SN Computer Science (2023) 4:365
https://doi.org/10.1007/s42979-023-01724-1

ORIGINAL RESEARCH

# Implementing Post-quantum Cryptography for Developers

**Julius Hekkala[1] · Mari Muurman[1] · Kimmo Halunen[2,3] · Visa Vallivaara[1]**

Received: 18 October 2022 / Accepted: 1 February 2023 / Published online: 29 April 2023


**Abstract**
Widely used public key cryptography is threatened by the development of quantum computers. Post-quantum algorithms have been designed for the purpose of protecting sensitive data against attacks with quantum computers. National Institute of Standards and Technology has recently reached the end of the third round of post-quantum standardization process and has published three digital signatures and one key encapsulation mechanism for standardization. Three of the chosen algorithms are based on lattices. When implementing complex cryptographic algorithms, developers commonly use cryptographic libraries in their solutions to avoid mistakes. However, most of the open-source cryptography libraries do not yet have post-quantum algorithms integrated in them. We chose a C++ cryptography library, Crypto++, and created a fork where we integrated four lattice-based post-quantum algorithms. We analyzed the challenges in the process as well as the performance, correctness and security of the implemented algorithms. The performance of the integrated algorithms was overall good, but the integration process had its challenges, many of which were caused by the mathematical complexity of lattice-based algorithms. Different open-source implementations of post-quantum algorithms will be essential to their easier use for developers. Usability of the implementations is also important to avoid possible mistakes when using the algorithms.

**Keywords** Post-quantum cryptography · Lattice cryptography · C++ · Programming library

## Introduction

In today's connected world, we send out millions of messages every day through different channels and networks. These messages can contain very sensitive information, such as financial details or health care data. Thus, it has become of the utmost importance to secure communications from untrusted adversaries. This security is achieved by using cryptography. With cryptographic algorithms, we can hide the sensitive information, so that adversaries are unable to access the concealed data. A common way to secure a channel or digitally sign documents is called public key cryptography which is used for example in the Transport Layer Security (TLS) protocol [1].

The foundation behind public key cryptography is a set of mathematical problems which are believed to be hard to solve. Many currently deployed public key algorithms are either based on the factorization of a large number produced by two prime numbers, like RSA [2], or on discrete logarithms, e.g., elliptic curve cryptography (ECC) [3]. A modern computer is unable to solve these problems in practice but the development of quantum computers will make both of these public key cryptographic algorithms vulnerable. Peter Shor developed the so-called Shor's algorithm already in 1994 and it can be used to solve the factorization problem [4] as well as the discrete logarithm problem [5] in polynomial time using a quantum computer.

This article is part of the topical collection "Advances on Information Systems Security and Privacy" guest edited by Steven Furnell and Paolo Mori.

✉ Visa Vallivaara
visa.vallivaara@vtt.fi

Julius Hekkala
julius.hekkala@gmail.com

Mari Muurman
mari.muurman@vtt.fi

Kimmo Halunen
kimmo.halunen@oulu.fi

[1] VTT Technical Research Centre of Finland, Kaitoväylä 1, Oulu, Finland

[2] Faculty of Information Technology and Electrical Engineering, University of Oulu, Oulu, Finland

[3] Department of Military Technology, National Defence University, Helsinki, Finland

It has become clear that when quantum computers reach their full potential, the current communication methods in use are no longer secure. Even though it is uncertain when the quantum threat will become a reality, it is necessary to prepare for it beforehand. Post-quantum cryptographic algorithms are designed for this purpose. The post-quantum algorithms utilize mathematical problems which are hard to solve for a classical computer as well as for a quantum computer. Several standardization authorities have begun official standardization processes to standardize post-quantum algorithms. For example, NIST (National Institute of Standards and Technology) has coordinated competition-like processes for the standardization of post-quantum key encapsulation methods (KEM) and digital signature algorithms.

The implementation of complex cryptographic algorithms in a correct manner can be difficult. It can be difficult to determine whether the implementation behaves correctly in all different scenarios and it is very easy to introduce subtle bugs in the implementation. Any mistakes in design or implementation can lead to security flaws, e.g., side channel vulnerabilities in the case of embedded systems. For this reason, cryptographic libraries are used in many programs instead of implementing the algorithms from scratch each time. However, post-quantum algorithms have rarely been implemented to open-source cryptographic libraries up to this point.

In this paper, we discuss the process of implementing post-quantum cryptographic algorithms into a programming language library. We chose Crypto++,[1] which is a C++ language general use cryptographic library with a wide variety of different algorithms and created a fork [2] of it to integrate post-quantum algorithms. Our work was first published as a conference paper [6] at ICISSP 2022 [7], and we expand upon it in this paper. To extend our work, in addition to improving the code in the fork we integrated another post-quantum algorithm into the fork, and used Known Answer Tests to further investigate the validity of the implementation.

This paper is divided as follows. First, we dive into the background of post-quantum algorithms and open-source implementations, and then, we describe the work done. After that follows results on testing and performance and security analysis. Then, we discuss the results and experiences during our work, after which the last chapter concludes this paper.

## Background

Quantum computers exploit the laws of quantum mechanics to solve complex mathematical problems more efficiently than a conventional computer. Quantum cryptography means cryptographic algorithms that run on quantum computers, whereas post-quantum cryptography means algorithms that run on a classical computer but protect the secured data against attacks coming from quantum computers. The first quantum algorithms have already been developed in the 1990 s.

Two quantum algorithms in particular have had an influence on the security of modern cryptographic algorithms. First, Grover's algorithm [8] speeds up a type of database search quadratically which results in a weakening of symmetric cryptographic algorithms. This can still be fixed by doubling the key sizes in the algorithms. Second, Shor's algorithm [4] provides an exponential speed up to the integer factorization problem and essentially breaks algorithms based on factorization and discrete logarithms. This means that a powerful quantum computer has the potential to break RSA, Diffie-Hellman and Elliptic Curve Cryptography (ECC) that is, essentially all currently used public key cryptography.

Theoretically, it would require a quantum computer with 1000 logical qubits to break a 160-bit elliptic curve cryptographic key and a quantum computer with 2000 logical qubits to factor a 1024-bit RSA modulus [5]. However, a powerful enough quantum computer to achieve this does not exist yet. For larger scale quantum computation, each logical qubit needs to be encoded into several physical qubits increasing the required number of the qubits. It has been recently evaluated that with 20 million noisy qubits it would be possible to factor the 2048-bit RSA integers in only 8 h [9]. Currently, IBM seems to be in the possession of the most powerful quantum computer measured in qubits with its 433-qubit Osprey quantum processor [10]. This means that in reality quantum computers capable of breaking public key cryptographic algorithms are still quite far away, but they are an upcoming threat that needs to be addressed.

Although the realization of powerful quantum computers is still far in the future, it is already highly important to recognize the threat and start preparing for it. One of the most important arguments to not delay the transition to post-quantum algorithms is the so called store-now, decrypt-later attack. In the attack the adversary collects the current encrypted information and waits until they have access to a quantum computer able to decrypt it. The attack makes systems which store confidential information for long periods of time, such as banking data or medical records, vulnerable. Another reason to acknowledge the need for post-quantum algorithms now, are long projects that are designed now and are planned to still be in use years, maybe decades later. A good example are vehicles which are going to be in use even 20 or 30 years later. It is also clear that the transition to post-quantum algorithms is going to take some time. [11] For all these reasons, it is highly important to recognize the threat of quantum computers already in advance and to start the transition to post-quantum algorithms.

