2024 IEEE/ACS 21st International Conference on Computer Systems and Applications (AICCSA) | 979-8-3315-1824-0/24/$31.00 ©2024 IEEE | DOI: 10.1109/AICCSA63423.2024.10912602

Performance Evaluation and Benchmarking of PQC CRYSTALS-Kyber on Embedded Devices

1st Mohamed Amine Mighri
National School of Engineering of Sousse
University of Sousse
Sousse, Tunisia
Mohamedamine.Mighri@eniso.u-sousse.tn

2nd Ahmed Benfarah
NOCCS Laboratory
University of Sousse
Sousse, Tunisia
Ahmed.Benfarah@eniso.u-sousse.tn

3rd Aref Meddeb
Faculty of Engineering
University of Sherbrooke
Sherbrooke, Canada
Aref.Meddeb@USherbrooke.ca

***Abstract*—The growing threat posed by quantum computing to classical cryptographic primitives has prompted significant research activities on Post-Quantum Cryptography (PQC). Among these, CRYSTALS-Kyber has emerged as a promising candidate due to its IND-CCA2 security and efficient performance. This paper presents a comprehensive benchmarking study focused on evaluating the performance of CRYSTALS-Kyber on embedded devices, with a particular emphasis on the popular Raspberry Pi platform.**

***Index Terms*—Post Quantum Cryptography, CRYSTALS-Kyber, Raspberry Pi.**

I. Introduction

In recent years, embedded devices have become ubiquitous in various applications, from Internet of Things (IoT) to industrial control systems. However, the resource-constrained nature of these devices presents unique challenges for implementing strong cryptographic primtives. Therefore, assessing the performance of PQC schemes [1] like CRYSTALS-Kyber [2] on such platforms is critical for determining their feasibility in real-world IoT deployments.

The main focus of this article lies on the performance evaluation of CRYSTALS-Kyber on embedded devices, particularly on the Raspberry Pi 3 platform. Through rigorous experimentation, we analyze CPU and RAM usage to assess the algorithm's adequacy for resource-constrained devices.

In addition to presenting empirical results, we conduct a comparative analysis, contrasting the performance of CRYSTALS-Kyber with state-of-the-art algorithms such as Elliptic Curve Diffie-Hellman (ECDH) [3]. This analysis not only sheds the light on the efficiency of CRYSTALS-Kyber, but also provides insights into the trade-offs involved in its implementation on embedded devices. We contribute to the understanding of PQC performance on embedded platforms, offering valuable insights on the integration of CRYSTALS-Kyber and similar algorithms into real-world IoT applications, where resource constraints are a primary concern.

Additionally, this study aligns with the recent publication of the NIST standard FIPS 203 for lattice-based key-encapsulation mechanisms, which highlights the ongoing advancements in PQC standards [4].

The remainder of this article is organized as follows. In Section II, we provide an overview of CRYSTALS-Kyber, detailing its specifications and software implementations using Java and C programming languages. Performance evaluation results on a laptop are also presented as a reference for comparison. In Section III, we evaluate the performance of CRYSTALS-Kyber on the Raspberry-Pi platform and compare it to classical crytographic schemes. We conclude by summarizing the most significant findings and by providing some future work directions.

II. Background on crystals-kyber

A. Kyber Specifications

CRYSTALS-Kyber [2], [5] is a robust PQC scheme [6], designed to resist attacks from quantum computers while upholding stringent security levels. As a KEM (Key Encapsulation Mechanism) [7], Kyber's specifications achieve the IND-CCA2 [8] security level (indistinguishability under chosen-ciphertext attack), ensuring robust protection against sophisticated cryptographic attacks.

Its specifications are carefully designed to strike a balance between security, efficiency and ease of implementation, rendering it applicable across diverse scenarios, including resource-constrained devices.

CRYSTALS-Kyber relies on the mathematical hardness of certain problems related to lattices, specifically a type of lattice called a module lattice [9]. Imagine a lattice as a giant grid with many holes (lattice points) arranged in a specific geometric pattern. The Learning With Errors (LWE) problem broadly refers to the challenge of distinguishing between structured data points on this lattice and random data points scattered around it. The Module Lattice problem is a variant of the lattice problem that focuses on lattices defined over module structures, which can be seen as an extension of vector spaces [10]. The Module LWE (MLWE) problem, which is central to the security of CRYSTALS-Kyber, combines elements of the LWE problem with the additional complexity introduced by module lattices, making it even harder to solve and thus ensuring stronger cryptographic security.

