Comprehensive review of state-of-the-art post-quantum cryptographic algorithms

Master's thesis

Authored by Adrian Flutur

Aalborg University
Department of Electronics and IT

AALBORG UNIVERSITY

Department of Electronics and IT
Aalborg University
http://www.aau.dk

Title:
Comprehensive review of state-of-the-art post-quantum cryptographic algorithms

Theme:
Cyber Security

Project Period:
Spring Semester 2024

Project Group:
1010

Participant(s):
Adrian Flutur (aflutu22@student.aau.dk)

Supervisor(s):
Edlira Dushku (edu@es.aau.dk)

Page Numbers: 43

Date of Completion:
May 30, 2024

Contents

Acronyms

KEM Key Encapsulation Mechanism. v, vi, 9, 15–19, 21, 23–30, 38, 40

NIST National Institute of Standards and Technology. vi, vii, 1, 2, 5–10, 13, 14, 16, 17, 38, 40

OQS Open Quantum Safe. 11

PQC Post-Quantum Cryptography. vi, vii, 1, 2, 4–18, 38, 40

SIG Digital Signature. v, vi, 9, 16, 18, 19, 22, 31–37, 39, 40

SoC System-on-a-Chip. 11

VM Virtual Machine. 11, 12

List of Figures

List of Tables

Abstract

In the era of rapid advancements in quantum computing, the field of Post-Quantum Cryptography (PQC) has garnered significant attention as a crucial research area dedicated to developing cryptographic algorithms resilient against both classical and quantum computer attacks. This project presents a review of the state-of-the-art post-quantum cryptographic algorithms, looking into their foundational mathematical problems, design principles, and performance metrics. We analyze a wide array of algorithm families of post-quantum cryptographic algorithms, with a particular emphasis on the algorithms participating in the National Institute of Standards and Technology (NIST) standardization contest. We overview the variations, strengths, weaknesses, and potential real-world applications of each algorithm. Our review not only highlights the algorithms that have already been selected for standardization but also looks at newly proposed algorithms and those that have been deprecated. Furthermore, we benchmark and compare the performance of these algorithms on regular computers and resource-constrained devices, drawing insights from both the latest research findings and our own results. This review serves as a resource for anyone seeking to navigate the complex landscape of post-quantum cryptographic algorithms, helping them choose the most appropriate algorithm for their systems and use cases.

Chapter 1

Introduction

The advent of quantum computing has raised significant concerns about the security of cryptographic systems that rely on the hardness of mathematical problems, such as factorization and discrete logarithms, to protect sensitive information. As quantum computers become increasingly powerful, the security of traditional cryptographic algorithms is facing unprecedented challenges. The potential development of large-scale quantum computers poses a significant threat to the current public-key cryptography infrastructure. The risk of a quantum attack on these systems grows, threatening the confidentiality and integrity of data transmitted over the internet. In response, researchers have been actively exploring the development of Post-Quantum Cryptography (PQC) - a new generation of cryptographic algorithms resistant to attacks from both classical and quantum computers.

The review begins by discussing the fundamental concepts of quantum computing and its impact on cryptography, highlighting the need for post-quantum cryptographic solutions. It then presents the various mathematical models and properties of PQC algorithms. By reviewing the latest research and developments in PQC, this project aims to provide a valuable resource for researchers and specialists seeking to understand the current landscape of post-quantum cryptography.

Furthermore, the review examines the ongoing standardization efforts led by NIST in the United States, which aims to identify and standardize the most suitable post-quantum cryptographic algorithms for future use, when standard cryptography will be discontinued. NIST have initiated a public competition to identify and standardize post-quantum cryptographic algorithms that can resist attacks from both classical and quantum adversaries. This project goes into the realm of post-quantum cryptography from the perspective of NIST's ongoing efforts to evaluate and select quantum-resistant algorithms. By providing an overview of NIST's PQC standardization process, we aim to shed light on the diverse range of candidate algorithms.

1.1 Research contributions

We evaluate each algorithm in terms of its underlying mathematical problem, design principles, and performance metrics. We provide an up-to-date review of the latest NIST PQC standardization process developments, analyzing algorithm performance based on metrics such as key size, execution time of cryptographic operations, and real-world practicality. By examining the new algorithms submitted to the ongoing rounds of the competition, we aim to highlight their potential advantages and drawbacks. Furthermore, we also look over the previously considered algorithms that have been removed from the standardization process due to security vulnerabilities.

While most research on post-quantum cryptographic algorithms today focuses on algorithm performance on regular computers running the x86 architecture, this review also extends the analysis to ARM platforms - widely used in mobile and embedded devices. We benchmark the NIST candidate algorithms on the ARM Cortex-A series processor series, running on Raspberry Pi 4. By assessing the performance of these algorithms on the ARM platform, we provide valuable insights into their suitability for resource-constrained environments and help developers optimize their implementations for better efficiency.

By addressing these particular aspects, we believe this project has the potential to make a significant contribution to the field of post-quantum cryptography. Our review provides a holistic view of the current state of post-quantum cryptographic algorithms and their practical implications, to help future researchers and developers select the best algorithm suited for their quantum computing requirements.

The rest of the project is organized as follows: Section 2 reveals background on quantum computers, PQC, the NIST standardization contest, and related work. In Section 3 we present the methods and steps used for performing the analysis. Section 4 displays the results and their analysis, and Section 5 concludes the project.

Chapter 2

Background

2.1 Post-quantum cryptography

2.1.1 Quantum computers

Quantum computers, a revolutionary advancement in computing technology, leverage the principles of quantum mechanics to perform calculations at an exponentially faster rate compared to traditional computers. While this breakthrough holds immense potential for various fields, it also poses a significant threat to the current state of internet security. Many widely used public-key cryptography algorithms, such as RSA and ECC, which form the backbone of secure online communications, are vulnerable to attacks by quantum computers. As the development of quantum computers progresses rapidly, it is crucial to explore and implement solutions that can withstand their immense computational power. This is where post-quantum cryptography comes into play.

The threat posed by quantum computers to our digital infrastructure may not be such a distant concern. Experts estimate that within the next 10 to 20 years, quantum computers could become powerful enough to crack the encryption algorithms currently in use. This poses a severe risk to the security of sensitive data, financial transactions, and confidential communications. As our reliance on digital technologies continues growing, it is imperative to proactively address this threat and develop robust solutions that can withstand quantum attacks.

As of early 2024, the most powerful quantum computers from giant companies such as IBM, Google, or Intel already have more than 100 qubits. One popular example is IBM's Osprey with 433 qubits [1]. However, these quantum computers are still noisy and incapable of running large-scale quantum algorithms such as Shor's or Grover's algorithms, which could threaten existing cryptographic systems. Shor's algorithm, developed by Peter Shor in 1994, is a quantum algorithm for integer factorization [2]. It is particularly significant because it can factor large integers exponentially faster than the best-known classical algorithms. The algorithm consists of two main parts:

- A quantum part that uses the quantum Fourier transform (QFT) to find the period of a function related to the integer to be factored.
- A classical part that uses the period found in the quantum part to determine the factors of the integer.

The quantum part of the algorithm exploits the superposition and entanglement properties of quantum bits (qubits) to perform the QFT efficiently. This allows the algorithm to find the period of the function in polynomial time, which is exponentially faster than classical methods. The impact of Shor's algorithm lies in its potential to break widely used public-key cryptography systems, such as RSA, which rely on the difficulty of factoring large integers. If a sufficiently large and error-corrected quantum computer is built, Shor's algorithm could render these cryptographic systems insecure.

On the other side, Grover's algorithm (developed by Lov Grover in 1996) is a quantum search algorithm that provides a quadratic speedup over classical search algorithms for unstructured search problems [3]. Grover's algorithm can find the desired element in approximately $\sqrt{N}$ steps, providing a quadratic speedup over classical search algorithms. This speedup is significant for large databases but less dramatic than the exponential speedup provided by Shor's algorithm for factoring.

It is estimated that a quantum computer with several thousand error-corrected qubits would be required to break current public-key cryptography, such as RSA and elliptic curve cryptography. Researchers are working on developing more reliable and scalable quantum computers, while also advancing post-quantum cryptography to ensure the long-term security of digital systems and communications.

2.1.2 Current state of PQC

Post-quantum cryptography refers to a set of cryptographic algorithms designed to be secure against attacks from both classical and quantum computers. These algorithms rely on different mathematical problems that are significantly harder for quantum computers to solve, providing a higher security level than traditional algorithms. PQC explores a range of mathematical frameworks, each with its own unique properties and advantages.

There are several families of post-quantum cryptographic algorithms, each based on different mathematical problems believed to be hard for both classical and quantum computers. The main families are listed below:

- Lattice-based cryptography: These algorithms rely on the difficulty of finding "short vectors" in a high-dimensional lattice. They are based on mathematical problems related to lattices, such as the Learning With Errors (LWE) and the Ring Learning With Errors (RLWE) problems. Examples include CRYSTALS-Kyber, CRYSTALS-Dilithium, and NTRU.

- Code-based cryptography: They utilize error-correcting codes, leveraging the complexity of decoding random linear codes, such as the McEliece cryptosystem and its variants, like the Classic McEliece and BIKE (Bit Flipping Key Encapsulation) schemes.
- Multivariate cryptography: These algorithms are based on the difficulty of solving systems of multivariate polynomial equations over finite fields. Examples of these include the Rainbow algorithm.
- Hash-based cryptography: Hash-based cryptography employs one-way hash functions to offer digital signatures resistant to quantum attacks. They use hash functions to construct digital signature schemes, such as the SPHINCS+ scheme.
- Isogeny-based cryptography: These algorithms are based on the difficulty of finding isogenies between elliptic curves, such as the SIKE (Supersingular Isogeny Key Encapsulation) scheme.

However, at this time there appears to be a limited number of reviews that evaluate the overall performance and effectiveness of various PQC algorithms and schemes.