[1] https://cryptopp.com

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

## Post-quantum Cryptography Standardization

The post-quantum cryptographic algorithms are designed to run on classical computers but meant to resist attacks from both classical and quantum computers. Different post-quantum algorithms are based on complex mathematical problems, such as lattices, coding theory or multivariate polynomials. In this paper, we focus on the lattice-based algorithms.

Standardization authorities have begun to formulate official standards for post-quantum algorithms. One of these authorities is NIST who began their standardization process in 2017 through a public, competition-like process. NIST's earlier competitions have led to the standardizations of the Advanced Encryption Standard (AES) [12] in 2001 and the Secure Hash Algorithm (SHA-3) [13] in 2015 which have been deployed widely all over the world. Unlike in their past competitions, NIST intends to choose multiple algorithms for standardization in order to have some variety in post-quantum cryptography. NIST received 82 submission packages out of which 69 met the requirements and were accepted. After public review and comments, seven finalists and eight alternative algorithms were chosen to enter the third round in 2020. These algorithms and the mathematical problems they are based on are presented in Table 1. The algorithms can be divided into two types—digital signatures and key encapsulation mechanisms (KEM), which can be used in key exchange. KEM algorithms are comprised of three parts—key generation, encryption (called encapsulation) and decryption (called decapsulation) algorithms [14].

Recently, in 2022, NIST has chosen three digital signature algorithms and one KEM algorithm to be standardized as well as four KEM algorithms to continue to the fourth round [17]. From the digital signatures, CRYSTALS-Dilithium [18], FALCON [19] and SPHINCS+ [20], which was one of the alternative schemes in third round, were chosen to be standardized. Both Dilithium and FALCON are based on structured lattices but were both selected for standardization since they are suitable for different types of applications. SPHINCS+ was also chosen in order not to rely solely on the security of lattices. SPHINCS+ is a stateless hash-based signature scheme and it brings some variation to the other two schemes to be standardized. However, because the SPHINCS+ scheme is different from the lattice-based schemes and still quite new, it is vulnerable to new attacks. A new forgery attack has already emerged against category five SPHINCS+ using SHA-256 [21]. This vulnerability questions the algorithm's maturity for standardization. NIST has chosen to standardize three signature schemes at the moment and to remove the other candidates from consideration. However, NIST has already issued a new call for proposals for post-quantum signatures [22] in order to possibly standardize more signature schemes in the future.

**Table 1** Third round algorithms of the NIST's post-quantum cryptography standardization process

| Algorithm | Usage | Mathematical problem |
|---|---|---|
| Classic McEliece | KEM | Goppa codes |
| CRYSTALS-Kyber | KEM | Lattices |
| NTRU | KEM | Lattices |
| SABER | KEM | Lattices |
| CRYSTALS-Dilithium | Digital signatures | Lattices |
| FALCON | Digital signatures | Lattices |
| Rainbow (broken [15]) | Digital signatures | Multivar. polynomials |
| BIKE | KEM | Codes |
| FrodoKEM | KEM | Lattices |
| HQC | KEM | Codes |
| NTRUPrime | KEM | Lattices |
| SIKE (broken [16]) | KEM | Isogenies of elliptic curves |
| GeMSS | Digital signatures | Multivar. polynomials |
| Picnic | Digital signatures | Symmetric primitives |
| SPHINCS+ | Digital signatures | Hash functions |

From the three structured lattice-based KEM finalists on third round, CRYSTALS-Kyber [23] was chosen for standardization. NIST found the specific problem, that Kyber is based on, more convincing than the other assumptions of the lattice-based KEMs [17]. Since the three lattice-based finalists were relatively comparable to each other, NTRU [24] and Saber [25] are no longer being considered for standardization. Some of the other KEM algorithms advanced to a fourth round and remain under the study of the community. The KEMs advancing to the fourth round are BIKE, Classic McEliece, HQC and SIKE. From these fourth round candidates, SIKE has already been broken after the announcement [16].

## Cryptographic Open-Source Software Libraries

To avoid mistakes in complex implementations of cryptographic algorithms, it is useful to rely on open-source cryptographic libraries. There are a great number of different cryptographic libraries available in many different languages which offer easy to use interfaces for a developer to add in their own software, yet many of the cryptographic libraries do not include any post-quantum algorithms.

There are a few example implementations in some cryptographic open-source libraries. For instance, the Open Quantum Safe project has developed *liboqs*, [3] a C library which specifically focuses on post-quantum cryptography. Another example is the PQClean project which collects

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

**Table 2** Post-quantum algorithms from NISTs standardization process in selected open-source libraries at the time of this work

| Library | Language | PQC Algorithms |
|---|---|---|
| OpenSSL | C | – |
| LibreSSL | C | – |
| Botan | C++ | McEliece<br>NewHope<br>XMSS |
| Crypto++ | C++ | – |
| Bouncy Castle | Java | SABER<br>McEliece<br>FrodoKEM |
| WolfCrypt | C | NTRU |
| Libgcrypt | C | – |

clean C implementations of NIST's post-quantum project's schemes. Their objective is to offer standalone C implementations that are highly tested and possibly valuable to other projects, such as the *liboqs* [26]. Also, SUPERCOP, [4] which is a benchmarking tool for PQC algorithms, contains many highly optimized implementations of the post-quantum schemes.

In Table 2, we have gathered post-quantum schemes that have been implemented in some of the commonly used open source cryptographic libraries. Most of the largest open-source cryptography libraries have not yet implemented any post-quantum algorithms. One of the most used open-source cryptographic libraries, OpenSSL, has stated that they will not be adding any post-quantum algorithms to the library before the algorithms have been standardized [27].

From Table 2, we can notice that McEliece [28] has been implemented in more than one library. McEliece is already a fairly old and trusted algortihm and thus probably chosen to be integrated into the libraries. It will most likely undergo little changes and thus needs less updating. NewHope [29] was eliminated from the third round of NIST's standardization process, since it was similar to KYBER which NIST preferred [30]. XMSS [31] is a post-quantum signature scheme that has not been part of the NIST standardization process.

Many of these algorithms have ungergone some small changes during the process which makes it difficult to keep the implementations up to date. Nevertheless, it is important to study the implementation of post-quantum algorithms in advance to see how they actually work as part of a library in practice and what are the challenges in the implementation process.

## Dangers and Challenges

If there are any mistakes left in the implementation of cryptographic algorithms, it can result in the weakening of the security and even make the whole system vulnerable. The implementation of cryptography is mostly based on translating very complex mathematical algorithms to platform-specific infrastructures which itself is already a common source of mistakes. One of the most famous cases of a mistake in a cryptographic implementation is Heartbleed [32], where a bug was found in the implementation of TLS in OpenSSL, a widely used cryptographic library. The vulnerability allowed the attacker to read data from protected memory locations on different popular HTTPS sites. A more recent example is the Java ECDSA vulnerability, where forgetting to check that values are not zero when verifying signatures resulted in false signatures being verified for any keys and messages. [5]

Even if the algorithms were implemented correctly, there are chances of misuse in case the design of the program is too complex. A survey about the causes of vulnerabilities in cryptographic softwares shows that only about 27% are actually cryptographic issues whereas 37% are memory or resource management issues [33]. This fact indicates that the greatest security concerns can be found in system-level bugs. They also found a correlation between the complexity of cryptography software and a higher density of vulnerabilities. Another study showed that only 17% of all the bugs were in the cryptographic libraries whereas the rest 83% were caused by misuses of the libraries in different applications and programs [34].

