POLITECNICO DI TORINO
Repository ISTITUZIONALE

Analysis and contributions to an open source Kyber library in Rust

Original
Analysis and contributions to an open source Kyber library in Rust / Medina, F., Molteni, M.C., Di Scala, A.J., Nava, L.. - (2024), pp. 1-6. (29th IEEE Symposium on Computers and Communications, ISCC 2024 Paris (France) 26 June 2024 through 29 June 2024) [10.1109/iscc61673.2024.10733712].

Availability:
This version is available at: 11583/2997642 since: 2025-03-20T12:01:24Z

Publisher:
Institute of Electrical and Electronics Engineers - IEEE

Published
DOI:10.1109/iscc61673.2024.10733712

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright
IEEE postprint/Author's Accepted Manuscript

©2024 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collecting works, for resale or lists, or reuse of any copyrighted component of this work in other works.

(Article begins on next page)

11 September 2026

Analysis and contributions to an open source Kyber library in Rust

1th Francesco Medina
Dipartimento di Automatica e Informatica
Politecnico di Torino
Torino, Italy
s280620@studenti.polito.it

2th Maria Chiara Molteni
Security Pattern
Vimercate, Italy
m.molteni@securitypattern.com

3th Antonio Josè Di Scala
Dipartimento di Scienze Matematiche
Politecnico di Torino
Torino, Italy
antonio.discala@polito.it

4th Lorenzo Nava
Security Pattern
Vimercate, Italy
l.nava@securitypattern.com

***Abstract*—This work focuses on analyzing the efficiency of a purely Rust-written Kyber library for ARM Cortex M4 microcontrollers. The analysis includes performance comparisons with a C-written Kyber library, as well as the identification and resolution of minor issues within the Rust-written Kyber library. Contributions to the library include proposed solutions for unhandled TRNG exceptions, enhancements in code quality, and improvements in code coverage, all of which have been approved by the library maintainers and released.**

***Index Terms*—ML-KEM, Rust, code analysis, ARM Cortex-M4, Crystals-Kyber, post-quantum cryptography, random number generator, lattice-based cryptography, code coverage.**

I. Introduction

In recent decades, we have witnessed a growing interest in quantum computing, which has led to an acceleration of research and development of quantum computers. However, in addition to its many positive consequences, quantum computing is becoming a potential risk for our cryptographic systems [17]. For this reason, a consistent line of research on post-quantum cryptography is being developed to find solutions to deal with this future threat.

Embedded systems are a category of devices that require great attention to error prevention aspects due to their hardware limitations, especially in a quantum attack context. One of the key principles of *Security by Design* is to incorporate specific security solutions early in the software life cycle, rather than postpone them. Toward this goal, in this work we focus our attention on Rust Embedded, which is a system programming language that introduces its own ownership model in order to mitigate common weaknesses such as memory leaks, null pointer dereferences, and data races.

In this article, we analyze and determine efficiency of a Kyber library written in Rust [3] when integrated with ARM Cortex M4 microcontrollers; we then resolve some minor issues of the library (accepted and released by the maintainers).

- Section II: we introduce in part II-A the state-of-the-art and some mathematical aspects of post-quantum cryptography. In part II-B we provide some background knowledge on the lattice KEM called CRYSTALS-Kyber, which is the focus of our analysis.
- Section III: part III-A provides an explanation of measurement techniques used in the analysis, and a comparison of computation time between the Rust-written Kyber library [3] and the state-of-the-art C-written library [8]. In part III-B, we presents some observations to the library regarding boundary value coverage.
- Section IV: this Section is all about our contributions to the library. In part IV-A, we present a solution that we proposed for the True Random Number Generator (TRNG) exceptions unhandled by the library until now, and its advantages. Part IV-B describes a contribution of the code quality and part IV-C presents an enhancement of the code coverage.
- Section V: final considerations and recommendations for future research.