Dimitrios Sikeridis et al. described the results of their performance analysis on a small subset of PQC algorithms in their study [4], where they showed that the PQC algorithms that had the best performance for time-sensitive applications were Dilithium and Falcon. When the hardware had floating point operations available, Falcon seemed to be a more suitable choice. Any other applications such as VPNs or SSH servers that did not need frequent connection establishments could make use of the other algorithms as well. Furthermore, it was shown that slower signing times can significantly impact the total throughput measured by a server. They expect that signatures and key sizes to have a significant impact on the duration of connection handshakes in real-world protocols.

Manohar Raavi et al. also attempted to analyze the performance of digital signature PQC algorithms in their paper [5]. For key generation, their data showed that Dilithium was between 102 and 32004 times faster than the other algorithms. When analyzing message signing, their results indicated that Dilithium was between 8 and 192 times faster than the other family of algorithms. Furthermore, Dilithium was the fastest for message verification as well, with up to 755 times faster than the other algorithms, but being only 0.01 milliseconds faster than Falcon. They have concluded that Dilithium was the fastest algorithm across all algorithms, having the best execution times and being the most memory efficient.

We believe this project will provide better insights based on a more complete performance analysis of NIST PQC algorithms. We benchmark each available algorithm and we discuss their strengths, drawbacks, and practicality.

2.2 NIST standardization contest

2.2.1 Current state of the standardization process

Recognizing the need for standardized post-quantum cryptographic algorithms, various organizations are actively involved in the evaluation and selection process. National Institute of Standards and Technology (NIST) is leading a crucial effort to standardize PQC algorithms [6]. Through a rigorous process of evaluation and testing, NIST aims to identify and standardize a set of algorithms that can be widely adopted and trusted. The European Telecommunications Standards Institute (ETSI) and the European Union Agency for Cybersecurity (ENISA) are also actively engaged in standardization efforts for PQC. These collaborative efforts are essential for ensuring interoperability, security, and trust across different systems and platforms, laying the foundation for a robust and secure global infrastructure.

After three rounds of evaluation and analysis, NIST has selected the first set of algorithms for standardization [7]. The public-key encapsulation mechanism chosen for standardization is CRYSTALS-Kyber, and the digital signature algorithms are CRYSTALS-Dilithium, Falcon, and SPHINCS+. These selections are based on the computational hardness of problems involving structured lattices, except for SPHINCS+, which uses an underlying mechanism based on hash functions.

The PQC standardization process has progressed with a fourth round, focusing on KEMs [8]. The KEMs under consideration are BIKE, Classic McEliece, HQC, and SIKE. Notably, there are no remaining digital signature candidates under consideration, leading NIST to issue a call for additional digital signature proposals. This call for submissions closed on the 1st of June 2023, and on the 17th of July 2023, NIST announced additional Digital Signature candidates for the PQC standardization process [9]. There are already numerous candidates and the process is still ongoing at the time of this writing.

NIST has expressed its gratitude to the community and submission teams for their contributions to the standardization process. They encourage continued participation from those whose schemes were not initially selected, emphasizing the importance of collective efforts in developing future post-quantum standards and selecting robust and secure algorithms to safeguard against the potential vulnerabilities posed by quantum computing advancements.

2.2.2 Discontinued algorithms

Amongst the several algorithms that were considered for the NIST PQC standardization process, some of them didn't make it through the selection criteria due to various reasons, such as being proven insecure after successful cracking attempts by security researchers outside of NIST.

During the NIST PQC standardization contest, the SIKE (Supersingular Isogeny Key Encapsulation) algorithm was found to be insecure [10, 11]. Belgian researchers Wouter

Castryck and Thomas Decru successfully cracked the SIKE algorithm in approximately 62 minutes using a single core on a six-core Intel Xeon CPU E5-2630v2 @ 2.60GHz. Their attack targeted the algorithm's reliance on supersingular isogenies, a mathematical concept that forms the basis of SIKE's security. This finding led NIST to determine that SIKE would not be standardized due to its vulnerability to the attack demonstrated by Castryck and Decru [12]. Following those events, the team behind SIKE added a note to their website to alert potential users of the fact that the algorithm should not be used anymore [13]. The cracking of SIKE highlights the ongoing challenges in developing cryptographic algorithms resistant to quantum computing threats. Despite the setback, it underscores the importance of rigorous testing and analysis in the cryptographic community to identify vulnerabilities early in the development process. This incident serves as a reminder of the dynamic nature of cryptography and the necessity for continuous research and adaptation to maintain the security of digital communications.

Another example of a successful attack on a PQC algorithm has been provided by Ward Beullens, a Post-doc at IBM Research. The Rainbow PQC algorithm was identified as insecure due to a practical key recovery attack published by Beullens [14]. This attack demonstrated that Rainbow could be compromised, leading to concerns about its security in the face of advancing quantum computing capabilities. The implications of this finding suggest that Rainbow is no longer a viable candidate for adoption in cryptographic systems designed to withstand quantum attacks. The attack on Rainbow highlighted the limitations of relying solely on computational complexity as a defense mechanism in cryptography. The size of the public key in the Rainbow signature scheme was another factor contributing to its insecurity. Before Beullens' work, the large size of the public key was already a disadvantage. However, the subsequent attacks necessitated even larger key sizes to maintain security, thus making the algorithm less practical for widespread use. In contrast, other PQC algorithms like Dilithium and Falcon, which rely on hard lattice problems, were not affected by the same vulnerabilities. These algorithms operate under a fundamentally different mathematical concept, offering a potential alternative for secure cryptographic systems in the quantum age. The discovery of the Rainbow algorithm's insecurity reveals the ongoing challenges in developing robust PQC solutions.

Given these considerations, we decided not to include SIKE and Rainbow in our tests. We assumed that their inherent flaws and vulnerabilities make them unsuitable for cryptographic applications in the context of preparing for the quantum era. Cryptographic systems require not just theoretical resistance to quantum attacks but also practical security that can withstand real-world scrutiny and exploitation. The focus shifts towards algorithms that offer stronger theoretical foundations, proven security properties, and practical efficiency, ensuring the integrity and confidentiality of communications in the face of evolving quantum computing capabilities.

2.3 PQC on ARM architectures

The current state of research regarding the performance of NIST PQC algorithms on ARM platforms, specifically Raspberry Pi devices, indicates a mix of challenges and opportunities. The performance of PQC algorithms varies significantly based on the optimization of the algorithm implementations. Optimized implementations tend to perform better across different hardware without causing significant bottlenecks, whereas poorly optimized implementations can lead to performance issues.

This variability suggests that the choice of algorithm and its implementation can have a substantial impact on their performance on ARM platforms like Raspberry Pi, which has also been shown by Sean Zakrajsek in his research [15]. However, they have also concluded that the performance of these algorithms tested on Raspberry Pi 4 was not significantly different compared to the same algorithms tested on Raspberry Pi 3B+. The reason behind this is that they only used the fastest implementations of the respective algorithms, along with the shortest key lengths and small sample sizes. Sean's work also revealed that some PQC algorithms may face compatibility issues when compiled on certain ARM platforms. For instance, the SUPERCOP toolkit, which includes various PQC algorithms, had difficulties compiling on the ARMv6 architecture of the Raspberry Pi Zero W. Three of the algorithms within their study encountered compilation problems on Raspberry Pi devices. This highlights potential barriers to deploying these algorithms on ARM platforms, particularly on older or less powerful models.

In 2024, Thomas E. Carroll et al. explored the possibility of using PQC in electric vehicles in their study [16]. They concluded based on the conducted experiments and assessments that they are confident PQC algorithms could be implemented and perform well in EV hardware.

Furthermore, Kathryn Hines et al. worked on a study where they analyzed the practical power consumption of a limited set of PQC algorithms on multiple devices, including Raspberry Pi 3 and 4 [17]. They demonstrated that it could be feasible to execute post-quantum algorithms on resource-constrained devices and power consumption is not likely to be a concern. It is also mentioned that PQC algorithms consume more power than the classical ciphers such as RSA on desktop environments, but they consume similar amounts of power as the classical ciphers when tested on the Raspberry Pis.

The individual performance of multiple PQC algorithms was also tested on Raspberry Pi 3 by Basel Halak et al. in their research [18]. Their results revealed that Kyber and Dilithium were the most resource-efficient solutions for embedded devices in a client-server scenario. Their results showed that the performance of these algorithms outperformed all other PQC algorithms, including the current schemes that use elliptic curve cryptography. The study has also mentioned that the digital signature scheme SPHINCS+ was revealed to have significant latency and energy costs, making it less suitable for IoT devices.

As NIST progresses towards finalizing its PQC standards, future research should focus

on optimizing the performance of these algorithms on a wider range of hardware such as ARM Cortex-A and Cortex-M series processors. Additionally, exploring the memory usage and other resource constraints of these algorithms on ARM platforms will be crucial for their deployment in resource-constrained environments.

This project aims to provide a more complete performance analysis of NIST PQC algorithms on ARM, for both KEMs and SIGs. While several studies point out performance metrics on specific embedded devices running a small set of algorithms, we try to compare results and metrics gathered from benchmarking all available algorithms using the Raspberry Pi 4 to get a full view of the picture. Lastly, we compare these results with the metrics gathered from the same benchmarks executed on a regular machine, so we can examine and point out each algorithm's strengths and drawbacks when using them on embedded platforms.

Chapter 3

Methodology

We will perform a series of experiments executing cryptographic operations using the NIST PQC algorithms on two distinct platforms: x86 and ARM. These experiments will follow the same setup and steps on both platforms. However, there will be two different sets of operations that will be executed for the KEM and SIG algorithms respectively. The operations for KEM algorithms will be: generating the keys, secret encapsulation, and secret decapsulation. The operations for SIG algorithms will be: generating the keys, secret signing, and secret verification. After gathering the data for all these algorithms, we will compare the results and provide insights based on the key sizes, execution times, and CPU clock cycles.