In addition, clear implementation and supporting documentations as well as testing environments have been discussed to have an impact on the correct usage of cryptographic algorithms. [35] For example, a research about NIST's post-quantum algorithm standardization process's submissions claims that there was a lack of guidelines and testing frameworks that could have helped the community to evaluate the schemes [26]. They also discussed the quality of the submitted codes. Many cryptographers are not specialized in software engineering which creates challenges in implementing the mathematical algorithms into proper code.

Another problem in implementing cryptographic algorithms is to make the execution constant time. Variations in functions' execution times can pose a serious threat to otherwise perfectly secure programs [36]. These timing attack vulnerabilities can be very hard to detect and prevent. The most powerful timing attack Spectre [37] in 2018 affected most CPUs revealing private data to the attacker.

From these examples, we can see how difficult it can be to implement cryptographic algorithms in practice. There are

[4] https://bench.cr.yp.to

[5] https://neilmadden.blog/2022/04/19/psychic-signatures-in-java.

many aspects to be taken into consideration. Post-quantum algorithms are even more complex to implement than the ones used in today's protocols. Even a small mistake can cause huge vulnerabilities if the developer is not aware of all the possible challenges. It will also take a considerable amount of time to finally make the transition to post-quantum cryptography and integrate post-quantum algorithms into protocols. For these reasons, it is highly important to do research on the implementation of post-quantum algorithms and recognize the possible dangers.

## Algorithm Integration

Many commonly used open-source cryptographic libraries have very few if any post-quantum public key algorithms implemented. Example implementations exist, but for the algorithms to spread into common use, it is necessary that they are implemented to libraries that the developers use. That way the threshold to try out and take post-quantum algorithms into use will be lowered.

Crypto++ [6] was chosen from the open-source libraries. According to Crypto++ wiki, the library has been used by a multitude of commercial and non-commercial projects [38]. Crypto++ is a C++ library but did not have any of the post-quantum algorithms present in the NIST standardization competition. In the yearly GitHub Octoverse report, [7] C++ is consistently among the ten most popular languages. We forked the library and started integrating the chosen algorithms into the fork. [8]

### Chosen Algorithms and the Mathematical Problems Behind Them

The three originally chosen algorithms were part of the third round of the NIST standardization process: KEM algorithms CRYSTALS-Kyber [23] and SABER [25] and digital signature algorithm CRYSTALS-Dilithium [39]. They were chosen as they were all seen as potential candidates for standardization. Afterwards, we also chose to integrate FrodoKEM [40], because while it was only part of the alternate candidates during the third round, it has been recommended to use by the German Federal Office for Information Security (BSI, *Bundesamt für Sicherheit in der Informationstechnik*) [41]. The integrated algorithms are also listed in Table 3. All of these algorithms are based on lattice problems. The hardness of lattice problems is based on Shortest Vector Problem (SVP), which is NP-hard under certain conditions [42].

[6] https://cryptopp.com

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

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

**Table 3** Integrated lattice-based algorithms in our work and their underlying security foundations

| Algorithm | Security problem |
|---|---|
| CRYSTALS-Kyber | Mod-LWE |
| CRYSTALS-Dilithium | |
| SABER | Mod-LWR |
| FrodoKEM | LWE |

In 2005, Regev published the Learning with Errors (LWE) problem [43], which is based on SVP. The LWE problem states that the distribution $(A, As + e)$ is difficult to distinguish from uniform. $A$ is a random matrix in $Z_q^{m \times n}$, $s$ a uniformly-random vector in $Z_q^n$ and $e$ is a random vector that has small coefficients.

Kyber and Dilithium are both based on the Module-LWE problem [44, 45]. Module-LWE is a special case of Ring-LWE [46] which extends the LWE problem to polynomial rings. The structure of the algorithms results in competitive performance, especially when utilizing NTT (number theoretic transform) to speed up polynomial multiplication. Kyber and Dilithium were in fact standardized after the third round [17].

SABER is based on the Module Learning with Rounding (Mod-LWR) problem which is the module version of LWR [47]. LWR is the LWE problem without the randomness and is thus deterministic while LWE is probabilistic. The design of SABER results in a much simpler structure: all integer moduli are powers of 2 and the algorithm requires less bandwith [25]. The downside is that NTT cannot be used for polynomial multiplication.

The security of FrodoKEM is based on the original LWE problem. While R-LWE and by extension, Mod-LWE, are much more efficient, FrodoKEM is a more concervative choice. The biggest benefit of the design in FrodoKEM is that in the future, some weaknesses could be found in the special cases of lattices that, e.g., Kyber and SABER are based on, and that these weaknesses would not be relevant in the case of generic lattices. [40]

### Goals in the Algorithm Integration Process

When integrating algorithms into a software library, there are certain things to be taken into account. Bernstein et al. [48] highlight simplicity and reliability when designing the API. They mention countermeasures taken to prevent vulnerabilities—among these reading randomness always from the OS instead of creating an own random number generator. They also comment that minimizing the code is an underappreciated goal in cryptographic software.

Green and Smith [35] display 10 principles for implementing usable cryptographic library APIs. As the designers

of the library will in most cases not be the developers mainly using it, usability is essential.

The subsequent list describes our approach to the integration process:

- Usability. According to Green and Smith [35], the API has to be easy to use and additionally easy to learn even for users without cryptographic expertise. In our work, these are the most important things to consider. The user has to be able to easily switch between different algorithms and parameter sets. It should be as self-evident as possible to the user how to use the algorithms and the code on the developer side should be as small as possible.
- Conformity of the code with the library. The code structure of the new algorithms and the way they are integrated to the fork resemble the other algorithms in the library. When possible, methods already present in the library are used instead of integrating new functions, e.g., when comparing bit arrays in constant time.
- Reliability. The integrated algorithms need to naturally work as intended and produce correct outputs.
- Prevent unnecessary code. We use library specific methods where applicable, and aim to produce little overhead in the code when integrating the algorithms to the fork.
- Performance. Good performance of the algorithm implementations is essential also for the usability of the library.

## Integrating the Algorithms

Algorithms that are based on lattice problems are much more mathematically complex than commonly used public key cryptographic algorithms, like RSA or elliptic curve cryptography. This results in the algorithms being more difficult to implement for average developers using only the specification [49]. There are also many pitfalls which could impact the security and performance of the implementations. As our goals involved the implementations being as reliable as possible and we also wanted to prevent unnecessary code, it would have made little sense to completely implement the algorithms from scratch.

For these reasons, we used implementations made by the developers of the algorithms, the most recent of which can be found on GitHub, as a basis. They were implemented in C, which made integrating them into a C++ library a very sensible choice. We added a class structure to each of the algorithms and changed the structure of the code wherever necessary. We utilized the API specified by the NIST standardization to guidelines as a baseline for the fork. One of our goals was that the code would be familiar with people that have used Crypto++ library before, so we structured the code similarly to other algorithms present in the library. We implemented an easy way for the users to switch between different security levels with the class structure.

