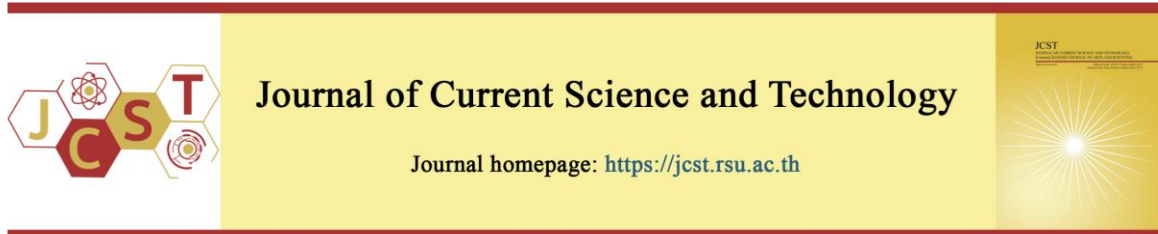
Journal of Current Science and Technology, April-June 2026 Vol. 16 No. 2, Article 183
ISSN 2630-0656 (Online)
Copyright ©2018-2026, Rangsit University

Cite this article: Junlachaiworakun, N., Panit, N., & Somsuk, K. (2026). Performance of CRYSTALS-Kyber on Raspberry Pi 5 under embedded-system constraints: A comparison with RSA and EC-KEM. *Journal of Current Science and Technology*, *16*(2), Article 183. https://doi.org/10.59796/jcst.V16N2.2026.183



Performance of CRYSTALS-Kyber on Raspberry Pi 5 under Embedded-System Constraints: A Comparison with RSA and EC-KEM

Nattapon Junlachaiworakun[1,2], Nirundon Panit[3], and Kritsanapong Somsuk[2,3,*]

[1]Department of Data Science and Information Technology, Faculty of Science, Udon Thani Rajabhat University, Udon Thani 41000, Thailand
[2]Graduate School, Udon Thani Rajabhat University, Udon Thani 41000, Thailand
[3]Faculty of Technology and Engineering, Udon Thani Rajabhat University, Udon Thani 41000, Thailand

*Corresponding author; E-mail: kritsanapong@udru.ac.th

Received 24 October 2025; Revised 23 December 2025; Accepted 19 January 2026; Published online 25 March 2026

Abstract

The emergence of quantum computers presents a significant risk to public key cryptography algorithms, including RSA and elliptic curve cryptography (ECC). CRYSTALS-Kyber, also called Kyber, was standardized by NIST as the Module-Lattice-based-Key Encapsulation Mechanism (ML-KEM, FIPS 203) to address quantum threats. However, its performance on modern embedded ARM platforms remains uncharacterized. The aim of this study is to benchmark the Kyber family, including Kyber-512, Kyber-768, and Kyber-1024, on the Raspberry Pi 5, which is selected to represent embedded-system constraints. In addition, RSA and EC-KEM were chosen as baseline comparisons. Execution time, memory consumption (RSS), and key/ciphertext sizes were measured over 1,000 iterations using custom Python scripts (liboqs 0.14.0 and the Python cryptography library) on 64-bit Ubuntu Linux under controlled parallel execution. The results show that Kyber achieves sub-millisecond execution times (0.05-0.21 ms) for key generation, encapsulation, and decapsulation with low variability (SD < 0.02 ms). Nevertheless, RSA-2048 requires 250-264 ms per operation, while EC-KEM ranges from 0.29 ms for X25519 to 3.04 ms for secp521r1. Memory consumption is comparable across all algorithms (24-25 MB RSS). Kyber's larger keys and ciphertexts (800-1568 bytes vs. 32-294 bytes for RSA/EC-KEM) present a latency-bandwidth trade-off for embedded deployments. Therefore, Kyber is computationally viable for latency-sensitive operations on modern ARM embedded systems. The evaluation focuses on discrete cryptographic operations, protocol-level integration and energy measurements are deferred to future work.

Keywords: *post-quantum cryptography; CRYSTALS-Kyber; embedded systems benchmarking; performance evaluation; key encapsulation mechanism*

1. Introduction

The development of quantum computers presents significant risks to several techniques in the field of public key cryptography. Well-known public key cryptography algorithms include RSA (Rivest et al., 1978) and Elliptic Curve Cryptography (ECC) (Koblitz, 1987; Miller, 1985), which are based on the challenges of integer factorization problem (IFP) (Mumtaz & Ping, 2019; Susilo et al., 2021; Boneh, 1999; Wiener, 1990) and elliptic curve discrete logarithm problems (ECDLP), respectively, for their security. Moreover, Shor's algorithm, which is one of the most efficient factoring algorithms based on quantum computing, provides polynomial-time attacks on these problems. This capability may affect the security of existing public key cryptography algorithms (Mosca, 2018) when large-scale quantum computers become operational (Shor, 1994; Larasati

& Kim, 2021). Therefore, many cryptographic algorithms based on traditional computers and used for secure communication, authentication, and key generation, face substantial long-term vulnerabilities.

To address this threat, Post-Quantum Cryptography (PQC), which is based on classical computers and resistant to quantum attacks is emerging as one of the most promising approaches for increasing the security level. In general, Kyber, standardized by the NIST in 2024, was presented as the Module-Lattice-based Key Encapsulation Mechanism (ML-KEM) under FIPS 203.

Previous studies have shown that Kyber's performance on resource-limited devices such as the Raspberry Pi 3 and ESP32 has been studied (Mighri et al., 2024; Fitzgibbon & Ottaviani, 2024). The launch of the Raspberry Pi 5, featuring a significantly enhanced ARM Cortex-A76 architecture, provides higher computing power and memory bandwidth. This advancement creates an opportunity to evaluate whether Kyber remains practical on this next-generation embedded platform.

This paper aims to address this limitation by conducting a platform-specific benchmarking study of Kyber on the Raspberry Pi 5 across the three standardized parameter sets, with direct comparisons to RSA and EC-KEM using similarly structured encapsulation processes. The evaluation considers computation time, memory usage, key and ciphertext sizes, and the stability of execution time over repeated runs. These aspects are central to determining whether an algorithm can be applied in resource-constrained environments.

The key contributions of this work are divided into three parts. The first part is a careful characterization of Kyber's performance on modern ARM-based embedded hardware. The second part is a variability analysis demonstrating execution-time consistency across different parameter sets. The last part is practical implications for developers considering PQC in embedded devices. However, this evaluation is limited to isolated cryptographic operations on a single platform and does not cover full protocol-level integration or energy consumption, which are left for the future work.