The core of Kyber specifications [5] includes three primitives:

- Key generation: This process creates a cryptographic public-private key pair. The public key is shared with

anyone you want to communicate with, while the private key is kept secret.
- Encapsulation: Here, the sender uses the recipient's public key to generate a shared secret key and encrypt a message. This shared key is only decrypted by the entity with the corresponding private key.
- Decapsulation: The recipient uses its private key to extract the shared key from the encrypted message.

Figure 1 depicts the operation steps of Kyber, including the generation of public/private keys pair, the encapsulation of a shared secret, and its subsequent decapsulation by the intended recipient.

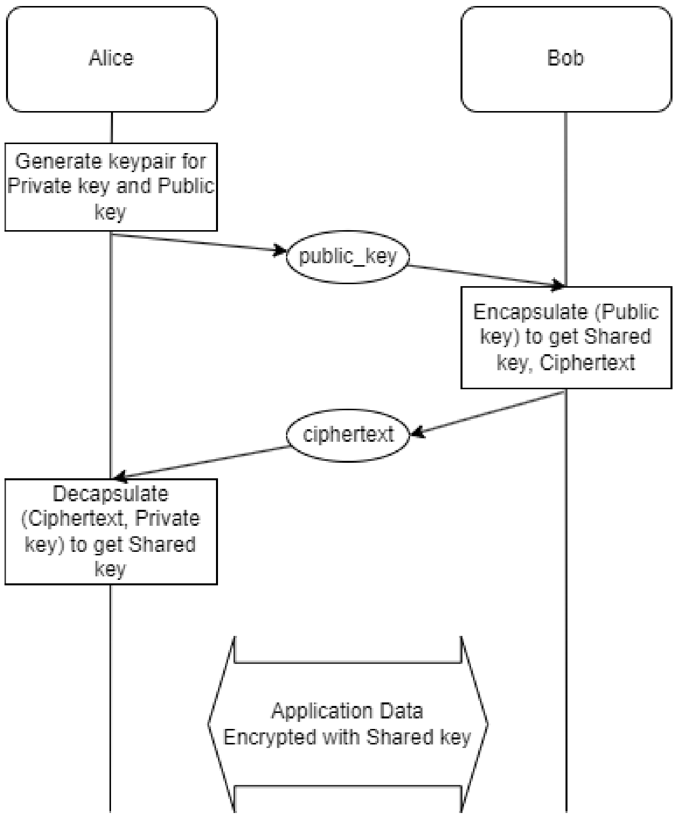
Alice
Bob
Generate keypair for Private key and Public key
public_key
Encapsulate (Public key) to get Shared key, Ciphertext
ciphertext
Decapsulate (Ciphertext, Private key) to get Shared key
Application Data Encrypted with Shared key

Fig. 1. Diagram illustrating Kyber KEM operation.

Three parameter sets for Kyber, called Kyber512 (NIST security level 1, $\approx$ AES-128), Kyber768 (NIST security level 3, $\approx$ AES-192), and Kyber1024 (NIST security level 5, $\approx$ AES-256) [5]. The parameter sets are listed in Table I.

TABLE I
PARAMETER SETS FOR KYBER SPECIFICATIONS

Algorithm	Security level	Public key size (in Bytes)	Private Key size (in Bytes)	Ciphertext size (in Bytes)
Kyber512	1	800	1632	736
Kyber768	3	1184	2400	1088
Kyber1024	5	1568	3168	1568

Overall, Kyber's specifications embody a careful balance between security, efficiency, and practicality, making it a promising choice for securing data communication on both conventional and emerging computing platforms [11].

B. Software Implementation with Java and C programming languages

To facilitate the practical adoption of Kyber, software implementations have been developed using popular programming languages such as Java and C. These reference implementations adhere to the specifications of the algorithm and provide a basis for further optimization and integration into various systems and platforms. In this subsection, we go through details of the Java Bouncy Castle [12] and C official reference implementation [13] of Kyber, highlighting their design principles, code structure, and functionality. Additionally, we present performance evaluation results obtained through testing these implementations in a laptop environment. These results serve as a benchmark for assessing the efficiency and effectiveness of Kyber in real-world scenarios.

1) Java implementation :

To evaluate the performance of Kyber variants, we ran a series of tests measuring key generation, encapsulation, and decapsulation times. We used Java implementation built on the Bouncy Castle libraries [12], a popular cryptography toolkit that includes the latest NIST PQC algorithms. The Bouncy Castle Crypto package is a Java implementation of cryptographic algorithms. The package is organised so that it contains a light-weight API suitable for use in any environment (including the newly released J2ME) with the additional infrastructure to conform the algorithms to the JCE framework. The tests were conducted on a laptop with an AMD Ryzen™ 5 5600H processor clocked at 3.3 GHz. The laptop also included 16 GBytes of DDR4 memory and a solid-state drive (SSD) for storage.