3.1 Preparing the testing environment

First, it is essential to establish the test machines and environments. The experiments will be conducted on two distinct architectures: x86 and ARM. Although the software configuration is largely similar for both architectures, there are minor differences that warrant discussion. The x86 architecture, known for its widespread use in personal computers and servers, will serve as one of the primary test platforms. Similarly, the ARM architecture, which has gained prominence in mobile and embedded devices, will be employed as the second test platform. While the fundamental software setup process remains consistent with that of the x86 architecture, there are certain considerations specific to ARM that must be taken into account. These may include differences in instruction sets, memory alignment, and optimization techniques.

3.1.1 Hardware configuration

The first step before conducting tests on PQC algorithms is to carefully select the appropriate hardware and architecture for the test machines. To maintain the integrity of the experiments and ensure the reliability of the collected data, it is crucial to document

and standardize the software setup process for both architectures. This includes specifying the operating system versions, software libraries, and configuration settings used. By establishing a consistent and well-defined environment, the experiments can be easily replicated and validated by other researchers. This transparency allows for a thorough understanding of the experimental conditions and aids in the interpretation of the results.

For our testing purposes, we will employ two distinct machines with different architectures and specifications. The first machine will be a Linux Virtual Machine (VM) on x86 architecture running Debian 12. Debian 12 is known for its stability, security, and extensive software repository, which makes it an ideal choice for setting up a controlled test environment. The VM will be hosted on a system equipped with a powerful Quad-core AMD Ryzen 7 5000 Series (x86-64) processor running at a base frequency of 3.2GHz. This processor architecture is commonly found in modern desktop systems today, making it a good standard environment for testing PQC algorithms. The VM will be allocated 4GB of RAM, which should be sufficient for running the algorithms and collecting performance metrics without encountering memory constraints. We will use the 64-bit version of Debian 12 on the VM, as it can take full advantage of the 64-bit architecture and address more memory compared to its 32-bit counterpart.

The second machine in our testing setup will be a Raspberry Pi 4 Model B, a popular single-board computer that features a Quad-core Cortex-A72 (ARM v8) 64-bit System-on-a-Chip (SoC) running at 1.8GHz. This machine represents a more resource-constrained device with an ARM architecture, which is widely used in embedded systems, mobile devices, and IoT applications. The Raspberry Pi 4 will have 2GB of RAM, which is a common configuration for this model and should be adequate for running PQC algorithms without experiencing severe memory limitations. On the Raspberry Pi 4, we will install the 64-bit version of Raspberry Pi OS, the official operating system for Raspberry Pi devices. This version of the OS is optimized for the ARM v8 architecture, ensuring better performance and compatibility compared to the 32-bit version. By testing PQC algorithms on the Raspberry Pi, we can evaluate their performance and behavior on a resource-constrained device with an ARM architecture, which is essential for understanding their applicability in a wide range of real-world scenarios.

3.1.2 Software configuration

To ensure a consistent testing environment across both the virtual machine and the Raspberry Pi, it is crucial to configure them identically, with the same set of tools and dependencies. This step is essential for obtaining accurate and comparable results when conducting experiments on PQC algorithms. For our experiments, we will be utilizing the *liboqs* library, which is an integral part of the Open Quantum Safe (OQS) project [19]. The OQS project aims to develop and promote the use of quantum-resistant cryptographic algorithms, ensuring the security of digital communications in quantum computing.

To begin the setup process, we must first acquire the source code of the liboqs library

from the designated GitHub repository. It is important to ensure that we are fetching the most up-to-date version of the code to guarantee compatibility with the latest PQC algorithms and any recent bug fixes or performance improvements. Once the source code has been obtained, the next step is to install all the necessary dependencies as outlined in the library's installation documentation. These dependencies include specific versions of compilers, libraries, and build tools that are required for the successful compilation and execution of the liboqs library. With the dependencies in place, we can proceed to build the liboqs library from its source code. To accomplish this, we will employ the CMake build file generator, which simplifies the process of creating build files across different platforms and environments. Afterwards, we will utilize the Ninja build system known for its fast and efficient build times to compile the library and generate the necessary binaries and artifacts. We must follow the same build process on both the Linux VM and the Raspberry Pi to maintain consistency in the testing environment. This includes using identical versions of the liboqs library, dependencies, and build tools, as well as any compiler flags or optimization settings that may impact the performance or behavior of the PQC algorithms.

Once the hardware and software components have been properly configured and the liboqs library has been successfully built on both the VM and the Raspberry Pi, we can move forward with conducting PQC experiments. These experiments will involve executing various operations on multiple families of PQC algorithms such as Kyber, Dilithium, SPHINCS+, BIKE, HQL, and more. We will collect performance metrics and resource utilization data and compare the results of the two architectures we have chosen for our tests. By comparing the results obtained from the Linux VM (running on x86 architecture) and the Raspberry Pi (running on ARM architecture), we can gain valuable insights into the performance characteristics and resource requirements of PQC algorithms across different hardware architectures and environments. This information can help with decisions regarding the selection and deployment of PQC algorithms in real-world scenarios, taking into account factors such as processing power and memory constraints. Moreover, the data collected from these experiments can contribute to the ongoing research and development efforts in the field of post-quantum cryptography. Providing empirical evidence of the performance and resource utilization of PQC algorithms on different platforms makes our findings valuable in the refinement and optimization processes of these algorithms, as well as guiding the design of future cryptographic systems.

With the hardware and software components properly configured, we can proceed with running PQC experiments on both the Linux VM and the Raspberry Pi, comparing the performance and resource utilization across the different architectures (x86 and ARM) and environments.

3.2 Review criteria

Mathematical models

We will outline the methodology employed to analyze the mathematical models at the foundation of the NIST PQC algorithms. The primary objective is to gain a basic understanding of the mathematical foundations upon which these algorithms are built. We do not evaluate the security claims or proofs for the PQC algorithms. These will be concluded by NIST during their standardization process.

Empirical analysis and measurements

Empirical analysis and measurements provide valuable insights into the performance, efficiency, and practicality of PQC algorithms. As previously mentioned, we will analyze execution times for key generation, encryption and decryption, and key sizes on both platforms to collect relevant metrics. The empirical analysis of PQC algorithms focuses on evaluating their performance using a testing suite based on the algorithm implementations located in the liboqs library. First, we measure the time these algorithms take to generate a new key pair. This metric represents the efficiency of the key generation process and its suitability for real-time applications. We then evaluate the performance of encryption and decryption functions, measuring the time required to encrypt and decrypt a given amount of data. Finally, we analyze the key sizes required by different PQC algorithms. Small key sizes are desirable for resource conservation and they are also particularly crucial when deploying PQC in resource-constrained environments like embedded devices.

Real world use-cases

This review does not intend to pick a single winner from the entire pool of candidates, but rather it tries to carefully analyze all possibilities and then offer practical advice on what algorithm may be suited for various real-world use cases. We start by benchmarking the algorithms according to the criteria already mentioned above, then we formulate opinions based on the results.

Chapter 4

Results

4.1 Model properties

This section provides an overview of the mathematical models that form the foundation of the PQC algorithms, illustrated in Tables 4.1 and 4.2. We also provide a list of different variants of each algorithm based on their documentation and the documentation available in the *liboqs* library, which can be found in Tables 4.4 and 4.5. In-depth specifications and implementation details for each algorithm can be found in the design papers which has been submitted to NIST, available on their respective websites.

Mathematical models

Algorithm	Mathematical model
BIKE	BIKE is based on the difficulty of decoding random quasi-cyclic codes. Quasi-cyclic codes are a class of error-correcting codes with a specific structure that allows for efficient computations. The security of BIKE relies on the hardness of the Quasi-Cyclic Syndrome Decoding problem, which involves finding the nearest codeword to a given vector in a quasi-cyclic code [20].
Classic McEliece	McEliece is based on the difficulty of decoding random linear codes. The security of McEliece relies on the hardness of the Syndrome Decoding problem, which involves finding the nearest codeword to a given vector in a random linear code. McEliece uses Goppa codes, a class of error-correcting codes, to construct the public key and perform encryption and decryption operations [21].

FrodoKEM	FrodoKEM is based on the hardness of the LWE problem, similar to Kyber. However, FrodoKEM uses a different structure for its keys and ciphertexts, which allows for a more conservative parameter selection and increased security at the cost of larger key sizes and slower operations compared to Kyber [22].
HQC	HQC is based on the difficulty of the Syndrome Decoding problem for Hamming Quasi-Cyclic (HQC) codes. HQC codes are a class of error-correcting codes with a quasi-cyclic structure and a specific distance metric called the Hamming distance. The security of HQC relies on the hardness of finding the nearest codeword to a given vector in an HQC code [23].
Kyber	Kyber is based on the hardness of the LWE and MLWE problems, which involve solving a system of noisy linear equations. The security of Kyber relies on the difficulty of distinguishing between random samples and samples from an LWE distribution. Kyber uses a structured lattice called a module lattice, which allows for smaller key sizes and faster operations compared to standard LWE-based schemes [24].
ML-KEM	ML-KEM is based on the hardness of the MLWE problem, similar to Kyber and CRYSTALS-Dilithium. However, ML-KEM uses a different algebraic structure for its keys and ciphertexts, which allows for a more efficient implementation and smaller key sizes compared to Kyber [25].
NTRU-Prime	NTRU Prime is based on the hardness of the Short Integer Solution (SIS) problem, which involves finding short vectors in a lattice. The algorithm uses polynomials over finite fields and applies operations such as polynomial multiplication and reduction modulo a composite modulus to generate public and private keys [26].

Table 4.1: Mathematical models for KEM PQC algorithms.

