Implementing Post-quantum Cryptography for Developers

Julius Hekkala[1][a], Kimmo Halunen[2,3][b] and Visa Vallivaara[1][c]
[1]*VTT Technical Research Centre of Finland, Kaitoväylä 1, Oulu, Finland*
[2]*University of Oulu, Faculty of Information Technology and Electrical Engineering, Oulu, Finland*
[3]*National Defence University, Department of Military Technology, Helsinki, Finland*

Keywords: Post-quantum Cryptography, Lattice Cryptography, C++, Programming Library.

Abstract: The possibility of a quantum computer threatens modern public key cryptography. Post-quantum cryptographic algorithms are designed to protect sensitive data and communications also against an attacker equipped with a quantum computer. National Institute of Standards and Technology is standardizing post-quantum algorithms that could replace currently used public key cryptographic algorithms in key exchange and digital signatures. Lattice-based cryptography is one of the post-quantum algorithm groups with the biggest potential. Cryptography libraries are used by developers in all kinds of different solutions, but currently the availability of post-quantum algorithms in open-source libraries is very limited. Implementing post-quantum algorithms into a software library involves a multitude of challenges. We integrated three lattice-based post-quantum algorithms into a fork of Crypto++, a C++ cryptography library. We analyzed challenges in the implementation process and the performance and security of the fork. Especially the complex mathematical ideas behind the algorithms make implementation difficult. The performance of the algorithms was satisfactory but analyzing the security of the implementation in more detail is needed.

1 INTRODUCTION

In the modern information society, the security of communications and data is essential. The people need to be able to trust that their sensitive data remains secure in the future. Cryptography is used to prevent adversaries from accessing this information. Public key cryptography is commonly used to create secure connections, e.g. in TLS (Rescorla, 2018), and to digitally sign documents.

Public key cryptographic algorithms are often based on mathematical problems that are thought to be hard to solve. The most notable example of a public key cryptographic algorithm is RSA (Rivest et al., 1978), which is based on factoring large numbers produced by two prime numbers. Another very commonly used group of algorithms, elliptic curve cryptography (ECC) (Hankerson et al., 2006), draws their security from discrete logarithms.

Both of these problems are difficult for modern computers, and it is easy to answer the increase of computer performance by increasing key sizes. Quantum computers, on the other hand, can solve these problems much quicker. Shor's algorithm can be used to solve factoring (Shor, 1997) and discrete logarithm problems (Proos and Zalka, 2003) in polynomial time.

If a powerful enough quantum computer is built, the communications and data secured by conventional means are no longer secure. Because it is possible that in the future the attackers will have quantum computers, it is important to prepare well in advance. For this purpose, post-quantum cryptographic algorithms are designed. They use mathematical problems that are difficult for classical as well as quantum computers. Different standardization authorities have started standardizing post-quantum algorithms, e.g. NIST has an ongoing process standardizing post-quantum algorithms for key encapsulation methods (KEM) (Shoup, 2001) and digital signatures.[1]

Implementing cryptography correctly is difficult. There are many possible mistakes that can be made when designing and implementing cryptosystems, which can then lead to the security of the

[a] https://orcid.org/0000-0002-7558-9687
[b] https://orcid.org/0000-0003-1169-5920
[c] https://orcid.org/0000-0002-9638-244X

[1] https://csrc.nist.gov/projects/post-quantum-cryptography

Hekkala, J., Halunen, K. and Vallivaara, V.
Implementing Post-quantum Cryptography for Developers.
DOI: 10.5220/0010786200003120
In *Proceedings of the 8th International Conference on Information Systems Security and Privacy (ICISSP 2022)*, pages 73-83
ISBN: 978-989-758-553-1; ISSN: 2184-4356

whole program faltering. That is why most programs use specific cryptographic libraries that were implemented by experts instead of programming the cryptographic functionality from scratch every time.

These libraries include interfaces that make it easier to incorporate cryptography in different programs, and ideally prevent the misuse of the cryptographic primitives. A clear drawback is that the algorithms available for use are the ones that are implemented in the library. At this time, many of the most prominent open-source cryptographic libraries do not offer any post-quantum algorithms. On the other hand, developers are accustomed to using certain libraries. Thus it is not very easy for developers to use post-quantum algorithms in their programs. For example, the current version of OpenSSL[2] does not include any post-quantum algorithms that are part of the NIST standardization process.

In this paper we present experiences and findings on implementing post-quantum cryptographic primitives in a programming language library. We also evaluate the performance of the implemented primitives. The paper is organized as follows. The second chapter shines light on post-quantum cryptography and different post-quantum algorithms. Then we detail the background of implementing post-quantum algorithms in cryptographic libraries. In fourth chapter, we present our work. Fifth chapter showcases the results and analysis. In the sixth chapter we discuss the results as well as any observations made during the integration process. Finally, seventh chapter concludes this paper.

