HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch

Département des Technologie de l'information et de la communication (TIC)
Filière Télécommunications
Orientation Sécurité de l'information

Travail de Bachelor

Vers un TLS post-quantique

Étudiant	**Filipe Fortunato**
Enseignant responsable	Prof. Alexandre Duc
Année académique	2019-2020

Yverdon-les-Bains, le 31 Juillet, 2020

Département des Technologie de l'information et de la communication (TIC)
Filière Télécommunications
Orientation Sécurité de l'information
Étudiant : Filipe Fortunato
Enseignant responsable : Prof. Alexandre Duc

Travail de Bachelor 2019-2020

Vers un TLS post-quantique

Résumé publiable

TLS is currently one of the most widely used Internet security protocols, with a vast range of applications, such as when browsing the internet or logging into one's bank accounts from any device. It is also the system providing the lockbox in a web browser's address bar, claiming that a given connection is secure, ensuring that only the connected server can read the transmitted data and that it is impossible to impersonate the server.

With the advent of quantum computing and increasingly powerful quantum computing platforms, an extremely devastating attack against the cryptographic hearth of the protocol is ever closer to being actualizable. That attack would completely wipe out the security from the currently in use TLS implementations.

That is one of the reasons as to why the American National Institute of Standards and Technology launched three years ago a contest to find cryptographic algorithms that would be able to withstand both quantum and conventional threats. Now, three years later, the competition is nearing its end, and only a handful of algorithms remain.

For this thesis, the decision to implement the Crystals-Kyber and Crystals-Dilithium algorithms inside the BearSSL TLS library was taken. The implementation entailed adapting and including the algorithms into the library. In-depth extensions and modifications of the TLS protocol were also required to conform with the challenges from the use of the selected quantum-resistant algorithms, such as much larger key, signatures, and ciphertext sizes.

The results of this thesis are a quantum-resistant TLS library, with very competitive performances compared to the original, along with a significant degree of potential code optimizations remaining, that could likely speed up the library by up to five-fold.

Étudiant :	Date et lieu :	Signature :
Filipe Fortunato	.....	.....
Enseignant responsable :	Date et lieu :	Signature :
Prof. Alexandre Duc	.....	.....

Préambule

Ce travail de Bachelor (ci-après TB) est réalisé en fin de cursus d'études, en vue de l'obtention du titre de Bachelor of Science HES-SO en Ingénierie.

En tant que travail académique, son contenu, sans préjuger de sa valeur, n'engage ni la responsabilité de l'auteur, ni celles du jury du travail de Bachelor et de l'Ecole.

Toute utilisation, même partielle, de ce TB doit être faite dans le respect du droit d'auteur.

HEIG-VD

Vincent Peiris
Chef de département TIC

Yverdon-les-Bains, le 31 Juillet 2020

Authentification

Le soussigné, Filipe Fortunato, atteste par la présente avoir réalisé ce travail et n'avoir utilisé aucune autre source que celles expressément mentionnées.

Yverdon-les-bains, le 31 Juillet 2020

Filipe Fortunato

Cahier des charges

Résumé du problème

L'arrivée potentielle des ordinateurs quantiques met à mal les systèmes cryptographiques à clef publique utilisés de nos jours. La communauté cryptographique se prépare avec l'appel du NIST à un standard post-quantique. Néanmoins, l'industrie risque de devoir attendre plusieurs années avant de pouvoir utiliser ce standard.

Le protocole TLS/SSL est particulièrement utilisé de nos jours. La plupart de nos communications sur internet utilisent ce protocole.

Cahier des charges

Tout d'abord, une étude des divers algorithmes de chiffrement et de signature post-quantique sera effectuée. L'algorithme choisi devra être le plus prometteur possible (concernant la standardisation du NIST) mais devra aussi être performant et posséder une licence utilisable.

Une fois l'algorithme choisi, il devra être intégré a une ou plusieurs librairies TLS/SSL existantes.Pour ce faire, les librairies les plus connues (OpenSSL/BoringSSL/BearSSL/...) seront d'abord étudiées afin d'en sélectionner les plus intéressantes.

La couche post-quantique devra ensuite y être ajoutée. Le logiciel résultant devra être stable, sûr (pas de side-channel attacks ou autres vulnérabilités, ...), et facile d'utilisation. A noter que ce travail de Bachelor implique de rendre toute l'infrastructure X509 de certificats post-quantique.

Livrables

Les délivrables seront les suivants :

- Un rapport concernant l'étude des divers algorithmes post-quantiques

- Une étude des diverses librairies TLS/SSL
- Une librairie TLS/SSL post-quantique (par exemple sous forme de patch)

Contents

Chapter 1

Introduction

1.1 General Context

The potential advent of quantum computers will inevitably render currently in use public key cryptography systems obsolete. At the time of this thesis, the global cryptography community is taking steps, such as the NIST standardisation process, towards developing quantum resistant cryptographic standards. Nonetheless, the industry will most likely have to wait for multiple years before such standards being usable in any sort of practical manner.

The TLS/SSL protocol is extremely and widely in use nowadays. Most Internet communications rely on that critical piece of software to guarantee their both their security and confidentiality.

It should be noted that this thesis is part of a series of three thesis currently in progress (the other two being "Vers un SSH post-quantique" and "Vers un VPN post-quantique"), as such, it should be noted that for most data collection and analysis related to cryptographic algorithms will most likely be similar between the three thesis.

1.2 Methodology

This thesis will consist of attempting to implement a quantum resistant cryptographic specification into an existing SSL/TLS Library as well as any required TLS protocol augmentations and/or changes.

In order to do so, multiple quantum-resistant encryption and signature algorithms will need to be analysed. The selected algorithms should be as promising as possible from a NIST standardisation standpoint, but also be performant and be licensed under usable terms.

Once the quantum-resistant algorithms selected, they should be integrated to one or multiple

existing SSL/TLS libraries. In order to do so, the most widely known libraries shall be taken into consideration (OpenSSL/BoringSSL/BearSSL/...) and the most interesting (from this thesis' standpoint) shall be selected.

Finally, a quantum-resistant layer shall be introduced to the selected library/libraries. The ensuing software should be stable, secure (no side-channel or similar vulnerabilities shall be introduced into it during this thesis' process) and simple to use. This thesis implies ensuring that the whole X509 certificate infrastructure is made quantum-resistant.

1.3 Thesis Summary

At the issue of a TLS library selection process, taking into account both previous similar projects and library functionalities, *BearSSL* was chosen as the TLS Library to be used in this thesis, due to its security centric design, as well as being written in modern C and not having had any project aiming to implement quantum-resistant cryptography. For further details about this choice, as well as the full list of TLS libraries considered, one can refer to Section 4.

1.3.1 Cryptographic Algorithm Selection

Due to the advice and direction of the supervising professor, A. Duc, It was decided early on to focus in the direction of lattice-based cryptographic algorithms, which would explain the preliminary selection containing a vast majority of such algorithms.

During the cryptographic algorithms analysis, the main encryption algorithms that were thoroughly analysed were the following:

- Crystals-Kyber
- NewHope (CCA variant)
- NTRU Prime
- NTRU

NTRU ended up being discarded rather early due to key generation speed issues, even tho it had the fastest encryption/decryption speed of all tested algorithms. After some more detailed testing, NTRU Prime was also discarded due to similar key generation speed issues, which left us with both Crystals-Kyber and NewHope. For the detailed analysis process, please refer to Section 5.

Prior to being able to chose a final algorithm, as both Crystals-Kyber and NewHope had very similar performances and memory footprints, a signature algorithm analysis was undertaken.

The following signature algorithms were considered for a thorough analysis:

- Crystals-Dilithium
- Falcon
- Sphincs$^+$

The results of the selection process were as follows, Sphincs$^+$ was at least two orders of magnitude slower that either of the other algorithms for both key generation, signature creation and verification, as well as a much larger signature size, which is a dealbreaker for TLS related applications.

Falcon and Dilithium, on the other hand, were much closer in terms of both key generation and signature creation/verification speed. However, due to the uncertainty of Falcon's Intellectual Property claims, which only grants a broad and widely usable license at the condition of being the winner of the standardisation call, Falcon had to be considered unsuitable for usage in such a thesis.

This resulted in **Crystals-Dilithium** being chosen as the signature algorithm to be implemented in this thesis.

Due to the selection of Crystals-Dilithium as the signature algorithm, and because of an assumption that since Crystals-Kyber was created by the same group, using the same class of quantum-resistant cryptography, that much of the code, logic and structures could end up being reused, led to the decision of choosing **Crystals-Kyber** as the encryption algorithm for this thesis.

This however, as can be seen in the implementation section 6.2, while the assumption was theorically sound, due to some fundamental design decisions within the reference implementations of *Kyber* and *Dilithium*, in practice, it was a much more time consuming task which was not pursued due to a lack of time.

1.3.2 Implementation

An initial overview of the TLS 1.2 handshake protocol was provided in Figure 6.1, which also highlight crital sections that will need to be reworked for the implementation of the Kyber and Dilithium crypto systems.

Then, the precise method for integrating the Kyber crypto system into a Diffie-Hellman like protocol was described in Figure 6.2.

Following that, an exhaustive list of the TLS 1.2 protocol changes and additions was drawn up, going over a new message extension required for supporting the Kyber key exchange construction, the new cryptographic identifiers required for most messages as well as new object identifiers required for the x509 standard.

A detailed overview of the Kyber and Dilithium implementations in BearSSL was then provided, going over all the code unification issues highlighted in the cryptographic algorithm selection, along with the implemented memory usage optimisations in order to better respect the BearSSL design philosophy and the state of integration into the automated testing frameworks of BearSSL.

Finally, a detailed description of the BearSSL quantum-resistant TLS implementation was provided, along with all the subtleties in handling all the aspects of the newly created TLS protocol, in particular with some memory related issues. This description also went over the major implementation limitations and hacks that had to be set in place.

1.3.3 Results

As can be seen in Table 7.1 and Figure 7.1, despite some performance anomalies, the resulting TLS handshake speed is more than acceptable, with roughly the same performance as one of the best conventional TLS cipher suites. The Kyber and Dilithium speeds when integrated to BearSSL were roughly 20 to 30% slower than the reference implementation, which was a quite expected result due to a lower optimisation potential ceiling with BearSSL's implementation.

Due to the quantum resistant key, cipher and signature sizes, it is not surprising to see handshake sizes roughly an order of magnitude larger than conventional TLS cipher suites.

Finally, when comparing BearSSL's performance with OpenQuantuSafe's quantum-resistance OpenSSL implementation, it can be seen that their code is roughly one order of magnitude faster than this thesis' results. This was not a suprising outcome, however, due to their code making use of optimised implementations as well as a faster TLS protocol revision and more optimisation possibilities.

Chapter 2

Quantum computing and Cryptography

This chapter shall cover some light background information about quantum computing and its effects and impact on widely used cryptography. Quantum resistant cryptographic systems are then introduced, along with the semantical differences in their usage for the purposes of key exchange schemes.

2.1 Consequences of the advent of quantum computing

Despite this thesis not going in details about quantum computing, it is nonetheless important to describe its history in broad strokes. With the development of quantum computing in the early 1980s, a whole new computing paradigm was created. Using this new computation paradigm, an efficient integer factorisation algorithm was developped by Peter Shor in the early 1990s (Reffered to as "Shor's algorithm" in this thesis) [1].

This development is highly relevant. Since major families of public key cryptography and signing are based upon the fundamental difficulty of factorizing large semiprime numbers or solving the "Discrete logarithm" family of problems in the case of RSA/ElGamal and ECDH/ECDSA respectively, among others.

Using Shor's algorithm, one would be able to trivialise the difficulty of both of those problems, thus rendering those cryptographic schemes useless.

Thankfully, Shor's algorithm requires the usage a quantum computing platform which can both hold a large amount of qubits as well preventing its inner state to become decoherent. This difficult requirement is the reason for which the above-mentioned cryptographic schemes are still in use.

2.2 Quantum suppremacy

Earlier, in late 2019, Google has publicly claimed to have reached quantum supremmacy [2], that is, they were able to compute using a quantum processor something that would be impractical to do using a traditional computer.

In the case of Google's claim, they estimated having reached quantum supremacy by comparing their quantum processor to a state-of-the-art traditional computer in the task of sampling the output of a pseudo-random quantum circuit.

While their claims and methodologies have since been challenged by other groups, such as IBM [3], nonetheless, it stands that it can be used to highlight a need for a speedy standardisation and implementation of a quantum resistant public key cryptographic encryption and signing scheme.

2.3 Quantum resistant cryptography

Due to the inherant weakness of the large number factorization and discrete logarithm problems when quantum computing enters the picture, there is a need to move away from those hard problems.

Current efforts toward quantum resistant asymmetric cryptography reside within five classes of cryptography systems [4]:

- **Hash-based cryptography**

  This system family relies upon a standard cryptographic hash function to perform digital signatures over files.

  It does not support message encryption, but exclusively message signatures.

  Hash-based cryptography relies entierly upon the security of the underlying hash function, as such, makes for a rather versatile signature scheme.

- **Code-based cryptography**

  This system family relies upon the use of error-correcting codes to provide encryption and digital signatures.

  However, code-based cryptographic schemes tend to have comparatively larger key sizes than other families, due to security issues arising when using reduced key sizes.

  Code-based cryptography relies upon the difficulty of decoding a linear code, which has been demonstrated to be NP-hard [5], thus suitable for cryptography.

- **Lattice-based cryptography**

This system family relies upon the use of lattices to provide encryption and digital signatures.