II. Preliminaries

A. Post-Quantum Cryptography

In recent years, we have been spectators of the advent of quantum computers [12]. Due to the power of quantum computing, some of the difficult or unmanageable mathematical problems upon which current cryptography is based can be solved with significantly less time and complexity compared to the capabilities of classical computers (and supercomputers). This implies that from the theoretical point of view, all cryptographic primitives, algorithms, and protocols currently considered secure may potentially become insecure in the future.

Two are the principal quantum algorithms that menace the classical cryptographic functions: the Grover and Shor algorithms. Lov Kumar **Grover** demonstrated in 1996 that, by exploiting the power of quantum computation, it is possible to weaken symmetric key cryptography [7]. Peter Williston **Shor** in 1994 demonstrated that public-key cryptography can be efficiently attacked with a quantum computer [17].

The main domain affected by the quantum threat is asymmetric cryptography, and then a lot of effort is put into the definition of new public-key algorithms. **NIST** initiated a process to solicit, evaluate, and standardize one or more quantum-resistant public-key algorithms [13]. Among all the proposed algorithms, at the end of the third round, four of them have been selected to be standardized: one for the key agreement and based on lattices (CRYSTALS-KYBER) and three for the digital signatures, two based on lattices (CRYSTALS-DILITHIUM and Falcon) and one based on hash functions (SPHINCS+). At the time of writing, the standardization process for these algorithms is in progress, and the first three standardized algorithms will be: **ML-KEM** (CRYSTALS-Kyber) [15], **ML-DSA** (CRYSTALS-DILITHIUM) [14], **SLH-DSA** (SPHINCS+) [16].

B. Lattice-based key agreement: ML-KEM

One of the finalist of the NIST competition is **CRYSTALS-KYBER**. This algorithm is designed in a **Key Encapsulation Mechanism** (KEM) flavour. The key encapsulation mechanism is a cryptographic technique that is generally used to establish a secure communication [5].

Kyber is an algorithm from the CRYSTALS suite [18]; same origin has Dilithium, the lattice-based algorithm for digital signatures. This post-quantum algorithm is IND-CCA2: this means that ciphertexts generated by this algorithm are also indistinguishable under an adaptive chosen ciphertext attack [4], [19].

Kyber relies on two main parameters: n, which is the length of vectors (equals to 256), and q, which is the modulus (equals to 3329). Operations in Kyber mainly involve vectors of length n with elements in $\mathbf{Z}_q$. There are three versions of Kyber: Kyber-512, Kyber-768 and Kyber-1024. The primary difference among them is the dimension of the lattice upon which each one relies.

Kyber and Dilithium undergo similar properties, listed below.

1) Lattices: **Lattices** are mathematical structures that have been studied since the 18th century and find broad applications in various mathematical fields. They are used in the design of cryptographic schemes starting from the late '90s [1], [6]. A lattice is a discrete additive subgroup of $\mathbf{R}^n$, i.e. a set of points in $\mathbf{R}^n$ with a dictated pattern, generated by a basis. The lattice's basis is a set of linearly independent vectors.

2) Closest vector problem: Many different properties and features of the lattices has been studied in the past years. In particular, one of them is the **closest vector problem** (CVP), on which also Kyber relies. Given a vector $v \in \mathbf{Q}^n$, the problem consists in finding a vector in the lattice that minimizes the distance from it [11].

3) Learning With Errors: The **learning with errors** is a mathematical problem that allows to conceal the value of a secret by introducing a noise to it. More specifically, let $\mathbf{Z}_q$ be the ring of integers modulo q, and $\mathbf{Z}_q^n$ be the set of vectors of n elements over $\mathbf{Z}_q$. Let $\mathbf{s} \in \mathbf{Z}_q^n$ be the secret vector, $\mathbf{v} \in \mathbf{Z}_q^n$ a vector, $e \in \mathbf{Z}_q$ the error, and $t = \mathbf{v} \cdot \mathbf{s} + e \in \mathbf{Z}_q$. Knowing t and $\mathbf{v}$ without knowing the applied error, it becomes hard to determine $\mathbf{s}$ [2].