2 BACKGROUND

Public key cryptography is an essential enabling component in the modern information society. It allows parties to exchange keys to securely communicate using encrypted traffic in protocols like TLS, and allows others to verify that a digital signature is valid and no changes have been made to the signed document afterwards.

Quantum computers are able to compute some tasks more efficiently than the classical computers. Grover's algorithm (Grover, 1996) makes the brute-force attack against cryptographic algorithms faster. This effect can be negated quite easily by doubling used key sizes in algorithms. This affects all possible algorithms, but public key algorithms are particularly threatened by the quantum algorithms.

Most public key algorithms are based on mathematical problems that are deemed hard to solve with a classical computer. These hard problems include factoring and discrete logarithms. While there are no known algorithms for classical computers that would solve them efficiently, there are quantum algorithms that solve them in polynomial time, most notably Shor's algorithm (Shor, 1997).

Currently there is no quantum computer powerful enough that it would break the modern cryptographic algorithms. Breaking 1024 bit RSA would require about 2000 qubits, while breaking 160 bit ECC would require about 1000 qubits (Proos and Zalka, 2003). Google reported to have built a 53 qubit quantum computer in 2019 (Arute et al., 2019). Very recently IBM announced that they have developed a 127 qubit quantum processor (Chow et al., 2021). Breaking the algorithms is still quite far away, but if somebody is able to develop a quantum computer with the required amount of qubits, the currently common public key cryptographic algorithms like RSA and ECC will be vulnerable. Then adversaries will be able to decrypt any new and also some old encryptions of data.

When the first quantum computer is available for malicious adversaries, any data or communication also sent before that might become vulnerable, as perfect forward secrecy is not present everywhere, e.g. older versions of TLS support ciphers without forward secrecy. That is why it is important to prepare for that possibility already.

2.1 Post-quantum Cryptography and Standardization

Post-quantum cryptography contains algorithms that run on classical computers but are specifically designed in a way that makes them resistant against attacks with both classical and quantum computers. There is a multitude of different types of post-quantum algorithms that are based on different mathematical problems. Some of the algorithms are based on coding theory, multivariate polynomials and hashing. In this paper, we concentrate on lattice-based post-quantum algorithms.

During the last years, different standardization authorities have started making official standards for post-quantum algorithms. NIST (National Institute of Standards and Technology) started a standardization process in 2017, and it is currently in the third round. Anyone was able to submit an algorithm to the process at the start, and from those submissions, the first round candidates for standardization were chosen. Each round, when the candidates were

[2]https://www.openssl.org

published, researchers had chance to try to find any weaknesses in the algorithms. Also, the submitters were able to make improvements based on the findings. At the end of each round, the amount of algorithms that are candidates for standardization got smaller.

The NIST post-quantum cryptography standardization process involves two types of algorithms - key encapsulation mechanism (KEM) algorithms and digital signature algorithms. KEM algorithms are meant to replace public key algorithms used especially for key exchange, e.g. in TLS. They consist of three separate parts: a key generation algorithm, an encryption (encapsulation) algorithm, and a decryption (decapsulation) algorithm (Shoup, 2001).

At the time of writing this, there are four KEM algorithms and three digital signature algorithms among the finalists, all presented in Table 1. The KEM algorithms are Classic McEliece which is based on Goppa codes and CRYSTALS-Kyber, NTRU and SABER which are based on lattices. The digital signature algorithms are CRYSTALS-Dilithium and FALCON which are based on lattices and Rainbow which is based on multivariate polynomials.

Table 1: The third round algorithms of the NIST post-quantum cryptography standardization process.

Algorithm	Mathematical Problem
Classic McEliece	Goppa codes
CRYSTALS-Kyber	Lattices
NTRU	Lattices
SABER	Lattices
CRYSTALS-Dilithium	Lattices
FALCON	Lattices
Rainbow	Multivar. polynomials

2.2 Chosen Algorithms

In this paper, we present work performed with three of the third round candidate algorithms in the NIST standardization process. Two KEM algorithms and one digital signature algorithm were integrated. The chosen algorithms were CRYSTALS-Kyber (Bos et al., 2018), SABER (D'Anvers et al., 2018) and CRYSTALS-Dilithium (Ducas et al., 2018). All three algorithms are based on lattice problems. The foundation of lattice problems used in cryptography is the Shortest Vector Problem (SVP). With some conditions SVP is an NP-hard problem, i.e. it does not have a deterministic solution that runs in polynomial time (Regev and Rosen, 2006).

Regev (2005) presented a problem called LWE (Learning with Errors) that is based on SVP. Different variants of LWE have been introduced, e.g. Ring-LWE (RLWE) and Module-LWE (MLWE). Kyber and Dilithium base their security on MLWE. SABER is based on the modular version of the Learning with Rounding problem, which is different from LWE in that it is deterministic.