Algorithm	Mathematical model
CRYSTALS-Dilithium	CRYSTALS-Dilithium is based on the hardness of the MLWE and SIS problems. The MLWE problem, similar to Kyber, involves solving a system of noisy linear equations over a module lattice. The SIS problem involves finding a short integer solution to a system of linear equations. Dilithium combines these problems to create a secure digital signature scheme [24].
FALCON	FALCON is based on the hardness of the SIS problem over NTRU lattices. NTRU lattices are a special class of lattices with a specific algebraic structure. FALCON uses the properties of NTRU lattices to construct a hash-and-sign digital signature scheme with fast signing and verification operations [27].
ML-DSA	ML-DSA is based on the hardness of the MLWE and SIS problems, similar to CRYSTALS-Dilithium. However, ML-DSA uses a different algebraic structure and signature scheme, which allows for more efficient implementations and faster signing and verification operations compared to Dilithium [28].
SPHINCS+	SPHINCS+ is based on the security of hash functions and their resistance to collision attacks. SPHINCS+ uses a multi-tree structure of hash-based one-time signature schemes, called Winternitz one-time signatures (WOTS), to construct a stateless hash-based signature scheme. The security of SPHINCS+ relies on the collision resistance of the underlying hash functions [29].

Table 4.2: Mathematical models for SIG PQC algorithms.

Algorithms, parameter sets, and key sizes

Tables 4.4 and 4.5 present all the algorithms analyzed in this review, along with their various parameter sets, NIST security levels, public and private key sizes, ciphertext sizes, shared secret sizes (for KEM algorithms), and signature sizes (for SIG algorithms). KEM algorithms use the shared secret when establishing a connection between two parties in cryptographic protocols such as TLS, similar to the way the Diffie-Hellman Key-Exchange algorithm works.

Security levels illustrated in Table 4.3 represent categories created by NIST, where algorithms in each category should be at least as hard to break as their reference classic algorithms such as AES or SHA. NIST has defined five security levels, each corresponding to different security strengths [6].

Security level	Reference classic algorithm
1	AES 128
2	SHA 256
3	AES 192
4	SHA 384
5	AES 256

Table 4.3: NIST security levels. A PQC algorithm being on a specific level should be as hard to break as that level's reference classic algorithm.

Algorithm	Parameter set	NIST level	PK size	SK size	CT size	SS size
BIKE	BIKE-L1	1	1541	5223	1573	32
	BIKE-L3	3	3083	10105	3115	32
	BIKE-L3	5	5122	16494	5154	32
Classic McEliece	Classic-McEliece-348864	1	261120	6492	96	32
	Classic-McEliece-348864f	1	261120	6492	96	32
	Classic-McEliece-460896	3	524160	13608	156	32
	Classic-McEliece-460896f	3	524160	13608	156	32
	Classic-McEliece-6688128	5	1044992	13932	208	32
	Classic-McEliece-6688128f	5	1044992	13932	208	32
	Classic-McEliece-6960119	5	1047319	13948	194	32
	Classic-McEliece-6960119f	5	1047319	13948	194	32
	Classic-McEliece-8192128	5	1357824	14120	208	32
	Classic-McEliece-8192128f	5	1357824	14120	208	32
FrodoKEM	FrodoKEM-640-AES	1	9616	19888	9720	16
	FrodoKEM-640-SHAKE	1	9616	19888	9720	16
	FrodoKEM-976-AES	3	15632	31296	15744	24
	FrodoKEM-976-SHAKE	3	15632	31296	15744	24
	FrodoKEM-1344-AES	5	21520	43088	21632	32
	FrodoKEM-1344-SHAKE	5	21520	43088	21632	32
HQC	HQC-128	1	2249	2305	4433	64
	HQC-192	3	4522	4586	8978	64
	HQC-256	5	7245	7317	14421	64
Kyber	Kyber512	1	800	1632	768	32
	Kyber768	3	1184	2400	1088	32
	Kyber1024	5	1568	3168	1568	32
ML-KEM	ML-KEM-512-ipd	1	800	1632	768	32
	ML-KEM-768-ipd	3	1184	2400	1088	32
	ML-KEM-1024-ipd	5	1568	3168	1568	32
NTRU-Prime	sntrup761	2	1158	1763	1039	32

Table 4.4: The list of all KEM candidate algorithms along with their parameters and specifications. Sizes are expressed in bytes. Abbreviations: PK = public key, SK = secret key, CT = ciphertext, SS = shared secret.

Algorithm	Parameter set	NIST level	PK size	SK size	SIG size
CRYSTALS-Dilithium	Dilithium2	2	1312	2528	2420
	Dilithium3	3	1952	4000	3293
	Dilithium5	5	2592	4864	4595
Falcon	Falcon-512	1	897	1281	752
	Falcon-1024	5	1793	2305	1462
	Falcon-padded-512	1	897	1281	666
	Falcon-padded-1024	5	1793	2305	128
ML-DSA	ML-DSA-44-ipd	2	1312	2560	2420
	ML-DSA-65-ipd	3	1952	4032	3309
	ML-DSA-87-ipd	5	2592	4896	4627
SPHINCS+	SPHINCS+-SHA2-128f-simple	1	32	64	17088
	SPHINCS+-SHA2-128s-simple	1	32	64	7856
	SPHINCS+-SHA2-192f-simple	3	48	96	35664
	SPHINCS+-SHA2-192s-simple	3	48	96	16224
	SPHINCS+-SHA2-256f-simple	5	64	128	49856
	SPHINCS+-SHA2-256s-simple	5	64	128	29792
	SPHINCS+-SHAKE-128f-simple	1	32	64	17088
	SPHINCS+-SHAKE-128s-simple	1	32	64	7856
	SPHINCS+-SHAKE-192f-simple	3	48	96	35664
	SPHINCS+-SHAKE-192s-simple	3	48	96	16224
	SPHINCS+-SHAKE-256f-simple	5	64	128	49856
	SPHINCS+-SHAKE-256s-simple	5	64	128	29792

Table 4.5: The list of all SIG candidate algorithms along with their parameters and specifications. Sizes are expressed in bytes. Abbreviations: PK = public key, SK = secret key, SIG = signature.

4.2 Model performance

The benchmarking algorithm

In this section, we present our proposed methodology for benchmarking the performance of selected algorithms using the *liboqs* library [19]. Our methodology aims to provide a clear understanding of the performance exhibited by the selected algorithms under both x86 and ARM platforms, considering metrics such as computational and runtime costs. We leveraged the liboqs library to ensure we have a standardized implementation and reliable results across different platforms. We used the provided source code and examples to create an abstraction layer over the implementation details located inside the library, so we could adjust runtime parameters and run specific operations directly. The following method was used to measure the performance of the PQC algorithms:

1. Initialize the liboqs library.
2. Select which KEM and SIG algorithms we want to run.

3. Select how many iterations we should use for computing the average results (defaults to 100).
4. Take each selected KEM algorithm, start a timer and a CPU clock counter, and execute the following operations individually: key generation, encapsulation, and decapsulation. The operations are executed by the liboqs underlying implementation code, then the results are gathered by our program.
5. Computing the average times and CPU cycles for the results.
6. Take each selected SIG algorithm, start a timer and a CPU clock counter, and execute the following operations individually: keypair generation, signing, and verification. The operations are executed by the liboqs underlying implementation code, then the results are gathered by our program.
7. Computing the average times and CPU cycles for the results.
8. Print the results.

The above steps provide a high-level overview of the algorithm. These steps ensured that our proposed method captured the essential performance characteristics of KEM algorithms accurately and efficiently. Below we have also provided the actual algorithm represented in pseudocode. The algorithm begins with the **main()** function. We have split KEM and SIG operations into two sub-routines named **kem_execute_and_measure(algorithm, operation)** and **sig_execute_and_measure(algorithm, operation)** for a clearer understanding of the whole process. These sub-routines end up calling functions inside the liboqs library. Essentially, the algorithm is a high-level wrapper over the liboqs implementations, making it easier to benchmark specific algorithms, iterate a specific number of times, provide custom parameters, and extract the results. The algorithm is presented below:

Algorithm 1 Benchmark algorithm - **main()**

Inputs
▷ kem_algs: list of all supported KEM algorithms
▷ sig_algs: list of all supported SIG algorithms
▷ iters: how many iterations to use for benchmarking (default is 100)
Output
▷ kem_results: a hashmap used to store the results for KEM algorithms
▷ sig_results: a hashmap used to store the results for SIG algorithms

for i = 1 to *kem_algs.length* **do**
  $alg \leftarrow algs[i]$
  $keygen\_results \leftarrow \{\}$
  $encaps\_results \leftarrow \{\}$
  $decaps\_results \leftarrow \{\}$

  for j = 1 to *iters* **do**
    $k \leftarrow$ **kem_execute_and_measure**(alg, KEM_KEYGEN)
    $keygen\_results.Add(k)$
    $e \leftarrow$ **kem_execute_and_measure**(alg, KEM_ENCAPS)
    $encaps\_results.Add(e)$
    $d \leftarrow$ **kem_execute_and_measure**(alg, KEM_DECAPS)
    $decaps\_results.Add(d)$
  end for

  $kem\_results[alg] = (AVG(keygen\_results), AVG(encaps\_results), AVG(decaps\_results))$
end for

for i = 1 to *sig_algs.length* **do**
  $alg \leftarrow algs[i]$
  $keypair\_results \leftarrow \{\}$
  $sign\_results \leftarrow \{\}$
  $verify\_results \leftarrow \{\}$

  for j = 1 to *iters* **do**
    $k \leftarrow$ **sig_execute_and_measure**$(alg, SIG\_KEYPAIR)$
    $keygen\_results.Add(k)$
    $e \leftarrow$ **sig_execute_and_measure**(alg, SIG_SIGN)
    $encaps\_results.Add(e)$
    $d \leftarrow$ **sig_execute_and_measure**(alg, SIG_VERIFY)
    $decaps\_results.Add(d)$
  end for

