function("pqc_kyber",
55         |b| b.iter(|| pqc_kyber::decapsulate(
56             &enc.ciphertext, &keys.secret_key)
57         )
58     );
59     group.bench_function("rjkyber",
60         |b| b.iter(|| rjkyber::decaps(
61             &keys.secret_key, &enc.ciphertext)
62         )
63     );
64 }
65
66 criterion_group!(
67     benches, bench_pqc_keygen,
68     bench_pqc_encaps, bench_pqc_decaps
69 );
70 criterion_main!(benches);

```

Listing 44: Benchmark code to compare RjKyber and pqc_kyber

Appendix F

Additional Benchmarks

F.1 RjKyber benchmarks for Kyber768 AVX2

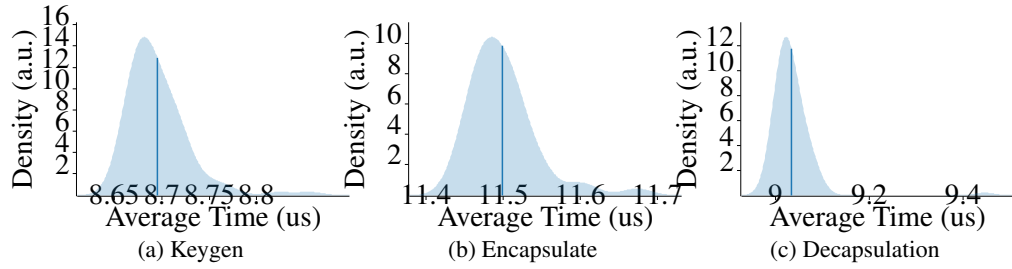


Figure F.1: High level interface functions

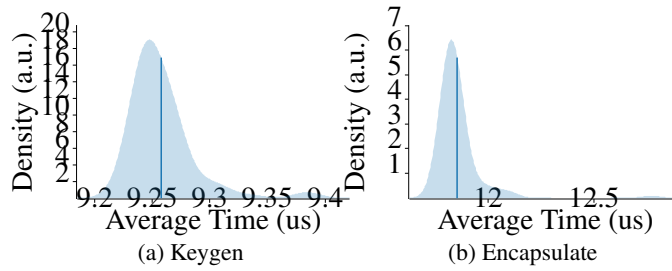