The three algorithms are all potential standards (Alagic et al., 2020). Since they all are based on lattice problems, comparison between the algorithms is more meaningful. Lattice-based algorithms have become one of the most potential replacements for the commonly used public key cryptographic algorithms, as is evident by the share of lattice-based algorithms present in the third round of the NIST standardization process. Even though they are relatively young, researchers are quite confident in their security. Another upside of lattice-based algorithms is that the key or ciphertext size is not massive, like is the case e.g. in algorithms that are based on multivariate polynomials.

2.3 Cryptographic Software Libraries

Cryptographic primitives are usually defined in mathematical notation, with mathematical formulae and operations. Implementing algorithms in a certain programming language is always a separate process from that with its own pitfalls. A mistake in the implementation will result in a security flaw even if the system is secure in theory (Lazar et al., 2014). It can be surprisingly difficult to implement cryptographic algorithms correctly.

That is why the general rule when needing cryptography is that there is no need to reinvent the wheel. There are plenty of open-source cryptographic libraries that have implemented the cryptographic primitives and offer a (hopefully) easy-to-use interface for the developer to integrate them in different programs. By using a cryptographic library one has succesfully avoided one of the biggest pitfalls.

If one wants to use common public key cryptographic algorithms like RSA or elliptic curve algorithms, there is an abundance of cryptographic libraries that implement these algorithms. On the other hand, if one wants to secure their program against quantum computers, options are very limited. There are example implementations of the algorithms and some open-source libraries that specifically focus on post-quantum cryptography, like *liboqs*[3] developed in the Open Quantum Safe project. The big and commonly used libraries often include very few post-quantum algorithms, if any at all.

[3]https://github.com/open-quantum-safe/liboqs

2.4 Dangers When using Open-source Cryptographic Libraries

Even if an algorithm has been designed to be secure, it does not mean that implementing or using it securely is easy. Mistakes in the implementation phase, misunderstandings and other mistakes undermine the security of the algorithm.

As all cryptography is implemented by people and people make mistakes, even high profile open-source libraries have bugs and vulnerabilities. One of the most well-known vulnerabilities is the Heartbleed bug, which was a mistake in the TLS implementation of OpenSSL and allowed adversaries to access critical information (Durumeric et al., 2014). Variation in the execution time of functions can lead to timing attacks, which can be very difficult vulnerabilities to prevent and detect (Almeida et al., 2016).

Even if the cryptographic algorithms are implemented correctly in the library, using them is not necessarily easy. In a study from 2014, out of the examined vulnerabilities 17 % were caused by mistakes made in the implementation of the library, and 83 % were caused by mistakes in the programs using the library, e.g. by the programs using the library incorrectly (Lazar et al., 2014). A study from 2013 found out that in over 88 % of the examined 10 000 Android applications in the Google Play store there was at least one mistake in the use of cryptographic interfaces (Egele et al., 2013).

These examples prove that smart design and clear interfaces need to be a priority when implementing cryptographic libraries. It is quite easy to make mistakes when using the functionality of the cryptographic libraries. A common mistake is not using cryptographically strong random number generators when generating randomness for an algorithm (Lazar et al., 2014).

It is not enough that the implementation of the cryptographic algorithm is correct and secure. Performance of the implementation is always crucial in real use, and has a great effect on what algorithms will be used, especially in cases where the operations will be performed often. In small IoT devices the choice of algorithms is very restricted.

2.5 Challenges in Implementing Post-quantum Cryptographic Algorithms

In principle, implementing post-quantum cryptographic algorithms does not fundamentally differ from implementing any other algorithm. As post-quantum algorithms run on the classical computer, unlike quantum cryptography that runs on a quantum computer, the challenges related to implementing cryptographic algorithms in general apply to implementing post-quantum algorithms.

A great challenge in implementing post-quantum cryptography is the mathematical complexity of the algorithms. This makes implementing the algorithm from just the specification very difficult for an average developer (Gaj, 2018). Currently used public key algorithms are based on mathematical problems that are much easier to understand.

Another challenge is the often greatly increased key size and sometimes the ciphertext size. Table 2 presents comparison of key sizes between post-quantum and commonly used public key algorithms. Especially the sizes of the private keys of the post-quantum algorithms are larger on the same security level as traditional public key algorithms. When implementing post-quantum cryptography on hardware, this becomes even more important. Because post-quantum algorithms use different operations than e.g. RSA, optimizing the hardware is more difficult (Gaj, 2018).

Table 2: Comparison of key sizes and ciphertext sizes of some post-quantum and traditional public key algorithms.