The Java implementation utilized the Bouncy Castle libraries for cryptographic operations. This information helps ensure our results can be replicated and compared to future studies. Our main focus was on how long each cryptographic operation took to run and how much memory it used. This will help us understand how efficient these algorithms are and if they're practical for real-world use. The Kyber variants were benchmarked for execution time performance; see Table II. The System.nanoTime() method was used to measure the execution time of cryptographic functions of the different Kyber variants. Every function was iterated 1000 times in a loop and then averaged for possible variation. Table III further describes the memory usage details observed during the execution of all the experiments, along with any inconsistency regarding resource usage by different Kyber variants.

As we can notice, Kyber512 has the fastest execution time for the three operations i.e., key pair generation, encapsulation, and decapsulation. Execution time increases as the algorithm complexity increases i.e., Kyber512 vs. Kyber768 vs. Kyber1024. This is due to the increased number of computations required for higher security levels.

The memory consumption shows a similar trend as execution time, with Kyber512 having the lowest consumption

TABLE II
EXECUTION TIME FOR JAVA KYBER IMPLEMENTATION IN A LAPTOP ENVIRONMENT

Algorithm	Kyber_keypair (ms)	Kyber_encaps (ms)	Kyber_decaps (ms)
Kyber512	0.054	0.055	0.071
Kyber768	0.080	0.081	0.100
Kyber1024	0.124	0.130	0.156

TABLE III
AVERAGE MEMORY CONSUMPTION OF JAVA KYBER IMPLEMENTATION IN A LAPTOP ENVIRONEMENT

Algorithm	Average Memory Consumption (MegaBytes)
Kyber512	6.305
Kyber768	6.306
Kyber1024	6.481

and Kyber1024 having the highest. The increase in memory consumption with increasing algorithm complexity is due to the larger data structures needed for more secure variants.

2) C official reference implementation :

In contrast to the Java implementation, the C official reference implementation offers a low-level approach, optimizing performance by leveraging the efficiency of the C programming language. Developed as a lightweight and portable solution, this implementation targets various platforms and architectures, ensuring compatibility and versatility. The design of the C implementation [13] emphasizes modularity and readability, allowing for easy comprehension and modification as needed. By adhering closely to the Kyber specifications while exploiting the capabilities of the C language, this implementation achieves a balance between speed and maintainability.

To provide insights into the performance of Kyber in a C environment, we conducted rigorous testing on the same laptop configuration as the Java experiments. By measuring execution time and memory usage across different Kyber variants, we aim to assess the comparative efficiency of the C implementation.

Table IV presents the execution times obtained from our experiments, the system time (sys/time library), and the function ”gettimeofday()” were used to collect time data in a passive mode. The time difference before and after a step of the algorithm was measured. This process was run in a loop 1000 times and we calculate the average. Additionally, some instances of the original CPU cycle measurement code were required to be removed from the Makefile for the program to run successfully. Table V outlines the memory usage observed during the execution, highlighting any discrepancies in resource consumption among the variants.

The execution time and memory consumption increase as the Kyber parameter size increases. This is because larger parameter sizes lead to more complex computations and require more memory to store the cryptographic keys.

TABLE IV
AVERAGE TIME EXECUTION OF C KYBER IMPLEMENTATION IN A LAPTOP ENVIRONMENT

Algorithm	Kyber_keypair (ms)	Kyber_encaps (ms)	Kyber_decaps (ms)
Kyber512	0.042	0.043	0.055
Kyber768	0.065	0.066	0.081
Kyber1024	0.092	0.095	0.12

TABLE V
AVERAGE MEMORY CONSUMPTION OF C KYBER IMLEMENTATION IN A LAPTOP ENVIRONMENT

Algorithm	Average Memory Consumption (MegaBytes)
Kyber512	0.960
Kyber768	0.964
Kyber1024	0.968

C. Comparative Analysis

Figure 2 shows a bar chart comparing the execution times of the studied cryptographic algorithms. These are identified by a combination of algorithm names, followed by the implementation language (Java or C) in parentheses. For this article, we chose to use a Java cryptography library called Bouncy Castle to implement the Elliptic Curve Diffie-Hellman (ECDH). ECDH is a widely used cryptographic primitive that allows two parties to generate a shared private key over an insecure channel. Bouncy Castle offers a well-tested, well-used ECDH, making it ideal for our research.