Most practical latice based cryptographic schemes rely upon half a dozen mathematical problems, whith the main ones being the Learning With Errors and its variants (Ring-LWE, Learning With Rounding, Ring-LWR), Shortest Vector Problem and Closest Vector Problem, which are known to be hard problems [6, 7, 8].

- **Multivariate-quadratic-equations cryptography**

  This system family relies upon the use of second-degree multivariate equations to provide both encryption and digital signatures.

  Its security relies on the hard problem of solving nonlinear equations over finite fields [9], thus is suitable for use in cryptographic applications.

- **Supersingular elliptic curve isogeny cryptography**

  This system family relies upon the use of supersingular elliptic curves and isogeny graphs to provide a construction analogous to Diffie-Hellman [10]. This allows for a more straight-forward replacement of existing public key encryption schemes, while still providing forward secrecy, but does not provide a signature scheme, only encryption. However, its inception is relatively much younger than the other families dating from the early 2010s compared to the 1990s, and its security claims have not had nearly as much time to be tested.

2.4 Key-encapsulation vs Diffie-Hellman

In the context of four of the five classes of quantum-resistant cryptography, their application to public key cryptography would rely on key encapsulation to construct a key exchange, with the exception of supersingular isogeny based cryptography, which is able to directly adopt the requirements and properties of Diffie-Hellman based key exchanges.

The main semantical difference between key exchange schemes based upon key encapsulation and the Diffie-Hellman construction is that with Diffie-Hellman, both parties of the exchange will agree upon a common key, where each of them can have an equal impact upon the final common key. In the context of key encapsulation based exchanges, one of the parties will be able to strictly impose an arbitrary key upon the other party.

It is however possible to construct a key exchange using key encapsulation as a primitive which does provide equal contribution to the final common key to both parties, but there is a major drawback to doing so. With a Diffie-Hellman based key exchange, it is possible to exchange a shared secret between two parties without prior communications in as little as two messages, one from each party, whereas with the construction abovementioned, the least amount of messages required to achieve the same result (I.E. equal impact upon the

final shared secret by either parties without prior knowledge of each other) are three, which prevents the establishment of shared secrets in a single round trip communication [11].

For further details about the key-encapsulation based key exchange, refer to Section 6.1.2.

Chapter 3

Existing TLS Post Quantum implementations

Prior to being able to decide upon a post quantum cryptographic suite and TLS Library to work on in this thesis, it can be useful to explore similar projects, aiming to implement post-quantum cryptographic primitives to a TLS library.

3.1 Analysis of existing solutions

The following are projects aiming to implement quantum resistant cryptography inside of TLS:

- **Cloudflare's post-quantum TLS experiments**[1]

  In mid-2019, Cloudflare decided to partner with Google to implement both lattice based and supersingular elliptic curve isogeny cryptography to BoringSSL, a TLS library, in an attempt to measure the real-world latency impact of the usage of post quantum cryptography in a web browsing TLS scenario.

  Cloudflare's experiment made use of the HRSS variant of the NTRU lattice based algorithms for key encapsulation as key exchange, as well as the SIKE isogeny-based encryption algorithm as a drop in replacement to the ECDHE key exchange protocol. It should also be noted that they used NIST level 1 parameter sets, likely in an attempt to reduce the cost impact of using quantum resistant cryptography. Interestingly, thed did not touch upon signature algorithms in their experiments, chosing to focus exclusively onto the key exchange aspect of the TLS handshake.

[1] `https://blog.cloudflare.com/towards-post-quantum-cryptography-in-tls/`

Their results showed that SIKE based key exchanges performed roughly two orders of magnitude slower in general than the curve25519 based X25519 kex exchange algorithm. It was also highlighted that their NTRU-HRSS performance was also roughly an order of magnitude slower than their non-quantum key exchange algorithm, mostly due to the slow key generation speed.

- **Microsoft's post-quantum TLS experiments**[2]

  Since mid-2018, Microsoft has been working along side with the Open Quantum Safe team to implement both lattice based and supersingular elliptic curve isogeny cryptography, as well as lattice and zero-knowledge proof based digital signature algorithms to OpenSSL, a TLS library.

3.2 Analysis of Open Quantum Safe's software

Open Quantum Safe[3] (refered to as OQS in this thesis) is a group developping a library dedicated to the implementations of various post quantum crpytographic algorithms, both encryption and digital signatures. Most candidates' cryptographic schemes of the ongoing NIST standardisation call are integrated into their library.

They have also integrated their library to both OpenSSL[4], a TLS library, as well as openSSH[5], a SSH software, in an attempt to evaluate the proposed quantum-resistant cryptography schemes.

Their implementation into OpenSSL relies on providing two types of key exchanges and authentication schemes, a purely quantum-resistant based scheme, or a hybrid scheme, making use of both quantum resistant and traditional key exchange schemes at the same time.

The existence of the hybrid scheme is aimed at a potential early deployment of quantum resistant cryptography, prior to the conclusion of the NIST standardisation process. In an attempt to provide some sort of safety net in case the selected quantum resistant cryptography algorithm ends up being vulnerable to some sort of attack, to still enjoy the relative protection of traditional crypto, during the transition period to fully quantum-resistant cryptography.

Another interesting detail to note, is that apart from the isogeny based cartographic scheme, all other quantum-resistant schemes use the default key encapsulation, with the server forcing a key upon the client, due to implementing TLS 1.3 protocol support. Which as is be discussed in the TLS protocol analysis in section 4, can lead to security issues.

[2] `https://www.microsoft.com/en-us/research/project/post-quantum-tls/`
[3] `https://openquantumsafe.org/`
[4] `https://github.com/open-quantum-safe/openssl`
[5] `https://github.com/open-quantum-safe/openssh`

As well as the fact that the OpenSSL Library can be compiled to use libOQS as a shared library (I.E if library updates should be decoupled with OpenSSL updates), this could lead to a potential security issue if an attacker is able to replace the existing library, or able to change the dynamic linking path to point to a stub of the library, which could perform key exchanges with static, compromised keys.

Those two issues combined resulted into the decision of rather than working on the Open Quantum Safe Library, to implement quantum-resistant cryptography into a TLS library for this thesis.

Chapter 4

TLS library selection

TLS/SSL is a protocol and more largely, an infrastructure that aims to provide a confidential and safe communication medium between two parties that potentially never met prior.

This chapter will highlight the specifics of the protocol, along with its two major revisions. Then, a few libraries shall be presented along with the library selection for this thesis. finally, the selected library shall be covered more in detail.

4.1 TLS protocol

The TLS protocol, along with its now long deprecated predecessor, SSL, is a protocol aiming to provide a secure connection between two peers that may never have met prior.

By "secure connection", the protocol aims more precisely to provide an encrypted connection between two peers, while guaranteeing that both peers are exactly who they claim to be, that is that they are properly authenticated. And finally, the protocol guarantees that the communications were not altered in transit.

Those three security features can be seen as the classic trifecta of Confidentiality, Integrity and Authenticity. To provide those security features, the protocol employs a fair amount of cryptographic primitives, including public-key encryption and signatures.

It should be noted that the protocol works in two major steps.

There is first a handshake process, which aims to establish the encrypted communication channel, and authenticate the peers. Then, there is the normal communication process, which employs the encrypted communication channel and provides additional integrity guarantees about the transmitted data.

While the normal communication process employs cryptographic primitives that may be

somewhat affected by the advent of quantum computing, such as symetric encryption schemes with Grover's algorithm, this is outisde the scope of this thesis. Indeed, this thesis is mostly tied to the NIST standardisation process, which is selecting public key encryption and signature schemes. Since such schemes are only in use in the handshake process of the protocol, it is only the handshake process that shall be made to be quantum resistant.

4.2 TLS1.2 vs TLS1.3

Prior to being able to select a TLS library to work with, a requirement analysis must be done on the two main TLS revisions, and it must be ascertained if either holds any advantage or disadvantage in the context of a quantum-resistant cryptography suite implementation.

Although TLS1.3 brought a large amount of changes, among which the pruning of legacy algorithms, forcing all key exchanges to be ephemeral, and much more, there is one change in particular that is very important for this thesis. This important constraint of TLS1.3 over 1.2 is the requirement of 1-RTT key exchange, that is, in a single round trip communication between the server and client, a common encryption key must have been established.

Due to the abovementioned restrictions of key encapsulation based algorithms' usages in key exchange schemes in Section 2.4, it would mean that the usage of quantum-resistant cryptography with TLS1.3 would either require the use of an isogeny based algorithm for their ability to strictly adhere to the requirements of Diffie-Hellman key exchange semantics, or to use directly a key encapsulation mechanism instead of a key exchange construction for TLS 1.3.

This specific trade-off is not one that must be taken lightly. Indeed, if one were to use the key encapsulation mechanisms as the key exchange in the context of a TLS handshake, one would inevitably put all the burden of key agreement upon the server. If said server was to be attacked or to contain a bug in such a way as to render the encapsulated key selection from the server to be predictable, any actor with that knowledge could be able to passively derive the pre-master secret associated with a given TLS handshake. Due to the fact that every other component of the master secret being public, such a decision could lead to devastating consequences.

Considering that two major TLS implementation projects already implemented isogeny based cryptography algorithms, being forced into the same algorithms would also be counterproductive to this thesis, as such, it has been decided to implement quantum-resistant cryptography in TLS version 1.2 in the scope of this project.

4.3 Library selection

In this thesis, the decision has been made to only consider TLS libraries in C or C++. This is mostly preference but also due to lack of experience with other languages that offer sufficient control over memory contents to properly store secrets (such as Rust).

As such, the following libraries among the ones considered deserve closer inspection :

- **OpenSSL**. Despite being the de-facto standard TLS library in use, it can definitely be considered a dated codebase, as well as having had some deep security issues in the past, such as Heartbleed, POODLE or DROWN [12, 13, 14] would still inspire distrust into the code quality from a security standpoint. Also, the fact that there are already projects by major global actors implementing quantum resistant cryptography in Openssl, as seen in Chapter 3, would make this a rather poor choice for this thesis.
- **BoringSSL**. This library aims to fix most of the dated codebase issues associated with OpenSSL, and fix a lot of security decisions [15]. It has nonetheless, like OpenSSL, already had a major project by two major global actors involving the implementation of quantum-resistant cryptography, as such it would most likely not be the most appropriate choice for this thesis.
- **WolfSSL**. This promising library has some very interesting design choices and decisions, such as primarily targeting embedded devices, and is quite widely adopted in the embedded device sector.

  It nonetheless also has a lattice-based quantum-resistant cryptographic scheme built-in since 2014 [16].
- **BearSSL**. This is quite an interesting library. Even tho it has been in development for far less time than the alternatives, with its first release dated late 2016, its development started in the wake of the disastrous vulnerability being disclosed for OpenSSL, as mentioned previously. It has a large focus on both being a secure implementation as well as being compatible with as many platforms as possible.

Mostly due to the fact that it has no such project for implementing a quantum-resistant cryptographic, its high modularity and simplicity of extension, its design decisions as well as its code quality and with the advice of the supervising professor, A. Duc, BearSSL has been selected for use in this thesis.

4.4 BearSSL

This library is rather promising, both from its design and implementation choices. It was built from the ground up in 2014 with security in mind. It boasts a restricted cipher suite

support, choosing to not implement insecure cryptographic algorithms such as DES. Choosing as well to not support legacy protocols such as the previous SSL protocol. It also employs "constant time" cryptography by default for all available algorithms as well as allocation-free code for the whole library.

Its SSL/TLS engine is coded in such a fashion as to avoid as much as possible memory related bugs and to easily support multiple sessions is parallel, in the form of a state machine, using the "T0" custom coding language, developped specifically for this purpose. It is also highly extensible, with the SSL/TLS engine design making it easy to add extensions or new cipher suites to the library. Most of the constant time code is done using a series of arithmetic primitive macros which allow for ease of extension of the secure code by adding new features.

Finally, it is also optimised to produce a small binary footprint, as well as using a low RAM amount, which is able to be used in most embedded devices. To that end, it is also able to work with smaller integer sizes for optimisation purposes on different platforms. That is, on specific platforms where computations with 16 bit integers are much faster than 32 or 64 bit integers, there is supporting code for all existing algorithms to be able to make use of this property in a constant time fashion.

Chapter 5

Quantum-resistant cryptographic algorithm selection

In this section, a selection of the NIST PQC round two candidates shall be analysed and compared to each other through various criteria to select a signature and encryption algorithm to be used in this thesis

5.1 Selection of quantum-resistant algorithms to consider

Due to the large amount of remaining candidates in the second round of the NIST standardisation process, that is, seventeen encryption schemes and nine signature schemes, a preselection of algorithms to consider for this thesis had to be done.

The main selection criteria were the following :

- **Performance**. Both for encryption and decryption speed as well as signature creation and verification speed, but also for key generation speed
- **Memory footprint**. Both key sizes and ciphertext sizes should be as small as possible.
- **License terms**. Both the provided software, as well as the underlying algorithms should be provided with workable open source licensing schemes, or better yet, be turned into the public domain

There also is to consider that, with the support of the supervising professor, A. Duc, it was decided to primarily focus on analysing Lattice based solutions due to the diversity of the applications of the underlying mathematical objects as well as the diversity of hard problems associated with them, and try to avoid any solutions developed by Microsoft, due to potential code licensing issues.

5.2 Algorithm Selection Process

It should be noted that since this thesis is part of a series of concrete post-quantum implementation thesis underway simultaneously, all 3 concerned participants (Jonathan Zaehringer, Pierre Kohler, and myself) took the opportunity to collaborate exclusively for the quantum resistant cryptographic selection process, in an attempt to broaden our data collection opportunities. As such, this section of the thesis should very much be considered a group effort, and thus, similarities between the 3 students' thesis in for the data collection and analysis chapter are more than likely.