1.1 Public-Key Cryptography and Threats

The development of quantum computers presents an existential threat to many public key cryptography algorithms. In fact, RSA (Rivest et al., 1978) and ECC (Koblitz, 1987; Miller, 1985) derive their security from IFP and ECDLP, respectively. Moreover, Shor's algorithm (Shor, 1994) provides polynomial-time solutions to both problems on quantum computers, rendering these cryptosystems vulnerable once sufficiently powerful quantum hardware becomes operational (Larasati & Kim, 2021). EC-KEM was developed to enhance key exchange efficiency (Van Assen et al., 2024) but remains fundamentally based on ECDLP and therefore shares the same quantum vulnerability. This threat has motivated the development and standardization of PQC algorithms to resist attacks from both classical and quantum computers. At present, no comprehensive benchmark has evaluated Kyber, RSA, and EC-KEM under consistent conditions on the Raspberry Pi 5. Furthermore, there is a lack of comprehensive examination of execution-time variability, which is crucial for latency-sensitive embedded systems.

1.2 Post-Quantum Cryptography and CRYSTALS-Kyber

To mitigate attacks from quantum algorithms, NIST selected PQC algorithms as cryptographic primitives resistant to both classical and quantum computers. Then, Kyber was selected and standardized as ML-KEM in accordance with FIPS 203. This standardization marks a pivotal transition in modern cryptographic infrastructure. ML-KEM is designed to replace conventional key generation techniques that are vulnerable to Shor's algorithm. The security level is based on the Module Learning with Errors (Module-LWE) problem. In addition, Module-LWE has been considered computationally difficult for both traditional and quantum attackers (Guo et al., 2021; Bisheh-Niasar et al., 2021; Jia et al., 2022; Jia et al., 2023). At present, three standardized parameter sets are defined for ML-KEM: ML-KEM-512, ML-KEM-768, and ML-KEM-1024. These parameter sets operate within polynomial rings with predetermined parameters, facilitating fast implementation while preserving robust security properties and enabling a range of security-performance trade-offs. Kyber is fundamentally designed for efficiency and side-channel awareness and is supported by reputable open-source libraries such as Open Quantum Safe's liboqs. These libraries provide standardized APIs and benchmarking tools to enable repeatable evaluation of performance on embedded devices. Recent studies demonstrate Kyber's viability on resource-constrained platforms (Mighri et al., 2024; Patterson et al., 2025), but comprehensive evaluation on modern ARM-based systems, such as

the Raspberry Pi 5, remains limited, motivating the detailed assessment presented in this paper.

1.3 Literature Review

1.3.1 PQC Benchmarking on Embedded Platforms

Several studies have examined PQC algorithms on constrained devices (Dong & Wang, 2025). For example, Mighri et al. (2024) benchmarked Kyber on the Raspberry Pi 3 and ESP32. The results showed that Kyber achieves lower execution times than ECC while maintaining acceptable memory and energy usage on low-power platforms. In addition, these findings indicate that Kyber can be implemented efficiently on earlier-generation embedded hardware under resource constraints. However, previous studies have not evaluated Kyber, RSA, and EC-KEM under consistent experimental conditions on the Raspberry Pi 5. Furthermore, none of these studies has conducted a thorough variability analysis using parameters such as standard deviation, p90, and p99 latencies.

1.3.2 Kyber Performance Studies

At present, Kyber's performance has been examined in many studies. In 2024, Fitzgibbon and Ottaviani (2024) defined computation time as a metric for PQC on constrained devices. Furthermore, Huang et al. (2020) and Bisheh-Niasar et al. (2021) demonstrated that hardware-oriented solutions may provide significant performance improvements on some systems. These studies indicate that Kyber's performance may be significantly enhanced by certain implementation decisions and that latency is an important factor for determining the appropriateness of PQC for limited systems. However, current research lacks a fully software-oriented, platform-specific evaluation of Kyber, RSA, and EC-KEM on the Raspberry Pi 5. Current studies are also deficient in providing an effective structure for defining comprehensive timing distributions and memory utilization under embedded-system constraints.

1.3.3 Cryptographic Evaluation on ARM-based Devices

Many PQC algorithms have been evaluated on embedded devices with ARM platforms. In 2024, Jati et al. (2024) developed a configurable hardware implementation of Kyber with side-channel protection for embedded systems. In addition, Sanal et al. (2021) demonstrated the optimization of Kyber on 64-bit ARM Cortex-A processors to establish baseline performance metrics for ARM platforms. Furthermore, Sisinni et al. (2018) emphasized the trade-offs between limited resources and real-time requirements in IoT deployments. These works demonstrate the feasibility of PQC on ARM and IoT devices. Nevertheless, current studies lack platform-specific benchmarking of Kyber, RSA, and EC-KEM on the Raspberry Pi 5 with execution-time variability metrics for latency-sensitive design. However, the extant literature addresses earlier-generation embedded platforms such as the Raspberry Pi 3 and ESP32 or specialized hardware implementations. The Raspberry Pi 5 is equipped with the ARM Cortex-A76 CPU, which represents a substantial upgrade from the Cortex-A53 and Cortex-A72 cores used in earlier models. This platform therefore defines a new performance context for Kyber that has not yet been well described. In addition, previous studies provide only limited analysis of execution-time variability, such as standard deviation and high-percentile latencies (p90, p99), even though these measures are important for latency-sensitive embedded use. At present, no published work has systematically compared Kyber, RSA, and EC-KEM under identical experimental conditions on a modern ARM-based platform.

The insufficient characterization of modern hardware and the lack of variability-focused analysis motivate this work. This study provides an extensive and replicable benchmark of Kyber on the Raspberry Pi 5 to inform embedded system decisions. Based on the gaps identified in the literature, this section presents the benchmarking results for Kyber, RSA, and EC-KEM on Raspberry Pi 5, based on the measurement protocols outlined in Section 3, while emphasizing quantitative trade-offs rather than assertions of algorithmic superiority.

2. Objectives

This study focuses on addressing the limitation in the practical evaluation of Kyber on the Raspberry Pi 5. Furthermore, it aims to measure the execution time, memory usage, and key and ciphertext overhead of Kyber-512, Kyber-768, and Kyber-1024 in accordance with the security–performance trade-offs defined in NIST FIPS 203. For comparison with Kyber, RSA and EC-KEM schemes with similarly structured encapsulation or encryption mechanisms were selected. Moreover, the objective is to determine whether the architectural improvements of the Raspberry Pi 5 translate into measurable performance gains and stability for deploying PQC. Therefore, this study can evaluate the practical feasibility of Kyber for IoT and edge-computing systems operating under resource constraints.