The algorithm implementations by the algorithms' designers have a couple of different versions that are a bit different internally. We used in our work the most basic versions that utilize SHAKE and other Keccak functions internally, but there are implementations that use AES to enable better performance on systems that support AES hardware acceleration. This is especially relevant when choosing what implementation to use in the production environment.

In the first phase, we integrated CRYSTALS-Kyber, CRYSTALS-Dilithium and SABER into the fork with the necessary tests and performance evaluation. To extend our work, we have added an implementation of FrodoKEM into the fork, as well as performed KATs (Known Answer Tests) to evaluate the correctness of the four algorithms in the fork. We also changed Kyber, Dilithium and FrodoKEM to use a common FIPS 202 [13] class in the fork for internal Keccak operations. The class is also based on the reference implementations of the algorithms.

## Challenges in the Integration Process

The lattice-based post-quantum algorithms are reasonably large in size and in the NIST standardization process, they feature different sets of parameters to achieve different security levels. There are a lot of parameters for each algorithm and they vary a lot between the security levels. One of the biggest challenges when integrating the algorithms into the fork has been to solve how the parameters and changes between them are handled. The C reference implementations that were used as a basis in the integration use compile-time constants. As this would undermine the flexibility when the developer wants to change the parameters, it was not a viable solution for us.

Two approaches were used in the integration process. With Kyber, SABER and then later with FrodoKEM, class templates were used to deal with the different parameters. A template class of the algorithm is created, with the variables depending on the template variables specified by the subclasses of each security level. That way it is easy for the developer to change between the security levels by choosing the correct subclass. With Dilithium, we decided to use member variables in the base class that are then changed by the subclass to specify the parameters. Both solutions work but the class templates make the integration somewhat easier. With member variables, other changes will follow. Because many methods in the algorithms use arrays whose sizes are specified by the parameters of that security level, the sizes of the arrays will vary. For example, in the Dilithium reference implementation, polynomials and vectors were implemented by using *structs*. The size of arrays in structs has to be known before compiling and that was not possible. We switched to using *std:array* for polynomials and *std:vector* for vectors. This caused some additional changes in

the methods, but as their inner data is possible to work on, it was not a huge problem.

In the integration process, debugging proved challenging quite many times. As the algorithms are complex and rely on randomness, it can be quite difficult to try and determine what the state of the variables should be at each point in the execution of the algorithm. We used reference implementations with randomness removed to find out where the algorithm does something wrong in the integrated C++ version and that way were able to find the bugs in the implementation phase. Known Answer Tests were later used to verify the correctness of the implementations.

Because we wanted the implementations to be compilable on Linux as well as with Visual Studio Compiler as in the original library, some changes had to be made in the integration phase. GCC supports variable sized arrays but it is not part of the official C++ standard, so at many points we had to switch to using *std::vector* instead. Making sure FrodoKEM and the newest changes also work with Visual Studio Compiler remains future work.

When we were integrating, e.g., Dilithium into the fork, the third round of the NIST PQC standardization process was still ongoing and the algorithm was not anymore up-to-date. As the algorithms are not established yet, there is quite a big risk that they somewhat change upon findings in research even now.

When implementing complex cryptographic algorithms, it is very easy to make mistakes that can be difficult to find. As an example, when making updates to the Dilithium implementation in the fork after having succesfully integrated the algorithm to the fork, the size of a vector was not correctly updated. The algorithm still ran without crashing, but the vector of wrong size resulted in the tests not succeeding. It took a substantial amount of time carefully debugging and inspecting the code for mistakes until the spot where the wrong constant was used could be found. Luckily, as the tests failed, it was clear that something was not correct in the algorithm implementation and thus committing this mistake to production would have been avoided in real development, as long as the tests are sufficient enough.

## Results of Testing and Analysis

After the integration process was completed, we tested that the algorithms work as they are supposed to with Known Answer Tests (KATs). We also studied the performance of the algorithms and compared the reference implementations to the performance with different compiler options. Then, we considered some security aspects in the implementations.

### Known Answer Tests

During their standardization process, NIST required each of the submission packages to include a file with Known Answer Test (KAT) values that can be used to determine the correctness of the algorithm implementations [50]. KATs include different fixed input values which produce certain output values. Many of the submission algorithms use random number generators which need to use specified values in order to produce the fixed output value. Our goal was to implement these KATs provided by NIST's third round submission packages into the fork of Crypto++ and check the correctness of the implementations of the post-quantum algorithms added to the fork. The KATs were used to locally test the algorithms but we did not commit them to the GitHub repository of the fork.

Since the post-quantum algorithms implemented to Crypto++ library were implemented in a way that they utilize as much as possible the functions already in the library, we needed to add a new random number generator *rng_KAT* to the library in order to force the algorithms to output the fixed results. Otherwise, the functions implemented into the fork did not require any changes or modifications.

We implemented the algorithms *PQCgenKAT_sign()* for Dilithium and *PQCgenKAT_kem()* for Kyber, SABER and Frodo-KEM which were all modified from the reference implementations provided by the algorithm designers to the NIST standardization process. Some modifications were needed since the parameter handling has been changed from the reference implementation. Also, we implemented a function to compare the results we generated with the provided KAT values.

For the KEM algorithms, Kyber and SABER, the KAT values we generated were identical to the given reference KAT values. Thus, the implementation to the Crypto++ had been successful. With the digital signature scheme, Dilithium, there was a problem when we compared our values to the KAT values from NIST's third round submissions. They were not identical, but we found out that the KAT values included in the NIST submission package had not been updated for the most recent changes made to the algorithm [18]. We then compared our results to KAT values generated by the up-to-date implementation found on GitHub. [9] These were an identical match. Thus, the Dilithium implementation in the fork of Crypto++ is also assumed to be correct based on the KATs.

After integrating Frodo-KEM using SHAKE based on the implementation of the algorithm designers, we also tested the implementation using KAT values available on

[9] https://github.com/pq-crystals/dilithium.

their GitHub repository. [10] In addition, these KAT tests were successful, so we could be quite confident on all of the algorithms behaving correctly (at least most of the time).

## Performance Analysis

The performance of the integrated algorithms was measured on a laptop (i7–10,875 H CPU @ 2.30 *GHz* x16) running Ubuntu 18.04. The implementations were first compiled with G++ using the default Crypto++ compiler options, and afterwards with having added *-march=native* compiler option. The performance was compared against the most recent version of the reference implementations found on GitHub. The performance was measured in clock cycles using the *__rdtsc()* instruction.

Table 4 illustrates the performance of the algorithms in the C++ fork when comparing to the original C language implementations. The table presents separately the runtimes of the key generation, encapsulation (i.e., encryption) and decapsulation (i.e., decryption) for KEM algorithms, and key generation, signing and verification of a signature for Dilithium. The CPU cycles are not absolute, i.e., the runtimes will change between executions depending on a multitude of factors. The numbers in the table are the average of 10 000 executions, so they are pretty indicative of the general performance of the algorithms. It is relevant that all the performance runs were performed on the same laptop in as similar conditions as possible. When at first compiling the implementations with default Crypto++ library options on Linux, especially the performance of SABER was a lot worse, and that is also prevalent in the table, the runtime basically doubling at times. A general trend is, that the integrated versions ran somewhat worse but not usually that much worse. FrodoKEM algorithms also were significantly slower on the integrated fork compared to the original implementation, when compiling with the default options.