The raw data used to produce the performance graphs is available in Tables 5.3 and 7.3

5.2.1 Prior considerations

First of all, of the available algorithms, according to the NIST PQC selection round 2 [17], one shall immediately be discarded from consideration, and that is SIKE, the singular algorithm implementing the supersingular elliptic-curve isogeny based cryptography.

The reasoning behind this is twofold, firstly the semantics of the cryptographic family allows it to directly interface with a Diffie-Hellman based key exchange scheme, which would both provide a simple drop-in replacement for the current ECDH SSL/TLS engine, and would provide immediately forward secrecy as well. This may seem like a desirable outcome, and to a certain degree, it is, but it would immediately mean that the selection process would be rendered pointless, as this is the only algorithm which can be used as an ECDH drop-in replacement.

Thus, the reasoning for its removal from considering is the following, if kept for consideration, it would be hands-down the most practical to put in place, underpinning the whole selection process. Thus, its removal from consideration is justified in wanting to pursue a KEM based approach to key exchange, as discussed in Section 2.4.

The second reasoning for its removal is that it has already been implemented by two major projects similar to this thesis, specifically, has been implemented in both BoringSSL and OpenSSL by Microsoft (the original algorithm's creators), as well as Google and Cloudflare in a joint experiment, as was discussed in Section 3.

Also of note, due to core properties of Code-based algorithms, their public key and ciphertext sizes tend to be quite larger than other schemes, which would render them quite undesirable for TLS purposes with a KEM based key exchange, due to the requirement of transferring public keys throughout the protocol.

This is also corroborated by the following graph in Figure 5.1, taken from the SUPERCOP benchmarking suite, highlighting both the public key and ciphertext sizes of all KEM algorithms from the second round. As can be seen, the vast majority of code-based algo-

rithms have between 4-16 times the public key and ciphertext sizes compared to lattice based solutions. Thus, the analysis step will be mostly focused on lattice based solutions.

5.2.2 Encryption algorithms preliminary comparison

First of all, here are the NIST security equivalence security levels, as defined in the NIST Post-Quantum Cryptography Call for Proposals [18].

- Category 1, any attack that breaks the relevant security must require comparable or greater computational resources than required for a key search of a block cipher with a 128-bit key, such as AES128.
- Category 2, any attack that breaks the relevant security must require comparable or greater computational resources than required for a collision search on a 256-bit hash function, such as SHA256.
- Category 3, any attack that breaks the relevant security must require comparable or greater computational resources than required for a key search of a block cipher with a 192-bit key, such as AES192.
- Category 4, any attack that breaks the relevant security must require comparable or greater computational resources than required for a collision search on a 384-bit hash function, such as SHA384.
- Category 5, any attack that breaks the relevant security must require comparable or greater computational resources than required for a key search of a block cipher with a 256-bit key, such as AES256.

Here is the list of algorithms still considered for this step:

- **Crystals-Kyber** with the 1024 variant and its category 5 NIST equivalence
- **NewHope CCA** with the 1024 variant and its category 5 NIST equivalence
- **NTRU Prime** with the sntrulpr857 variant and its category 4 NIST equivalence
- **NTRU** with the hrss701 variant and its category 3 NIST equivalence
- **Classic McElliece** with the mceliece460896 variant and its category 3 NIST equivalence
- **Saber v1.1** with the Saber-KEM variant and its category 3 NIST equivalence

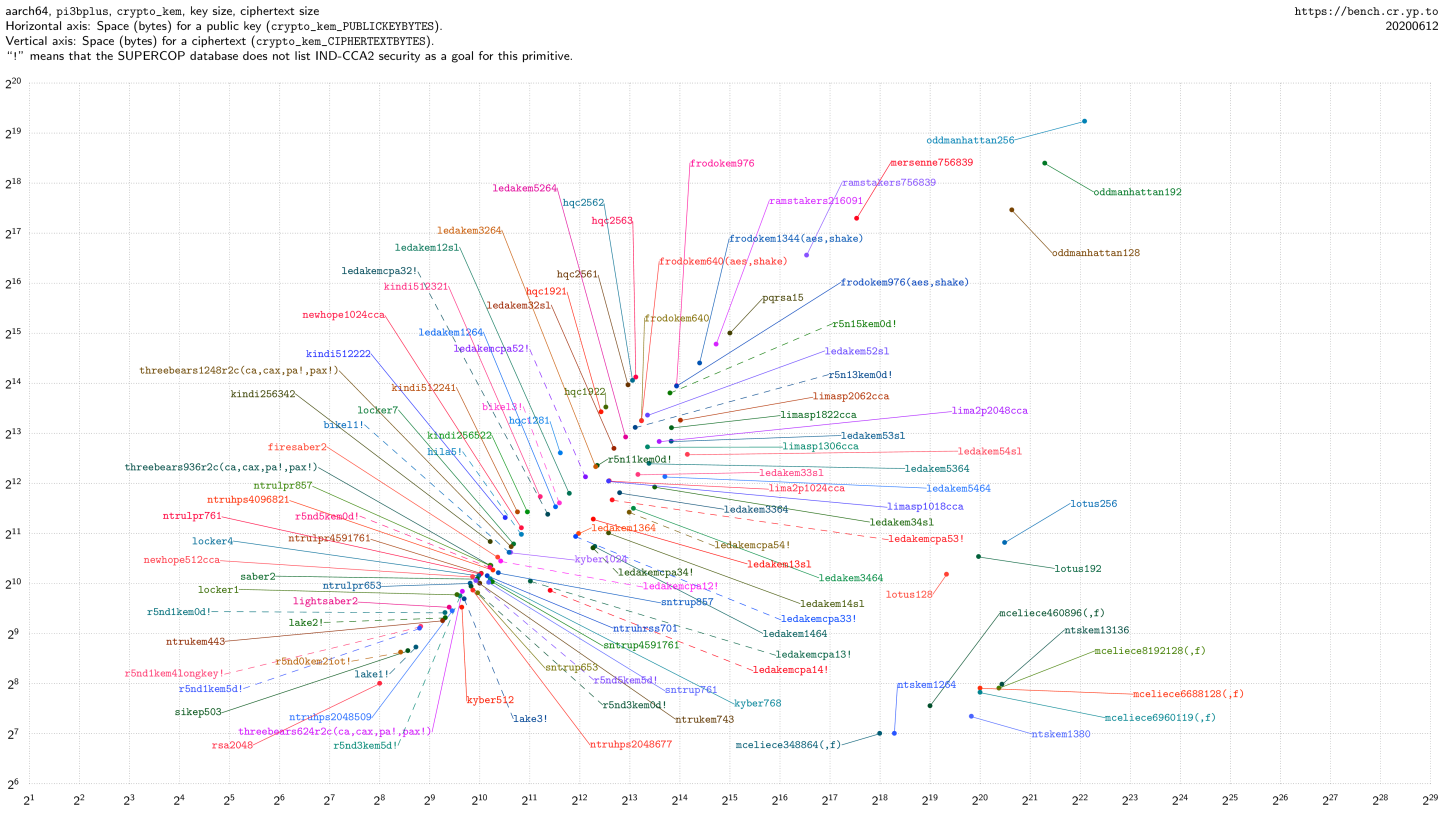

Figure 5.1: Comparison of public key and cipher text sizes for KEM based cryptography algorithms, Robocop, 2020. `https://bench.cr.yp.to/graph/aarch64-pi3bplus-kem-pkbytes,cbytes.pdf`

Hardware and software specifications			
OS	Kernel	CPU	RAM
Linux Mint 19.3	4.19	`i7-7700HQ @4.8GHz`	`1x16GiB DDR4 @2400MHz`
Arch Linux	5.4	`i7-4810MQ @2.80GHz`	`2x 8GiB DDR3 @1600MHz`
macOS 10.15.4	-	`i5-6287U @3.1GHz`	`2x 4GiB LPDDR3 @2133MHz`

Table 5.1: Hardware and software specifications

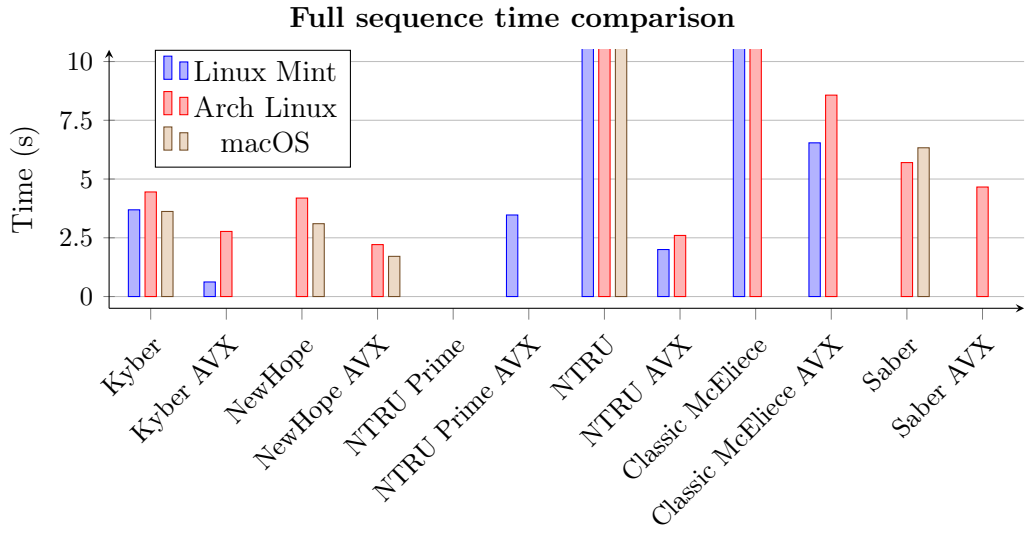

Figure 5.2: Preliminary encryption algorithm performance analysis graph

Figure 5.1 highlights the different hardware configurations used in the performance comparisons.

What follows in Figure 5.2 are the results of the initial performance benchmarks, the testing methodology was to simply measure the time duration to generate 10000 keys, encapsulate 10000 secrets, and decapsulate the 10000 secrets back to back. This is quite a crude testing, but it allows to quickly draw a picture about their respective performance to discard the ones that would not fit the TLS protocol requirements of quick key gen and encapsulation/decapsulation.

At first glance, it can be noticed that Classic McElliece is missing from this graph, this is due to its speed being around two orders of magnitude slower than other algorithms, thus would render the graph unusable. Classic McElliece completed the benchmark in 2205 seconds for its reference implementation and 200 seconds for its optimised implementation.

At this point, the only algorithm that we were able to be safely discard is Classic McElliece,

since every other algorithm was roughly two orders of magnitude faster.

Further more, for NTRU Prime, only the sntru variant was kept, due to it performing similarly to the other variant, and bringing faster encryption and decryption. Finally, Saber was also removed, due to performing similarly to Kyber and NewHope, but slightly slower in both the reference and optimised implementations.

Thus, the following algorithms proceeded to further testing:

- **Crystals-Kyber** with the 1024 variant and its category 3 NIST equivalence
- **NewHope CCA** with the 1024 variant and its category 3 NIST equivalence
- **NTRU Prime** with the sntrulpr857 variant and its category 4 NIST equivalence
- **NTRU** with the hrss701 variant and its category 3 NIST equivalence

5.2.3 Encryption algorithms comparison

In Figure 5.3, a comparison of all remaining encryption algorithms' performances can be seen, this time with separate timings for key gen, encapsulation and decapsulation, while using the same methodology of 10000 runs of each, as well as more testing platforms.

What is immediately apparent is that Kyber and NewHope's key generation speed are at least two orders of magnitude faster than both NTRU and NTRU Prime, with the encapsulation and decapsulation times being to a lesser extent quite faster (3 to 10 times faster), for their reference implementations. When comparing optimised implementations, however, the difference reduces greatly, with similar if not slightly better timings for NTRU and NTRU Prime, however, since there is no guarantee of being able to implement the optimised code into BearSSL, choosing either NTRU Prime or NTRU would require other issues with either Kyber or NewHope.

Then, these are the licensing terms of each algorithm.

- **Crystals-Kyber**'s reference code is released into the public domain [19], following the Creative Commons "No Rights Reserved" scheme, as well as any intellectual property claims have been relinquished to either the algorithm or reference/optimised code according to the NIST submission documents [20].
- **NewHope**'s reference code is released into the public domain [21], as well as any intellectual property claims have been relinquished to either the algorithm or reference/optimised code according to the NIST submission documents [22].
- **NTRU Prime**'s reference code, as far as can be seen, is not released under any specific licensing terms, intellectual property claims have been relinquished to either the

algorithm or reference/optimised code according to the NIST submission documents [23].

- **NTRU**'s reference code is released into the public domain [24], following the Creative Commons "No Rights Reserved" scheme, as well as any intellectual property claims have been relinquished to either the algorithm or reference/optimised code according to the NIST submission documents [25].

Finally, this means that both Kyber and NewHope's licensing terms are usable, and as their ciphertext/public key sized are quite reasonable, as can be seen in Figure 5.1, the final decision resides between Crystals-Kyber and NewHope CCA for the encryption algorithm.

However, prior to deciding which to use, the signature algorithms must be compared.

5.2.4 Signature algorithms comparison

Of all available round two algorithms, particular attention shall be placed on lattice based signature algorithms, this is due to the selection of a lattice based encryption algorithms, thus one can surmise that shared concepts and code paths could be able to be extracted, as well as key generation semantics. This decision, in hindsight, ended up being a poor one, and should not have had as much weight as is did. As can be seen in Section 6.2, in practice, a rather substantial rework of both kyber and dilithium's code would have had to take place to unify their code base.