Algorithm	**Priv. key** (bytes)	**Public key** (bytes)	**Output** (bytes)
Kyber-512	1632	800	768
Kyber-768	2400	1184	1088
Kyber-1024	3168	1568	1568
LightSaber	1568	672	736
Saber	2304	992	1088
FireSaber	3040	1312	1472
RSA-3072	384*	384*	384
RSA-7680	960*	960*	960
RSA-15360	1920*	1920*	1920
Curve25519	251	256	

* Modulo.

It is important that there are implementations of post-quantum cryptographic algorithms in different languages and for different problems, so that replacing algorithms that are not quantum secure is easier. Table 3 shows the current amount of implemented asymmetric post-quantum algorithms in some of the most commonly used open-source libraries. Of the algorithms in the table not mentioned before, XMSS is a hash-based post-quantum signature scheme and NewHope is a lattice-based KEM algorithm that was removed from the NIST standardization process after the second round.

Many of the largest open-source cryptographic

Table 3: Availability of post-quantum asymmetric algorithms in selected open-source libraries at the time of the work.

Library	Language	Algorithms
OpenSSL	C	-
LibreSSL	C	-
Botan	C++	McEliece NewHope XMSS
Crypto++	C++	-
Bouncy Castle	Java	NewHope McEliece Rainbow...

libraries have not implemented any of the algorithms. For this, there are most likely quite a few reasons. The standardization is still ongoing and many are still awaiting which algorithm will be standardized. Also, many of the algorithms have undergone changes, mostly small parameter changes, during the process. As many of the algorithms are quite young in age, it is possible that there are still vulnerabilities to be found that greatly decrease their security.

3 ALGORITHM INTEGRATION

The availability of the NIST post-quantum algorithms in open-source cryptographic libraries is scarce. There are example implementations, mostly in C, and some libraries that specialize in the post-quantum niche. It would be a lot easier for the end-users of these algorithms, i.e. developers, researchers and hobbyists, if the algorithms were included in the cryptographic libraries they are used to using. That way the user will not have to learn how to use the new cryptographic library and how its interfaces work, reducing the threshold for trying out and adopting post-quantum algorithms.

We chose Crypto++[4] as the library under examination, as it did not have any of the algorithms from the NIST standardization process before. According to the Github 2020 The State of the Octoverse report,[5] C++ was the 7th most popular programming language in Github in 2020 and has been consistently popular over the years. We used the reference implementations of SABER, Dilithium and Kyber as a basis and integrated them into a fork[6] of Crypto++.

[4]https://cryptopp.com

[5]https://octoverse.github.com

[6]https://github.com/juliushekkala/cryptopp-pqc

3.1 Goals in the Integration Process

Bernstein et al. (2012) emphasize simplicity and reliability when designing an API for a cryptographic programming library. They also present countermeasures taken when designing how the library works to prevent vulnerabilities, e.g. by always reading randomness from the OS cryptographic random number generator instead of implementing their own complicated way to generate randomness. In that context, they also mention code minimization as an underappreciated goal in cryptographic software.

Forler et al. (2012) present their cryptographic API that is resistant against misuse. Namely, it is designed in a way that avoiding nonce reuse and plaintext leaking is easier. Green and Smith (2016) present 10 principles for implementing cryptographic library APIs in a usable way. Because a cryptographic library is used by other people than those who developed it, usability needs to be a focus when designing the cryptographic API.

When integrating the post-quantum algorithms into the fork of Crypto++, the following were our goals:

- Usability. Green and Smith (2016) mention the API needs to be easy to learn even for users without cryptographic expertise in addition to being easy to use. We see these as the most important aspects in our case. The API needs to be clear and concise. Changing parameters has to be easy.
- Conformity of the code with other algorithms of the library. As far as possible, the structure of the new algorithms' code is similar to the algorithms already present in the library. Also, library specific functions are used where applicable.
- Reliability. The algorithms work as specified and return the correct outputs.
- Prevent unnecessary code. This can be achieved e.g. by using library functions where appropriate.
- Performance. Algorithms need to be as fast as possible, without any unnecessary overhead.

3.2 Realization of the Goals

Because the chosen algorithms, Kyber, Dilithium and SABER, are quite a bit more complex than traditional public key algorithms, implementing them based on just the specification is really difficult. That is why we decided to use the reference implementations provided by the designers of the algorithms to the third round of the NIST standardization process.

Because the reference implementations were implemented with C and the language of the Crypto++ library was C++, choosing to work with the reference implementations directly as a basis was very advantageous. This way we were able to prevent much unnecessary code. We created a fork of the at the time most current version of Crypto++. The reference implementations were imported to the fork and changed from C code to C++ code where necessary.

A major goal in the implementation process was that a potential user would be able to use the implemented post-quantum algorithms, e.g. change between the different security levels, in a similar way to other algorithms already present in the library. Class structure was added, as well as the API through which the user is able to use the algorithms. The API specified by NIST for the reference implementations works as a basis, as we saw no reason to invent the wheel anew, in other ways we tried to have the API work as similarly as other algorithms in Crypto++.