We investigated the cause of the performance loss especially in the case of SABER, and managed to find the cause. The design of SABER uses integer moduli that are powers of 2 [25]. This design choice simplifies the algorithm a lot, but there is one downside—SABER cannot use NTT to optimize polynomial multiplication and has to use some other algorithm internally to ensure the performance of the algorithm. The SABER team combines Toom-Cook and Karatsuba multiplication to speed up polynomial multiplication [25]. While this enables very good performance for SABER, the performance decreased by about 100% in the fork when essentially the same construction was used with minimal changes.

We found the default compiler options used in compiling Crypto++ were the cause. While it does use the *-O3* optimization flag, the SABER C implementation uses *-march=native* flag to tell the compiler to optimize the code for the specific hardware it is being compiled on. Adding this flag to the fork compiler options massively improved the performance of SABER, bringing it very close to the performance of the original implementation. Adding the flag also generally improved the performance of all algorithms, especially FrodoKEM, but also Kyber and Dilithium, as illustrated in Table 4.

As the *-march=native* option is not available on, e.g., Visual Studio Compiler (MSVC), it poses a challenge for extracting the optimal performance on Windows. The performance of the algorithms would likely also be better in general if we had integrated implementations that use AES instead of SHAKE operations. Also, the integrated implementations are the bare bones versions without any platform specific optimizations, so in real environments the performance would most likely be better.

It can be easily seen from Table 4 that the performance of FrodoKEM is consistently significantly worse when comparing to Kyber and SABER, the operations being tens of times slower than the respective operations in Kyber and SABER on the same security level.

## Security Analysis

The integrated versions of Kyber, SABER and Dilithium were analyzed with Valgrind [11] in order to find any possible memory leaks. No memory leaks were found in our C++ implementations. Analyzing FrodoKEM remains future work. Another important security aspect that must be taken into consideration when implementing cryptographic algorithms are side-channel attacks and their prevention. Even though the candidates for NIST's standardization process are widely studied on their efficiency and theoretical security, their side-channel security research has been relatively scarce. In this implementation, we are mostly interested in timing analysis.

Timing attacks [51] can be used to extract some information about the long-term secret key by analyzing the execution times of the cryptographic algorithms. The attack can easily be avoided by using constant-time executions. In many cryptographic algorithms, modular reductions are a usual source of timing leakage since they are often implemented using conditional statements. The designers of Kyber and Dilithium have taken this fact into consideration by using Montgomery [52] and Barrett reductions

[10] https://github.com/Microsoft/PQCrypto-LWEKE.

[11] https://valgrind.org

**Table 4** Performance of Kyber, SABER, FrodoKEM 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* enabled

| Algorithm | Function | CPU cycl (ref. impl.) | CPU cycl (C++ def.) | Run-time incr.(%) | CPU cycl (C++ opt.) | Run-time incr.(%) |
|---|---|---|---|---|---|---|
| Kyber-512 | Key gen | 62,344 | 76,790 | 23 | 72,786 | 17 |
| | Encaps | 82,072 | 93,951 | 14 | 89,443 | 9 |
| | Decaps | 96,599 | 106,133 | 10 | 103,530 | 7 |
| Kyber-768 | Key gen | 104,760 | 128,795 | 23 | 119,795 | 14 |
| | Encaps | 125,573 | 153,051 | 22 | 144,601 | 15 |
| | Decaps | 144,242 | 170,077 | 18 | 163,542 | 13 |
| Kyber-1024 | Key gen | 167,130 | 199,451 | 19 | 193,092 | 16 |
| | Encaps | 191,038 | 231,329 | 21 | 225,035 | 18 |
| | Decaps | 215,149 | 252,671 | 17 | 249,050 | 16 |
| LightSaber | Key gen | 38,488 | 76,673 | 99 | 49,296 | 28 |
| | Encaps | 50,390 | 96,471 | 91 | 55,150 | 9 |
| | Decaps | 56,267 | 110,647 | 97 | 58,798 | 4 |
| Saber | Key gen | 72,221 | 147,247 | 104 | 80,930 | 12 |
| | Encaps | 90,387 | 176,890 | 96 | 92,024 | 2 |
| | Decaps | 98,053 | 199,656 | 104 | 97,490 | -1 |
| FireSaber | Key gen | 133,318 | 236,223 | 77 | 143,944 | 8 |
| | Encaps | 141,194 | 279,257 | 98 | 144,228 | 2 |
| | Decaps | 153,039 | 310,587 | 103 | 153,255 | 0 |
| FrodoKEM-640 | Key gen | 4,400,569 | 5,874,930 | 34 | 4,557,691 | 4 |
| | Encaps | 8,883,762 | 11,603,364 | 31 | 8,720,604 | -2 |
| | Decaps | 8,991,188 | 11,554,022 | 29 | 8,706,781 | -3 |
| FrodoKEM-976 | Key gen | 9,882,989 | 12,684,395 | 28 | 9,683,134 | -2 |
| | Encaps | 17,252,083 | 26,120,073 | 51 | 16,892,008 | -2 |
| | Decaps | 17,173,616 | 25,987,239 | 51 | 16,795,400 | -2 |
| FrodoKEM-1344 | Key gen | 18,322,636 | 22,906,380 | 25 | 16,925,600 | -8 |
| | Encaps | 33,663,030 | 48,947,802 | 45 | 32,449,574 | -4 |
| | Decaps | 32,692,148 | 48,761,607 | 49 | 32,273,491 | -1 |
| Dilithium2 | Key gen | 173,738 | 191,309 | 10 | 172,128 | -1 |
| | Sign | 773,783 | 809,549 | 5 | 777,489 | 0 |
| | Verif | 192,578 | 213,248 | 11 | 195,329 | 1 |
| Dilithium3 | Key gen | 315,315 | 343,565 | 9 | 309,369 | -2 |
| | Sign | 1,223,055 | 1,332,790 | 9 | 1,250,971 | 2 |
| | Verif | 301,953 | 340,751 | 13 | 309,892 | 3 |
| Dilithium5 | Key gen | 475,905 | 525,782 | 10 | 451,494 | -5 |
| | Sign | 1,530,003 | 1,608,641 | 5 | 1,480,852 | -3 |
| | Verif | 506,734 | 557,814 | 10 | 489,525 | -3 |

[53]. SABER, however, uses power-of-two moduli in order to avoid variable-time operations. This makes the implementation of SABER constant time, too.

Even if the algorithms execute in constant time, they can be vulnerable to other side-channel attacks, such as power analysis and electromagnetic analysis. A recent study [54] has conducted a correlation power analysis targeting the Toom-Cook-based and NTT-based multiplication in lattice-based KEMs revealing the secret operands involved in these operations. The attacks can be mitigated with masking and hiding countermeasures, as has been done in the side-channel resistant implementation of SABER [55]. Another research targets the aforementioned Barrett reduction in CRYSTALS-Kyber to obtain the secret key [56]. The Barrett reduction was used to prevent timing attacks but it can expose the implementation to a chosen-ciphertext clustering attack.