Once this decision was made, major attention was directed to the following signature algorithms:

- **Crystals-Dilithium** with the IV variant and its category 3 NIST equivalence
- **Falcon** with the 1024 variant and its category 4-5 NIST equivalence (due to lack of closer to NIST 3 equivalence)
- **Sphincs$^+$** with the shake-192s variant and its category 3 NIST equivalence

Then, these are the licensing terms of each algorithm.

- **Crystals-Dilithium**'s reference code is released into the public domain [26], following the Creative Commons "No Rights Reserved" scheme, as well as any intellectual property claims have been relinquished to either the algorithm or reference/optimised code according to the NIST submission documents [27].
- **Falcon**'s reference code is released into the MIT Licensing scheme [28], however, the intellectual property is covered under the US patent **US7308097B2** according to the NIST submission documents [29]. It is also stated (in page 20 of the NIST IP statement)

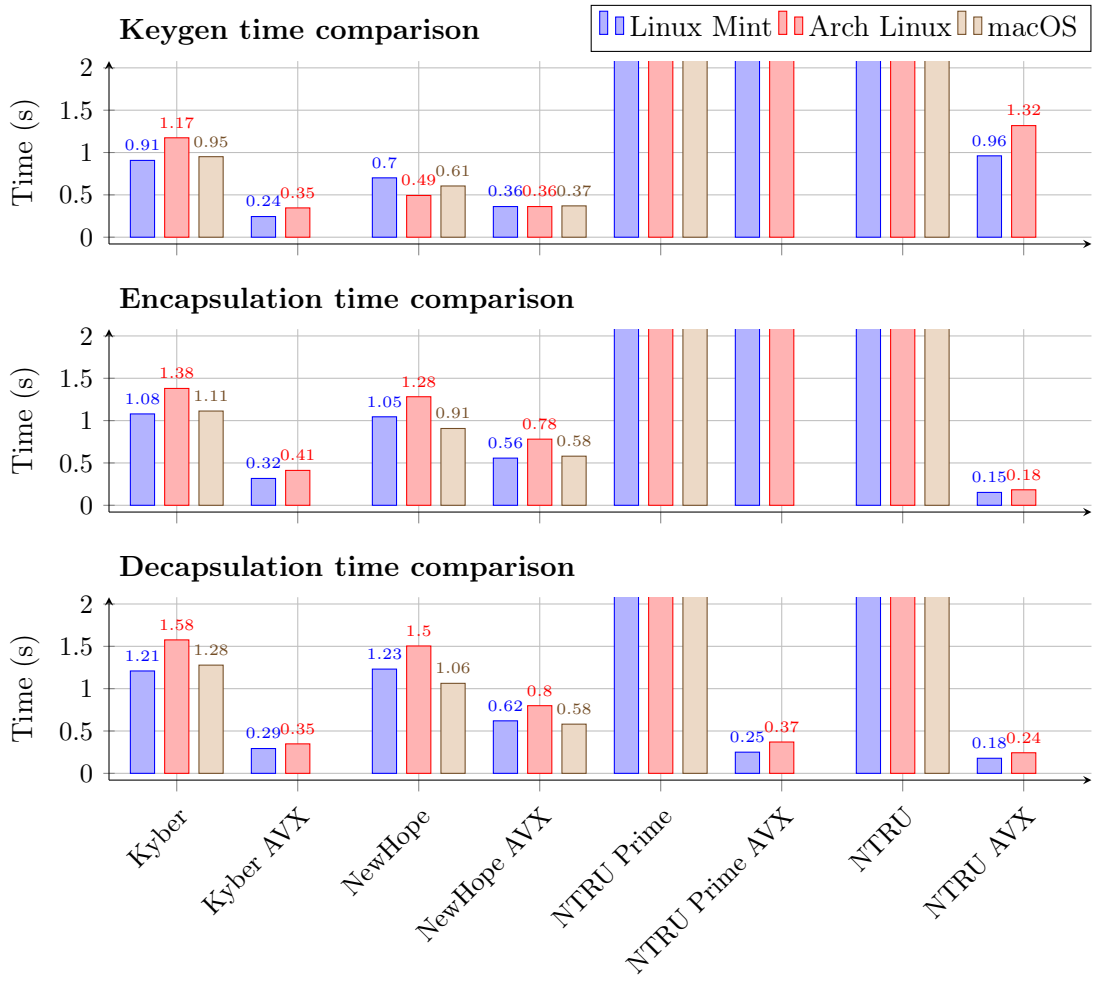

Figure 5.3: Final encryption algorithm performance analysis graph

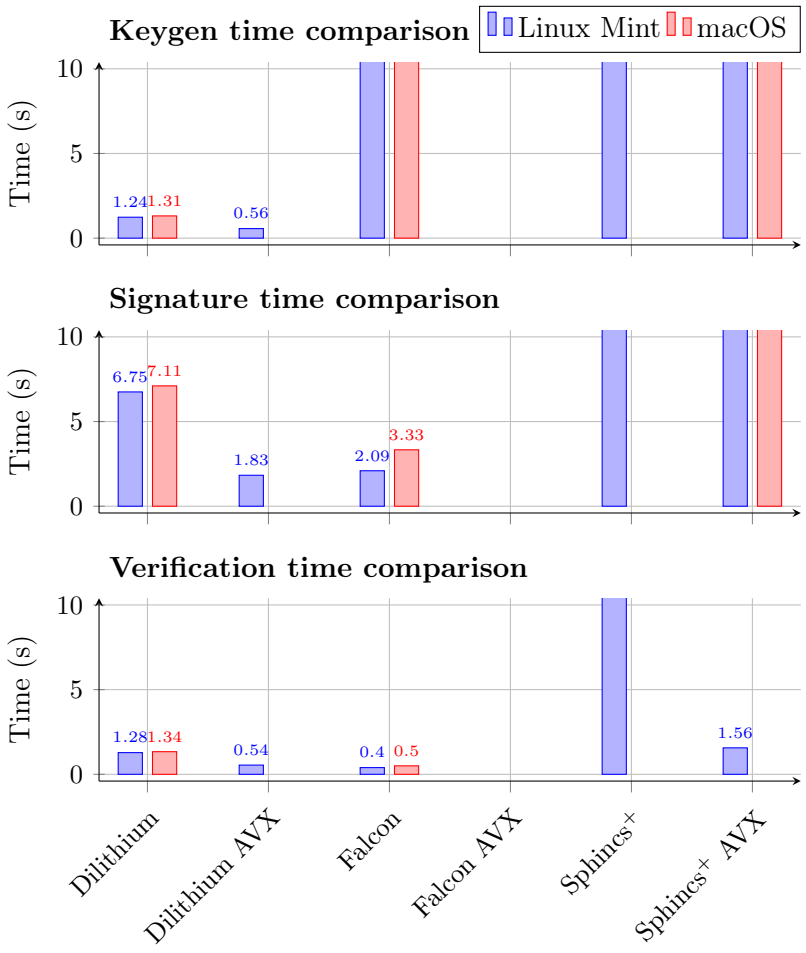

Figure 5.4: Signature algorithm performance analysis graph

that if Falcon were to be selected for the final standard, the patent holder would offer a non-exclusive license, without compensation, for the implementation of the standard.

- **Sphincs$^+$**'s reference code is released into the public domain [30], following the Creative Commons "No Rights Reserved" scheme, as well as any intellectual property claims have been relinquished to either the algorithm or reference/optimised code according to the NIST submission documents [31].

Speed comparisons for the selected algorithms are available in Figure 5.4.

It can be immediately noticed that key generation speed is MUCH, much faster for Dilithium than for either of falcon or Sphincs$^+$, which is not that much of a deal breaker for TLS applications, where key generation for certificates will be quite sparse. On the other hand,

Signature algorithm size comparison (size in Bytes)			
Algorithm name	Private key size	Public key size	signature size
Crystals-Dilithium4	3856B	1760B	3366B
Falcon1024	2305B	1793B	∼1273B
Sphincs$^+$-Shake-192s	96B	48B	17'064B

Table 5.2: Comparison of public/private key and cipher text sizes for signature algorithms.

signature creation and verification speeds of Falcon are quite faster than Dilithium, between 4-5 times faster to be precise, which can be quite more interesting in a TLS setting. However, when taken into consideration that the Falcon code package does not provide AVX testing code, whereas Dilithium does, one could compare the optimised results from Dilithium to the reference falcon code. With this comparison, both Falcon and Dilithium are quite adapted to TLS integration.

In the end, however, Sphincs$^+$'s signature and verification speeds were orders of magnitude larger than either of the lattice based solutions, which makes it quite unsuitable for TLS integration.

A table of size comparisons for different signature algorithms analysed can be found in Figure 5.2. As it can be seen, the key size of Sphincs$^+$ is extremely smaller than the lattice based counterparts, with roughly an order of magnitude between the two, although, this is at the cost of a larger signature size, between 5 and 15 times larger that the lattice based algorithms. This trade off is quite unfavorable for TLS in a signature algorithm. Indeed, since signatures will have to be sent through network packets between peers communicating though TLS, the larger the signature size, the higher the latency, and the higher the chance for network corruption of the signatures. This is the final reason as to why the selected Sphincs$^+$ algorithm is unsuitable for TLS based usages, and shall not be used in this thesis.

This finally leaves us with two competive lattice based signature algorithms in Falcon and Crystals-Dilithium.

5.3 Final Encryption and Signature Algorithm Selection

In summary, the remaining encryption algorithms were the following :

- **Crystals-Kyber**
- **NewHope CCA**

And the remaining signature algorithms were the following :

Raw data for encryption algorithm performance graphs						
Crystals-Kyber - 1024						10'000 times
	ref			avx		
	keygen	enc	dec	keygen	enc	dec
Linux Mint	907 ms	1079 ms	1209 ms	244 ms	317 ms	293 ms
Arch Linux	1174 ms	1380 ms	1576 ms	346 ms	412 ms	348 ms
MacOS	951 ms	1113 ms	1278 ms	N/A	N/A	N/A
NewHope CCA - 1024						10'000 times
	ref			avx		
	keygen	enc	dec	keygen	enc	dec
Linux Mint	701 ms	1045 ms	1231 ms	362 ms	557 ms	620 ms
Arch Linux	883 ms	1282 ms	1504 ms	493 ms	781 ms	799 ms
MacOS	605 ms	907 ms	1063 ms	370 ms	580 ms	581 ms
NTRU Prime - sntrup857						1'000 times
	ref			avx		
	keygen	enc	dec	keygen	enc	dec
Linux Mint	92.764 s	10.459 s	30.163 s	1.237 s	0.477 s	0.025 s
Arch Linux	128.467 s	14.430 s	41.866 s	1.762 s	0.696 s	0.037 s
MacOS	106.902 s	12.410 s	33.719 s	N/A	N/A	N/A
NTRU - hrss701						10'000 times
	ref			avx		
	keygen	enc	dec	keygen	enc	dec
Linux Mint	55.268 s	3.738 s	10.295 s	0.961 s	0.152 s	0.178 s
Arch Linux	60.737 s	3.711 s	10.988 s	1.318 s	0.183 s	0.243 s
MacOS	48.831 s	2.772 s	7.947 s	N/A	N/A	N/A

Table 5.3: Raw data used for the creation of the encryption speed comparison graphs.

Raw data for signature algorithm performance graphs						
Crystals-Dilithium - 4						10'000 times
	ref			avx		
	keygen	enc	dec	keygen	enc	dec
Linux Mint	1236 ms	6749 ms	1280 ms	563 ms	1828 ms	544 ms
MacOS	1313 ms	7106 ms	1335 ms	N/A	N/A	N/A
Falcon - 1024						10'000 times
	ref			avx (Tests not available)		
	keygen	enc	dec	keygen	enc	dec
Linux Mint	142.2 s	2.094 s	395 ms	N/A	N/A	N/A
MacOS	188.55 s	3.331 s	0.50 s	N/A	N/A	N/A
Sphincs$^+$ - Shake-192s robust						10'000 times
	ref			avx		
	keygen	enc	dec	keygen	enc	dec
Linux Mint	181.2 s	3600 s	78.1 s	40 s	850 s	1.562 s
MacOS	N/A	N/A	N/A	65 s	1280 s	N/A

Table 5.4: Raw data used for the creation of the signature speed comparison graphs.

- **Crystals-Dilithium**
- **Falcon**

All of them perform similarly, they all are released under usable licensing schemes and have similar memory footprints.

However, Falcon's Intellectual Property caveat of being patented, with a conditional licensing scheme that assumes it is selected as standard leads it to not being quite as suitable for this project than Crystals-Dilithium. Indeed, should this project lead to a usable product, basing it upon a conditional licensing scheme does incur unnecessary risks of producing unusable software. As such, this project shall use **Crystals-Dilithium** as its quantum-resistant signature algorithm.

Considering the use of Crystals-Dilithium as a signing algorithm, one can reasonable assume that the encryption algorithm developed by the Crystals team, based upon the same lattice cryptographic family, that both of them would share a non-insignificant part of their code, logic and key formats. As can be seen in Section 6.2, while this assumptions was correct, due to the scale of this thesis as well as the design of the reference code packages provided by the Crystals team, there was not enough time to profit from this property, as such, it should not have had as much weight as it had upon the decision of the final encryption alogirthm.

This assumption could have led to a potentially poor choice of algorithm and thus deserves to be marked as a methodological error in this thesis, despite NewHope having been rejected

from Round 3 of the NIST standardisation process in hindsight.

Thus, the selected encryption algorithm for this project is **Crystals-Kyber**.