Figure F.2: High level interface functions with automatic random number generation

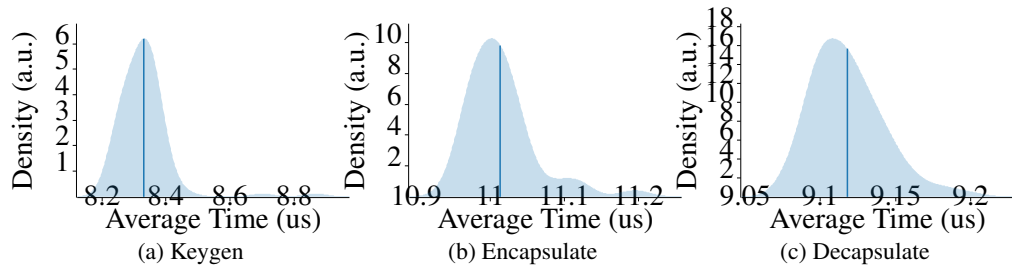


Figure F.3: Rust bindings for Jasmin IND-CCA KEM functions

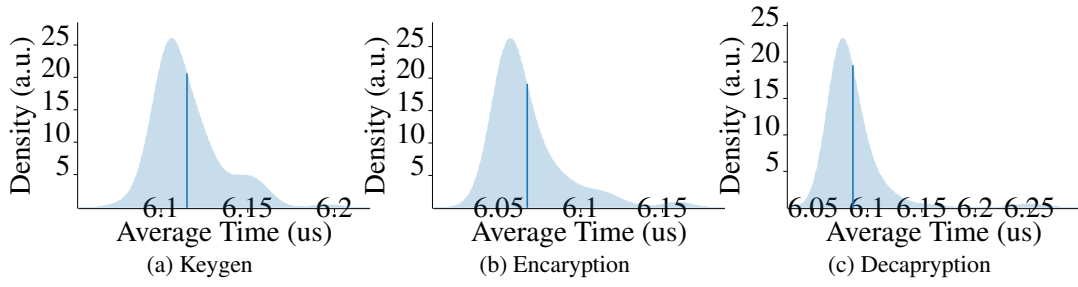


Figure F.4: Rust bindings for Jasmin IND-CPA PKE functions

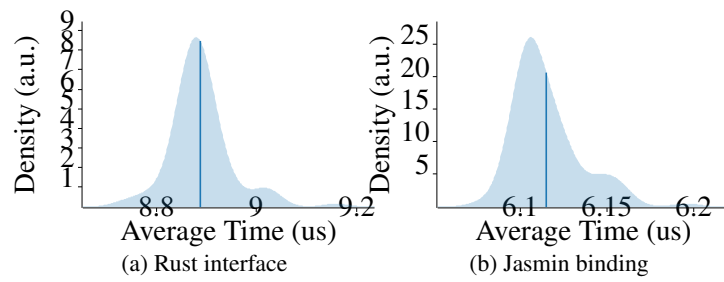


Figure F.5: Comparison between RjKyber high-level keygen and the corresponding Jasmin binding

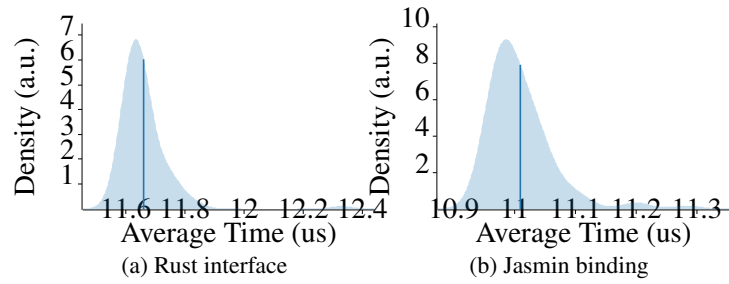


Figure F.6: Comparison between RjKyber high-level encapsulation and the corresponding Jasmin binding

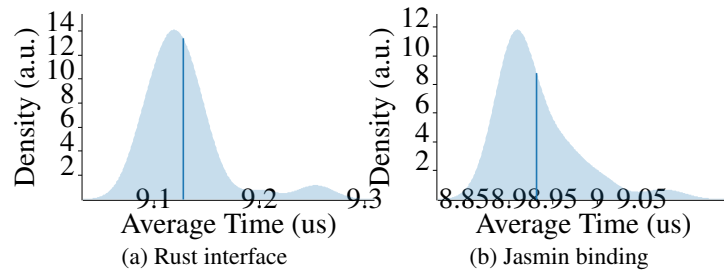


Figure F.7: Comparison between RjKyber high-level decapsulation and the corresponding Jasmin binding