4) Addition and multiplications between polynomials: Vectors in $\mathbf{Z}_q^n$ can be considered as polynomials with coefficients in $\mathbf{Z}_q$. The majority of the operations on which the Kyber algorithm is based are additions and multiplications between polynomials in the ring $\frac{\mathbf{Z}_q[x]}{x^n+1}$. From the computational point of view, addition between polynomials is a fast operation; instead, the multiplication is much slower and requires a great amount of memory. To speed it up, for the first time in a cryptographic algorithm has been introduced the **Number Theoretic Transform** (**NTT**) [9]. Roughly speaking, it is a generalization of the Discrete Fourier Transform over a finite field, which allows performing polynomial multiplication of high degree on integers sequences over a ring more efficiently.

III. Analysis on Kyber library written in Rust on a ARM Cortex-M4

All our the experiments [10] were conducted using an open-source pure Rust implementation of Kyber called **pqc_kyber** [3], installed on a STM32F407VGT6 board, equipped with an ARM Cortex-M4 core.

A. Performances evaluation

Speed tests conducted on the Kyber library written in C have already have been conducted and officially documented by pqm4 [8], which is a popular repository of post-quantum crypto C-written algorithms for the ARM Cortex-M4. Nowadays, there are no speed tests related to the Rust implementation of the Kyber library.

To measure speed of keygen, encaps and decaps algorithms, in terms of clock cycles, pqm4 used a popular C library called LibOpenCM3. Such a library provides an Hardware Abstraction Layer (HAL) in C for ARM Cortex-M microcontrollers.

Currently, Rust community is trying to improve the language's integration with embedded systems in order to offer compatibility and capabilities as those offered by the LibOpenCM3 library.

In order to assess the consistency and accuracy of the results obtained, we employ two distinct methods to measure clock cycles, comparing them to the results previously presented by pqm4 for the C-written Kyber library.

We expect to obtain very similar outcomes from both approaches, as the clock cycles required to execute an algorithm are expected to remain consistent regardless of the measurement method used. From the results obtained and listed in Table I and from the graphs in Figures 1, 2 and 3, in correspondence with the green lines, we observe that the SysTick and DWT methods produce very similar values, which are aligned with the expectations.

At this stage, we compare our tests with those provided by pqm4. The two mechanisms we employed to measure clock cycles are the following:

- SysTick: which is a CPU's peripheral system timer that is used to measure elapsed cycle counts. This count is accurate for clock cycles up to 24-bits.

- Data Watchpoint and Trace (DWT): this unit contains 4 counters. For our analysis, the read-only clock cycle counter register (CYCCNT) is relevant. This register increments on each clock cycle when the processor is not halted in debug state.

In particular, we use the same clock configurations adopted by pqm4:

1) HSE (High-Speed External) clock = 8 MHz
2) PLL (Phase-Locked Loop) clock = 24 MHz
3) PLL48CLK = 48 MHz (required for the TRNG module)

In the graphs shown in Figures 1, 2 and 3, lower values indicate better performance.

The results show that for the Decaps (Kyber-512), KeyGen and Encaps (all Kyber versions) operations, the C-written Kyber is faster than the Rust-written one. However, for the Decaps function in Kyber-768 and Kyber-1024, the Rust version is **as fast as** the C one, because for some functions of Decaps, the C compiler must remain conservative in its optimizations. We observed this behavior, by comparing the assembly files of C and Rust, both obtained by disassembling the binaries generated by their respective compilers. Among all the functions that are called internally by the Decaps' IND-CPA Decryption function and whose optimization is not as efficient as the Rust compiler's, the one that has the most impact on the level of optimization is the $ntt()$. In particular, the $ntt()$ function requires a pointer to the vector of elements of Z_q as a parameter and gradually modifies them internally. The C compiler identifies this particular method of modifying the input parameter as a potential case of aliasing, namely the scenario where two or more pointers point to the same memory location. This can lead to unexpected behavior, especially when the memory location is modified through one pointer while being accessed through another. Therefore, this circumstance force the compiler to refrain from applying optimizations that could result in errors or undesired behaviors. In the Rust code version, the implementation of the $ntt()$ function involves a mutable ($\&mut$) reference to the vector of elements of Z_q as an input parameter, ensuring that no one else can modify it besides the owner, thereby avoiding aliasing. This language property favors optimization and allows Rust compiler to make extensive use of multiple optimization mechanisms (without any constraints), such as reordering, scheduling, loop unrolling and constant propagation (pre-computing already known variable operations). We noticed that the compiler unrolls the $ntt()$'s loops and pre-calculates the address offsets at the beginning of the assembly code. This advantage is more significant for Decaps with k=3 and k=4, where the $ntt()$ function takes as input a vector with many more elements to process.