As we can notice, ECDH (green bars) exhibits the highest processing time compared to Kyber (blue and orange bars). This is due to the inherent complexity of the ECDH calculation compared to simple operations in Kyber. In terms of ECDH activity, ECDH Sect571r1 has the longest duration, followed by ECDH Sect409r1 and ECDH Sect283r1. This means that the safety level of the ECDH curve (which is reflected in the curve name) can influence the timing. Curves with high degrees of safety generally require complex calculations, resulting in slower execution times. When comparing Java and C implementations for the same variant (e.g., Kyber512), C implementation (orange bars) exhibits lower execution times than Java implementation (blue bars) because C is a compiled language, whereas Java is an interpreted language.

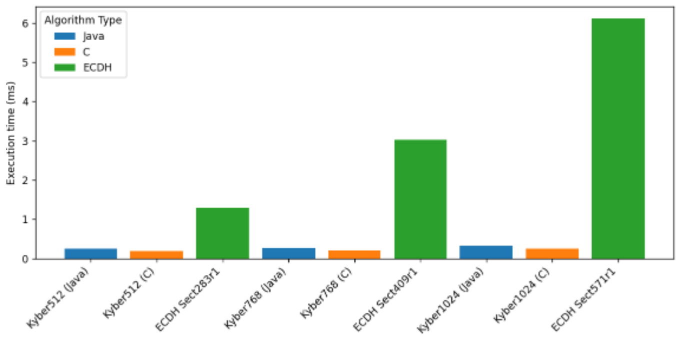

Fig. 2. Execution time performance comparison.

In conclusion, the software implementations of Kyber utilizing Java and C programming languages provide valuable insights into the algorithm's performances. The Java implementation, leveraging libraries such as Bouncy Castle and Kyber, showcases a comprehensive analysis of execution time and memory usage across different Kyber variants. Meanwhile, the C official reference implementation emphasizes performance optimization through a low-level approach, ensuring compatibility and versatility across various platforms and architectures.

These results highlight the efficiency of both implementations, with the C implementation demonstrating slightly improved execution times and significantly reduced memory consumption compared to its Java counterpart.

III. CRYSTALS-KYBER PERFORMANCE EVALUATION ON EMBEDDED DEVICES

Evaluating cryptographic algorithms on resource-constrained devices is crucial for ensuring secure and efficient operations in real-world IoT applications. In this section, we assess the performance of CRYSTALS-Kyber on the Raspberry Pi 3 Model B v1.2 platform and the ESP-WROOM-32 embedded card. These devices were chosen because they represent two widely used categories in IoT deployments: the Raspberry Pi 3, a popular single-board computer with relatively higher processing power and versatility, and the ESP-WROOM-32, a microcontroller-based platform designed for low-power, cost-effective IoT solutions. By testing on both devices, we cover a broad spectrum of IoT scenarios, from more computationally capable nodes to highly constrained edge devices. This dual-platform approach provides a comprehensive baseline for assessing cryptographic performance in various industrial applications.

A. *Performance evaluation on Raspberry Pi 3*

The Raspberry Pi 3 is characterized by small size and low cost, which makes it suitable for various applications including IoT, edge computing and education. It has a Broadcom BCM2837 processor that uses the Quad-Core ARM Cortex-A53 clocked at 1.2 GHz and 1 GByte LPDDR2 SDRAM memory that offers modest, yet capable computing power suitable for a wide range of applications. Table VI summarizes the Raspberry Pi 3 Model B characteristics.

TABLE VI
OVERVIEW OF THE RASPBERRY PI 3 CHARACTERISTICS.

	Raspberry Pi 3
Processor	64-bit quad-core ARM Cortex-A53
Clock frequency	1200 MHz
RAM	1024 MByte
Wi-Fi	Yes
Bluetooth	Yes
Power supply	5V, 2.5A

The speed of the CRYSTALS-Kyber algorithm is tested at different security levels. For this purpose, the official software of Kyber is utilized. The software is downloaded and a special file called *"test_speed.c"* is created to run the speed tests. This file then calls other files named *"test_speed512", "test_speed768", and "test_speed1024"*, depending on the desired security level. The run-time of the three cryptographic primitives of Kyber is measured by the created files. Challenges were encountered in obtaining time measurements due to compatibility issues between Linux CPU cycles and Raspberry Pi CPU. To address this issue, system time and the *"gettimeofday()"* function are utilized, and were executed in a loop to average over 1000 cycles. Additionally, the Makefile is edited to remove CPU cycle measurement code.