Platform independency was also desirable, at least that using the fork would be possible on both Linux and Windows operating systems.

3.3 Challenges in the Integration Process

Throughout the implementation process, the biggest issue that needed a lot of consideration was the question of how the multitude of parameters should be handled. The C reference implementations manage the parameters using compile-time constants, and as that would require the user to compile the library again if they want to change the security levels of the post-quantum algorithms, it was not a suitable choice for us. Two different solutions were used for this in the implementations. In the implementations of Kyber and SABER, templates were used to handle the parameters. A base class of the algorithm was created, that had two template parameters that were used to define all the algorithm parameters. When using the algorithm, the user would call a subclass of the base class that creates an instance of the base class with the correct template parameters.

Using templates is quite an elegant solution, unless there is a need for a lot of template parameters. In case of Kyber and SABER it was just two, but for Dilitihium we would have needed quite a list to make it work. For code clarity, it was decided to go another route with Dilithium. With Dilithium we ended up using just member variables, that are changed by the subclass creating an instance of the base class. This caused some additional compulsory changes. In the reference implementation, polynomials and vectors have been implemented using *structs*. The size of arrays in structs needs to be defined before compiling, and this was not possible, as there was variation in the sizes between security levels. Polynomials were switched to work with *std::array* and vectors with *std::vector*. Even if this caused in turn some changes in the methods, it was not very difficult, as *std::array* and *std::vector* make it possible to work on their inner data.

Debugging presented quite a challenge at times. As the algorithms involve a lot of different advanced mathematical operations and rely heavily on randomness, it is not easy to determine whether the state of the program is correct at some point. We used the reference implementation with randomness removed in the debugging process to find out where the execution of the C++ version goes wrong. That way we were able to find bugs created in the integration process.

Platform independency caused some trouble, as using variables to define arrays is possible with GCC compiler, but not on many others, as it is not in accordance with the official C++ standard. Because of this we had to change from using arrays to using *std::vector* at many points.

Because the NIST standardization process is not finished, the details of the algorithms and the parameters can change. When we started evaluating the implementation, the security levels of Dilithium had radically changed from the version that was implemented to the work. We had to update Dilithium to an up-to-date version to evaluate it against the newest version.

4 RESULTS

The performance of the algorithms integrated to the fork was analysed by measuring clockcycles with *__rdstc()* instruction and compared against the measured performance of the reference implementations on the same device. All measurements were done on a laptop (i7-10875H CPU @ 2.30 *GHz* x16) running Ubuntu 18.04 and the programs were initially compiled with G++ using the predetermined compiler options of Crypto++ and the reference implementations respectively. Using different compilers and compiler optimizations will bring forth differences in the performance.

4.1 Performance Analysis

When the SABER algorithm first was implemented in the library, the performance was not satisfactory. Compared to Kyber, the integration had decreased the performance by much more. The runtime of the algorithm was double of the reference implementations, even though not many changes had been made. After debugging, it was found what part of SABER was the bottleneck. One design choice of SABER is to use integers moduli that are powers of 2 (D'Anvers et al., 2018). While this makes the other parts of the implementation simpler, it prevents SABER from using the number theoretic transform (NTT) that is used in Kyber to speed up calculations. The polynomial multiplication algorithm is interchangable and needs to be decided by the implementers of the algorithm. In the reference implementation, the authors use Toom-Cook-Karatsuba multiplication. For the sake of time and resource management, we used the construction provided by the authors in the reference implementation. Even though the same construction was used with very little changes, the performance decrease was close to 100 %.

Inspecting the compiler options used in Crypto++ by default and the reference implementation revealed the cause. After debugging, the compiler option *-march=native* used in the reference implementation was found to be the culprit. With this option enabled, the compiler will detect the architecture of the system and automatically optimize the code for that architecture. Without this option, the compiler was not able to optimize Toom-Cook-Karatsuba construct in the way that was intended by the authors of the algorithm. While changing the compiler options of the library on Linux was successful, Microsoft Visual Studio Compiler does not offer similar functionality, and the optimization has to be done separately for each type of machine.

Table 4 presents the performance measurements of the integrated algorithms, with the average runtime of different versions of the algorithms measured in CPU clockcycles. The performance of the integrated C++ versions, both with and without the *-march=native* GCC optimization flag, was compared against the reference implementations. Without optimization, both KEM algorithms were slower than the reference implementation, especially SABER. Optimization improved performance of both algorithms, but the difference was radical with SABER, which is emphasized with red and green colors in the runtime change in Table 4 respectively. With *-march=native* enabled, the performance of SABER was even better than that of the reference implementation.