  $sig\_results[alg] = (AVG(keypair\_results), AVG(sign\_results), AVG(verify\_results))$
end for
return (kem_results, sig_results)

Algorithm 2 KEM benchmark algorithm - **kem_execute_and_measure(algorithm, operation)**

Inputs
▷ algorithm: any of the supported KEM algorithms
▷ operation: any of the supported operations.
Can be one of: KEM_KEYGEN, KEM_ENCAPS, KEM_DECAPS
Output
▷ the benchmark results after running the specified operation using the given algorithm, as a tuple (time, cpu_cycles)

$start\_time \leftarrow get\_time()$
$start\_cpu\_cycle\_counter()$

if operation = KEM_KEYGEN **then**
  $execute\_keygen(algorithm)$
else if operation = KEM_ENCAPS **then**
  $execute\_encaps(algorithm)$
else
  $execute\_decaps(algorithm)$
end if

$cpu\_cycles \leftarrow end\_cpu\_cycle\_counter()$
$end\_time \leftarrow get\_time()$
$time \leftarrow end\_time - start\_time$

return (time, cpu_cycles)

Algorithm 3 SIG benchmark algorithm - **sig_execute_and_measure(algorithm, operation)**

Inputs
▷ algorithm: any of the supported SIG algorithms
▷ operation: any of the supported operations.
Can be one of: SIG_KEYPAIR, SIG_SIGN, SIG_VERIFY
Output
▷ the benchmark results after running the specified operation using the given algorithm, as a tuple (time, cpu_cycles)

$start\_time \leftarrow get\_time()$
$start\_cpu\_cycle\_counter()$

if operation = SIG_KEYPAIR **then**
  $execute\_keypair(algorithm)$
else if operation = SIG_SIGN **then**
  $execute\_sign(algorithm)$
else
  $execute\_verify(algorithm)$
end if

$cpu\_cycles \leftarrow end\_cpu\_cycle\_counter()$
$end\_time \leftarrow get\_time()$
$time \leftarrow end\_time - start\_time$

return (time, cpu_cycles)

Results - KEM algorithms

In the following figures, we have listed the results of running the KEM algorithms using the benchmarking program provided above (Figures 4.1 up to 4.6). Then, we listed the CPU cycles that have been observed while running those operations in Tables 4.6-4.7. We displayed the results for all KEM algorithms and operations, for both x86 and ARM architectures. The operations are, in order: key generation (Figures 4.1-4.2), secret encapsulation (Figures 4.3-4.4), and secret decapsulation (Figures 4.5-4.6). The values are presented in microseconds (μs).

Figure 4.1: KEM algorithms keygen time benchmark (x86)

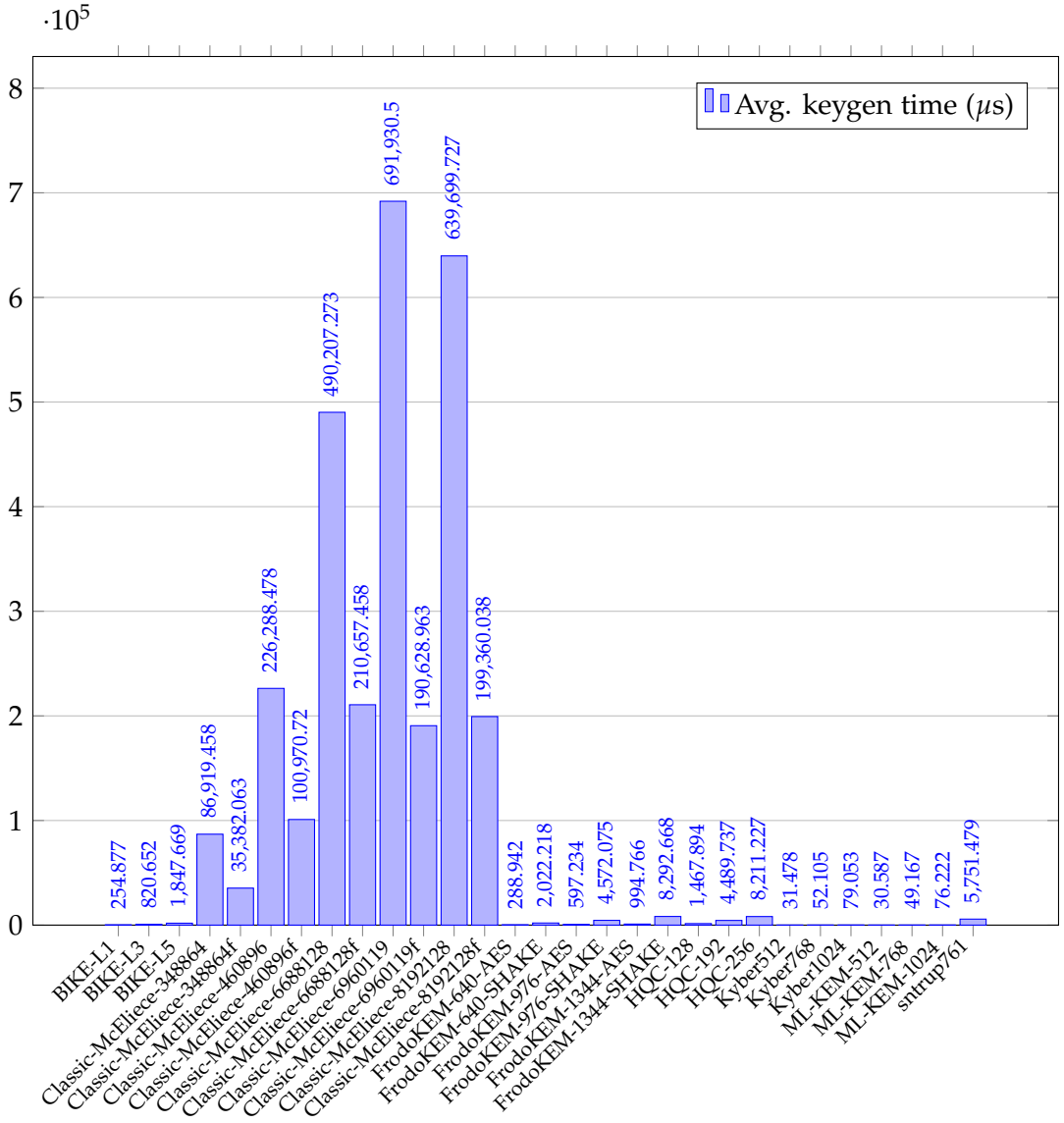
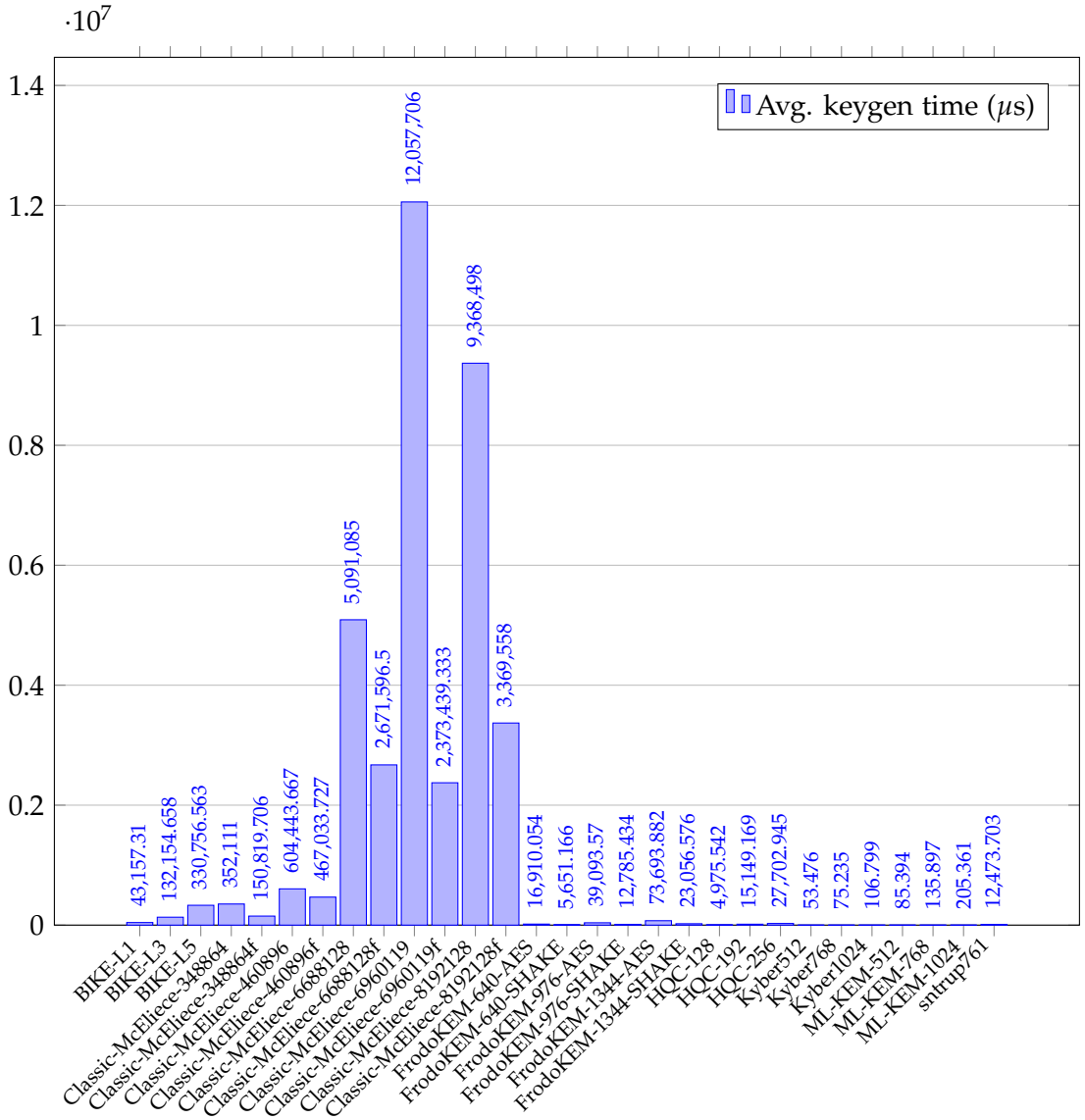


Figure 4.2: KEM algorithms keygen time benchmark (ARM)



As we can directly observe, the key generation times of all variants of Classic McEliece are the highest amongst all other algorithms, on both x86 and ARM platforms. The fastest times on both platforms are represented by the Kyber and ML-KEM algorithms. However, Kyber512 running on ARM was 70% slower compared to x86, while ML-KEM-512 running on ARM reported 180% slower times compared to x86. Surprisingly, BIKE seems to be an astonishing 16500% slower on ARM compared to x86.