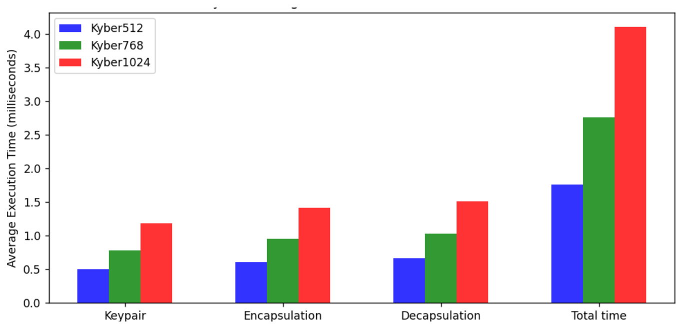

Fig. 3. Kyber Average Execution Time on Raspberry Pi 3 environment.

As can be seen from Figure 3, the execution time increases with the security level. For example, the key pair generation time for Kyber512 is about 0.498 milliseconds, while for Kyber1024 it is about 1.184 milliseconds. This indicates a clear linear trend in the increase of execution time with increasing security levels. As for the impact on cryptographic operations, all primitives (key pair generation, encapsulation, and decapsulation) experience a proportional increase in execution time as the security level rises. There is not a significant disparity in the degree of effect on these operations; they are all equally affected by the increase in security level. We also carried out performance evaluations of Kyber by counting CPU cycles in addition to the execution time measurements. By doing this, we understand much better the computational efficiency of the algorithm, which is crucial when analyzing its performance on resource constrained devices such as the Raspberry Pi. Figure 4 presents the average CPU cycles required for key pair generation, encapsulation and decapsulation at different security levels of CRYSTALS-Kyber.

We notice a clear exponential trend in the number of CPU cycles required for cryptographic operations as the security level passes from Kyber512 to Kyber1024. Notably, the increase in CPU cycles for key generation, encapsulation and decapsulation is substantial, indicating heightened computational requirements with higher security levels. In particular, the increase in CPU cycles for key generation is noteworthy, with Kyber1024 requiring significantly more cycles compared to Kyber512. Similarly, both encapsulation and decapsula-

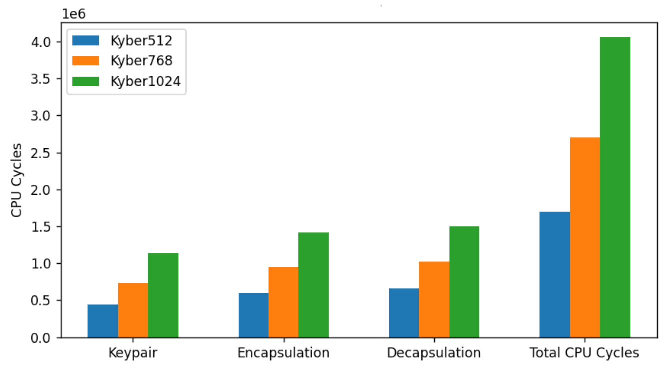

Fig. 4. Kyber CPU cycles on Raspberry Pi 3.

tion operations exhibit a considerable rise in CPU cycles as the security level increases. This trend emphasizes the impact of security level on computational requirements across cryptographic primitives, with each operation experiencing a proportional increase in computational complexity.

To dig deeper into Kyber's algorithm performance, comparisons have been made with different Kyber variants and the ECDH using the sect283r1 binary field Weierstrass curve that offers the equivalent of a 128 bits security and for that we implemented (ECDH) using the Tiny ECDH library. The library supports 10 standard NIST curves of varying sizes, including 5 pseudo-random curves and 5 Koblitz curves. These curves provide security levels ranging from 80 to 256 bits symmetrically equivalent security. It is worth noting that only results from comparisons with ECDH sect283r1 are presented, as other curves resulted in significantly longer execution times compared to Kyber. Essentially, curves providing the equivalent of 128 bits of security require substantially much more time than Kyber1024, which offers the equivalent of 256 bits of security. Hence, comparing with ECDH sect283r1 is deemed sufficient for our study.

Table VII highlights that all Kyber variants are significantly faster than ECDH Sect283r1.