The most important consideration is that Rust language natively gives more **security** advantages with **zero cost abstraction** with respect to C language. Generally, for most programming languages, enhancing security implies a reduction in performances. In this context, it is also important to emphasize that these two libraries have different maturity levels. Indeed, C library is maintained by a big community and many developers work constantly on it. On the other hand, the Rust-written Kyber library is relatively young (currently at version 0.7.1), requires further improvements and efforts to be correctly comparable to the C library. In this perspective, the **results** that we achieved can be read as very **promising**.

Table I: Speed comparison at 24MHz. First rows refers to pqm4 benchmark results. The remaining rows refer to the two measurements techniques we applied to the Rust library. The average, minimum and maximum value of each algorithms are reported for each KEM method.

Kyber Ver.	Measure Method	Lang	Key Gen [cycles] (mean)	Key Gen [cycles] (min)	Key Gen [cycles] (max)	Encaps [cycles] (mean)	Encaps [cycles] (min)	Encaps [cycles] (max)	Decaps [cycles] (mean)	Decaps [cycles] (min)	Decaps [cycles] (max)
512	pqm4	C clean	636181	635670	648917	843945	843433	856680	940320	939808	953055
768	pqm4	C clean	1059876	1057827	1071809	1352934	1350884	1364866	1471055	1469005	1482987
1024	pqm4	C clean	1649604	1646417	1686328	2016366	2013177	2053070	2159906	2156716	2196609
512	SysTick timer	Rust	782915	782912	782968	1075512	1075505	1075566	1037024	1037020	1037077
768	SysTick timer	Rust	1271579	1271574	1271630	1572956	1572949	1573007	1466918	1466912	1466965
1024	SysTick timer	Rust	1782451	1782444	1782497	2327229	2327220	2327277	2193613	2193604	2193659
512	DWT unit	Rust	784455	784455	784456	1077356	1077356	1077356	1042176	1042176	1042177
768	DWT unit	Rust	1278644	1278644	1278644	1571206	1571206	1571208	1468604	1468604	1468604
1024	DWT unit	Rust	1772551	1772551	1772551	2330897	2330896	2330898	2187373	2187373	2187374

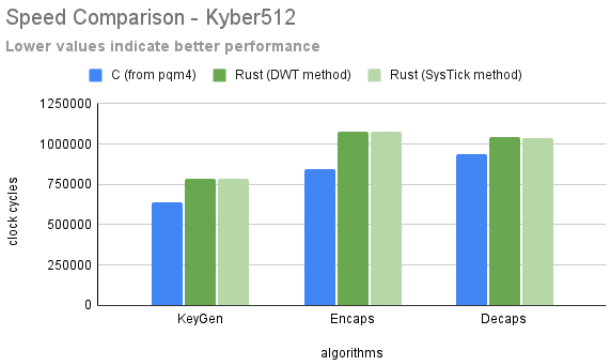

Figure 1: Clock cycles for Kyber512 measured on STM32F4.