Figure 4.3: KEM algorithms encaps time benchmark (x86)

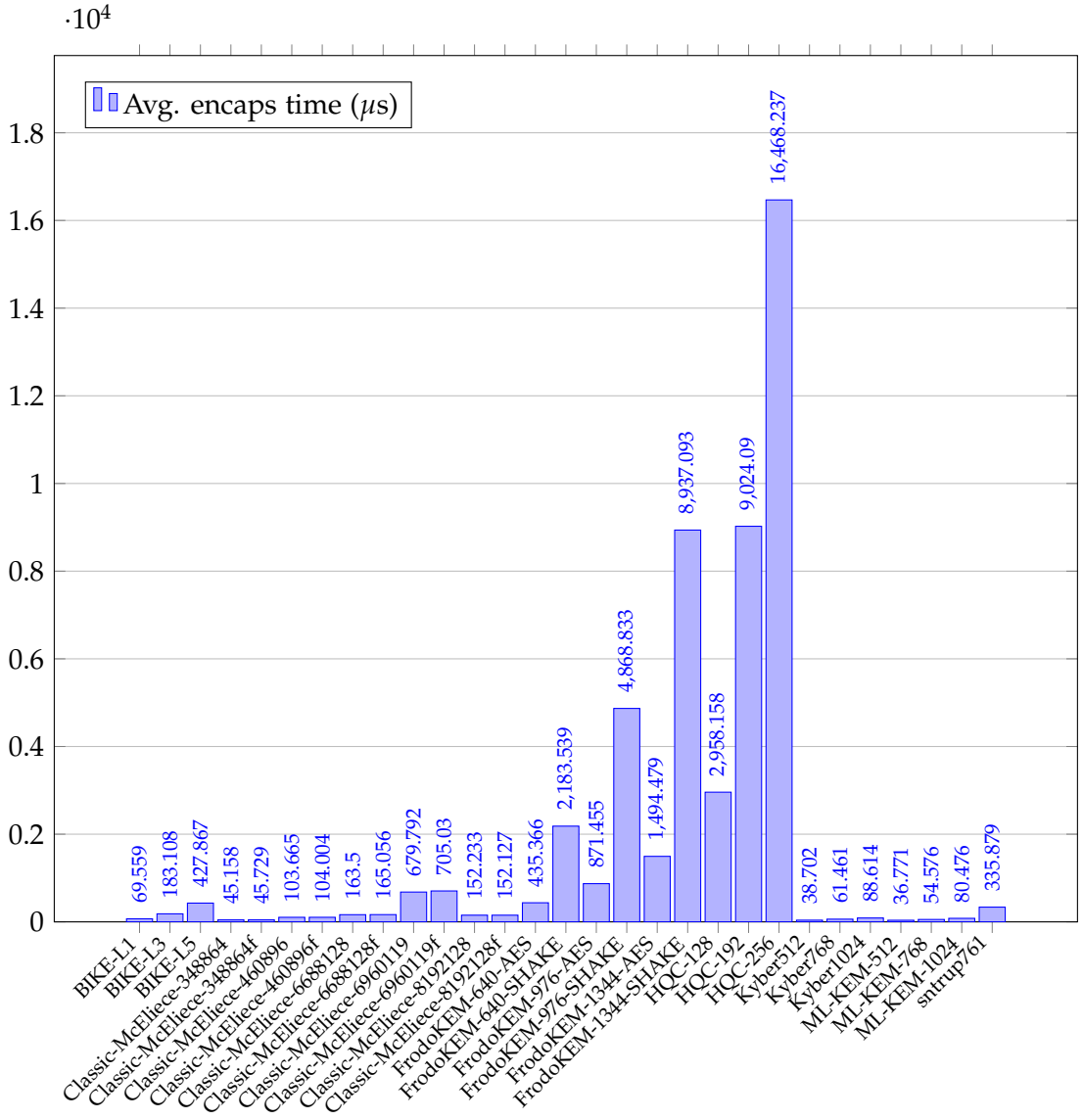
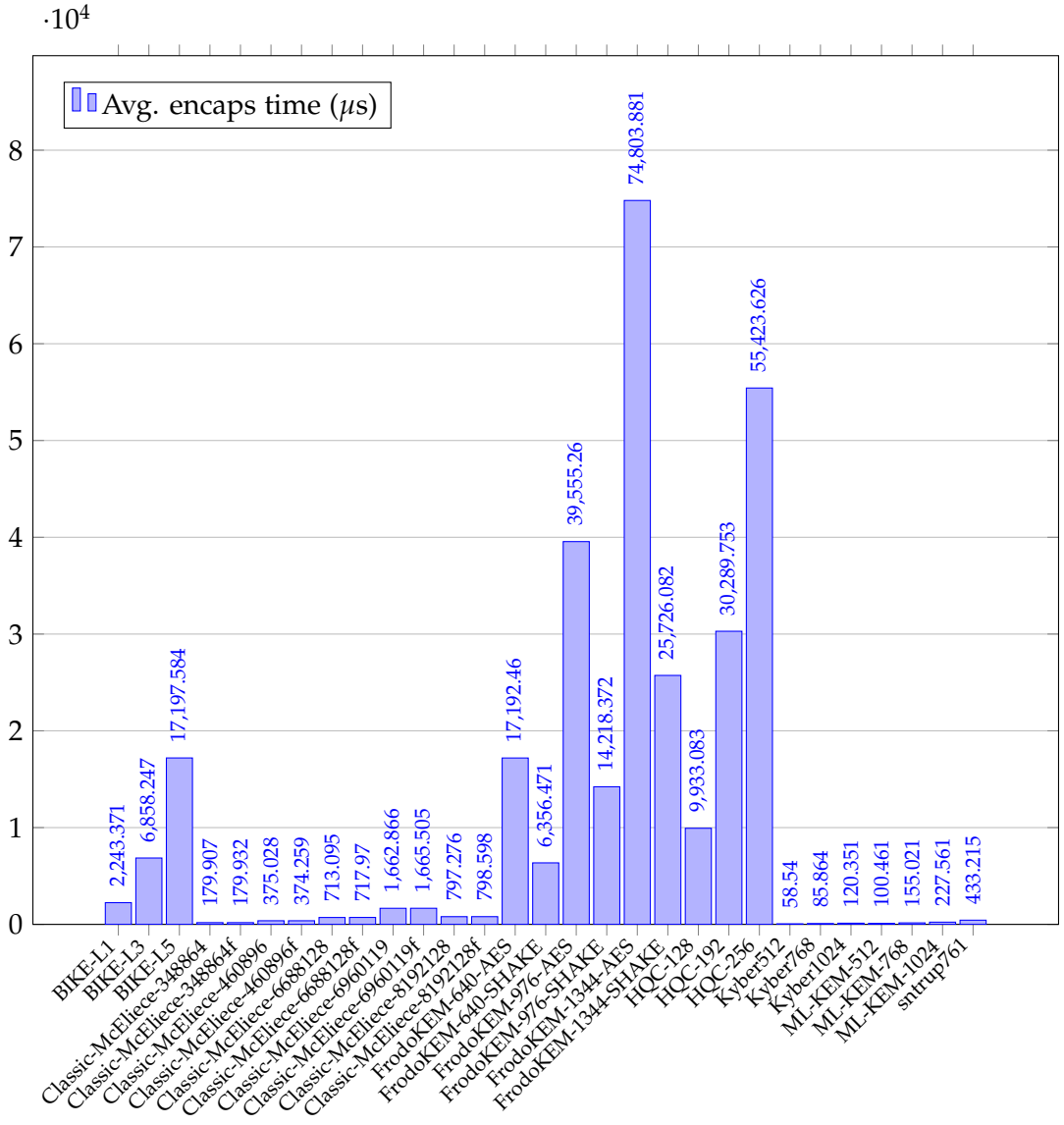


Figure 4.4: KEM algorithms encaps time benchmark (ARM)



For the secret encapsulation operation, FrodoKEM and HQC have reported the slowest times, being several orders of magnitude slower than ML-KEM, Kyber, or Classic McEliece. Contrary to the very large initial key generation times, McEliece has the third fastest encapsulation times. The fastest times on both platforms are represented, again, by the Kyber and ML-KEM algorithms. Just like before, Kyber512 running on ARM is 50% slower compared to x86, while ML-KEM-512 running on ARM is 170% slower compared to x86. Furthermore, FrodoKEM and BIKE are several times slower on ARM as well.