3. Materials and Methods

This section presents the materials and the proposed method for the implementation. To ensure alignment with the benchmarking methodology, this section details how each performance metric was measured under controlled and repeatable conditions. Every reported value can be traced to a clearly defined experimental procedure. The implementation shown in Figure 1 outlines each phase of the performance evaluation. The method includes the sequence of environment configuration, benchmark execution, data collection, and result aggregation for Kyber, RSA, and EC-KEM on the Raspberry Pi 5 platform. These stages were established to clarify the desired outcomes, methodologies, and experimental limitations uniformly applied to all algorithms, without extending any cryptographic security claims beyond the established analytical parameters.

3.1 Experimental Setup and Procedure

To evaluate the performance of each cryptographic algorithm on a contemporary embedded ARM-based platform, all experiments were conducted on a Raspberry Pi 5. This system was configured with Ubuntu 24.04 64-bit and executed using Python 3.11. In addition, benchmarking was performed using a custom Python script, pqcbenchmsmulti.py developed to support reproducible measurement through controlled iteration counts, warm-up phases, and parallel execution across CPU cores. The script coordinated 1,000 measurement iterations per configuration following a brief warm-up phase, logged all timing and memory samples, and output the aggregated statistics used in Tables 1–4 directly from the raw data, thereby preserving full traceability between the implementation and the reported results.

The experimental software stack comprised liboqs version 0.14.0 for PQC, the build and toolchain environment on the Raspberry Pi 5 used cmake 3.28.3-1build7, gcc 4:13.2.0-7ubuntu1, g++ 4:13.2.0-7ubuntu1, ninja-build 1.11.1-2, git 1:2.43.0-1ubuntu7.3, and libssl-dev 3.0.13-0ubuntu3.6, providing a fully specified compilation context for reproducing the experiments, cryptography library version 43.0.0 for classical cryptographic primitives, including RSA and ECC operations, and the Python runtime and operating system components reported automatically at execution time. Moreover, to utilize the multi-core architecture of the Raspberry Pi 5, the benchmarking framework employed Python’s multiprocessing module with 4 worker processes, because this matched the number of physical CPU cores (quad-core ARM Cortex-A76 at 2.4 GHz). This configuration maximizes CPU utilization while avoiding context-switching overhead from oversubscription. Each worker process functioned as an individual entity performing identical cryptographic tasks, with the aggregate of 1,000 measurement iterations evenly allocated among all workers (250 iterations per core). In this model, each worker executed key generation, encapsulation or encryption, and decapsulation or decryption processes, with the outcome per-operation latencies regarded as independent observations applied to calculate the average, standard deviation, and percentile metrics presented in the Results section. This parallel execution model reflects multi-core behavior in embedded systems while maintaining process segregation for reliable measurement. Furthermore, CPU was not explicitly configured, allowing the Linux scheduler to allocate workers among cores organically according to the normal scheduling parameters of Ubuntu 24.04. A preliminary phase of 100 repetitions (25 per worker) was conducted prior to data collection to stabilize CPU frequency scaling and cache behavior, with these warm-up measurements omitted from all published findings. All benchmark processes operated at the default priority, and real-time scheduling rules were not implemented. Thus, actual embedded-system conditions are maintained.

Additionally, for every experimental test, information about the execution environment, operating system, CPU architecture, hostname, and Python version was automatically recorded to facilitate traceability and replication. No manual tuning of operating system parameters was applied beyond the default Ubuntu configuration; this preserves a realistic embedded-software environment while still allowing other researchers to reproduce the benchmark by following the documented software versions and configuration details.

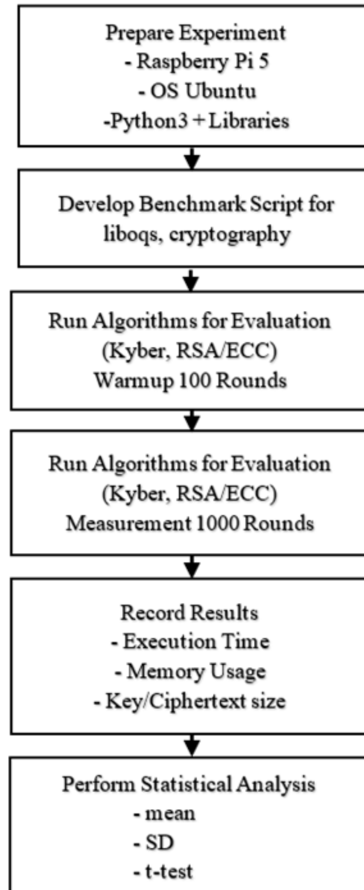


Figure 1 Flowchart of the performance-evaluation process for Kyber compared with RSA and EC-KEM on the Raspberry Pi 5 platform

3.2 Algorithms & Security Levels

In this section, the evaluation focuses on the efficacy of Kyber. In addition, it is compared with RSA and ECC. Furthermore, all cryptographic algorithms were evaluated under uniform experimental settings to provide consistent performance comparisons. Within this benchmarking scope, the selected algorithms are considered as representatives of lattice-based, integer-factorization-based, and elliptic-curve-based public-key schemes. The analysis focuses only on execution time, memory usage, and data size measured on the Raspberry Pi 5.

3.2.1 CRYSTALS-Kyber

To implement Kyber using Python with liboqs version 0.14.0, a Python library was chosen for the implementation process. In addition, all standard parameters were selected for the evaluation, including Kyber-512, Kyber-768, and Kyber-1024. The benchmark evaluated the standard KEM operations specified by the library interface for each configuration: key generation, encapsulation, and decapsulation processes. These three parameters correspond to the NIST-standardized ML-KEM-512, ML-KEM-768, and ML-KEM-1024 security levels. This allows the analysis of how performance scales as key and ciphertext sizes increase under a consistent implementation and hardware environment. These parameters differ in key and ciphertext dimensions, along with computational expense, allowing the examination of performance scaling across configurations.

3.2.2 RSA

In this paper, RSA was evaluated as a classical public key cryptography baseline. The cryptography library (version 43.0.0), with OAEP padding and SHA-256, was selected for the implementation. In addition, two different key sizes were selected for the implementation, including RSA-1024 and RSA-2048.