B. Boundary value coverage observations

We performed a manual review of the inputs and outputs of the functions in the library. In particular, for both the Rust and C implementations of the Kyber library, we noted that the $ntt()$ function accepts as input a mutable vector of *i16*, i.e. integer values in the interval $[-32768, 32767]$. However, from Kyber's specification, the ntt function accepts as input a vector of elements in $\mathbf{Z}_q$, i.e. integers in the interval $[0, q-1]$, or equivalently in $[-\frac{q}{2}, \frac{q}{2}]$, with q equals to 3329 [18]. Note that not all the elements of *i16* type are **acceptable inputs** for the *ntt* function.

Testing the ntt function as a black box, we tried to pass as input almost all combinations of test vectors containing *i16* values. We observed that, for some test vectors, an **integer overflow** occurs. This means that the result of the sum operation exceeds the maximum representable value for the used data type.

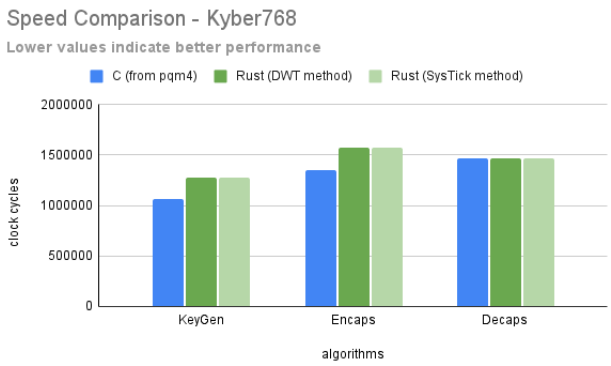

Figure 2: Clock cycles for Kyber768 measured on STM32F4.

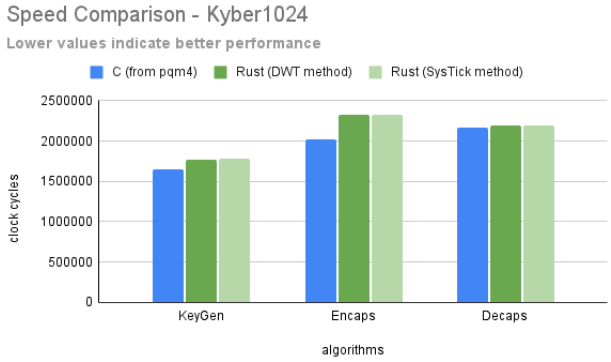

Figure 3: Clock cycles for Kyber1024 measured on STM32F4.

There is no range validity check inside the ntt function, because it is assumed that the input values respects the theoretical range $[-\frac{q}{2}, \frac{q}{2}]$.

The same behavior occurs in the *montgomery_reduce*, *poly_compress* and *barrett_reduce* library's functions. These functions are declared to accept integers with a certain range (i.e *i32* for the first function and *i16* for the other two), but not all values are supposed to be valid inputs. Consequently, in the Kyber context they do not produce any kind of errors, but when run as isolated functions they trigger overflow errors.

Since it is good practice to make functions testable in an isolate mode, it is preferable to define an input range validity control in the library, in order to prevent the overflow cases described before and introduce vulnerable code.

IV. Contribution to the Library

A. Solution for the Unhandled TRNG exceptions

A Random Number Generator (RNG) plays a critical role in cryptography. Random numbers are used for generating session keys, initialization vectors (IVs), or cryptographic nonces. Nondeterministic RNG or True RNG (TRNG) produces randomness based on an unpredictable physical source, known as the entropy source, which is beyond human control. The Kyber library function *randombytes* calls the *fill_bytes* method (from the *RngCore* trait), which uses a TRNG to fill an input vector with bytes of random numbers.

Since TRNGs are hardware modules, they can **fail** or produce **undefined behaviors** that can compromise the security of the library. To avoid this kind of problems, instead of *fill_bytes*, we suggested using method *try_fill_bytes* (also documented and coming from *RngCore* trait): this function is declared to be safer than the previous one, because it is able to intercept an exception and propagate control back to the caller, so that it can handle the exception correctly without terminating the process.