Chapter 6

Implementation

In this chapter, the required TLS protocol changes and additions, such as the creation of a new extension, shall be covered extensively.

Following that, the implementation of Kyber and Dilithium in BearSSL shall also be covered.

Finally, an overview of the main quirks and difficulties concerning implementing quantum-resistant cryptography into BearSSL.

6.1 TLS 1.2 protocol changes and additions

This section will give an in-depth overview of the TLS 1.2 handshake protocol. It will then be followed by a detailed description of the Kyber-KEM based key exchange construction. Finally, an exhaustive list of the TLS 1.2 protocol changes and additions required to accommodate the Kyber-KEM based key exchange.

6.1.1 TLS 1.2 handshake protocol overview

First of all, Figure 6.1 represents a slightly simplified sequence diagram of the TLS 1.2 handshake protocol that will need to be revised and/or extended to accommodate the needs of a quantum-resistant cipher suite.

In particular, it represents a handshake during which an Elliptic Curve Diffie-Hellman Ephemeral (ECDHE) cipher suite was in use. A cipher suite is said to be ephemeral, if during the key exchange process unique key pairs for both the client and server, not the ones associated with their certificates, are in use. The use of such cipher suites holds some interesting properties such as forward secrecy (in the event the server certificate private key is obtained by a third party. they would still be unable to decrypt any previously

established connections). Which is not the case for non-ephemeral cipher suites. Thus, because the implementation of an ephemeral quantum-resistant cipher suite holds roughly the same challenges as a non-ephemeral cipher suite, this thesis shall by design only implement ephemeral quantum-resistant cipher suites, due to the abovementioned advantage they hold over non-ephemeral cipher suites.

What follows is a non-exhaustive step by step description of the TLS 1.2 handshake protocol. Only the aspects relevant to this thesis are covered by this description. For the full specification of the TLS 1.2 handshake protocol, please refer to its RFC [32].

- During the *ClientHello* phase, among other information, the list of supported cipher suites and extensions is sent to the server.
- During the *ServerHello* phase, the server will choose which cipher suite to use in the connection as well as a list of extensions, containing potentially a *ClientCertificateRequest*, to authenticate the client.
- During the *ServerCertificate* phase, the server will provide a x509 certificate containing the public key associated to the server with regard to signature verification, as well as its hostname, and optionally a series of intermediate certificates required for the client to be able to establish a chain of trust.
- During the *ServerKeyExchange* phase, the server will generate an elliptic curve key pair and will provide its public key, as well as a signature of the generated public key, generated using the private key associated with the provided certificate.
- Once the client receives both the *ServerCertificate* and *ServerKeyExchange*, it will verify that the certificate provided by the server is valid and can be linked, with the potential help of its intermediate certificates, to a trust anchor to ensure the authenticity of the server certificate. The client will then verify that the hostname indicated in the certificate matches the server the client is trying to connect to. Finally, the client will then use the public key contained inside the certificate to verify that the contents of the *ServerKeyExchange* message were actually signed by the server itself. If at any point, one of the verifications fails, the handshake is aborted. At this point, the client has been able to properly authenticate the server.
- During the optional *ClientCertificate* phase, the client will also provide a x509 certificate containing the public key associated to the client with regard to signature verification, and optionally a series of intermediate certificates required for the server to be able to establish a chain of trust.
- During the *ClientKeyExchange* phase, the client will generate an elliptic curve key pair and will provide its public key, as well as an optional signature of the generated public key, generated using the private key associated with the provided certificate, if the

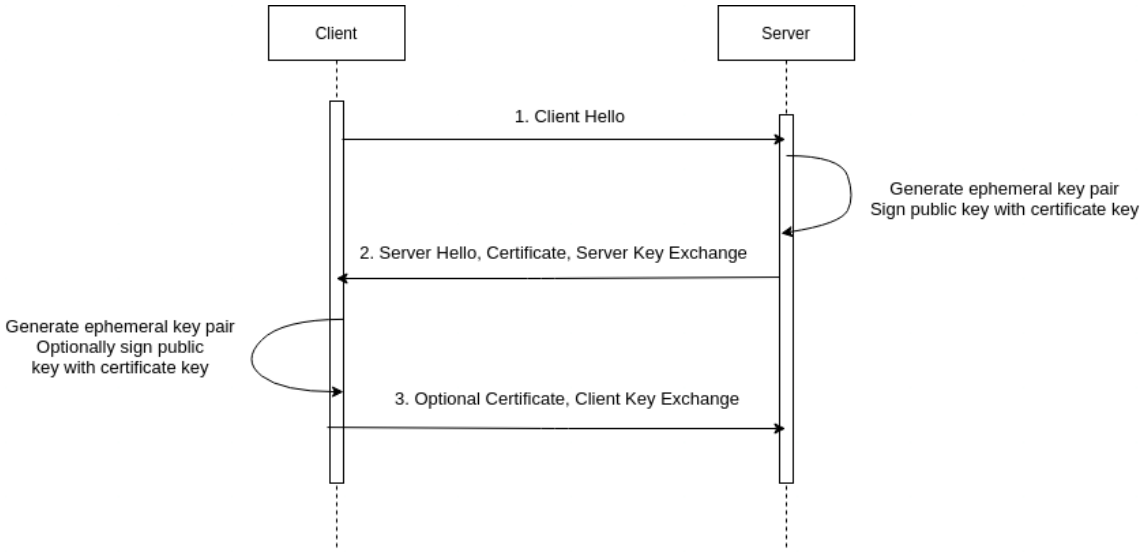

Figure 6.1: Sequence diagram of part of a TLS 1.2 handshake using an ECDHE cipher suite

server requested a client authentication. At this point, the client is able to generate a shared secret using the elliptic curve Diffie-Hellman process.

- Once the server receives both the *ClientCertificate*, if requested, and *ClientKeyExchange*, it will verify that the certificate provided by the client is valid and can be linked, with the potential help of its intermediate certificates, to a trust anchor to ensure the authenticity of the client certificate. Finally, the server will then use the public key contained inside the certificate to verify that the contents of the *ClientKeyExchange* message were actually signed by the client itself. If at any point, one of the verifications fails, the handshake is aborted. At this point, the server has been able to properly authenticate the client if required. At this point, the server is able to generate a shared secret using the elliptic curve Diffie-Hellman process. Thus, a confidential and authenticated communication channel has been established at this point, and the handshake is finalised.

6.1.2 Kyber-KEM based key exchange

Before detailing the changes and additions to the TLS protocol, it is important to note how exactly, the quantum-resistant key exchange will take place. This completely ignores the authentication aspect of the handshake protocol, which will be covered in more detail further in the section, to focus solely on the key exchange.

Due to the Crystals-Kyber variant in use bearing no impact on the key exchange construction

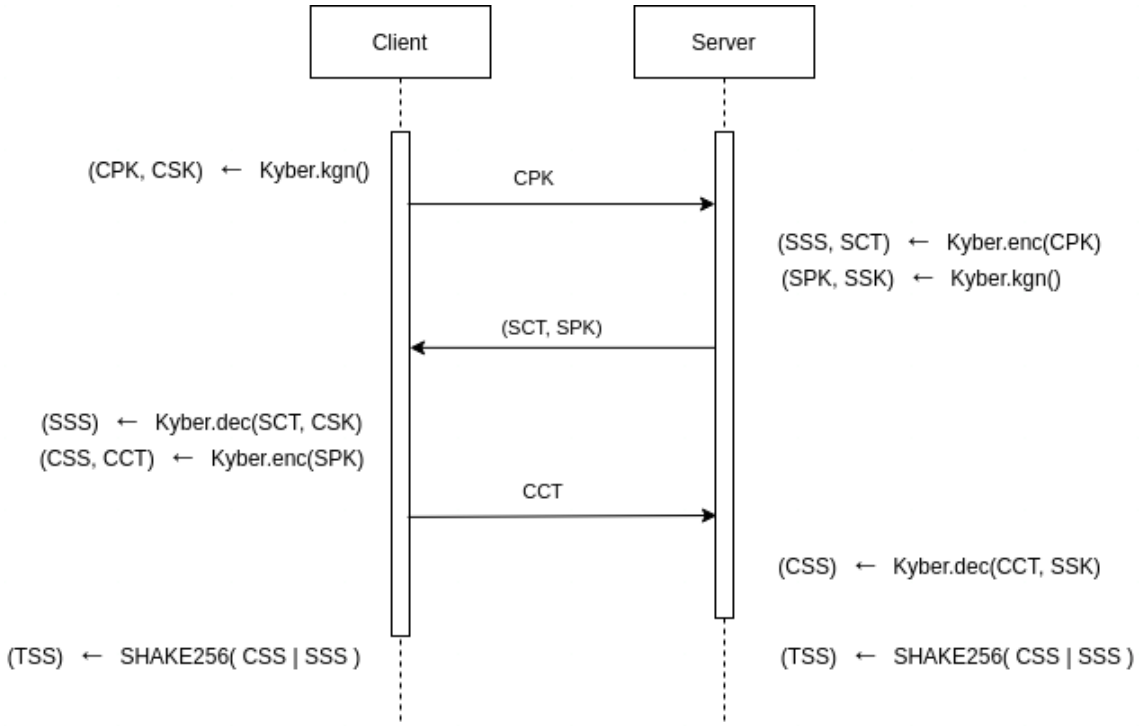

Figure 6.2: Sequence diagram of the Kyber-KEM based key exchange protocol

itself, it shall be left out.

The Key exchange construction is based upon the Kyber Unilateral Authenticated Key Exchange construction detailed in the Crystals-Kyber paper [33].

An existing implementation of the original protocol can be found in the reference code package of Crystals-Kyber.

Its usage, however, will not make use of static keys as shown in the Kyber.UAKE Figure of the paper, due to the TLS protocol already providing authentication mechanisms distinctly from the key exchange when an ephemeral key exchange is in place.

The usage of the Shake256 function as a key derivation function was done due to the function being already widely in use throughout the Kyber algorithm, and also to match the existing implementation of the original protocol.

Figure 6.2 highlights the key exchange protocol.

- The client shall generate a Kyber key pair (CPK, CSK) and transmit its public key (CPK) to the server, storing its associated private key (CSK).
- The server shall then use the client public key (CPK) to generate a shared secret (SSS)

and its associated ciphertext (SCT) containing the generated shared secret (SSS) it will store its generated shared secret (SSS).

- The server shall then generate a Kyber key pair (SPK, SSK) storing its associate private key (SSK).
- The server shall then transmit the tuple (SCT, SPK) containing its ciphertext as well as its public key to the client.
- The client shall then use its private key (CSK) to decapsulate the server shared secret (SSS) from the received ciphertext (SCT).
- The client shall then use the server public key (SPK) to generate a shared secret (CSS) and its associated ciphertext (CCT) containing the generated shared secret (CSS).
- The client shall use both its own shared secret (CSS) along with the server's shared secret (SSS) to derive the true shared secret (TSS). This derivation shall be as follows: TSS = SHAKE256 (CSS|SSS) meaning that the true shared secret is the output of the SHAKE-256 hash function with the concatenation of the server shared secret to the client shared secret as its input.
- The client shall then transmit its ciphertext (CCT) to the server.
- The server shall then use its private key (SSK) to decapsulate the client shared secret (CSS) from the received ciphertext (CCT).
- The server shall use both its own shared secret (SSS) along with the client's shared secret (CSS) to derive the true shared secret (TSS). This derivation shall be as follows: TSS = SHAKE256 (CSS|SSS) meaning that the true shared secret is the output of the SHAKE-256 hash function with the concatenation of the server shared secret to the client shared secret as its input.

It should be noted that due to the use of the SHAKE hash function family in use for the true shared secret derivation, which offers arbitrary output size capabilities, the client and server MUST have decided ahead of time in a true shared secret length.

6.1.3 TLS 1.2 detailed list of changes and additions

What follows is a detailed list of all changes and additions made to the TLS 1.2 protocol itself, without considering the implementation aspect with BearSSL, to accommodate quantum resistant cipher suites.

Each addition or change shall be annotated with ♦ for readability sake.

6.1.3.1 *ClientHello* changes

♦ The biggest addition to the TLS protocol is a *custom extension* used in the *ClientHello* phase of the protocol.

This new extension is required due to wanting to use the three message key exchange protocol presented in Section 6.1.2, which would not work with the current semantics of the TLS key exchange protocol as is, using only two messages.

This, however, can luckily be adapted to fit into the TLS protocol, with the help of a client hello extension. As such, the *Kyber client hello anticipated key exchange extension* is born.

The extension is formatted as per Figure 6.3. The extension uses the 0XDADD identifier. This is a purely arbitrary decision, due to it being unassigned by the IANA [34] The extension is structured as such for the following reasons. The first length field is required to be able to read the full extension. In the second level, the *Kyber key material length* is used to determine the full size of all keys encoded back-to-back. Finally, each key is composed of the Kyber key type, as a sanity check for its announced length, finally, the proper key length, followed by the actual bytes of the key.

Such a construction is advantageous in that it allows for a given client to provide multiple potential ephemerals keys, at different security levels, in case the server has a minimum security level requirement.

This extension fully respects the specified format in Section 7.4.1.4 of RFC5246 [32].

♦ Four new algorithms have been added to the *SignatureAlgorithms ClientHello* extension, following the format in Section 7.4.1.4.1 of RFC5246 [32].