In addition to presenting the performance of Kyber and SABER, table 4 includes Dilithium performance after updating it to the newest parameter set. Without any custom optimization, the C++ implementation was very close in performance to the original C implementation. With *-march=native* enabled, the C++ implementation siginificantly outperformed the C implementation.

4.2 Security Analysis

The implementation was analyzed with Valgrind[7] to find out possible memory leaks. No memory leaks were found in the C++ implementation. Side channel attacks are always a risk when implementing cryptographic algorithms, especially when optimization is present. Side channel attack analysis remains future work.

Naturally, we had to be careful that there are no possible parts in the algorithm where an attacker could determine secrets based on execution times. Constant time functions were used e.g. for comparing the equality of byte arrays.

5 DISCUSSION

The possible emergence of quantum computers presents a threat especially to the current public key infrastructure. As there are systems where data and communications need to be secure for years of time, preparing for possible quantum computers is necessary. Lattice-based algorithms are an emerging group of post-quantum algorithms that are one potential replacement for the currently used non-quantum-resilient algorithms. As they are still quite young, it is entirely possible that systematic attacks will be found against them. The current understanding is that the algorithms of this paper, Kyber, Dilithium and SABER, are not affected by serious attacks.

As the NIST standardization process is still ongoing, it is understandable that many big libraries have not implemented the algorithms. The algorithms may still undergo and have undergone changes during the process, as was the case with Dilithium during our work. Also, it is always possible that serious vulnerabilities are found. Like with AES, usually the standardized algorithm will spread to use and others will be less relevant. From that perspective, it makes

[7] https://valgrind.org

Table 4: Performance of Kyber, SABER and Dilithium on different security levels. The reference implementation performance is compared first against performance when compiling the fork with default options, and then when compiling with *-march=native* eanabled. The red and green color are used to emphasize the difference in the performance of SABER with and without the option.

Algorithm	**Function**	**CPU cycles** (ref. impl.)	**CPU cycles** (C++ default)	**Runtime increase** (%)	**CPU cycles** (C++ with opt.)	**Runtime increase** (%)
Kyber-512	Key gen.	70527	77452	10	70106	-1
	Encaps.	90177	103454	15	93220	3
	Decaps.	106539	124722	17	114415	7
Kyber-768	Key gen.	119089	137032	15	124357	4
	Encaps.	141759	170474	20	152433	8
	Decaps.	162677	196737	21	180212	11
Kyber-1024	Key gen.	181525	213370	18	191942	6
	Encaps.	205682	251775	22	226342	10
	Decaps.	230457	283941	23	259733	13
LightSaber	Key gen.	43321	84330	95	46551	7
	Encaps.	56580	106420	88	51789	-8
	Decaps.	62496	121883	95	55970	-10
Saber	Key gen.	80317	158330	97	80856	1
	Encaps.	100271	193770	93	91811	-8
	Decaps.	109088	218243	100	97211	-11
FireSaber	Key gen.	132859	257348	94	127774	-4
	Encaps.	156832	305424	95	143594	-8
	Decaps.	169193	340069	101	153203	-9
Dilithium2	Key gen.	216791	201662	-7	175171	-19
	Sign.	920338	863546	-6	788545	-14
	Verif.	214877	228392	6	201943	-6
Dilithium3	Key gen.	350129	370364	6	321002	-8
	Sign.	1382591	1397358	1	1300339	-6
	Verif.	338629	364654	8	320600	-5
Dilithium5	Key gen.	532097	550770	4	471764	-11
	Sign.	1674901	1697241	1	1569844	-6
	Verif.	562763	594885	6	518180	-8

sense not to implement algorithms that will not be used.

5.1 Biggest Challenges in Implementing Lattice-based Post-quantum Cryptography

There is a multitude of challenges affecting implementing lattice-based post-quantum cryptography. They are harder to implement than algorithms that are based on factoring and discrete logarithms, as the mathematical operations and theory are much more complex. This makes creating an efficient and well optimized implementation quite difficult. Implementing based on just the specification is challenging, and without the reference implementations as a basis, the implementation would not have been that successful.

Debugging posed a problem in the implementation process. It can be hard to find the problematic parts in the implementation as the implementations are complicated. Because many of the lattice-based algorithms are not deterministic, the output is every time different for the same input, and it requires extra effort to be able to debug the implementation. In our work we used the reference implementations in the debugging process, by determining with that program what the state of the variables should be in each part of the code. That way we were able to find out where mistakes had been made. If no such reference implementation is easily available, debugging becomes tedious.

In general use, the key and output sizes and the use of memory of lattice-based algorithms are not a problem. In memory-constrained environments, though, it is not possible to use every lattice-based algorithm. Also, some communication protocols are not suitable for lattice-based algorithms because of e.g. constraints on the message length.