Furthermore, to be consistent and follow the error pattern adopted by the library, we added to the set of *KyberErrors* a new type: *RandomBytesGenerator*. We also made sure that in case this exception occurs, the error is propagated up to the caller function without internally using the Rust's $unwrap$ method, since it would immediately interrupt the process freeing the stack. This proposal was accepted and released by the library community.

This kind of problem has a worst-case scenario, depicted in Figure 4, and it involves the **IND-CCA Key Generation** algorithm. It is considered a worst-case scenario because the TRNG module failure occurs after the execution of the most computationally intensive functions (*IND_CPA_KeyPair* and *SHA3-256*).

More in detail, the processed data are progressively stored in the stack as it is generated and processed. At this point, when the second call for getting random numbers fails, it throws a panic event, which is responsible for freeing the stack and abort the process. This scenario represents a non-negligible issue, because IoT devices are specifically designed to be used intensively, leading to a higher probability of failure with service disruption.

Our proposed solution shown in Figure 5, brings several advantages to the library:

1) Security: TRNG peripherals can generate unstable outputs that can be exploited for predictive attacks. This solution avoids having these kinds of vulnerabilities.
2) Failure Recovery: it allows the library to recover from failures without losing the already processed and stored data, by giving a second chance to the TRNG or trying an alternative software path or fallback mechanisms.
3) Robustness: it guarantees that unexpected errors do not abort the whole process.
4) Error Reporting: meaningful and readable error messages for this kind of error are useful for the debugging process of the library.

The same type of solution can be applied throughout the library where there is need for random number generation, such as **IND-CCA Encapsulate** algorithm: issue as in Figure 6 and solution shown in Figure 7.

B. Contributions to enhance Code Quality

We notice that the *store64* function was created to store a 64-bit integer as a byte array in little-endian order. To reach a *single point of responsibility and failure* for a certain

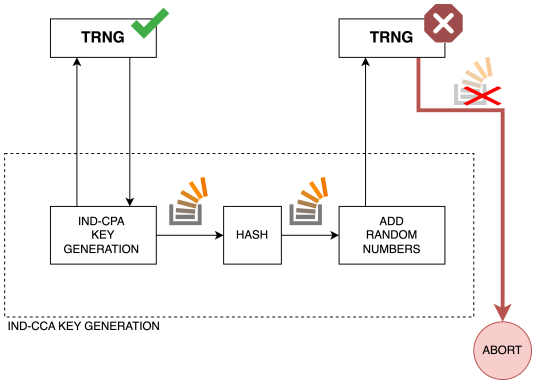

Figure 4: IND-CCA Key Generation RNG issue. The red color indicates a failure that results in the termination of the process and the removal of the stack. The green check indicates a successful call.

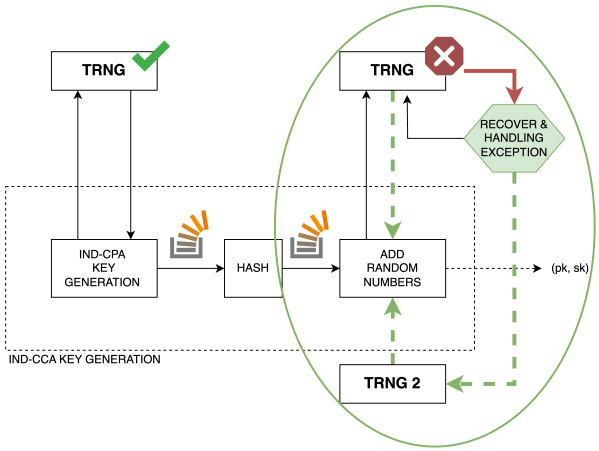

Figure 5: Our IND-CCA Key Generation RNG solution. The red color indicates a failure; this time, the exception is intercepted, giving the programmer the opportunity to take two different management paths, indicated by the green color. The first path gives the TRNG module a second chance, while the second path has a fallback mechanism.