## Discussion

A quantum computer with enough qubits will be able to break commonly used public key cryptography. While we cannot be sure that there will ever be a quantum computer powerful enough, the development of quantum computers has been fast during recent years. Because the threat from quantum computers is realistic some day in the future, it is imperative that data and communications start to be secured against them before it is too late. This is where post-quantum algorithms step in. They run on classical computers but are based on different mathematical problems that the quantum computers cannot easily solve.

Implementing cryptography by oneself is difficult and leads to potentially severe security problems. Open-source cryptographic libraries have a huge role both in commercial and non-commercial development. It is essential that post-quantum algorithms are implemented to open-source libraries, so that the threshold for developers and organizations to migrate to post-quantum or hybrid algorithms will lower.

### Biggest Challenges in the Implementation of Post-quantum Algorithms

Implementing post-quantum, in our case lattice-based algorithms, is not simple. The mathematical problems behind the algorithms are complex, especially when comparing to commonly used public key algorithms like RSA. This makes finding mistakes in the implementations more difficult, and implementing the algorithm based on just the specification requires special expertise. Without the reference implementations, integrating this many algorithms into our fork would not have been possible.

Debugging is also a challenge in the implementation phase. Because the algorithms specifications consist of multiple sub-algorithms and there is randomness involved, it can be difficult to determine what the state of the different variables should be at different points of execution. The compiled algorithms might seemingly execute correctly but return completely wrong results. In the integration phase, we made use of the reference implementations with randomness removed to debug our integration and find where the execution of the program goes wrong. KATs are then useful for validating the correctness of the ready implementation.

Integrating the post-quantum algorithms into existing systems will be a large-scale challenge in the near future. It is probably too optimistic to think that replacing public key cryptography with post-quantum algorithms one-to-one without any other changes is realistic. There are likely to be underlying dependencies in the systems, and the often increasing key and ciphertext sizes bring a challenge to, e.g., communications.

The algorithms are quite new and the implementations will become more efficient and secure. There has been work done in the recent years on, e.g., hardware optimized implementations of the different algorithms, e.g., [57, 58].

Kyber and Dilithium were already standardized by NIST [17]. Still, new findings like the attack on SPHINCS+ [21] and how SIKE was completely broken [16], even if the algorithms are not based on lattice problems, reasonably bring doubts about the security of the other post-quantum algorithms. Because the lattice-based algorithms are still new, it is entirely possible that there are weaknesses still to be found both in the structure of the algorithms and implementations. That might put off organizations from changing to post-quantum cryptography. In any case, it can be argued, that starting to use the post-quantum algorithms is important, so that they can be integrated into different systems and any overlooked mistakes are found as early as possible.

### Did we Succeed in Achieving our Goals?

After integrating the algorithms into the fork, a good question to ask would be if the goals we set beforehand were reached. While a good set of the goals were successfully met, there still remain some open questions.

The performance of the algorithms was in general satisfactory. An important thing to notice is that it depends a lot on the optimization of software and hardware. In general, the performance of the integrated algorithms in the fork was in the similar area when comparing to the original reference implementations—meaning that the integration into the library did not add much overhead into the implementation. Because the performance of the algorithms is in most cases very much dependent on the environment that the algorithms run in and what optimizations are possible there, using a library implementation might not always be the most optimal solution. In case the encryption and decryption do not happen that often, small differences in performance are not significant.

Usability was one of the main goals we set. This still remains kind of an open question, because answering this would need feedback and testing from developers themselves. A focus was set on having the algorithms as usable as possible and the API enabled by the class structure allows the developer to change between algorithms and parameter sets, i.e., algorithm versions in a decently smooth manner. Speaking of usability, adding exception handling, e.g., in case of wrong parameter sizes or other mistakes, would enhance the usability of the fork but might also have a negative effect on the readability of the code and in some cases add unnecessary overhead.

One of the reasons we decided to use the reference implementations instead of doing our own implementations from scratch was indeed preventing unnecessary

code. We could say that this has been achieved, as we also decided to use methods already present in the library wherever we could. This way we have been able to at least not worsen the readability of the code when comparing to the original implementations. We also created a separate FIPS 202 class based on the reference implementations. This class is used by all the integrated algorithms other than SABER for Keccak operations, e.g., absorb and squeeze operations, minimizing the code to avoid repetition.

We also aimed at misuse resistance. The structures of the reference implementations quite decently support this. The user is not able to change where the randomness comes from, so it is ensured that it is cryptographically secure against user mistakes. Still, as there is practically no checks if the variables supplied by the user are of the correct size, simple mishaps by a developer might lead to interesting results.

### Comparing the Implemented Algorithms

When we chose the algorithms to be implemented, the third round of the NIST post-quantum standardization process was still ongoing and none of the algorithms had been standardized. This summer, NIST standardized CRYSTALS-Dilithium among the digital signature algorithms and CRYSTALS-Kyber among the KEM algorithms [17]. SABER and FrodoKEM were dropped from the standardization process.

In our work, both SABER and Kyber seemed like suitable candidates for standardization. The performance of both of them is good, and while SABER is a bit more simple from the design, Kyber has the advantage of being from the same algorithm family as Dilithium. NIST cites the problem of Kyber being more convincing compared to SABER as the reason for preferring to standardize Kyber [17]. It is most likely, that after being standardized, the popularity and usage of Kyber will rise substantially compared to SABER. It is possible, though, that even though it is standardized, weaknesses might be found in Kyber or the Mod-LWE problem it is based on, that could cause people to prefer using other algorithms.

While SABER and Kyber are quite similar, FrodoKEM differs from them a lot. It bases its security on the plain LWE problem, which causes its performance to be significantly worse, as could be seen in Sect. 4.2. The benefit of the structure of FrodoKEM is following: if structured lattice algorithms like Kyber and SABER were broken, FrodoKEM could still be secure [17]. Ultimately, the bad performance was the main reason why FrodoKEM was dropped. FrodoKEM has been recommended by other organizations, like the German BSI [41], because of the security advantage. Although using Kyber (or SABER) is much more practical in most scenarios, using FrodoKEM remains a good option when the performance is not that important, i.e., when the operations are not performed that frequently or the confidence in security is of the highest importance.

### Limitations and Future Work

There are identifiable limitations in our study. We have only focused on the integration of four lattice-based post-quantum algorithms, leaving out other algorithm types. Also, as our study only focuses on integrating algorithms into an existing open-source library, this leaves the process of creating an original implementation of an algorithm directly into the open-source library out of the scope of the study. The study could also always be extended to more open-source libraries than just one.

The implementations in the fork can always be updated and improved and the fork also needs to be maintained. In the current version, only basic versions of the algorithms using SHAKE are included. AES functionality could be added, as well as AVX2 (*Advanced Vector Extensions 2*) optimized versions of the algorithms, which were also included in the reference implementations.

The fork can be used to research the usage of the algorithms in different environments by different developers. Also, the security of the implementations can be further examined, e.g., with side channel analysis and fuzzing. The algorithm implementations are not optimized for hardware, so side channel vulnerabilities in this early stage of development are possible. The usability of the implemented algorithms could still be further examined, since that was not possible in the scope of this paper even though it was one of our goals.