5.2 Were the Implementation Goals Achieved?

In general, the goals we set in 3.1 were mostly achieved. The library specific functions were used wherever possible, and the structure is quite similar to using other algorithms. Performance was also satisfactory, especially after compiler optimization. There were some things, though, that left a bit to be desired.

One of our goals was to make the algorithms in the fork as usable as possible. Now, using cryptographic primitives correctly is never going to be the easiest thing to do, but as we stayed very close to the structure of the API specified by NIST, using the algorithms is quite simple. It would be possible to evaluate the usability of the algorithms in the library by using C++ developers as evaluators. For usability, one thing that is missing from the current implementation is handling coding errors, e.g. using too small buffers. Currently this leads in most cases to just the compiled executable crashing. Exception handling would improve the usability, but it is important to consider the points of code where it could be useful for the developer in order not to introduce bloated code.

Another one of the goals was preventing unnecessary code. Again, we partly succeeded - e.g. the hash functions in the reference implementations were as a rule exchanged to using already present library implementations. With better planning and carefully designing the class structure more could have been achieved. Kyber and Dilithium are based on the same framework and work very similarly, and the code could have been designed in a way that they use common code components for e.g. NTT functionality. In the current implementation the NTT functionality (essentially being the same) can be found twice, for each Kyber and Dilithium separately. This was mostly caused by our process of integrating each algorithm one by one. A more careful consideration of the overall picture would have been beneficial.

One of the points to take into account when designing a cryptographic API that we mentioned in 3.1 was misuse resistance. The API in our implementation is quite resistant against misuse. The algorithms are designed in a way that the user does not supply any own nonces to the algorithms, so nonce reuse will not be an issue. Additionally, the randomness needed in the algorithms is generated inside the algorithms and not supplied by the user. Still, it is quite possible that the users would be able to use the algorithms wrong, and would need more testing for definite results.

While the measured performance of the integrated algorithms was satisfactory, it would be interesting to examine it more thoroughly. For example, the performance of SABER is even better in the integrated C++ version when using *-march=native* than in the original C version. One could try to find out whether this is because of the changes made in our work or because of some compiler optimization differences, et cetera.

5.3 Comparison of the Implemented Algorithms

The NIST post-quantum standardization process is still underway. Therefore it is meaningful to make some comparisons between SABER and Kyber, as both are potential standards (Alagic et al., 2020). Based on our work, one cannot make definite conclusions about which of the algorithms is ultimately better.

As the difference in the key and output sizes is small and performance is not radically different, these should not be the most importnat point in standardization decisions. From these aspects, both of the algorithm are suitable for standardization.

Kyber is structurally a lot more complex than SABER. SABER does put a lot of focus on achieving as much simplicity in the algorithm as possible, which probably makes it easier to implement. Kyber is part of the same framework as Dilithium, which is one of the candidates to be standardized as a digital signature algorithm. This is a clear advantage for Kyber.

5.4 Future Work

The implementation can be used to further examine and evaluate the algorithms. The security of the algorithms and the implementation can be examined for example by fuzzing. The fork can be used to examine the applicability of the algorithms in different protocols and scenarios. Different compiler optimizations can be experimented with and investigated whether they cause side channel vulnerabilities.

As the algorithms change during the NIST standardization process, they need to be updated to keep up with the changes. Maintenance and improvement of the library is needed, too. Currently only the reference versions of the algorithm implementations are integrated to the library, while all the algorithm submissions also included AVX2 (*Advanced Vector Extensions 2*) optimized versions.

Integrating them would improve the performance on devices that can use those processor instructions.

Naturally, other post-quantum algorithms can be implemented into the created fork. Another challenge would be to implement e.g. a Java version of the algorithms, as there are no Java reference implementations available that we know of.

6 CONCLUSIONS

Post-quantum cryptography aims to provide an answer to the emergence of quantum computers that threaten especially the public key cryptographic ecosystem. Lattice-based algorithms are one type of post-quantum algorithms that are likely to increase in use in the coming years. We integrated into a cryptographic library fork three lattice-based algorithms from the third round candidates of NIST post-quantum cryptography standardization process, namely KEM algorithms Kyber and SABER and digital signature algorithm Dilithium.

We examined the challenges and possible pitfalls of implementing post-quantum cryptographic algorithms in software libraries. The mathematical complexity of the algorithms and difficult to understand specification provide a challenge for the people implementing the algorithms, and extra attention is required not to create possible security issues. If one algorithm is easier to implement than another, it is an immense advantage, as easier implementation means less risk of implementation errors.

ACKNOWLEDGEMENTS

This research was supported by PQC Finland project funded by Business Finland's Digital Trust program.