Those new algorithms are for the new Dilithium signature algorithm, using value 0x04 on the *SignatureAlgorithm* enum of the RFC

It has been decided to only support SHA-2 and its variants for the new Dilithium signature algorithms, as such, the only valid values for those algorithms are as follows:

- 0x0404 SHA2-256 With Dilithium signature
- 0x0304 SHA2-224 With Dilithium signature
- 0x0504 SHA2-384 With Dilithium signature
- 0x0604 SHA2-512 With Dilithium signature

Those same algorithms have also been added to the *SignatureAndHashAlgorithm* field in the *CertificateRequest* message, as defined in Section 7.4.4 of RFC5246 [32], as well as the *SignatureAlgorithm* field in *CertificateVerify*, as defined in Section 7.4.8 of RFC5246 [32].

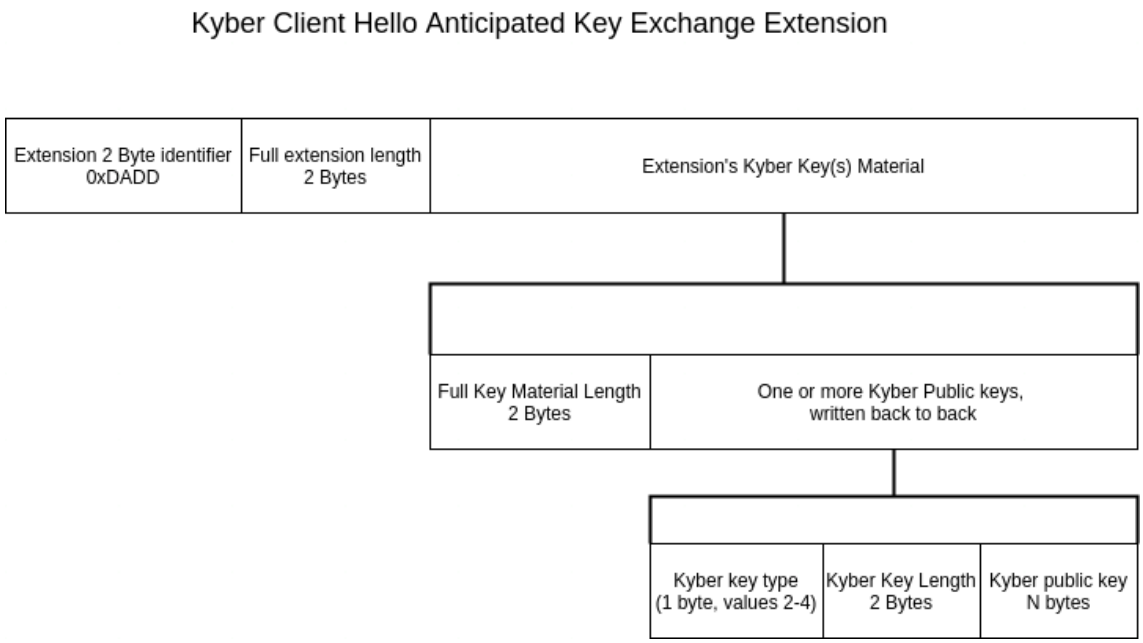

Figure 6.3: Kyber client hello anticipated key exchange extension data structure

6.1.3.2 *ServerHello, SeverKeyExchange & CertificateRequest* changes

♦ A new server key exchange type for the field *ECCurveType*, as defined in Section 5.4 of RFC4492 [35] Value 0x04 designates a Kyber key exchange signed with Dilithium. It has been decided NOT to support Kyber key exchanges signed with any other signature schemes, since allowing a quantum resistant key exchange to be authenticated by a non-quantum resistant signature would lead to all sorts of attacks, such as Man-in-the-Middle attacks, by adversaries holding a sufficiently capable quantum computing system.

This new server key exchange type is formatted as follows.

- 2 Bytes Kyber server ciphertext length
- X Bytes Actual Kyber ciphertext
- 2 Bytes Dilithium signature length
- Y Bytes Dilithium signature over Kyber ciphertext
- 2 Bytes Kyber server public key length
- Z Bytes Actual Kyber server public key

One could notice that unlike traditional ECDHE key exchanges, the ciphertext is signed, instead of the actual Kyber public key.

This was done in purpose. While this signature over the ciphertext instead of the public key does not diminish the actual authenticity claims of the traditional key exchange, since it fits as well with the semantical logic of the key exchange. Which is that the server certifies knowing a fundamental secret to the construction of the shared secret. In the case of traditional ECDHE key exchanges, that fundamental part is the associated private key, without which, the shared secret would not be equal for the two peers. In the case of this Kyber key exchange, the fundamental part is the underlying server shared secret component of the true shared secret, as per Section 6.1.2 and Figure 6.2.

It should also be noted in all Kyber reference security modes, a ciphertext is always at most as large as a public key (in the case of Kyber-1024), but could be slightly smaller, as in the case of Kyber-786 and Kyber-512. Thus, this would slightly (to a likely negligible amount) accelerate the computation of the hash to sign.

However, if such a decision would prove to be detrimental to the security of the protocol, the change to signing the public key instead of or in addition to the ciphertext would be trivial in the implementations.

♦ A new client certificate type value as been added to the enum *ClientCertificateType*, in Section 7.4.4 of RFC5246 [32].

Value 0x63 designates a Dilithium client certificate

6.1.3.3 *ClientKeyExchange & CertificateVerify* changes

♦ In the case of the *KeyExchangeAlgorithm* value being 0x63, in the struct *ClientKeyExchange* in Section 7.4.7 of RFC5246 [32], Then a new client key type shall be used.

It consists simply of a Kyber ciphertext signed with the server's Kyber public key, prepended by the ciphertext length.

♦ In the case of the *CertificateVerify* struct, if the signature algorithm is one of the newly added signature algorithms, and the client certificate is a Dilithium certificate, the contents shall be a standard Dilithium signature of the Client Key Exchange public key (as well as the client and server random).

♦ One new cipher suite was created. However, it is very realistic and simple to implement new ones, relying on other cryptographic primitives outside of the handshake protocol.

The following is the new cipher suite, along with its suite ID, which is unassigned by the IANA [36].

- 0xDECO TLS_KYBR_DLTHM_WITH_CHACHA20_POLY1305_SHA256

The decision to focus on the implementation of a Chacha20+Poly1305 cipher suite as the only quantum-resistant cipher suite is because the targeted hardware segment for BearSSL is mostly embedded devices. Such devices rarely have an AES hardware-engine, which would mean much slower performances since purely software implementations of AES don't take as much advantage of modern hardware than Chacha20 [37].

6.1.4 New X509 certificate constants

When supporting x509 certificates both containing Dilithium keys and/or being signed with Dilithium keys, there is a fair amount of custom Object Identifiers (OID) that have to be defined.

The following OIDs have been added to the x509 standard.

- 1.2.840.10045.4.4.1 dilithium-with-SHA224
- 1.2.840.10045.4.4.2 dilithium-with-SHA256
- 1.2.840.10045.4.4.3 dilithium-with-SHA384
- 1.2.840.10045.4.4.4 dilithium-with-SHA512
- 1.2.840.10045.2.2 id-dilithiumPublicKey

One can note that the OIDs insert themselves into the 1.2.840.10045 OID structure, which was defined in 1998 as the ANSI X9.62 standard [38].

It was done so, simply by a lack of knowledge of other relevant OID structures.

Other than the newly created OIDs, the signatures and public keys can simply be inserted as-is into the existing ASN.1 structure of a given X509 certificate.

6.2 Kyber and Dilithium implementation in BearSSL

Firstly, it should be noted that due to the target platforms for BearSSL being mainly embedded devices without x86 and AVX2 instruction sets, it was decided to focus on the reference implementation of both algorithms, instead of their optimized versions.

It was however decided to only implement Kyber's original variant, and ignore the 90s version. This is mostly due to concerns of performance in embedded devices, where the 90s usage of AES instead of Keccak as a Pseudo-random number generator is much slower due to a lack of hardware implementation.

Finally, it has also been decided to NOT implement Dilithium's deterministic signature mode. Indeed, while such a mode can be quite very useful for platforms without proper randomness

generation, deterministic signature schemes have in the past lead to TLS vulnerabilities, such as with Ed25519 [39].

6.2.1 Kyber and Dilithium code unification practical issues

Due to the design of Kyber and Dilithium's code, relying liberally on compiler macros and defines throughout the code to determine loop lengths, mathematical constants, and structure sizes. A large amount of code rework and restructure had to take place to support all security levels of both Kyber and Dilithium at the same time, without having to rely on wrapping multiple binaries to provide all available security levels which while faster and easier to set up, would have most likely substantially affected the final library size, which would go against BearSSL's philosophy.

Also, while both Kyber and Dilithium are based upon the same hard problem, namely Module-LWE, they use widely different moduli, which in turn affects deeply the memory space required to store their polynomials.

Both Kyber and Dilithium employ 255th-degree polynomials, but due to the different moduli, twelve bits are required for Kyber to represent the polynomial coefficients and twenty-three bits for Dilithium, which means that for optimal performance due to memory alignment, Dilithium polynomials would take thirty-three bits of space, compared to the sixteen bits for Kyber polynomials. Also of note, due to the different moduli and the code being accelerated with Montgomery modular multiplications, there would be a need to propagate the different moduli to every function that requires working with Montgomery modular multiplications. The usage of number-theoric transforms to accelerate multiplications also implies widely different look-up tables for zeta values, yet again due to different moduli, which further exacerbates the need to propagate moduli information further into the codebase.

Those major remarks make is such that while in theory, the unification of the codebase would be rather simple, whereas in practice, not so much. It can be estimated that with roughly 20-30 hours of work, the codebase could have been neatly and cleanly unified, however, due to a lack of time and the scope of the thesis, it was decided to not do so, and leave the duplicated codebase, while hoping for the best from compiler space optimizations.

It should also be noted that the Dilithium and Kyber cryptographic primitives such as Shake and SHA-3 have been replaced with calls the built-in BearSSL Keccak engine, thus removing any requirement from external third party code other than Crystals' Kyber and Dilithium reference implementations.

6.2.2 Kyber and Dilithium memory usage optimisations

Another aspect of the Kyber and Dilithium implementations is the relatively large amount of used RAM required to store all the polynomial while working on them. While such concerns

	Reference code		BearSSL code	
	Dilithium4	Kyber-1024	Dilithium4	Kyber-1024
Keygen	59'392 Bytes	14'336 Bytes	17'408 Bytes	5'120 Bytes
Decrypt / Verify	56'320 Bytes	23'040 Bytes	30'720 Bytes	16'896 Bytes
Encrypt / Sign	78'848 Bytes	17'920 Bytes	30'720 Bytes	11'776 Bytes
Fast sign (BearSSL only)	-	-	61'440 Bytes	-

Table 6.1: Comparison of the local polynomial memory usage in specific Kyber and Dilithium functions between the reference implementation and the one available in BearSSL.

were outside the scope of the reference implementation, with some simple memory usage optimizations such as seeding the public matrix row by row just in time before its usage, as well as reusing the underlying polynomials when no overlap in usage can be detected, a non-negligible amount of memory can be recouped.

Table 6.1 highlights the differences in ram usage exclusively by polynomials in various Kyber and Dilithium functions. NB: Due to the restriction of no dynamic allocation in direct BearSSL library code, regardless of the security variant chosen, the locally reserved polynomial amount shall be equivalent to the highest supported security level, that is Dilithium4 and Kyber1024, thus comparisons between the reference implementation's memory usage for other security levels should be meaningless.

It can be seen that we get up to 70% less memory usage in the case of a Dilithium4 keygen. It should be noted, however, that Kyber's memory usage was not optimized as in-depth as Dilithium.

Regarding the *Fast sign* mode listed for Dilithium4 in BearSSL's case. In the case of a Dilithium signature, the signature is created in a rejection based system, where the tentative signature is rejected if it would end up leaking information about secrets. In such a case, since the tentative signature is generated using the public matrix for a matrix-vector multiplication step, storing the matrix in memory does take a considerable amount of memory, but will make it so that the time-consuming rejection sampling of the matrix from the seed row-by-row only takes place once. Thus, the *Fast sign* mode listed is merely a purposeful optional lack of memory usage-optimization, to combat computational trade-offs, rather than a speed optimization per se.

Enabling this signature setting in the Dilithium header ends up speeding up the signature time by an average of 25%, but increases memory usage by 100%, it is nonetheless important to note that subtlety.

6.2.3 Kyber and Dilithium integration with BearSSL testing frameworks

Kyber and Dilithium were properly integrated into the existing BearSSL speed test framework, however, they were not integrated into the crypto accuracy framework.

Indeed, the provided Known-answer tests by the Crystals team rely upon a specific kind of deterministic random bit generator (DRBG) which is not available in BearSSL, and its implementation would be outside the scope of this thesis. As such, the correctness of the underlying Kyber and Dilithium BearSSL implementation was done by pinning the DRBG output on both the reference and BearSSL code, such that the DRBG returns always the same byte, and the outputs of all functions were compared between the BearSSL and reference code.

Detailed instructions to run such tests can be found within the annexed code package.

6.3 BearSSL quantum-resistant TLS implementation

Now that the two new quantum-resistant cryptographic primitives are implemented in BearSSL and that the TLS 1.2 protocol has been adapted and revised to include a quantum-resistant handshake protocol, it only remains to be implemented in BearSSL itself.

Another thing to note is that the SSL/TLS engine itself is written in a custom, purpose-built language, T0. This has the advantage of somewhat painlessly implementing a highly complex state machine without the risk of encountering memory-related bugs, and with a trivial parallel processing potential. However, it should be noted that the T0 language works with reverse polish notation and that some code control structures are quite unintuitive to use, such as switch statements, conditional branching, and stack management.