Other post-quantum algorithms could be added to the fork as well. An option would also be to create implementations of the algorithms in other languages, e.g., Java. Both directions are needed in order to have the PQC algorithms available in as many use cases and environments as possible.

## Conclusions

Currently common public key cryptography is mostly based on elliptic curves and integer factorization. These mathematical problems can be easily solved by a powerful enough quantum computer. Post-quantum cryptography is a solution to mitigate against this threat.

In our paper, we have demonstrated how to integrate some of the candidate algorithms from the NIST post-quantum cryptography standardization contest to a widely used

programming library. The integration has not been without its challenges but our work shows that this is possible and can be done in some cases without much added overhead in efficiency. However, the new proposed candidates for PQC are not easy to implement and thus it is very important to continuously validate the algorithms and possible changes in them. Vulnerabilities in cryptographic libraries are usually very critical and if the cryptography is impacted, then most of other security guarantees cannot be kept.

It is also important to notice that usability issues in cryptographic libraries have been a great contributor towards mistakes and bugs in code. In our work one of the main goals was to make the new algorithms as intuitive and easy to use as possible for the developers utilising the Crypto++ library. The complexity of the cryptographic algorithms (especially PQC algorithms) adds to this and it is still necessary to improve on the usability of the libraries as well as the understanding of developers on the possibilities and limitations of the implementations.

While NIST already has standardized some post-quantum algorithms, and the process to potentially standardize other algorithms is still ongoing, a lot of work remains to be done. Open-source cryptographic libraries will play an important role in enabling developers to use post-quantum cryptography. Further research on the security of the algorithms as well as new implementations in different environments are necessary so that the confidence in the algorithms and usage of post-quantum algorithms rises.

**Funding** Open Access funding provided by Technical Research Centre of Finland (VTT). This research was supported by the PQC Finland project funded by Business Finland's Digital Trust program (Diary number 7188/31/2019).

## Declarations

**Conflict of interest** On behalf of all authors, the corresponding author states that there is no conflict of interest.

**Ethical Approval** This article does not contain any studies with human participants performed by any of the authors.



## References

1. Rescorla E. The transport layer security (TLS) Protocol Version 1.3. RFC Editor (2018). https://doi.org/10.17487/RFC8446. https://rfc-editor.org/rfc/rfc8446.txt
2. Rivest RL, Shamir A, Adleman L. A method for obtaining digital signatures and public-key cryptosystems. Commun ACM. 1978;21(2):120–6.
3. Hankerson D, Menezes AJ, Vanstone S. Guide to elliptic curve cryptography. Springer, New York (2004). https://doi.org/10.1007/b97644
4. Shor PW. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM J Comput. 1997;26(5):1484–509.
5. Proos J, Zalka C. Shor's discrete logarithm quantum algorithm for elliptic curves. Quant Inform Comput. 2003;3(4):317–44.
6. Hekkala J, Halunen K, Vallivaara V. Implementing post-quantum cryptography for developers. In: Proceedings of the 8th International Conference on Information Systems Security and Privacy - ICISSP, pp. 73–83 (2022). https://doi.org/10.5220/0010786200003120
7. https://icissp.scitevents.org/?y=2022. Accessed on 19.12.2022
8. Grover L.K. A fast quantum mechanical algorithm for database search. In: Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing, pp. 212–219 (1996)
9. Gidney C, Ekerå M. How to factor 2048 bit RSA integers in 8 hours using 20 million noisy qubits. Quantum 5, 433 (2021). https://doi.org/10.22331/q-2021-04-15-433
10. Gambetta J. Quantum-centric supercomputing: The next wave of computing. https://research.ibm.com/blog/next-wave-quantum-centric-supercomputing. Accessed on 19.12.2022 (2022)
11. Joseph D, Manzano M, Tricot J, Pinuaga FD, Leichenauer S, Hidary J. Transitioning organizations to post-quantum cryptography. Nature. 2022;605:237–43. https://doi.org/10.1038/s41586-022-04623-2.
12. Dworkin MJ, Barker EB, Nechvatal JR, Foti J, Bassham Lawrence E, Roback E. J.F.D. Announcing the advanced encryption standard (AES). 2001. https://doi.org/10.6028/NIST.FIPS.197.
13. Dworkin M.J. SHA-3 Standard: permutation-based hash and extendable-output functions (2015). https://doi.org/10.6028/NIST.FIPS.202
14. Shoup V. A proposal for an ISO standard for public key encryption. Cryptology ePrint Archive, Report 2001/112. https://eprint.iacr.org/2001/112.pdf (2001)
15. Beullens W. Breaking rainbow takes a weekend on a laptop. Cryptology ePrint Archive, Paper 2022/214 (2022). https://eprint.iacr.org/2022/214
16. Castryck W, Decru T. An efficient key recovery attack on SIDH (preliminary version). Cryptology ePrint Archive, Paper 2022/975 (2022). https://eprint.iacr.org/2022/975
17. Alagic G, Apon D, Cooper D, Dang Q, Dang T, Kelse J, Lichtinger J, Miller C, Moody D, Peralta R, Perlner R, Robinson A, Smith-Tone D, Liu Y.-K (2022) Status report on the third round of the NIST post-quantum cryptography standardization process. https://csrc.nist.gov/publications/detail/nistir/8413/final
18. Ducas L, Kiltz E, Lepoint T, Lyubashevsky V, Schwabe P, Seiler G, Stehlé D (2021) CRYSTALS-Dilithium - algorithm specifications and supporting documentation (Version 3.1)
19. Fouque P.-A, Hoffstein J, Kirchner P, Lyubashevsky V, Pornin T, Prest T, Ricosset T, Seiler G, Whyte W, Zhang Z (2020) Falcon: Fast-Fourier Lattice-based Compact Signatures over NTRU
20. Aumasson J.-P, Bernstein D.J, Beullens W, Dobraunig C, Eichlseder M, Fluhrer S, Gazdag S.-L, Hülsing A, Kampanakis P, Kölbl