REFERENCES

Alagic, G., Alperin-Sheriff, J., Apon, D., Cooper, D., Dang, Q., Kelsey, J., Liu, Y.-K., Miller, C., Moody, D., Peralta, R., et al. (2020). Status report on the second round of the NIST post-quantum cryptography standardization process. https://csrc.nist.gov/publications/detail/nistir/8309/final.

Almeida, J. B., Barbosa, M., Barthe, G., Dupressoir, F., and Emmi, M. (2016). Verifying Constant-Time Implementations. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 53–70, Austin, TX. USENIX Association.

Arute, F., Arya, K., Babbush, R., Bacon, D., Bardin, J. C., Barends, R., Biswas, R., Boixo, S., Brandao, F. G., Buell, D. A., et al. (2019). Quantum supremacy using a programmable superconducting processor. *Nature*, 574(7779):505–510.

Bernstein, D. J., Lange, T., and Schwabe, P. (2012). The Security Impact of a New Cryptographic Library. In *Progress in Cryptology – LATINCRYPT 2012*, pages 159–176, Berlin, Heidelberg. Springer Berlin Heidelberg.

Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J. M., Schwabe, P., Seiler, G., and Stehlé, D. (2018). CRYSTALS-Kyber: a CCA-secure module-lattice-based KEM. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 353–367. IEEE.

Chow, J., Dial, O., and Gambetta, J. (2021). IBM Quantum breaks the 100-qubit processor barrier. https://research.ibm.com/blog/127-qubit-quantum-processor-eagle. Accessed: 2021-11-26.

D'Anvers, J.-P., Karmakar, A., Roy, S. S., and Vercauteren, F. (2018). Saber: Module-LWR based key exchange, CPA-secure encryption and CCA-secure KEM. In *International Conference on Cryptology in Africa*, pages 282–305. Springer.

Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schwabe, P., Seiler, G., and Stehlé, D. (2018). CRYSTALS-Dilithium: A Lattice-Based Digital Signature Scheme. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(1):238–268.

Durumeric, Z., Li, F., Kasten, J., Amann, J., Beekman, J., Payer, M., Weaver, N., Adrian, D., Paxson, V., Bailey, M., and Halderman, J. A. (2014). The Matter of Heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, IMC '14, pages 475–488. Association for Computing Machinery.

Egele, M., Brumley, D., Fratantonio, Y., and Kruegel, C. (2013). An Empirical Study of Cryptographic Misuse in Android Applications. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, page 73–84, New York, NY, USA. Association for Computing Machinery.

Forler, C., Lucks, S., and Wenzel, J. (2012). Designing the API for a Cryptographic Library. In *Reliable Software Technologies – Ada-Europe 2012*, pages 75–88, Berlin, Heidelberg. Springer Berlin Heidelberg.

Gaj, K. (2018). Challenges and Rewards of Implementing and Benchmarking Post-Quantum Cryptography in Hardware. In *Proceedings of the 2018 on Great Lakes Symposium on VLSI*, GLSVLSI '18, page 359–364, New York, NY, USA. Association for Computing Machinery.

Green, M. and Smith, M. (2016). Developers are Not the Enemy!: The Need for Usable Security APIs. *IEEE Security Privacy*, 14(5):40–46.

Grover, L. K. (1996). A fast quantum mechanical algorithm for database search. In *Proceedings of the*

twenty-eighth annual ACM symposium on Theory of computing, pages 212–219.

Hankerson, D., Menezes, A. J., and Vanstone, S. (2006). *Guide to elliptic curve cryptography*. Springer Science & Business Media.

Lazar, D., Chen, H., Wang, X., and Zeldovich, N. (2014). Why does cryptographic software fail? A case study and open problems. In *Proceedings of 5th Asia-Pacific Workshop on Systems*, pages 1–7.

Proos, J. and Zalka, C. (2003). Shor's discrete logarithm quantum algorithm for elliptic curves. *Quantum Information & Computation*, 3(4):317–344.

Regev, O. (2005). On Lattices, Learning with Errors, Random Linear Codes, and Cryptography. In *Proceedings of the Thirty-Seventh Annual ACM Symposium on Theory of Computing*, STOC '05, page 84–93, New York, NY, USA. Association for Computing Machinery.

Regev, O. and Rosen, R. (2006). Lattice Problems and Norm Embeddings. In *Proceedings of the Thirty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '06, page 447–456, New York, NY, USA. Association for Computing Machinery.

Rescorla, E. (2018). The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446. https://rfc-editor.org/rfc/rfc8446.txt.

Rivest, R. L., Shamir, A., and Adleman, L. (1978). A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Commun. ACM*, 21(2):120–126.

Shor, P. W. (1997). Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *SIAM Journal on Computing*, 26(5):1484–1509.

Shoup, V. (2001). A Proposal for an ISO Standard for Public Key Encryption. Cryptology ePrint Archive, Report 2001/112. https://eprint.iacr.org/2001/112.pdf.