F.2 RjKyber benchmarks for Kyber768 Reference

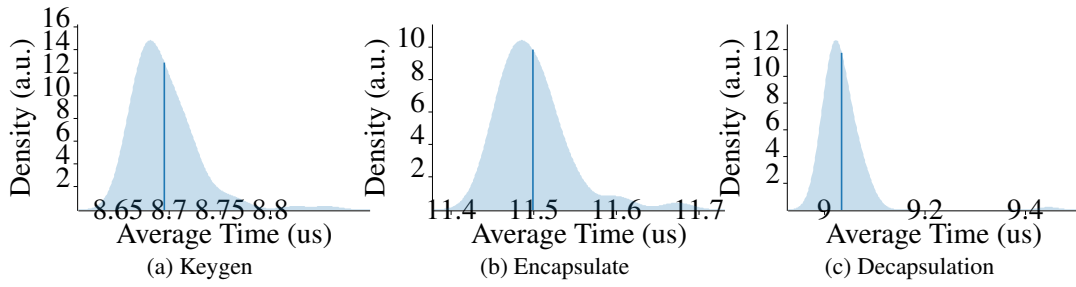


Figure F.8: High level interface functions

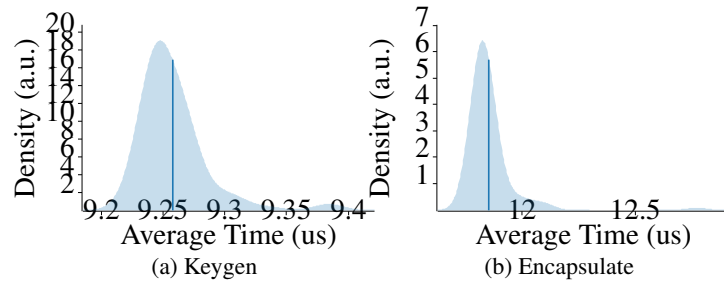


Figure F.9: High level interface functions with automatic random number generation

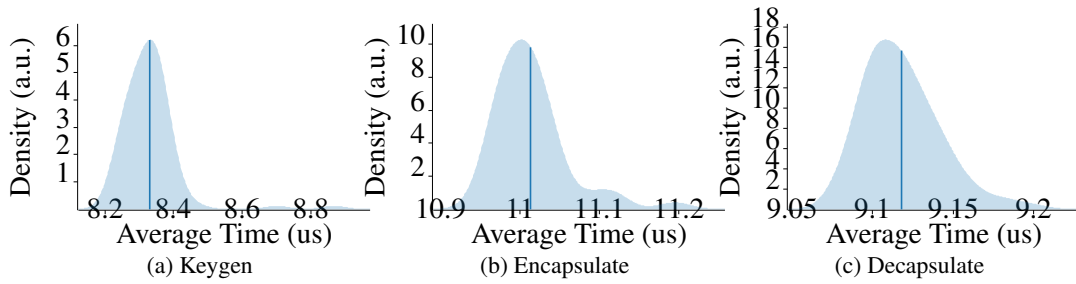


Figure F.10: Rust bindings for Jasmin IND-CCA KEM functions

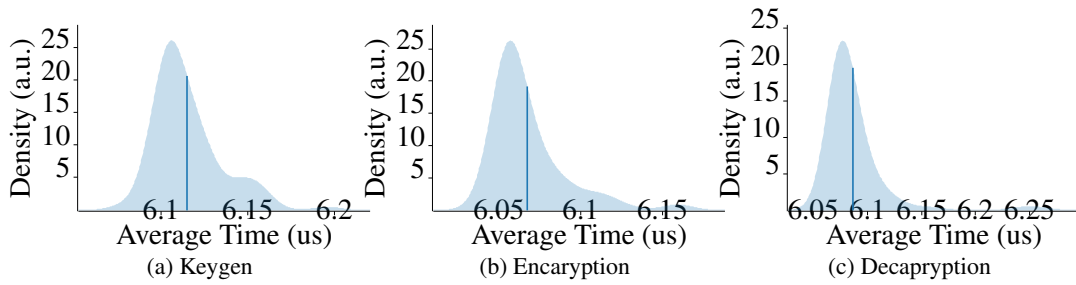


Figure F.11: Rust bindings for Jasmin IND-CPA PKE functions

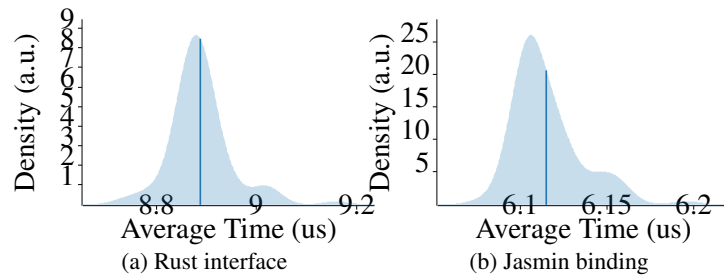


Figure F.12: Comparison between RjKyber high-level keygen and the corresponding Jasmin binding

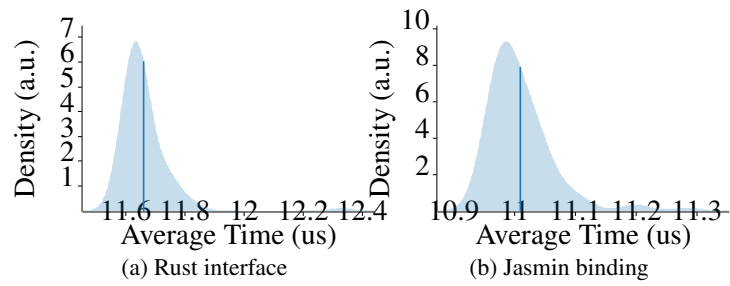