S, Lange T, Lauridsen M.M, Mendel F, Niederhagen R, Rechberger C, Rijneveld J, Schwabe P, Westerbaan B. SPHINCS+ - Submission to the 3rd round of the NIST post-quantum project. v3.1 (2022)
21. Perlner R, Kelsey J, Cooper D. Breaking category five SPHINCS+ with SHA-256. Cryptology ePrint Archive, Paper 2022/1061 (2022). https://eprint.iacr.org/2022/1061
22. https://csrc.nist.gov/projects/pqc-dig-sig/standardization/call-for-proposals. Accessed on 14.10.2022
23. Bos J, Ducas L, Kiltz E, Lepoint T, Lyubashevsky V, Schanck J.M, Schwabe P, Seiler G, Stehlé D. CRYSTALS-Kyber: a CCA-secure module-lattice-based KEM. In: 2018 IEEE European Symposium on Security and Privacy (EuroS &P), pp. 353–367 (2018). IEEE
24. Chen C, Danba O, Stein J, Hülsing A, Rijneveld J, Schanck J.M, Schwabe P, Whyte W, Zhang Z. Ntru algorithm specifications and supporting documentation. (2019)
25. D'Anvers J.-P, Karmakar A, Roy S.S, Vercauteren F. Saber: Module-LWR based key exchange, CPA-secure encryption and CCA-secure KEM. In: International Conference on Cryptology in Africa, pp. 282–305 (2018). Springer
26. Kannwischer M, Schwabe P, Stebila D, Wiggers T. Improving software quality in cryptography standardization projects, pp. 19–30 (2022). https://doi.org/10.1109/EuroSPW55150.2022.00010
27. https://www.openssl.org/roadmap.html. Accessed on 5.8.2022
28. McEliece RJ. A public key cryptosystem based on algebraic coding theory. (1978)
29. Alkim E, Schwabe P. Newhope algorithm specifications and supporting documentation. (2019)
30. Alagic G, Alperin-Sheriff J, Apon D, Cooper D, Dang Q, Kelsey J, Liu Y.-K, Miller C, Moody D, Peralta R. et al.: Status report on the second round of the NIST post-quantum cryptography standardization process. https://csrc.nist.gov/publications/detail/nistir/8309/final (2020)
31. Buchmann J, Dahmen E, Hülsing A. Xmss - a practical forward secure signature scheme based on minimal security assumptions. In: Yang, B-Y. (Ed.) Post-Quantum Cryptography, pp. 117–129. Springer, Berlin, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25405-5_8
32. Durumeric Z, Li F, Kasten J, Amann J, Beekman J, Payer M, Weaver N, Adrian D, Paxson V, Bailey M, Halderman JA. The matter of heartbleed. In: Proceedings of the 2014 Conference on Internet Measurement Conference. IMC '14, pp. 475–488. Association for Computing Machinery, New York, NY, USA (2014)
33. Blessing J, Specter MA, Weitzner DJ. You really shouldn't roll your own crypto: an empirical study of vulnerabilities in cryptographic libraries (2021). https://doi.org/10.48550/arXiv.2107.04940
34. Lazar D, Chen H, Wang X, Zeldovich N. Why does cryptographic software fail? A case study and open problems. In: Proceedings of 5th Asia-Pacific Workshop on Systems, pp. 1–7 (2014)
35. Green M, Smith M. Developers are not the enemy!: The need for usable security APIs. IEEE Secur Privacy. 2016;14(5):40–6. https://doi.org/10.1109/MSP.2016.111.
36. Almeida J.B, Barbosa M, Barthe G, Dupressoir F, Emmi M. Verifying constant-time implementations. In: 25th USENIX Security Symposium (USENIX Security 16), pp. 53–70. USENIX Association, Austin, TX (2016). https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/almeida
37. Kocher P, Genkin D, Gruss D, Haas W, Hamburg M, Lipp M, Mangard S, Prescher T, Schwarz M, Yarom Y. Spectre attacks: exploiting speculative execution. arXiv (2018). https://doi.org/10.48550/ARXIV.1801.01203.
38. https://www.cryptopp.com/wiki/Related_Links. Accessed on 14.10.2022
39. Ducas L, Kiltz E, Lepoint T, Lyubashevsky V, Schwabe P, Seiler G, Stehlé D. CRYSTALS-Dilithium: a lattice-based digital signature scheme. IACR Trans Cryptograph Hardware Embedded Syst. 2018;2018(1):238–68.
40. Bos J, Costello C, Ducas L, Mironov I, Naehrig M, Nikolaenko V, Raghunathan A, Stebila D. Frodo: Take off the ring! practical, quantum-secure key exchange from lwe. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 1006–1018 (2016)
41. BSI TR-02102-1: Kryptographische Verfahren: Empfehlungen und Schlüssellängen. Version 2022-01. Technical report, Bundesamt für Sicherheit in der Informationstechnik (2022)
42. Regev O, Rosen R. Lattice problems and norm embeddings. In: Proceedings of the Thirty-Eighth Annual ACM Symposium on Theory of Computing. STOC '06, pp. 447–456. Association for Computing Machinery, New York, NY, USA (2006)
43. Regev O. 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, pp. 84–93. Association for Computing Machinery, New York, NY, USA (2005)
44. Brakerski Z, Gentry C, Vaikuntanathan V. (Leveled) Fully homomorphic encryption without bootstrapping. In: Proceedings of the 3rd Innovations in Theoretical Computer Science Conference, pp. 309–325 (2012)
45. Langlois A, Stehlé D. Worst-case to average-case reductions for module lattices. Designs Codes Cryptography. 2015;75(3):565–99.
46. Lyubashevsky V, Peikert C, Regev O. On ideal lattices and learning with errors over rings. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques, pp. 1–23 (2010). Springer
47. Banerjee A, Peikert C, Rosen A. Pseudorandom functions and lattices. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques, pp. 719–737. Springer (2012)
48. Bernstein D.J, Lange T, Schwabe P. The Security Impact of a New Cryptographic Library. In: Progress in Cryptology – LATINCRYPT 2012, pp. 159–176. Springer, Berlin, Heidelberg (2012)
49. Gaj K. 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, pp. 359–364. Association for Computing Machinery, New York, NY, USA (2018)
50. https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization/Call-for-Proposals. Accessed on 8.8.2022
51. Kocher PC. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In: Koblitz, N. (ed.) Advances in Cryptology — CRYPTO '96, pp. 104–113. Springer, Berlin, Heidelberg (1996). https://doi.org/10.1007/3-540-68697-5_9
52. Montgomery PL. Modular multiplication without trial division. Math Comput. 1985;44:519–21.
53. Barrett P. Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor. In: Advances in Cryptology — CRYPTO' 86, pp. 311–323. Springer, Berlin, Heidelberg (1987). https://doi.org/10.1007/3-540-47721-7_24
54. Mujdei C, Beckers A, Mera J.M.B, Karmakar A, Wouters L, Verbauwhede I. Side-Channel Analysis of Lattice-Based Post-Quantum Cryptography: Exploiting Polynomial Multiplication. Cryptol. ePrint Arch. Paper 2022/474 (2022). https://eprint.iacr.org/2022/474
55. Beirendonck M.V, D'Anvers J.-P, Karmakar A, Balasch J, Verbauwhede I. A Side-Channel Resistant Implementation of SABER.

Cryptology ePrint Archive, Paper 2020/733 (2020). https://doi.org/10.1145/3429983. https://eprint.iacr.org/2020/733
56. Sim B-Y, Park A, Han D-G. Chosen-Ciphertext clustering attack on CRYSTALS-KYBER using the side-channel leakage of barrett reduction. IEEE Internet Things J. 2022. https://doi.org/10.1109/JIOT.2022.3179683.
57. Zhao Y, Chao Z, Ye J, Wang W, Cao Y, Chen S, Li X, Li H. Optimization space exploration of hardware design for crystals-kyber. In: 2020 IEEE 29th Asian test symposium (ATS), pp. 1–6 (2020). https://doi.org/10.1109/ATS49688.2020.9301498
58. Zhu Y, Zhu M, Yang B, Zhu W, Deng C, Chen C, Wei S, Liu L. Lwrpro: An energy-efficient configurable crypto-processor for module-lwr. IEEE Trans Circuits Syst I Regular Papers. 2021;68(3):1146–59. https://doi.org/10.1109/TCSI.2020.3048395.