These key sizes were chosen because they are widely used reference points in embedded-system studies and offer a practical comparison to the three Kyber parameter sets, while remaining computationally feasible for repeated benchmarking on the Raspberry Pi 5. The benchmark analyzed key generation, encryption, and decryption processes for each key size. RSA is considered as a classical baseline under identical hardware and execution circumstances.

3.2.3 Elliptic Curve–Based KEM (EC-KEM)

Elliptic Curve–based Key Encapsulation Mechanism (EC-KEM) were evaluated by using a constructed KEM design based on ECC. In this construction, EC-KEM was implemented using ephemeral–static elliptic curve Diffie Hellman (ECDH) combined with a key derivation function, where the ephemeral public key was transmitted as the ciphertext and the shared secret was derived by both parties. The following ECC configurations were evaluated: X25519, secp256r1, secp384r1, and secp521r1. For each EC-KEM configuration, the benchmark measured key generation, encapsulation, and decapsulation operations. This constructed EC-KEM was used only as a performance baseline to reflect ECC-style key exchange behavior within the same benchmarking framework and was not intended to define or propose a new standardized KEM primitive. This construction was performed for performance benchmarking and comparative evaluation and does not represent a standardized KEM specification.

3.2.4 Interpretation Scope

The evaluated algorithms employ various cryptographic interfaces and assumptions. Performance outcomes are therefore analyzed in terms of execution time, data size, and resource utilization for the specified processes, without assuming complete equality across algorithm families. Accordingly, the comparisons reported in the Results and Discussion sections are limited to the measured trade-offs observed on the Raspberry Pi 5 and should not be interpreted as general claims about algorithmic superiority, security strength, or deployment readiness beyond the conditions examined in this study.

3.3 Measurement Methodology

Performance was assessed using a custom benchmarking script designed to measure execution time, memory usage, and data size under controlled and repeatable conditions. In line with reproducibility requirements, all measurement procedures were documented so that another researcher can determine how each reported metric was calculated using only this section and the software versions specified in Section 3.1.

3.3.1 Operation Definition and Execution

Performance was measured for individual cryptographic operations. The tasks differed by algorithm family and comprised key generation, encapsulation for Kyber and EC-KEM, and encryption for RSA, along with the corresponding decapsulation or decryption steps. Each task was run independently, with no reuse of state or intermediate values between iterations. A brief warm-up period preceded data collection to limit initialization effects and short-term runtime fluctuations, and all reported results excluded these warm-up runs. For each algorithm-operation pair, the script executed exactly 1,000 measurement iterations in total across all worker processes to ensure that the latency statistics in Tables 1 and 4 were computed from an identical sample size under every configuration.

3.3.2 Timing Measurement

The computation time was measured using the time.perf_counter() function, which provides a monotonic high-resolution clock appropriate for performance evaluation. The execution time, measured in milliseconds, indicates the overall duration of a single cryptographic process from initiation to completion. Each experimental configuration was executed for a predetermined number of iterations. The total iterations for parallel processes were equally allocated across worker processes. The timing information gathered from all workers was combined and evaluated as separate observations. From this aggregated sample of 1,000 observations per configuration, the script computed the minimum, maximum, mean, median, standard deviation, and the 90th and 99th percentiles directly, without applying smoothing, outlier removal, or any additional post-processing beyond standard statistical formulas. The presented metrics include the minimum, maximum, mean, median, standard deviation, and the 90th and 99th percentiles. All time metrics are shown as operation latency during concurrent execution rather than total throughput.

3.3.3 Parallel Execution Model

Benchmarks were conducted using multiprocessing module in Python. Each worker process executed identical cryptographic operations

separately, without exchanging any internal state. Timing information from all workers was then aggregated for statistical analysis. This model measures the latency of each operation under realistic multi-core use on the Raspberry Pi 5 while keeping processes isolated. The timing distributions reported in the Results section therefore reflect both variation within each core and scheduling effects that are relevant to embedded deployment.

3.3.4 Memory Measurement

Memory consumption was analyzed using the RSS of each worker process during execution. RSS values were retrieved through the psutil library, with the Linux /proc interface used as a fallback when necessary. RSS values were collected after each benchmark run and averaged across all workers. For each setup, the script took several RSS samples from each worker across 1,000 time cycles and then calculated the mathematical average of all samples across all workers. The averaged data shown in Table 2 provide the foundation for the subsequent ANOVA analysis. The reported figures therefore represent the typical process-level memory usage associated with each cryptographic operation.

3.3.5 Data Size Measurement

In this paper, the sizes of the key and ciphertext were measured in bytes. The dimensions of public keys, private keys, and ciphertexts for Kyber and EC-KEM were obtained directly from the serialized outputs used in the benchmark. In EC-KEM, public keys and ciphertexts were represented as raw elliptic-curve points. The lengths of the key and ciphertext for RSA were derived from the serialized objects generated by the cryptography library. Therefore, all size values presented in Table 3 are derived from the actual byte strings produced in a uniform software environment, rather than from theoretical parameter specifications.

3.3.6 Result Aggregation

For each algorithm and operation, timing and memory samples from all iterations and worker processes were aggregated. Statistical summaries were computed directly from the aggregated dataset, with no outlier removal or additional post-processing. Percentile values (p90 and p99) were computed from the aggregated timing samples without smoothing or filtering. This collection method generated an equivalent dataset for each configuration that supports all descriptive statistics and subsequent hypothesis testing. It provides a direct relationship between the numbers shown in the Results section and the raw data obtained from the embedded device equipped with the Raspberry Pi 5.

3.4 Environment Control

All evaluations in this section were performed in a controlled setting to reduce external sources of variation unrelated to the encryption techniques. No supplementary user applications were active during the measurements, and the system was dedicated to benchmark activities. Furthermore, Raspberry Pi 5 operated with the default power-management and CPU frequency-scaling rules of Ubuntu 24.04 to provide a software environment without manual modifications that may affect repeatability. The operating-system settings, power setup, and hardware conditions were the same in every experiment. A short warm-up period was implemented before data collection to stabilize performance characteristics. The system was monitored to prevent overheating and unexpected configuration changes. In general, important environmental information, including the operating-system version, CPU architecture, Python runtime, and toolchain versions, was documented to allow independent replication of the findings.