TABLE VII
ECDH vs. Kyber on Raspberry Pi 3 environment

	Kyber512	**Kyber768**	**Kyber1024**	**ECDH Sect283r1**
Execution time (ms)	1.765	2.764	4.114	2775.516
CPU Cycles	1696446	2707131	4056566	2726717108
Code size (Bytes)	17851	17903	19467	11551

For instance, Kyber512 is approximately 1573 times faster than Sect283r1, while Kyber1024 is still approximately 675 times faster. The CPU count also shows that Kyber significantly outperforms ECDH. For example, Kyber512 is significantly faster than Sect283r1 for equivalent security level (128-bit), boasting a speedup of over 1600 times. Even Kyber1024, offering the highest security (256-bit), is hundreds of times faster than Sect283r1. This exceptional performance in comparison to ECDH makes Kyber a compelling choice for resource-constrained devices and applications requiring stringent delays, where faster encryption is crucial.

B. *Performance evaluation on ESP-WROOM-32*

We consider here the ESP-WROOM-32 Lolin32 module. It is a small-size microcontroller with a relatively low cost that can be used in a range of applications, including IoT, edge computing and learning projects. The Lolin32 development board is based on the ESP32 microcontroller, containing a dual-core Tensilica LX6 processor with a clock frequency of 240 MHz, that also incorporates 520 KBytes of SRAM. Table VIII provides an overview of the ESP-WROOM-32 characteristics.

TABLE VIII
Overview of the ESP-WROOM-32 characteristics

	Specifications
Processor	Xtensa dual-core (or single-core) 32-bit LX6 microprocessor
Clock Speed	160 or 240 MHz
Performance	Up to 600 DMIPS
Co-processor	Ultra low power (ULP) co-processor
Memory	520 KBytes SRAM
Wireless Connectivity	Wi-Fi: 802.11 b/g/n
Bluetooth	v4.2 BR/EDR and BLE

The NIST C reference implementation is incorporated to the ESP-WROOM-32 platform through the ESP IDF development framework. By doing this, we could exploit the ESP32 microcontroller's full potential regarding cryptographic operations. The version 5.2.1 of the ESP-IDF development framework was used and compilation was performed using the default GCC in GNU Compiler Collection, version 11.4.0 compiler. The compiler optimization was turned off to get the actual performance of the board. Figure 5 shows experimental results of the execution time of key generation, encapsulation and decapsulation for each Kyber variant. We used a similar approach to measure execution time on the ESP-WROOM-32 as we did on the Raspberry Pi. We relied on *system time*() and the *gettimeofday*() functions.

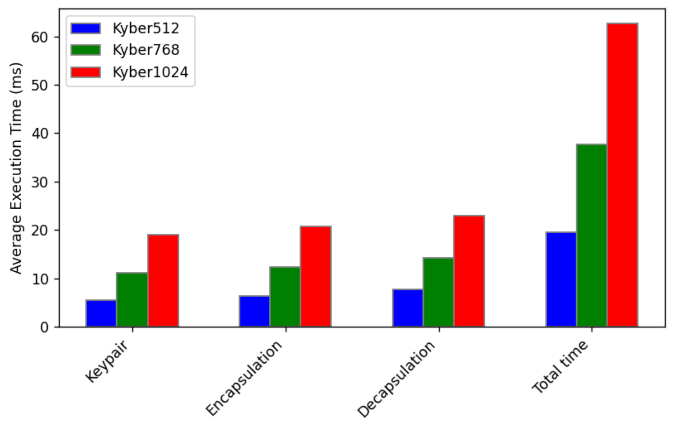

Fig. 5. Kyber Average Execution Time on ESP-WROOM-32 environment.

The average execution time of Kyber on ESP-WROOM-32 exhibits a clear increasing trend across all cryptographic operations as the security level progresses from Kyber512 to Kyber1024. Notably, there is a substantial increase in execution time for key pair generation, encapsulation, and decapsulation operations. For instance, the average execution time for key pair generation increases from 5.423 ms for Kyber512 to 18.939 ms for Kyber1024. Similarly, both encapsulation and decapsulation operations experience a significant rise in execution time with increasing security levels. When comparing the increase in execution times across cryptographic primitives, it is clear that the increase is not uniform. Key generation generally exhibits the highest increase in execution time, followed by encapsulation and decapsulation. This suggests that the impact of security level on execution time varies across different cryptographic primitives.

For finer-grained performance metrics, we also recorded the CPU cycles required by every cryptographic operation presented in Figure 6. We used the ESP32 cycle counter with the help of the *"esp_cpu_get_cycle_count()"* function to obtain the CPU cycle count before and after launching the activity. We accessed finer cycle counts that allowed more accurate performance evaluations.

Results indicate a clear increase in the computational overhead when the security parameter is increased, a reflection of the well-known tradeoff between security and performance.