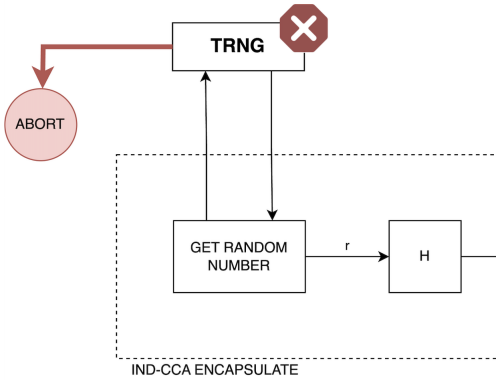

Figure 6: IND-CCA Encapsulation RNG issue. The red color indicates a failure, resulting in process termination and stack removal.

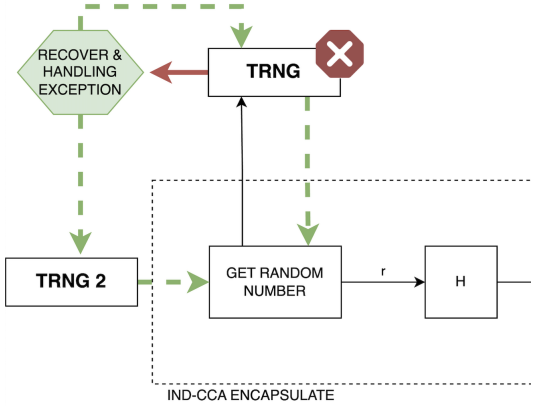

Figure 7: Our IND-CCA Encapsulation RNG solution. As in figure 5, this solution allows for a failure recovery. The first path gives the TRNG module a second chance, while the second path has a fallback mechanism.

type of functionality, we proposed to substitute some lines in the code to better exploit the capabilities of the *store64* function. For the sake of clarity, we report the code lines in the function *keccak_squeeze*, highlighting in red the lines that are substituted with the new code lines in green.

```
pub fn store64(x: &mut [u8], mut u: u64) {
    for i in x.iter_mut().take(8) {
        *i = u as u8;
        u >>= 8;
    }
}
```

```
pub fn keccak_squeeze(
    out: &mut [u8],
    mut outlen: usize,
    s: &mut [u64],
    mut pos: usize,
    r: usize,
) -> usize {
    let mut idx = 0;
    while outlen > 0 {
        if pos == r {
            keccakf1600_statepermute(s);
            pos = 0
        }
        let mut i = pos;
        let mut w = i/8;
        while i < r  && i < pos+outlen {
          out[idx]=(s[i/8]>>8*(i%8)) as u8;
          i += 1;
          idx += 1;
          store64(&mut out[idx..],s[w]);
          i += 8;
          w += 1;
          idx += 8;
        }
        outlen -= i - pos;
        pos = i;
    }
    pos
}
```

This solutions offers several advantages.

- Isolation of failures: since the *store64* function is called by 4 different functions, if something goes wrong, it is easy to trace back to the issue.
- Modularity and code reuse: clear responsibility for the *store64* function allow a better maintainability and reusability.
- Readability and testing: this solution facilitates the unit testing and the integration testing.

C. Enhancement of the Code Coverage

During the code coverage analysis process, some areas were not covered by the library's test suite available before March 16, 2023. In particular, the most important issue was the absence of tests covering addressing the branch for validating the dimension of the public key vector (input to the encapsulate function), and the private key (input to the decapsulate function). We proposed to the library maintainers to include additional tests for these sections, resulting in an increase in line coverage from 94.19% to 94.48%.

V. Conclusion and future works

In this work, we presented a detailed analysis of the official Crystal-Kyber Rust library [3]. We found some issues, and proposed solutions to improve the software quality and maturity of the library. All the solutions were accepted by the community.

We proposed an enhanced prevention and management of errors and malfunctions of the RNG peripheral. We have also provided tests regarding the impacts of adopting the Rust language for this PQC library, which is a novelty with respect to the current state of the art.