4. Results

The aim of this section is to present the benchmarking outcomes for Kyber, RSA, and EC-KEM implemented on the Raspberry Pi 5 in relation to the measurement techniques specified in Section 3. All results are derived from the measurement procedures described in Section 3 and are presented without post-hoc normalization or outlier removal. The analysis examines how execution time, memory usage, key and ciphertext sizes, and execution-time variability together define the performance profiles of the evaluated schemes on this embedded platform. This allows later sections to distinguish between confirmed behavior and effects that are specific to the Raspberry Pi 5. In addition, to support the interpretation in Section 5, each subsection first summarizes the core numerical findings and then discusses their implications for latency–memory–bandwidth trade-offs and stability versus peak performance in embedded-system design.

4.1 Execution Time

Execution time was measured for three cryptographic operations: key generation, encapsulation or encryption, and decapsulation or decryption. For each algorithm and operation, the latency values

reported in Table 1 represent the arithmetic mean per operation, computed from 1,000 measurement iterations following a warm-up phase. Under parallel execution, timing samples from all worker processes were aggregated and treated as independent observations. Figure 2 presents the mean execution time on a logarithmic scale to facilitate comparison across algorithms with substantially different latency ranges.

For all Kyber parameter sets, key generation, encapsulation, and decapsulation execute within the sub-millisecond range. Mean execution time increases gradually from Kyber-512 to Kyber-1024 but remains below 0.25 ms for every operation. This supports prior embedded-device studies showing that Kyber achieves very low latency compared with classical public-key schemes, while extending those results to Raspberry Pi 5 under a multi-core benchmarking setup (Mighri et al., 2024; Fitzgibbon & Ottaviani, 2024). In contrast, RSA exhibits much higher latency. RSA-1024 requires around 48–50 ms per operation, while RSA-2048 exceeds 250 ms on average for key generation, encryption, and decryption. These timings are several orders of magnitude higher than those measured for Kyber on the same platform, underscoring a quantitative latency trade-off between lattice-based and integer-factorization-based schemes under identical benchmark conditions rather than a universal performance hierarchy across all systems. EC-KEM schemes occupy an intermediate position between Kyber and RSA. In general, key generation is comparable in speed to Kyber, whereas encapsulation and decapsulation take roughly three to six times longer when X25519 and secp256r1 are considered. For secp384r1 and secp521r1, these operations extend into the multi-millisecond range, reflecting the higher cost of increased curve size. The average execution times for all algorithms and operations are provided in Table 1.

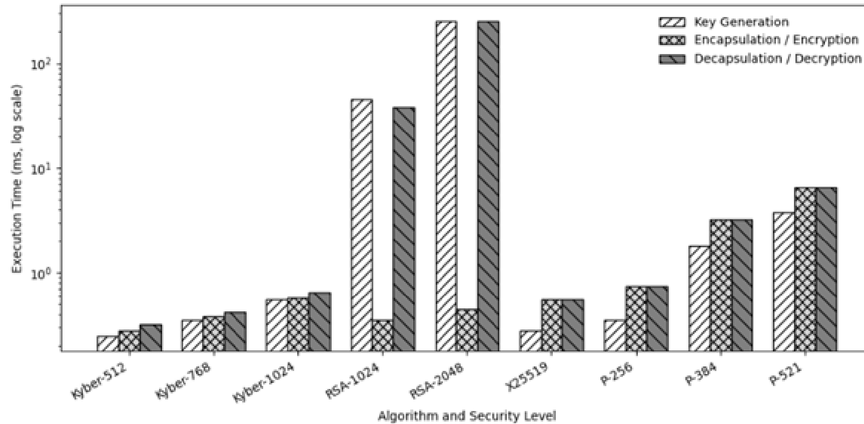


Figure 2 Mean execution time (milliseconds, log scale) for key generation, encapsulation or encryption, and decapsulation or decryption of Kyber, EC-KEM, and RSA on the Raspberry Pi 5

Table 1 Execution Time of Kyber, RSA, and EC-KEM

Algorithm	Key Generation (ms)	Encapsulation or Encrypt (ms)	Decapsulation or Decrypt (ms)
Kyber512	0.0512	0.0848	0.1107
Kyber768	0.0634	0.1154	0.1537
Kyber1024	0.0831	0.1496	0.2094
RSA-1024	48.1992	47.5365	49.4472
RSA-2048	250.9672	252.3937	263.5725
EC-KEM-X25519	0.0574	0.3081	0.2934
EC-KEM-P256 (secp256r1)	0.0504	0.3563	0.3920
EC-KEM-P384 (secp384r1)	0.3607	1.8165	1.9192
EC-KEM-P521 (secp521r1)	0.5916	2.9565	3.0386

These values represent the arithmetic mean per operation, computed from 1,000 measurement iterations aggregated across all worker processes. Pairwise statistical testing further supports the observed execution-time differences. Welch's t-tests indicate statistically significant differences between ML-KEM and RSA-2048 across all operations ($p < 0.001$).

Taken together, the execution-time results reveal a structured latency trade-off: Kyber achieves the lowest latency, EC-KEM configurations occupy a middle range whose cost increases with curve size, and RSA incurs the highest delay on the Raspberry Pi 5. For embedded deployments where CPU time is the dominant constraint and bandwidth is adequate, these measurements motivate selecting Kyber when very short and predictable key-establishment delays are required, while acknowledging that this conclusion is specific to the benchmarked platform and workload.

4.2 Memory Usage

Memory consumption was evaluated using RSS of each worker process during benchmark execution. The results reported in Table 2 represent the average RSS, computed by aggregating memory samples collected across all iterations and worker processes.

Table 2 Average memory Usage (RSS)

Algorithm	Avg RSS (Byte)	Avg RSS (MB)
Kyber512	25,206,590	24.04
Kyber768	25,381,780	24.21
Kyber1024	25,357,530	24.18
RSA-1024	25,322,370	24.15
RSA-2048	25,321,470	24.15
EC-KEM-X25519	26,037,640	24.83
EC-KEM-P256 (secp256r1)	25,762,470	24.57
EC-KEM-P384 (secp384r1)	25,650,520	24.46
EC-KEM-P521 (secp521r1)	25,743,360	24.55

The RSS values range from 24 MB to 24.8 MB. Furthermore, Kyber, RSA, and EC-KEM demonstrate similar memory usage, with slight variations across the techniques. In addition, EC-KEM instances using bigger elliptic curves exhibit somewhat elevated RSS values, although the discrepancies in total memory usage are minimal. Within the constraints of the Raspberry Pi 5 and the software stack described in Section 3, these results indicate that none of the evaluated algorithms is memory-bound for the standalone operations considered in this benchmark; the dominant resource trade-off is therefore between latency and key or ciphertext size, rather than between latency and process-level memory footprint. A one-way ANOVA shows no statistically significant differences in mean RSS across the evaluated algorithms ($p \geq 0.05$), reinforcing the conclusion that the observed execution-time differences are not driven by substantial variations in RSS under the tested conditions.