Figure 4.5: KEM algorithms decaps time benchmark (x86)

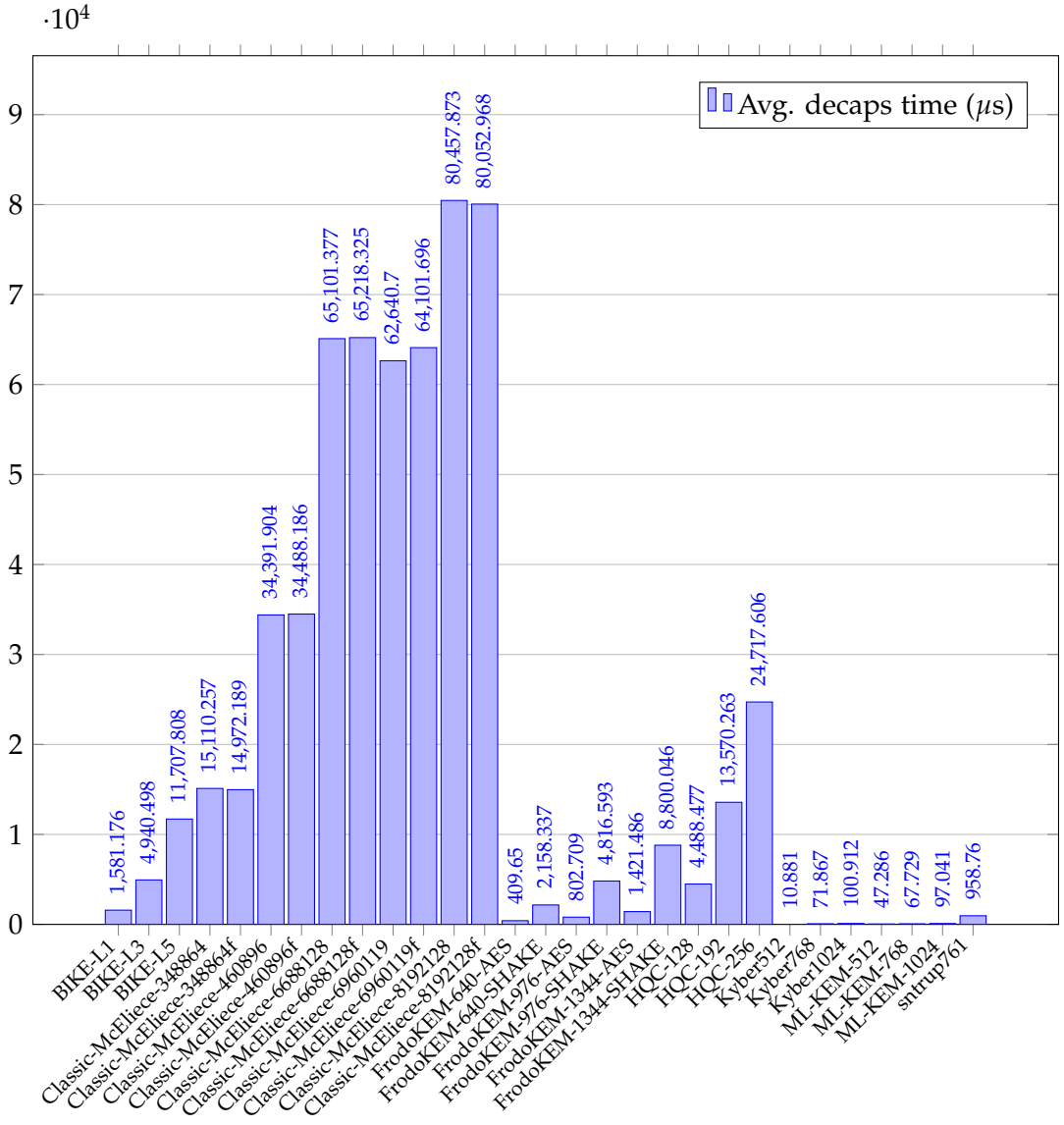
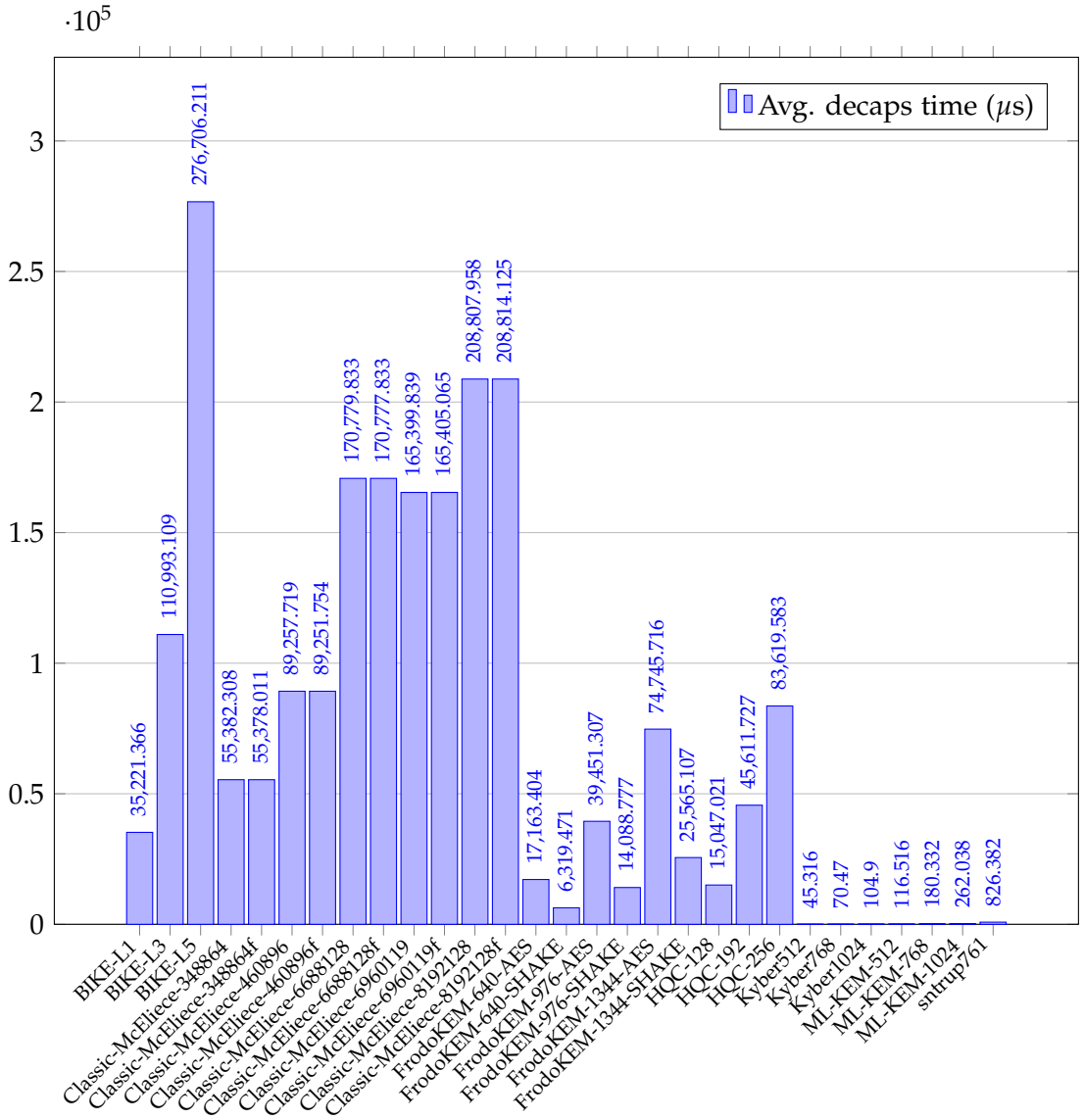


Figure 4.6: KEM algorithms decaps time benchmark (ARM)



As for the decapsulation operation, Classic McEliece, HQC, and BIKE are the slowest among all algorithms, with Kyber and ML-KEM taking the lead as the fastest algorithms for the third time. FrodoKEM and NTRU Prime deliver reasonable times but are overall slower than Kyber or ML-KEM. On x86, Kyber512 running on ARM is 316% slower compared to x86, while ML-KEM-512 running on ARM is 146% slower compared to x86. BIKE is slower on ARM again, this time being about 2100% slower.

Algorithm	Parameter set	Keygen CPU cycles	Encaps CPU cycles	Decaps CPU cycles
BIKE	BIKE-L1	813900	222043	5049663
	BIKE-L3	2620801	584691	15778430
	BIKE-L5	5903899	1367096	37411798
Classic-McEliece	Classic-McEliece-348864	277751053	144205	48284499
	Classic-McEliece-348864f	113062779	146002	47843256
	Classic-McEliece-460896	723105611	331118	109899019
	Classic-McEliece-460896f	322651777	332197	110206402
	Classic-McEliece-6688128	1566461648	522302	208031152
	Classic-McEliece-6688128f	673157103	527296	208405417
	Classic-McEliece-6960119	2211072205	2172064	200168445
	Classic-McEliece-6960119f	609155633	2252693	204837204
	Classic-McEliece-8192128	1653715821	486318	257103456
	Classic-McEliece-8192128f	637055677	485978	255809455
FrodoKEM	FrodoKEM-640-AES	923146	1391033	1308876
	FrodoKEM-640-SHAKE	6461651	6977126	6896618
	FrodoKEM-976-AES	1908273	2784523	2564844
	FrodoKEM-976-SHAKE	14609490	15557806	15390881
	FrodoKEM-1344-AES	3178513	4775324	4542057
	FrodoKEM-1344-SHAKE	26498705	28557939	28120072
HQC	HQC-128	4690398	9452380	14342412
	HQC-192	14346264	28835990	43363265
	HQC-256	26237862	52623777	78984616
Kyber	Kyber512	100495	123578	146947
	Kyber768	166402	196301	229548
	Kyber1024	252506	283057	322345
ML-KEM	ML-KEM-512	97646	117404	151008
	ML-KEM-768	157004	174295	216329
	ML-KEM-1024	243469	257046	309991
NTRU-Prime	sntrup761	18378651	1073166	3063519

Table 4.6: KEM algorithms CPU cycles for all operations (x86)