Since the study of post-quantum cryptography is a nowadays effort, our analysis could inspire some future works. For example, the Kyber algorithm is in the process of being standardized (ML-KEM), and then analysis of its implementations in different languages is useful to ensure the security of the algorithm and the safe-usage from end-users. In addition, the Rust community is continuously growing, especially for embedded systems, and this implies that code reviews of Rust code will be much more spread, and some solutions that we found in this work could be analysed and re-employed.

Moreover, our analysis has been performed only on a STM32F407VGT6 board; the next step in this direction is to test other hardware platforms, also open-source, such as the RISC-V architectures.

Finally, since power consumption is a critical issue, in particular when the target is an embedded device, we plan to do further evaluations and optimizations about the power efficiency of the library.

Acknowledgment

Security Pattern and Antonio J. Di Scala are partially supported by the QUBIP project (https://www.qubip.eu), funded by the European Union under the Horizon Europe framework program [grant agreement no. 101119746]. Antonio J. Di Scala is member of GNSAGA of INdAM and of CrypTO, the group of Cryptography and Number Theory of the Politecnico di Torino. He is also partially supported by project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union – NextGenerationEU.

References

[1] Miklós Ajtai. Generating hard instances of lattice problems. *Electron. Colloquium Comput. Complex.*, TR96, 1996. https://dl.acm.org/doi/pdf/10.1145/237814.237838.

[2] David Balbás. The hardness of lwe and ring-lwe: A survey. Cryptology ePrint Archive, Paper 2021/1358, 2021. https://eprint.iacr.org/2021/1358.

[3] Mitchell Berry. Kyber. https://github.com/Argyle-Software/kyber.

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

[5] Sana Farooq, Ayesha Altaf, Faiza Iqbal, Ernesto Thompson, Debora Vargas, Isabel De la Torre Díez, and Imran Ashraf. Resilience optimization of post-quantum cryptography key encapsulation algorithms. *Sensors*, 23:5379, 06 2023.

[6] Oded Goldreich, Shafi Goldwasser, and Shai Halevi. Public-key cryptosystems from lattice reduction problems. In Burton S. Kaliski, editor, *Advances in Cryptology — CRYPTO '97*, pages 112–131, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[7] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '96, page 212–219, New York, NY, USA, 1996. Association for Computing Machinery.

[8] Matthias J. Kannwischer, Richard Petri, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. PQM4: Post-quantum crypto library for the ARM Cortex-M4. https://github.com/mupq/pqm4.

[9] Zhichuang Liang and Yunlei Zhao. Number theoretic transform and its applications in lattice-based cryptosystems: A survey, 2022.

[10] Francesco Medina. Analysis and Contributions for Kyber written in Rust. https://github.com/francescomedina/kyber-rust-thesis.

[11] Daniele Micciancio. *Closest Vector Problem*, pages 212–214. Springer US, Boston, MA, 2011.

[12] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, 2000.

[13] National Institute of Standards and Technology. Nist ir 8240 - status report on the first round of the nist post-quantum cryptography standardization process. Technical report, U.S. Department of Commerce, Washington, D.C., 2019.

[14] National Institute of Standards and Technology. Module-lattice-based digital signature standard - draft. Technical report, U.S. Department of Commerce, Washington, D.C., 2023.

[15] National Institute of Standards and Technology. Module-lattice-based key-encapsulation mechanism standard - draft. Technical report, U.S. Department of Commerce, Washington, D.C., 2023.

[16] National Institute of Standards and Technology. Stateless hash-based digital signature standard - draft. Technical report, U.S. Department of Commerce, Washington, D.C., 2023.

[17] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM J. Comput.*, 26(5):1484–1509, oct 1997.

[18] Crystal team. Crystals - cryptographic suite for algebraic lattices: Kyber, 2020.

[19] Changji Wang, Yang Liu, and Jung-Tae Kim. An ind-cca2 secure key-policy attribute based key encapsulation scheme. In *2009 International Conference on Multimedia Information Networking and Security*, volume 1, pages 449–453, 2009.