Given the broadly similar memory usage across schemes in this setting, the next subsection turns to key and ciphertext sizes, which more directly affect bandwidth and storage requirements and thus complete the latency–memory–bandwidth trade-off picture for embedded deployments on Raspberry Pi 5.

4.3 Key and Ciphertext Sizes

Key and ciphertext sizes were obtained from the serialized key and ciphertext objects produced during the benchmarks. The values reported in Table 3 represent the exact byte lengths of public keys, private keys, and ciphertexts for each algorithm.

Table 3 Key and Ciphertext Sizes

Algorithm	Public Key (Byte)	Private Key (Byte)	Ciphertext (Byte)
Kyber512	800	1632	768
Kyber768	1184	2400	1088
Kyber1024	1568	3168	1568
RSA-1024	162	635	128
RSA-2048	294	1219	256
EC-KEM-X25519	32	32	32
EC-KEM-P256 (secp256r1)	91	138	91
EC-KEM-P384 (secp384r1)	120	185	120
EC-KEM-P521 (secp521r1)	158	241	158

Sizes are reported in bytes and correspond to serialized key and ciphertext objects generated during execution.

Kyber requires larger public keys, private keys, and ciphertexts than both RSA and EC-KEM. For example, Kyber-1024 generates public keys and ciphertexts of over 1.5 kB, whereas RSA-2048 and EC-KEM require only a few hundred bytes. In the EC-KEM implementations, the ciphertext includes the serialized public key produced during encapsulation. Therefore, the size of the ciphertext increases with the

parameters of the underlying curve X25519 and secp256r1 provide extremely small outputs, but secp384r1 and secp521r1 result in somewhat larger sizes.

These measurements reveal a concrete trade-off on Raspberry Pi 5 between Kyber's very low latency (Section 4.1) and its larger key and ciphertext sizes, which can increase storage and communication overhead in bandwidth-constrained environments. For applications where communication bandwidth or persistent storage is the primary constraint, the compact keys and ciphertexts of RSA or small-curve EC-KEM may remain attractive despite their higher latency; conversely, when latency and timing predictability dominate, the larger Kyber artifacts may be acceptable given typical embedded memory and link capacities. Despite Kyber's increased bandwidth and storage requirements, the resulting key and ciphertext dimensions remain within acceptable limits for modern embedded and edge-computing contexts on the Raspberry Pi 5.

Because latency and bandwidth alone do not determine suitability for embedded applications, the next subsection examines execution-time variability to assess how predictably each scheme behaves under repeated operation and to distinguish stability from peak performance on the evaluated platform.

4.4 Execution-Time Variability

To examine performance stability, summary statistics were derived from the combined timing samples for each algorithm and operation. The information in Table 4 presents the mean, median, standard deviation, and the 90th and 99th percentiles (p90 and p99), with all values reported in milliseconds.

For Kyber, the median and mean values are closely aligned across all parameter sets and processes, and standard deviations remain small. The percentile results indicate that upper-tail execution durations are close to the average, reflecting steady runtime behavior. On Raspberry Pi 5, this pattern indicates that, within the benchmarked configuration, Kyber operations rarely experience large latency spikes relative to their mean values. In other words, Kyber provides not only low average latency but also high execution-time stability, which is advantageous for timing-sensitive key-establishment routines that must satisfy tight response bounds. RSA has greater variability, particularly RSA-2048, as characterized by a larger standard deviation and a wider gap between percentile values. This trend indicates susceptibility to higher computational demands and intermittent delay surges. The EC-KEM schemes exhibit variable performance. Implementations using smaller curves, such as X25519 and secp256r1, have stable runtimes. However, larger curves, such as secp384r1 and secp521r1, show increased variability, especially during encapsulation and decapsulation processes. In general, these patterns show that as algorithmic complexity and per-operation delay increase, execution-time stability often decreases. This emphasizes the distinction between rapid and reliable methods, such as Kyber, and others whose performance is affected by sporadic prolonged delays, such as RSA-2048 and large-curve EC-KEM. The inferential results align with the descriptive time distributions shown in Table 4 and validate the previously noted variations in variability. These findings should be understood only as distinctions at the measurement level on this platform.

Therefore, the variability analysis builds on earlier studies of PQC (Mighri et al., 2024; Fitzgibbon & Ottaviani, 2024). In addition, mean latencies and full timing distributions (mean, median, standard deviation, p90, and p99) for Kyber, RSA, and EC-KEM under identical conditions on the Raspberry Pi 5 are reported. This allows stability-versus-peak trade-offs to be assessed explicitly when algorithms are chosen for latency-sensitive embedded workloads.

4.5 Summary

The benchmarking results present three primary patterns. First, Kyber operates within the sub-millisecond range (0.05–0.21 ms) with little variation (SD < 0.02 ms). Notably, it is much faster than RSA-2048 (250–264 ms) and typically outpaces most EC-KEM settings. Second, memory usage is comparable across all schemes (24–25 MB RSS, $p > 0.05$). This implies that process-level memory does not differentiate the algorithms under the evaluated circumstances. Third, Kyber's performance advantage is associated with greater key and ciphertext sizes (800–1568 bytes compared to 32–294 bytes). Then, a trade-off between latency and bandwidth in embedded environments is presented. The identified patterns provide the foundation for the deployment-focused study in Section 5.