Once those initial hurdles overcome, complex state machine modifications are quite easy to do, thanks to the custom language compiler's strong stack usage analyzer, it is quite rare to write state machine code with unexpected side effects. However, it should be noted that the resulting code is a nightmare to debug, as such, if one were to write wrong state machine logic, it would usually take between twenty minutes to two hours to locate and fix the root of the problem, which ends up happening shockingly often due to the abovementioned unwieldiness of the control structures.

Most of the SSL/TLS engine changes were quite straight forward ones, unlike the x509 certificate engine ones, due to the rigidity of the ASN.1 certificate format, as well as the rather large scope of the engine's purpose inside of BearSSL.

It should also be noted that due to the larger Kyber and Dilithium message sizes, for keys, ciphertexts, and signatures, the underlying SSL engine structure had to be enlarged by a non-insignificant amount, now being roughly eight times larger.

While this in itself is unavoidable, with some couple thousand bytes that could likely be saved with some proper memory usage optimization and overlapping, this memory usage explosion led to a nasty bug that wasted roughly 2 weeks of development time with the SSL/TLS engine development.

Source Code : Excerpt from src/ssl/ssl_hs_common.t0 : 142-149

```
// Define a word that evaluates as an address of a field within the
// engine context. The field name (C identifier) must follow in the
// source. For field "foo", the defined word is "addr-foo".
: addr-eng:
    next-word { field }
    "addr-" field + 0 1 define-word
    0 16383 "offsetof(br_ssl_engine_context, " field + ")" + make-CX
    postpone literal postpone ; ;
```

As can be seen above, this snippet of T0 code contains a non-trivial macro, serving to create an inner label evaluating to the offset in the context of a C field name.

Due to the nature of t0 code being written in reverse polish notation, it is not easy to figure out which parameters are sent to which methods, and considering that most of the methods that can be seen in this snippet have no documentation whatsoever, it was quite difficult to even notice that this snippet of code was the actual root cause of a slew of segmentation faults.

The issue was that the *16383* $(2^{14} - 1)$ magic number used to be *8191* $(2^{13} - 1)$, with no documentation what so ever as to what that original value's expected effects were. To the best of my knowledge, that magic number represents a sort of boundary for possible ranges of offsets for values in the context, which would short-circuit with the newly enlarged context.

Implementation of Kyber+Dilithium into the SSL/TLS testing framework, that is, in the x509 certificate parsing automated testing framework, as well as in the testing client/server software was rather straight-forward.

6.4 Implementation limitations and hacks

First of all, the only proper hack employed for the implementation can be seen in the *SignatureAndHashAlgorithm* field storage inside of the given SSL/TLS engine context.

Source Code : Excerpt from inc/bearssl_ssl.h : 2835-2866

```
/**
 * \brief Get the hash functions and signature algorithms supported by
 * the server.
```

```
4  *
5  * This value is a bit field:
6  *
7  *   - If RSA (PKCS#1 v1.5) is supported with hash function of ID `x`,
8  *     then bit `x` is set (hash function ID is 0 for the special MD5+SHA-1,
9  *     or 2 to 6 for the SHA family).
10 *
11 *   - If ECDSA is supported with hash function of ID `x`, then bit `8+x`
12 *     is set.
13 *
14 *   - If DILITHIUM is supported with hash function of ID `x`, then bit
15 *     `25+x` is set. (only algorithms with ID 3 to 6 are supported)
16 *
17 *   - Newer algorithms are symbolic 16-bit identifiers that do not
18 *     represent signature algorithm and hash function separately. If
19 *     the TLS-level identifier is `0x0800+x` for a `x` in the 0..11
20 *     range, then bit `16+x` is set.
21 *
22 * "New algorithms" are currently defined only in draft documents, so
23 * this support is subject to possible change. Right now (early 2017),
24 * this maps ed25519 (EdDSA on Curve25519) to bit 23, and ed448 (EdDSA
25 * on Curve448) to bit 24. If the identifiers on the wire change in
26 * future document, then the decoding mechanism in BearSSL will be
27 * amended to keep mapping ed25519 and ed448 on bits 23 and 24,
28 * respectively. Mapping of other new algorithms (e.g. RSA/PSS) is not
29 * guaranteed yet.
30 *
31 * \param cc   client context.
32 * \return  the server-supported hash functions and signature algorithms.
33 */
```

Indeed, if one looks at the *bearssl_ssl.h* header, between lines 2835 and 2866, it is defined that the context stores the supported signature+hash algorithm combinations inside of a 32 bit, bitfield as flags.

Before this thesis' work, all the bits of the bitfield were already assigned. It should also be noted, that due to implementation limitations of the SSL/TLS state machine, it can only read up to 32-bit values from the context, with limited possibilities to bypass this issue, all requiring extensive rework of the underlying logic to work properly. Thus, since the lower half of the bitfield was already taken up by RSA and ECDSA signature which are quite commonplace with TLS 1.2, any change there would be quite far from the path of least resistance.

This leaves us with the upper 16 bits, which are already mapped to other, newer, TLS1.3+ algorithms. While repurposing those TLS 1.3+ algorithms bit field entries could have been a workable solution for a TLS 1.2 implementation, in practice, it would be quite undesirable to break support for existing algorithms.

In the end, it was decided to compromise. Due to Dilitihum signatures being only supported with 4 SHA-2 variants by design, and when taking into consideration that according to IANA's *TLS SignatureScheme* value registry [36], the 0x800x values are only utilized up to 0x800B, which would mean that there are currently 4 available bits in the bitfield that would not be in use at the moment, there is thus barely enough place to insert the Dilithium algorithms in a non-breaking way into the bitfield. However, if new algorithms in the 0x800x range were to be implemented in the future, there would be a non-trivial conflict between the quantum-resistant custom algorithms and new standardized algorithms, which is an unenviable situation.

It should also be noted that while this hack is not the cleanest solution, this aspect of BearSSL's codebase has already been stated to need to be refactored by its main developer, which when done, would likely render this whole conflict moot.

In the case of the new TLS client hello extension, as defined in Figure 6.3, it is not fully supported by the underlying BearSSL code. Only the first Kyber key shall be read, and all others will be ignored. This was done in an attempt to not complexify the client's context, which would have had to keep track of multiple ephemeral Kyber keys, which would inflate by a non-negligible amount the size of the above-mentioned context.

In the case of the x509 engine, mixed certificate chains with Dilithium and conventional signature algorithms are supported, due to the desire to still allow for a server to be able to use a Dilithium certificate while the above intermediate certificates have not been re-issued as Dilithium certificates. While this would most likely be impossible to obtain a legitimate certificate from a third-party certificate authority, due to certificate request logistical issues, those issues are outside of the scope of this thesis.

While mixed certificate chains using conventional signature algorithms to certify a Dilithium certificate could be desirable, the other mixed chain, using Dilithium to certify a conventional certificate would be less than desirable, however, there is no support in the BearSSL x509 engine to distinguish the two, or even to prevent the support of either.

Chapter 7

Results

In this chapter, the main results of this thesis will be presented, as well as a comparison with existing TLS standards, and other quantum resistant TLS implementations.

7.1 Performance comparisons

In the context of this thesis, the main metric of interest is the TLS handshake performance. In particular, how fast the handshake is established with a peer, as well as how much more bandwidth is taken to do so. Both in the context of BearSSL with pre-existing ciphersuites, as well as with third party quantum-resistant TLS solutions.

7.1.1 BearSSL handshake speed

Firstly, as can be seen in Table 7.1, there is a table of all handshake durations with diverse cipher suites and security levels.

The exact cryptographic primitives in use for each category was as follows :

- **PQ security 1** consists of quantum-resistant primitives with an announced NIST security level of 1, that is Kyber-512 based key exchange along with Dilithium2 signature
- **PQ security 2** consists of quantum-resistant primitives with an announced NIST security level of 3, that is Kyber-768 based key exchange along with Dilithium4 signature
- **ECDHE/ECDSA** consists of traditional elliptic curve primitives, that is a Curve25519 based ECDHE key exchange, along with a P256 ECDSA signature

For each handshake, only a single certificate was configured for each peer, as well as using the same certificate type, signed with the same key type for both parties.

It should immediately be noted that the two purple cells were highlighted due to having a much larger standard deviation to the other tests, indeed, the typical standard deviation is every other test ranged between 3 and 6 ms, whereas for the two atypical results, their standard deviation was 12 and 20 ms. For the other results, however, the values are much more interesting, for the localhost tests, we obtained the expected results.

However, it can immediately be seen that for the Raspberry PI and WSL system, the handshakes without client authentication with a certificate were much slower, between 1.5 and 4 times slower than when client authentication was in place, even being faster than the EC based handshake, which is highly different to the localhost testing.

While the root cause for this discrepancy has not been found with certainty, it is speculated that it may be due to the underlying network stack buffering the network transmission until enough data should be sent which is likely to be equal to the MTU of the interface, until or an arbitrary timeout is reached. Indeed, with the use of a quantum-resistant cipher suite, the increased sizes would mean that during the *ClientKeyExchange* + *Certificate* + *CertificateVerify* step, the resulting network packet would be at least 6-7 Kilobytes large, whereas, in every other situation, the packet size never reaches above 1.2 Kilobytes, which incidentally, is under the default MTU threshold of most systems.

With everything considered, the handshake performance is well within acceptable limits, with a side-effect of possibly being much faster than conventional cipher suites in the case of mutually authenticated handshakes.

It is also important to keep in mind that the integrated Kyber and Dilithium implementations do not take advantage of vectorization, which would likely bring the resulting performance faster than conventional cipher suites, by a non-negligible amount. While an AVX accelerated reference implementation of Kyber and Dilithium exists for x86 platforms, on ARM, such an implementation is not yet available.

7.1.2 BearSSL handshake size

While the speed performance of the newly created cipher suite is quite acceptable, the full handshake size does roughly increase by an order of magnitude.

What follows in Table 7.2 is the sizes of the different handshakes from Table 7.1.

One can immediately see that the quantum-resistant handshakes end up being roughly an order of magnitude larger than their elliptic curve based counterparts. This increase could end up having some nasty side effects for typical TLS use cases.

Such as when considering some TLS use cases, such as a large number of interconnections during the usual internet usage. Such as where some arbitrary website will invoke three to

Without client authentication			
	PQ security 1	PQ security 3	ECDHE/ECDSA
localhost	7.771 ms	8.412 ms	7.437 ms
Raspberry PI 4	56.77 ms	68.88 ms	61.79 ms
Windows 10 WSL	53.96 ms	51.45 ms	56.02 ms
With client authentication			
	PQ security 1	PQ security 3	ECDHE/ECDSA
localhost	11.86 ms	12.50 ms	9.202 ms
Raspberry PI 4	29.54 ms	41.27 ms	65.51 ms
Windows 10 WSL	11.29 ms	18.70 ms	59.74 ms

Table 7.1: BearSSL handshake duration table,averaged over 20+ samples each.

	With client auth.	Without client auth.
PQ security 1	15'222 Bytes	9'499 Bytes
PQ security 3	23'034 Bytes	14'143 Bytes
ECDHE/ECDSA	1'978 Bytes	1'369 Bytes

Table 7.2: BearSSL handshake length.

five third-party TLS protected API calls that will probably return 100-200 KB of data, this new connection overhead could have devastating effects on users' usual network usage.

7.1.3 BearSSL crypto speed

In Figure 7.1, the performances of the Dilithium and Kyber integrations to BearSSL are compared to their reference and optimised implementations, the raw data can be found in Table 7.3.

As can be seen, BearSSL's code implementation is roughly 20 to 40% slower than the reference implementation. With the exception of the Dilithium signature implementation without FastSign, which as described Chapter 6.2.2, halves the memory usage, but makes signature creation rougly 80% slower than reference.

The most likely reason for this significant performance difference is due to the genericisation of Kyber and Dilithium's code. Indeed, since the referene code was using compiler macros to define every single parameter, the compiler could have behaved in a much more over-zealous fashion with its optimisation. Whereas in the case of BearSSL's implementation, size attributes and such information must be carried through all function calls, and thus further reduce the optimisation potential.

Another aspect to consider is that BearSSL's code is compiled with the size optimisation flag *-Os*, rather than the reference's level 3 speed optimisation flag *-O3*. There is also the aspec of

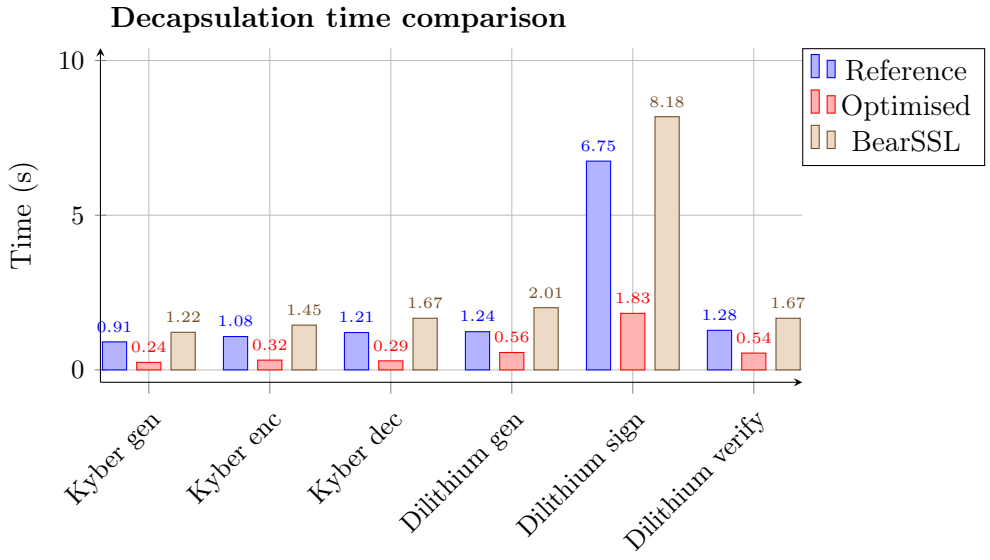

Figure 7.1: Final encryption algorithm performance analysis graph

BearSSL being a library rather than an executable, thus requiring the generation of position independent code with the *-fPIC*, which when taken into consideration with the binary size optimisation flag aspect, will definitely reduce code inlining possibilities throughout the code, leading to yet again, further performance costs.