Figure F.13: Comparison between RjKyber high-level encapsulation and the corresponding Jasmin binding

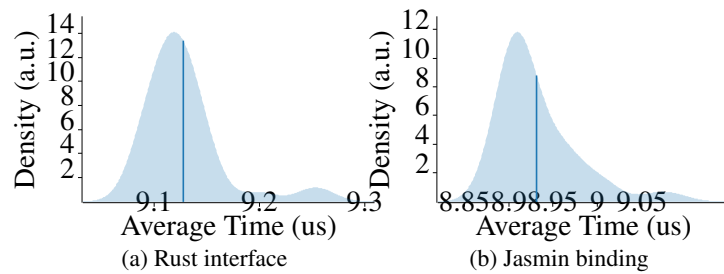


Figure F.14: Comparison between RjKyber high-level decapsulation and the corresponding Jasmin binding

F.3 Comparison between RjKyber and pqc_kyber using Kyber768 AVX2

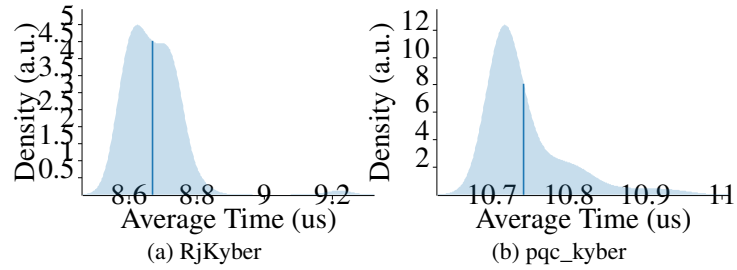


Figure F.15: Comparison between RjKyber key generation and the corresponding pqc_kyber function

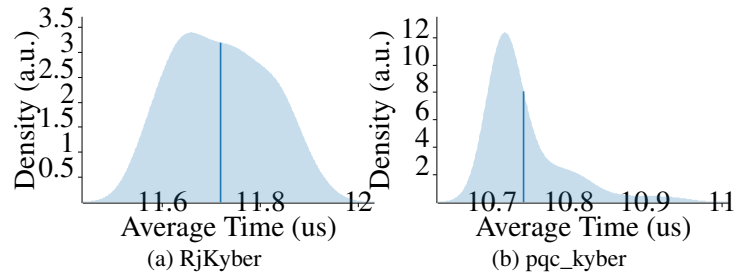


Figure F.16: Comparison between RjKyber encapsulation and the corresponding pqc_kyber function

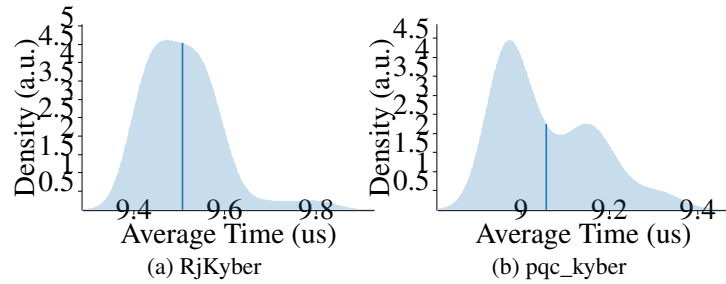


Figure F.17: Comparison between RjKyber decapsulation and the corresponding pqc_kyber function