Table 4 Statistical metrics of execution time (milliseconds) for Kyber, RSA, and EC-KEM based on 1,000 measurement iterations

Algorithm	Operation (ms)	Mean (ms)	Median (ms)	Standard Deviation (ms)	p90 (ms)	p99 (ms)
Kyber512	Key Generation	0.0512	0.0500	0.0066	0.0551	0.0728
	Encapsulation	0.0848	0.0823	0.0082	0.0922	0.1161
	Decapsulation	0.1107	0.1076	0.0142	0.1175	0.1443
Kyber768	Key Generation	0.0634	0.0600	0.0105	0.0681	0.0869
	Encapsulation	0.1154	0.1086	0.0509	0.1227	0.1947
	Decapsulation	0.1537	0.1467	0.0595	0.1594	0.2265
Kyber1024	Key Generation	0.0831	0.0773	0.0844	0.0878	0.1429
	Encapsulation	0.1496	0.1427	0.0689	0.1551	0.2250
	Decapsulation	0.2094	0.2002	0.1043	0.2137	0.2894
RSA-1024	Key Generation	48.1992	48.2000	0.7246	48.8000	49.8000
	Encrypt	47.5365	47.5000	0.5318	48.0000	49.0000
	Decrypt	49.4472	49.4000	0.8123	50.0000	51.0000
RSA-2048	Key Generation	250.9672	251.0000	1.1834	252.0000	255.0000
	Encrypt	252.3979	252.3000	1.1267	253.5000	256.0000
	Decrypt	263.5725	263.5000	1.4729	265.0000	268.0000
EC-KEM-X25519	Key Generation	0.0574	0.0545	0.0234	0.0614	0.0941
	Encapsulation	0.3081	0.2843	0.1853	0.3068	0.5318
	Decapsulation	0.2934	0.2827	0.1283	0.2923	0.3665
EC-KEM-P256 (secp256r1)	Key Generation	0.0504	0.0488	0.0091	0.0544	0.0772
	Encapsulation	0.3563	0.3415	0.1414	0.3587	0.4632
	Decapsulation	0.3920	0.3727	0.1607	0.3998	0.6461
EC-KEM-P384 (secp384r1)	Key Generation	0.3607	0.3555	0.0472	0.3637	0.4342
	Encapsulation	1.8165	1.7694	0.2804	1.8218	3.5208
	Decapsulation	1.9192	1.8497	0.3472	1.9310	3.8036
EC-KEM-P521 (secp521r1)	Key Generation	0.5916	0.5843	0.0953	0.5924	0.6583
	Encapsulation	2.9565	2.8596	0.4597	2.9687	5.2228
	Decapsulation	3.0386	2.9681	0.3459	3.0694	4.9225

5. Discussion

Experimental results demonstrate that Kyber outperforms RSA and EC-KEM in execution speed across all operations tested on the Raspberry Pi 5. This latency advantage is platform-specific and reflects the particular software stack and workload conditions tested; system designers should weigh it against Kyber's increased key and ciphertext sizes when selecting a key-encapsulation mechanism for embedded systems. The difference is particularly apparent in encapsulation and decapsulation, where Kyber reduces latency by almost an order of magnitude compared with RSA-2048, and this pattern appears not only in mean values but also in percentile-based timing distributions that reflect runtime variance. The inferential comparisons (Welch's t-tests) support the latency separation observed in the measured means and percentile metrics. For applications requiring minimal session-setup delays, adopting Kyber on the Raspberry Pi 5 substantially reduces per-operation latency if bandwidth and storage constraints permit.

Although Kyber, as a lattice-based scheme, uses larger keys and ciphertexts, memory consumption during execution remains similar across all algorithms, with average RSS differing only slightly under identical conditions and a one-way ANOVA showing no significant differences in mean RSS ($p > 0.05$). Under this runtime environment, process-level memory usage is dominated by the Python interpreter and shared libraries; consequently, deployment trade-offs center on latency and bandwidth rather than RSS. Kyber's larger key material imposes negligible additional memory cost for the isolated operations benchmarked here. When compared with prior work on Raspberry Pi 3-class devices and other low-power IoT systems, the current results follow a similar qualitative trend in which Kyber appears to be the lowest-latency option among the evaluated public-key mechanisms under their respective experimental setups, even though absolute timing values shift with hardware and implementation changes. These consistencies reflect platform-specific tendencies rather than the universal

superiority of lattice-based schemes across all embedded architectures.

The low and steady latency of Kyber facilitates its use in time-sensitive key setup for embedded devices on this platform. The combination of low mean latency and minimal timing variance positions Kyber as a suitable candidate for time-critical key-establishment on the Raspberry Pi 5, particularly when predictable worst-case delays are required. This study examines standalone cryptographic procedures in a controlled environment; protocol overhead, system integration, and concurrent workloads may affect performance in real-world applications. This evaluation is limited to algorithm-level benchmarks on a single platform and excludes energy consumption and protocol-level analysis, restricting generalization to heterogeneous IoT environments. Future work should incorporate energy measurements, multi-platform evaluations, and protocol-level benchmarking to support broader deployment decisions.

6. Conclusion

When the selected algorithms are implemented on Raspberry Pi 5, the results indicate that Kyber reduces latency compared with RSA and EC-KEM across all evaluated operations. Although Kyber uses larger keys and ciphertexts, its memory consumption is comparable to that of the other algorithms. The findings of this research indicate that Kyber is suitable for latency-sensitive key setup under the evaluated circumstances. Nevertheless, they do not support claims of enhanced security or equivalent performance on other platforms.

The primary deployment trade-offs concern latency and bandwidth rather than memory. The observed performance order aligns with previous research on embedded systems. These results describe behavior on this platform and should not be construed as evidence that post-quantum approaches are prepared to supplant conventional algorithms in all IoT environments.

7. Limitations and Future Work

This work is limited to implementing the selected cryptographic algorithms on a single Raspberry Pi 5 platform and excludes energy measurements, protocol-level evaluation, and concurrent workload analysis, restricting its applicability to diverse embedded environments. Future research incorporating multi-platform comparisons, energy profiling, and protocol-level benchmarking is needed to provide system designers with a comprehensive foundation for deployment decisions.

8. Abbreviations

Abbreviation	Full Term
ECC	Elliptic Curve Cryptography
EC-KEM	Elliptic Curve–based Key Encapsulation Mechanism
ECDH	Elliptic Curve Diffie–Hellman
ECDLP	Elliptic Curve Discrete Logarithm Problem
IFP	Integer Factorization Problem
IoT	Internet of Thing
liboqs	Open Quantum Safe library
ML-KEM	Module-Lattice-Based Key Encapsulation Mechanism
MLWE	Module Learning with Errors
ms	millisecond
PQC	Post-Quantum Cryptography
RSA	Rivest–Shamir–Adleman
p90	90th Percentile
p99	99th Percentile

9. Acknowledgements

This work was supported by Graduate School, Udon Thani Rajabhat University, Udon Thani, 41000, Thailand and Faculty of Technology and Engineering, Udon Thani Rajabhat University, Udon Thani, 41000, Thailand.