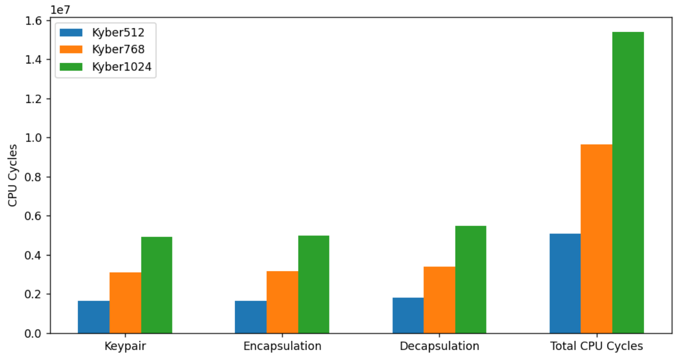

Fig. 6. Kyber CPU cycles on ESP-WROOM-32.

As we can notice, the increase in CPU cycles for all operations (key pair generation, encapsulation and decapsulation) is more exponential than linear. The ratios of increase are consistently around 1.5 to 2 times as the security level increases from Kyber512 to Kyber1024. This suggests a trend that is closer to exponential rather than linear. In summary, as the security level increases, the computational cost grows substantially, suggesting an exponential trend, and all cryptographic operations are similarly impacted by the higher security levels. We compared the performances of different Kyber variants and the ECDH using the sect283r1 binary field Weierstrass curve that offers the equivalent of 128-bits security. The same methodology was applied to both Kyber and ECDH implementations. The ESP IDF development framework was used to structure the code, ensuring functionality and optimization on the ESP32 platform. This approach allowed for a fair and accurate performance evaluation by leveraging the robust development environment provided by ESP IDF, which is well-suited for ESP32's architecture.

TABLE IX
ECDH vs. Kyber Performance comparison on ESP-WROOM-32

	Kyber512	**Kyber768**	**Kyber1024**	**ECDH Sect283r1**
Execution time (ms)	19.524	37.638	62.632	7601.339
CPU Cycles	5090172	9636322	15375759	3017540740
Code size (Bytes)	199685	199557	199845	178909

We can notice as shown in Table IX that the different Kyber variants are significantly faster than ECDH sect283r1. For instance, Kyber512 is approximately 389 times faster than sect283r1, while Kyber1024 is still approximately 121 times faster. The CPU cycle count also shows that Kyber outperforms ECDH.

For example, Kyber512 is significantly faster than sect283r1 for the same security level (128-bit), boosting a speedup of over 593 times. Even Kyber1024, offering the highest security level, is nearly 196 times faster than sect283r1. This exceptional performance makes Kyber a compelling choice for resource-constrained devices and applications requiring high throughput, where fast encryption is crucial. Nevertheless, from a code size point of view, ECDH sect283r1 slightly outperforms all Kyber variants. This point might be critical when it comes to system memory.

C. Benchmarking summary

Kyber demonstrates superior performance over ECDH across the three tested platforms: a laptop, a Raspberry Pi 3, and an ESP-WROOM-32 . This performance gap is highlighted by Kyber's lower execution time and CPU cycle count compared to ECDH. Table X summarizes benchmarking tests of both algorithms (Kyber and ECDH) across the different platforms.

For instance, at a 128-bit security level, Kyber512 is executed in just 0.14 milliseconds on a laptop, whereas ECDH (Sect283r1) requires about 202 milliseconds. This result underscores Kyber's efficiency. The disparity in performance is even more evident on less powerful devices like the Raspberry Pi 3 and ESP-WROOM-32 . On the Raspberry Pi 3, Kyber512 is run in 1.765 milliseconds, while ECDH (Sect283r1) requires more than 2775 milliseconds, making Kyber over 1500 times faster. The benchmarking results clearly indicate that Kyber is a more efficient and higher-performing alternative to ECDH on all tested platforms, at a small cost of additional system memory usage. This advantage is particularly critical for resource-constrained devices, where Kyber's reduced execution time and CPU cycle count can lead to significant battery saving and enhanced overall system performance.