7.1.4 BearSSL comparison with Liboqs

The first thing that should be remembered is that per Chapter 3, liboqs's OpenSSL implementation relies upon the TLS 1.3 protocol, rather that this thesis' TLS 1.2 protocol in BearSSL, which means thus that they rely upon the direct use of a KEM as a key exchange when using Kyber.

It should also be noted that liboqs is also using the AVX optimized variants of the Kyber and Dilithium cryptosystems.

As can be seen in Table 7.3, Liboqs' usage of the AVX variant of Kyber and Dilithium gives them a fair advantage regarding handshake speed.

Their handshake benchmark tool with Dilithium2 and Kyber-512 was able to conclude in average, with a very low standard deviation, a handshake every *0.45 ms*. Their results are roughly an order of magnitude faster than the ones found in this thesis, as per Table 7.1.

Part of the drastic difference can be seen by their usage of the faster *KEM as a key exchange protocol* in conjunction with the TLS 1.3 protocol, as well as their usage of AVX optimized

Raw data for BearSSL implementations of Kyber-1024 and Dilithium4			
Crystals-Kyber - 1024			10000 times
	keygen	enc	dec
Reference	907 ms	1079 ms	1209 ms
Optimised	244 ms	317 ms	293 ms
BearSSL	1216 ms	1448 ms	1670 ms
Liboqs	295 ms	363 ms	306 ms
Crystals-Dilithium - 4			10000 times
Reference	1236 ms	6749 ms	1280 ms
Optimised	563 ms	1828 ms	544 ms
BearSSL	2010 ms	8184 ms	1670 ms
BearSSL w/o FastSign	-	11670 ms	-
Liboqs	545 ms	1519 ms	650 ms

Table 7.3: data used for the creation of the implementations speed comparison graphs.

code and their ability to compile with the *-O3* optimization compiler flag.

Regarding handshake Size, their handshake without client authentication with Kyber-512 and Dilithium2 uses 7'915 bytes, whereas this thesis' uses 9'499 Bytes, likely due to an extra Kyber public key + ciphertext transfer.

All things considered, their software can boast excellent performance, along with better integration into third-party software, such as curl or Nginx.

7.2 Future improvements

While the resulting TLS implementation is performant enough for day to day usage, especially when compared with existing BearSSL cipher suites, it remains that there is a non-negligible amount of work required to implement the new BearSSL library into practical software such a Curl, Nginx, or Firefox.

It also is quite reassuring that the performance without using vectorized implementations of Kyber and Dilithium is enough for the TLS use case, which leads to a rather low hanging fruit, performance improvement wise, consisting of the implementation of Kyber and Dilithium AVX and Neon variants.

Then, there is the unification of Kyber and Dilithium's codebase, which while not likely difficult, as per Chapter 6.2, does require a fair bit of time to do.

Finally, the most interesting and time-consuming task would be to safely and properly implement Kyber and Dilithium using the available constant time BearSSL mathematical constructs for different word sizes.

Chapter 8

Conclusion

This thesis set out to implement an efficient, secure, and simple to use quantum-resistant TLS library. I believe that such a goal was achieved, indeed, boasting similar performances to some of the fastest conventional TLS cipher suites in BearSSL.

While the constant time claims of the implemented Kyber and Dilithium algorithms were not verified due to a lack of time, one can however reach a certain degree of confidence when taking in account that most of the critical parts of the algorithms, such as the polynomial uniform rejection sampling, noise generation and modular multiplication with Montgomery or NTT multiplication, were directly taken from the reference implementation. Along with the replacement of the Keccak based primitives in use replaced with the BearSSL's inner implementations, which are believed to be constant time.

Finally, due to having implemented the algorithm fully within the BearSSL API conventions, as well as inside the provided sample codes, along with a slew of test Dilithium certificates with their associated Dilithium keys, its usage is as simple as the conventional BearSSL library usage.

It should however be noted that while this thesis set out to ensure the full x509 infrastructure was quantum-resistant, in practice, due to BeaSSL's limited x509 support, such as not respecting rejection lists, this goal was only partially achieved. Nonetheless, all x509 features supported by BearSSL were made quantum-resistant.

Bibliography

[1] P.W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*. IEEE Comput. Soc. Press, 1994.

[2] Frank Arute, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph C. Bardin, Rami Barends, Rupak Biswas, Sergio Boixo, Fernando G. S. L. Brandao, David A. Buell, Brian Burkett, Yu Chen, Zijun Chen, Ben Chiaro, Roberto Collins, William Courtney, Andrew Dunsworth, Edward Farhi, Brooks Foxen, Austin Fowler, Craig Gidney, Marissa Giustina, Rob Graff, Keith Guerin, Steve Habegger, Matthew P. Harrigan, Michael J. Hartmann, Alan Ho, Markus Hoffmann, Trent Huang, Travis S. Humble, Sergei V. Isakov, Evan Jeffrey, Zhang Jiang, Dvir Kafri, Kostyantyn Kechedzhi, Julian Kelly, Paul V. Klimov, Sergey Knysh, Alexander Korotkov, Fedor Kostritsa, David Landhuis, Mike Lindmark, Erik Lucero, Dmitry Lyakh, Salvatore Mandrà, Jarrod R. McClean, Matthew McEwen, Anthony Megrant, Xiao Mi, Kristel Michielsen, Masoud Mohseni, Josh Mutus, Ofer Naaman, Matthew Neeley, Charles Neill, Murphy Yuezhen Niu, Eric Ostby, Andre Petukhov, John C. Platt, Chris Quintana, Eleanor G. Rieffel, Pedram Roushan, Nicholas C. Rubin, Daniel Sank, Kevin J. Satzinger, Vadim Smelyanskiy, Kevin J. Sung, Matthew D. Trevithick, Amit Vainsencher, Benjamin Villalonga, Theodore White, Z. Jamie Yao, Ping Yeh, Adam Zalcman, Hartmut Neven, and John M. Martinis. Quantum supremacy using a programmable superconducting processor. *Nature*, 574(7779):505–510, October 2019.

[3] Edwin Pednault, John Gunnels, Dmitri Maslov, and Jay Gambetta. On "quantum supremacy". IBM Research blog, 2019. `https://www.ibm.com/blogs/research/2019/10/on-quantum-supremacy/`.

[4] Daniel J. Bernstein. Introduction to post-quantum cryptography. In *Post-Quantum Cryptography*, pages 1–14. Springer Berlin Heidelberg, 2009.

[5] E. Berlekamp, R. McEliece, and H. van Tilborg. On the inherent intractability of certain coding problems (corresp.). *IEEE Transactions on Information Theory*, 24(3):384–386, 1978.

[6] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM*, 56(6):1–40, September 2009.

[7] Miklós Ajtai. The shortest vector problem in l2 is np-hard for randomized reductions (extended abstract). In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, STOC '98, page 10–19, New York, NY, USA, 1998. Association for Computing Machinery.

[8] O. Goldreich, D. Micciancio, S. Safra, and J.-P. Seifert. Approximating shortest lattice vectors is not harder than approximating closest lattice vectors. *Information Processing Letters*, 71(2):55 – 61, 1999.

[9] Jintai Ding and Bo-Yin Yang. *Multivariate Public Key Cryptography*, pages 193–241. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[10] David Jao and Luca De Feo. Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. In *Post-Quantum Cryptography*, pages 19–34. Springer Berlin Heidelberg, 2011.

[11] Jacqueline Brendel, Marc Fischlin, Felix Günther, Christian Janson, and Douglas Stebila. Challenges in proving post-quantum key exchanges based on key encapsulation mechanisms. Cryptology ePrint Archive, Report 2019/1356, 2019. `https://eprint.iacr.org/2019/1356`.

[12] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, and J. Alex Halderman. The matter of heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, IMC '14, page 475–488, New York, NY, USA, 2014. Association for Computing Machinery.

[13] Bodo Möller, Thai Duong, and Krzysztof Kotowicz. This poodle bites: Exploiting the ssl 3.0 fallback. 2014.

[14] Nimrod Aviram, Sebastian Schinzel, Juraj Somorovsky, Nadia Heninger, Maik Dankel, Jens Steube, Luke Valenta, David Adrian, J. Alex Halderman, Viktor Dukhovni, Emilia Käsper, Shaanan Cohney, Susanne Engels, Christof Paar, and Yuval Shavitt. DROWN: Breaking TLS with SSLv2. In *25th USENIX Security Symposium*, August 2016.

[15] A Langley. Boringssl. imperialviolet, oct. 2015.

[16] Quantum-safe wolfssl. www.wolfssl.com, Blog post, 2015. `https://www.wolfssl.com/quantum-safe-wolfssl-2/`.

[17] NIST. *Post-Quantum Cryptography, Round 2 Submissions*, 2019 (accessed July 29, 2020). `https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization/Call-for-Proposals`.

[18] NIST. *Post-Quantum Cryptography, Call for Proposals*, 2017 (accessed July 26, 2020). https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization/Call-for-Proposals.

[19] *Crystals-Kyber's Code licensing terms.* https://github.com/pq-crystals/kyber/blob/master/LICENSE.

[20] *Crystals-Kyber's NIST Intellectual Property statements.* https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/round-1/ip-statements/CRYSTALS-Kyber-Statements.pdf.

[21] *NewHope's Code licensing terms.* https://github.com/newhopecrypto/newhope/blob/master/README.md.

[22] *NewHope's NIST Intellectual Property statements.* https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/round-2/updated-ip-statements/NewHope-Statements-Round2.pdf.

[23] *NTRU Prime's NIST Intellectual Property statements.* https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/round-1/ip-statements/NTRU-Prime-Statements.pdf.

[24] *NTRU's Code licensing terms.* https://github.com/newhopecrypto/newhope/blob/master/README.md.

[25] *NTRU's NIST Intellectual Property statements.* https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/round-2/updated-ip-statements/NTRU-Statements-Round2.pdf.

[26] *Crystals-Dilithium's Code licensing terms.* https://github.com/pq-crystals/dilithium/blob/master/LICENSE.

[27] *Crystals-Dilithium's NIST Intellectual Property statements.* https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/round-1/ip-statements/CRYSTALS-Dilithium-Statements.pdf.

[28] *Falcon's Code licensing terms.* https://falcon-sign.info/impl/README.txt.html.

[29] *Falcon's NIST Intellectual Property statements.* https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/round-1/ip-statements/Falcon-Statements.pdf.

[30] *Sphincs$^+$'s Code licensing terms.* https://github.com/sphincs/sphincsplus/blob/master/LICENSE.

[31] *Sphincs$^+$'s NIST Intellectual Property statements.* https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/round-2/updated-ip-statements/SPHINCS-Plus-Statements-Round2.pdf.

[32] E. Rescorla. The transport layer security (tls) protocol version 1.2, 8 2008. RFC 5246.

[33] Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. Crystals – kyber: a cca-secure module-lattice-based kem. Cryptology ePrint Archive, Report 2017/634, 2017. https://eprint.iacr.org/2017/634.

[34] *IANA's Extension type values assignment table.* https://www.iana.org/assignments/tls-extensiontype-values/tls-extensiontype-values.xhtml.

[35] E. Rescorla. Elliptic curve cryptography (ecc) cipher suites for transport layer security (tls), 6 2006. RFC 4492.

[36] *IANA's Extension type values assignment table.* https://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml.

[37] Daniel A. F. Saraiva, Valderi Reis Quietinho Leithardt, Diandre de Paula, André Sales Mendes, Gabriel Villarrubia González, and Paul Crocker. PRISEC: Comparison of symmetric key algorithms for IoT devices. *Sensors*, 19(19):4312, October 2019.

[38] ANSI. Public key cryptography for the financial services industry: The elliptic curve digital signature algorithm (ecdsa). ANSI X9.62, 2005.

[39] Niels Samwel, Lejla Batina, Guido Bertoni, Joan Daemen, and Ruggero Susella. Breaking ed25519 in wolfssl. Cryptology ePrint Archive, Report 2017/985, 2017. https://eprint.iacr.org/2017/985.

List of Figures

List of Tables

Appendix A

LogBook

Date	Description
20.02.2020	Kick-off meeting with Prof. Alexandre Duc
02.2020	Study of the project specifications
26.02.2020	Second meeting with Prof. Alexandre Duc
02.2020	Research about the nist and it's call for proposals
02.2020 - 03.2020	Round 2 NIST algorithm analysis and selection
02.2020 - 03.2020	TLS libraries analysis and selection
11.03.2020	Third meeting with Prof. Alexandre Duc
18.03.2020	Fourth meeting with Prof. Alexandre Duc
03.2020 - 04.2020	Kyber crypto-system clean-up and integration in BearSSL
03.2020 - 04.2020	TLS protocol in depth analysis
04.2020 - 05.2020	Kyber crypto-system clean-up and integration in BearSSL
05.2020 - 06.2020	Dilithium crypto-system cleanup and integration in BearSSL
05.2020 - 06.2020	Initial quantum-resistant TLS protocol design phase
06.2020 - 07.2020	Quantum-resistant TLS implementation + Testing

Table A.1: Logbook