10. CRediT Statement

Nattapon Junlachaiworakun: Writing – Original Draft, Data Curation, Software, Methodology, Investigation, Formal Analysis.
Nirundon Panit: Validation, Software, Writing – Review & Editing, Methodology, Investigation.
Kritsanapong Somsuk: Supervision, Project Administration, Validation, Conceptualization, Writing – Review & Editing, Methodology, Formal Analysis, Visualization.

11. References

Bisheh-Niasar, M., Azarderakhsh, R., & Mozaffari-Kermani, M. (2021). Instruction-set accelerated implementation of CRYSTALS-Kyber. *IEEE Transactions on Circuits and Systems I: Regular Papers*, *68*(11), 4648-4659. https://doi.org/10.1109/TCSI.2021.3106639

Boneh, D. (1999). Twenty years of attacks on the RSA cryptosystem. *Notices of the AMS*, *46*(2), 203-213.

Dong, B., & Wang, Q. (2025). Epquic: Efficient post-quantum cryptography for quic-enabled

secure communication [Conference presentation]. *Proceedings of the Great Lakes Symposium on VLSI 2025*, New York, US. https://doi.org/10.1145/3716368.3735199

Fitzgibbon, G., & Ottaviani, C. (2024). Constrained device performance benchmarking with the implementation of post-quantum cryptography. *Cryptography*, *8*(2), Article 21. https://doi.org/10.3390/cryptography8020021

Guo, W., Li, S., & Kong, L. (2021). An efficient implementation of KYBER. *IEEE Transactions on Circuits and Systems II: Express Briefs*, *69*(3), 1562-1566. https://doi.org/10.1109/TCSII.2021.3103184

Hofstede, R., Jonker, M., Sperotto, A., & Pras, A. (2017). Flow-based web application brute-force attack and compromise detection. *Journal of Network and Systems Management*, *25*(4), 735-758. https://doi.org/10.1007/s10922-017-9421-4

Huang, Y., Huang, M., Lei, Z., & Wu, J. (2020). A pure hardware implementation of CRYSTALS-KYBER PQC algorithm through resource reuse. *IEICE Electronics Express*, *17*(17), 20200234-20200234. https://doi.org/10.1587/elex.17.20200234

Jati, A., Gupta, N., Chattopadhyay, A., & Sanadhya, S. K. (2024). A configurable crystals-kyber hardware implementation with side-channel protection. *ACM Transactions on Embedded Computing Systems*, *23*(2), 1-25.

Jia, W., Xue, G., Wang, B., & Hu, Y. (2022). Module-LWE-Based key exchange protocol using error reconciliation mechanism. *Security and Communication Networks*, *2022*(1), Article 8299232. https://doi.org/10.1155/2022/8299232

Jia, W., Zhang, J., & Wang, B. (2023). Hardness of Module-LWE with semiuniform seeds from Module-NTRU. *IET Information Security*, *2023*(1), Article 2969432.

Koblitz, N. (1987). Elliptic curve cryptosystems. *Mathematics of Computation*, *48*(177), 203-209. https://doi.org/10.1090/S0025-5718-1987-0866109-5

Larasati, H. T., & Kim, H. (2021). Quantum cryptanalysis landscape of shor’s algorithm for elliptic curve discrete logarithm problem [Conference presentation]. *International Conference on Information Security Applications*. Springer, Cham. https://doi.org/10.1007/978-3-030-89432-0_8

Mighri, M. A., Benfarah, A., & Meddeb, A. (2024). Performance evaluation and benchmarking of pqc CRYSTALS-Kyber on embedded devices [Conference presentation]. *2024 IEEE/ACS 21st International Conference on Computer Systems and Applications (AICCSA).* IEEE, Sousse, Tunisia. https://doi.org/10.1109/AICCSA63423.2024.10912602

Miller, V. S. (1985). Use of elliptic curves in cryptography [Conference presentation]. *Conference on the theory and application of cryptographic techniques.* Springer Berlin Heidelberg, Berlin, Heidelberg.

Mosca, M. (2018). Cybersecurity in an era with quantum computers: Will we be ready? *IEEE Security & Privacy*, *16*(5), 38–41. https://doi.org/10.1109/MSP.2018.3761723

Mumtaz, M., & Ping, L. (2019). Forty years of attacks on the RSA cryptosystem: A brief survey. *Journal of Discrete Mathematical Sciences and Cryptography*, *22*(1), 9-29. https://doi.org/10.1080/09720529.2018.1564201

Patterson, J. C., Buchanan, W. J., & Turino, C. (2025). Energy consumption framework and analysis of Post-Quantum Key-Generation on embedded devices. *Journal of Cybersecurity and Privacy*, *5*(3), Article 42. https://doi.org/10.3390/jcp5030042

Rivest, R. L., Shamir, A., & Adleman, L. (1978). A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, *21*(2), 120-126. https://doi.org/10.1145/359340.359342

Sanal, P., Karagoz, E., Seo, H., Azarderakhsh, R., & Mozaffari-Kermani, M. (2021). Kyber on ARM64: Compact implementations of Kyber on 64-bit ARM Cortex-A processors. In J. García-Alfaro et al. (Eds.), *Security and privacy in communication networks* (LNICST, Vol. 398, pp. 424–440). Springer. https://doi.org/10.1007/978-3-030-90022-9_23

Shor, P. W. (1994). Algorithms for quantum computation: discrete logarithms and factoring [Conference presentation]. *Proceedings 35th annual symposium on foundations of computer science.* IEEE, Santa Fe, US. https://doi.org/10.1109/SFCS.1994.365700

Sisinni, E., Saifullah, A., Han, S., Jennehag, U., & Gidlund, M. (2018). Industrial internet of things: Challenges, opportunities, and directions. *IEEE Transactions on Industrial*

Informatics, *14*(11), 4724-4734. https://doi.org/10.1109/TII.2018.2852491

Susilo, W., Tonien, J., & Yang, G. (2021). Divide and capture: An improved cryptanalysis of the encryption standard algorithm RSA. *Computer Standards & Interfaces*, *74*, Article 103470.

Van Assen, J., Kromes, R., & Erkin, Z. (2024). Auditable Medical Data Sharing through Recoverable Key Agreement [Conference presentation]. *2024 6th Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS).* IEEE, Berlin, Germany. https://doi.org/10.1109/BRAINS63024.2024.10732363

Wiener, M. J. (1990). Cryptanalysis of short RSA secret exponents. *IEEE Transactions on Information Theory*, *36*(3), 553-558. https://doi.org/10.1109/18.54902