TABLE X
BENCHMARKING OF KYBER AND ECDH IN DIFFERENT ENVIRONMENTS

	Algorithm		**Laptop**		**Raspberry Pi 3**		**ESP-WROOM-32**	
			Execution Time (ms)	**CPU Cycle**	**Execution Time (ms)**	**CPU Cycle**	**Execution Time (ms)**	**CPU Cycle**
128	**Kyber512**	**Key Gen**	0.042	120439	0.498	441820	5.423	1639107
		Enc	0.043	163051	0.604	601042	6.352	1640163
		Dec	0.055	208534	0.663	659584	7.749	1810902
		Total	**0.14**	**492024**	**1.765**	**1696446**	**19.524**	**5090172**
192	**Kyber768**	**Key Gen**	0.065	221800	0.778	728593	11.058	3081611
		Enc	0.066	235563	0.956	954109	12.396	3172551
		Dec	0.081	307826	1.030	1024429	14.184	3382160
		Total	**0.212**	**765198**	**2.764**	**2707131**	**37.638**	**9636322**
256	**Kyber1024**	**Key Gen**	0.092	351587	1.184	1134684	18.939	4916966
		Enc	0.095	424625	1.420	1421388	20.720	4970964
		Dec	0.12	456534	1.510	1500494	22.973	5487829
		Total	**0.307**	**1232638**	**4.114**	**4056566**	**62.632**	**15375759**
128	**ECDH**	**Sect283r1**	**202.311**	**575745174**	**2775.516**	**2726717108**	**7601.339**	**3017540740**

IV. CONCLUSION

In this article, we evaluated the performance of CRYSTALS-Kyber on resource-constrained devices, specifically the Raspberry Pi 3 and the ESP Wroom 32, to assess its feasibility for real-world IoT applications. Experimental results demonstrate that Kyber is effective in terms of runtime on these devices, exhibiting better performances compared to Elliptic Curve Diffie-Hellman (ECDH), at a small cost of additional system memory. This indicates that Kyber is indeed a viable option for secure communications in environments with limited computational resources. Future work will focus on evaluating Kyber's performance in real use-case scenarios, as well as conducting a detailed analysis of its memory and energy consumption to ensure it meets the requirements of various IoT applications.

REFERENCES

[1] Manish Kumar. Post-quantum cryptography Algorithm's standardization and performance analysis. *Elsevier Array*, 15:1–27, 08 2022.

[2] Joppe W. 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. In *IEEE European Symposium on Security and Privacy London, United Kingdom*, pages 353–367, April 2018.

[3] Mlynek Petr, Raso Ondrej, Fujdiak Radek, Pospichal Ladislav, and Kubicek Pavel. Implementation of Elliptic Curve Diffie Hellman in ultra-low power microcontroller. In *IEEE 38th International Conference on Telecommunications and Signal Processing*, pages 662–666, 07 2015.

[4] NIST. Module-Lattice-Based Key-Encapsulation Mechanism Standard. https://csrc.nist.gov/pubs/fips/203/final, 2024. [Accessed 02-09-2024].

[5] NIST. Kyber — pq-crystals.org. https://pq-crystals.org/kyber/index.shtml, 2021. [Accessed 22-04-2024].

[6] Daniel J. Bernstein and Tanja Lange. Post-quantum cryptography. *Nature*, 549:188–194, 09 2017.

[7] Tsunekazu Saito, Keita Xagawa, and Takashi Yamakawa. Tightly-Secure Key-Encapsulation Mechanism in the Quantum Random Oracle Model. *IACR Cryptol. ePrint Arch.*, 10822, 2017.

[8] Jiang Haodong, Zhang Zhenfeng, Chen Long, Wang Hong, and Ma Zhi. IND-CCA-Secure Key Encapsulation Mechanism in the Quantum Random Oracle Model, Revisited. *Lect. Notes Comput. Sci*, 10993:96–125, 2018.

[9] Micciancio Danieleand Regev Oded. *Lattice-based Cryptography*, pages 147–191. Springer Berlin, Heidelberg, 2009.

[10] Das Dipayan, Hoffstein Jeffrey, Pipher Jill, Whyte William, and Zhang Zhenfei. Modular lattice signatures, revisited. *Designs, Codes and Cryptography*, 88:1–32, 03 2020.

[11] Yiming Huang, Miaoqing Huang, Zhongkui Lei, and Jiaxuan Wu. A pure hardware implementation of CRYSTALS-KYBER PQC algorithm through resource reuse. *IEICE Electronics Express*, 17(17):1–6, 08 2020.

[12] org.bouncycastle.pqc.crypto.crystals.kyber package summary - bcprov-jdk15to18 1.77 javadoc — javadoc.io. https://javadoc.io/doc/org.bouncycastle/bcprov-jdk15to18/1.77/org/bouncycastle/pqc/crypto/crystals/kyber/package-summary.html. [Accessed 28-04-2024].

[13] GitHub - pq-crystals/kyber — github.com. https://github.com/pq-crystals/kyber. [Accessed 28-04-2024].