Algorithm	Parameter set	Keygen CPU cycles	Encaps CPU cycles	Decaps CPU cycles
BIKE	BIKE-L1	43157188	2243245	35221271
	BIKE-L3	132154590	6858141	110992996
	BIKE-L5	330756640	17197480	276706061
Classic-McEliece-348864	Classic-McEliece-348864	352110831	179823	55382157
	Classic-McEliece-348864f	150819742	179842	55377900
	Classic-McEliece-460896	604443221	374939	89257620
	Classic-McEliece-460896f	467033484	374171	89251647
	Classic-McEliece-6688128	5091085312	712856	170779674
	Classic-McEliece-6688128f	2671596288	717669	170777805
	Classic-McEliece-6960119	12057706368	1662518	165399783
	Classic-McEliece-6960119f	2373439147	1665121	165404903
	Classic-McEliece-8192128	9368497408	796964	208807776
	Classic-McEliece-8192128f	3369557376	798264	208813995
FrodoKEM	FrodoKEM-640-AES	16909924	17192346	17163302
	FrodoKEM-640-SHAKE	5651052	6356375	6319373
	FrodoKEM-976-AES	39093476	39555122	39451170
	FrodoKEM-976-SHAKE	12785274	14218248	14088638
	FrodoKEM-1344-AES	73693745	74803777	74745566
	FrodoKEM-1344-SHAKE	23056439	25725917	25564972
HQC	HQC-128	4975410	9932981	15046931
	HQC-192	15149058	30289608	45611601
	HQC-256	27702894	55423494	83619430
Kyber	Kyber512	53376	58428	45229
	Kyber768	75128	85748	70377
	Kyber1024	106696	120239	104810
ML-KEM	ML-KEM-512	85298	100363	116427
	ML-KEM-768	135809	154898	180242
	ML-KEM-1024	205272	227469	261949
NTRU-Prime	sntrup761	12473576	433120	826296

Table 4.7: KEM algorithms CPU cycles for all operations (ARM)

The CPU cycles required for running all operations are also provided for completeness. While CPU cycles might not be very accurate in regards to how many low-level operations the CPU executes within one cycle, they can provide additional profiling data and reassurance that our runtime times above are indeed correctly measured.

Results - SIG algorithms

In the next figures, we have listed the results of running the SIG algorithms using the same benchmarking program (Figures 4.7 up to 4.12). Then, we listed the CPU cycles that have been observed while running those operations in Tables 4.8-4.9. We displayed the results for all SIG algorithms and operations, for both x86 and ARM architectures. The operations are, in order: keypair generation (Figures 4.7-4.8), signing (Figures 4.9-4.10), and verifying the signature (Figures 4.11-4.12). The values are presented in microseconds (μs).

Figure 4.7: SIG algorithms keypair generation time benchmark (x86)

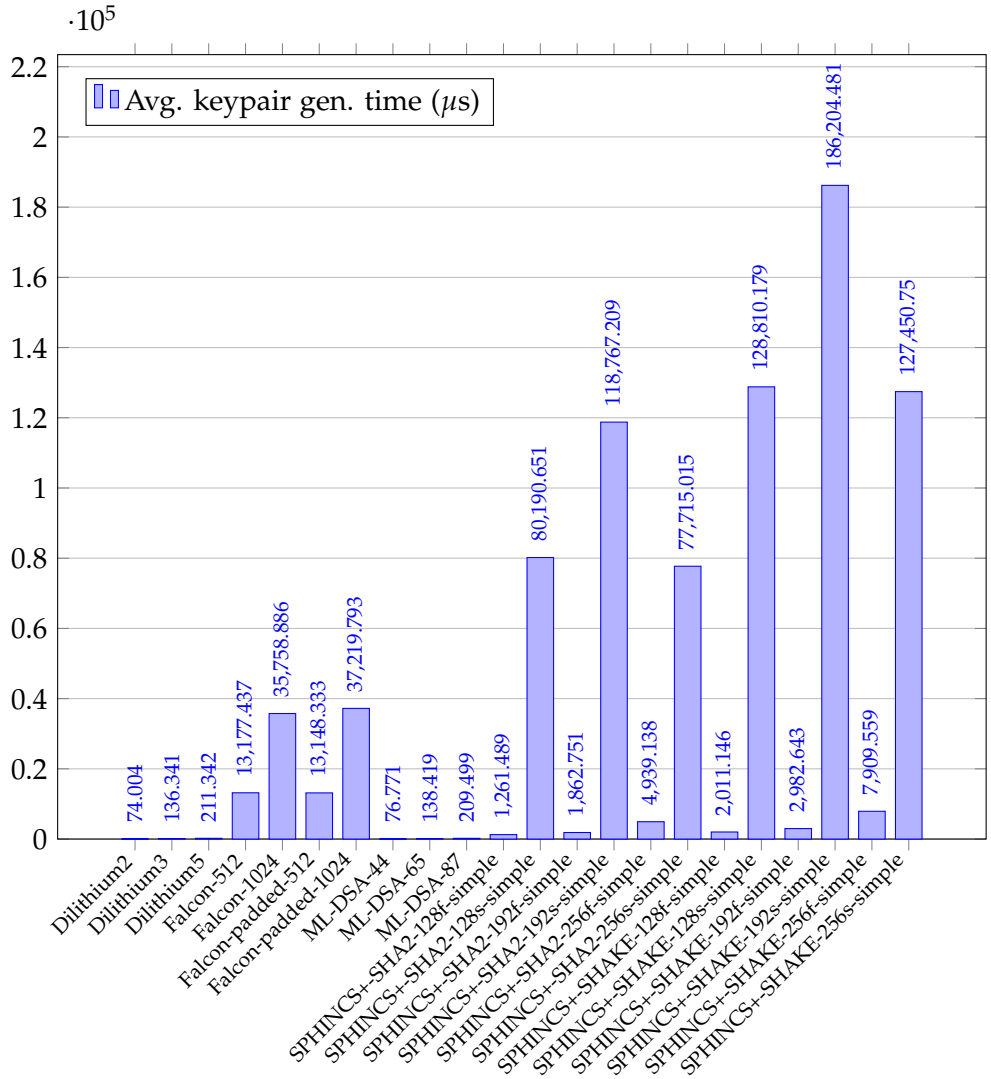
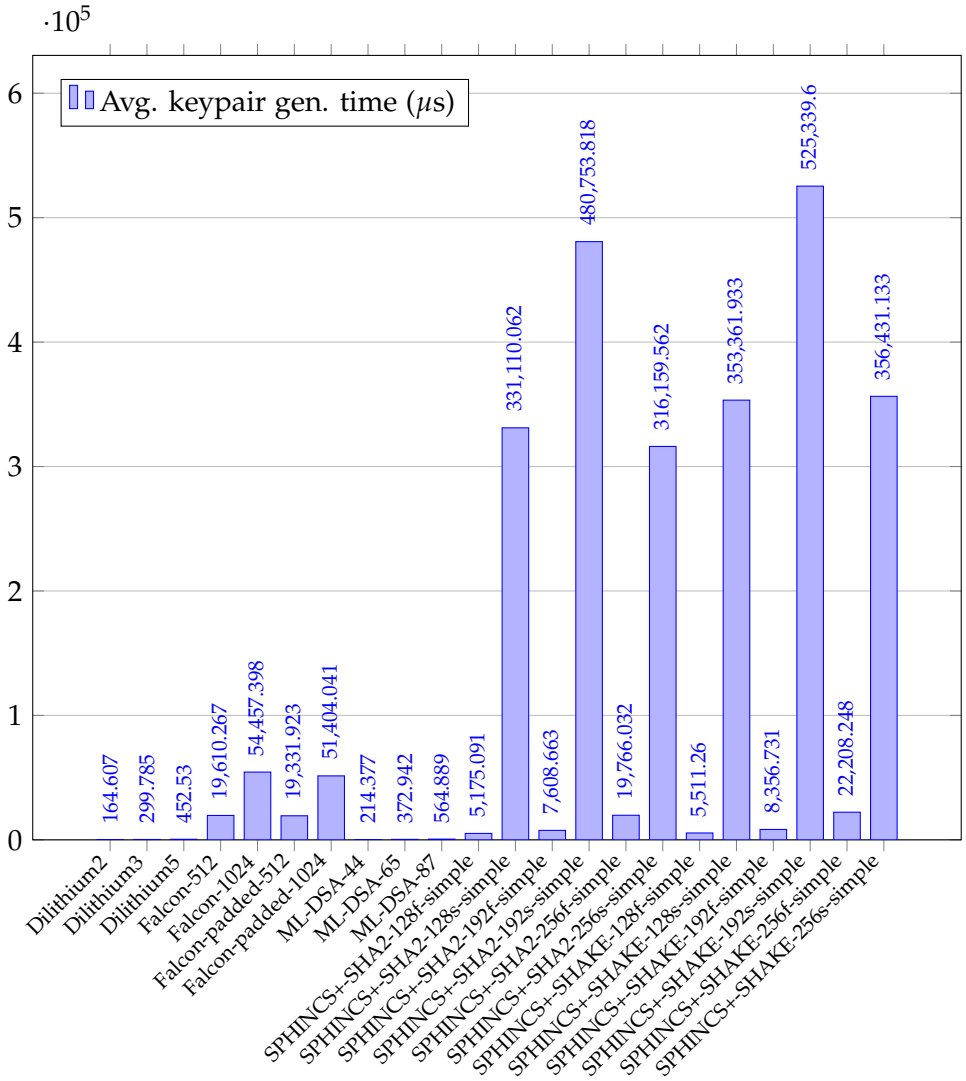


Figure 4.8: SIG algorithms keypair generation time benchmark (ARM)



In the keypair generation step for SIG algorithms, SPHINCS+ and Falcon had the slowest times, while Dilithium and ML-DSA performed best. On ARM, Dilithium2 was 120% slower than on x86, and ML-DSA-44 was 180% slower than its x86 counterpart.

Figure 4.9: SIG algorithms sign time benchmark (x86)

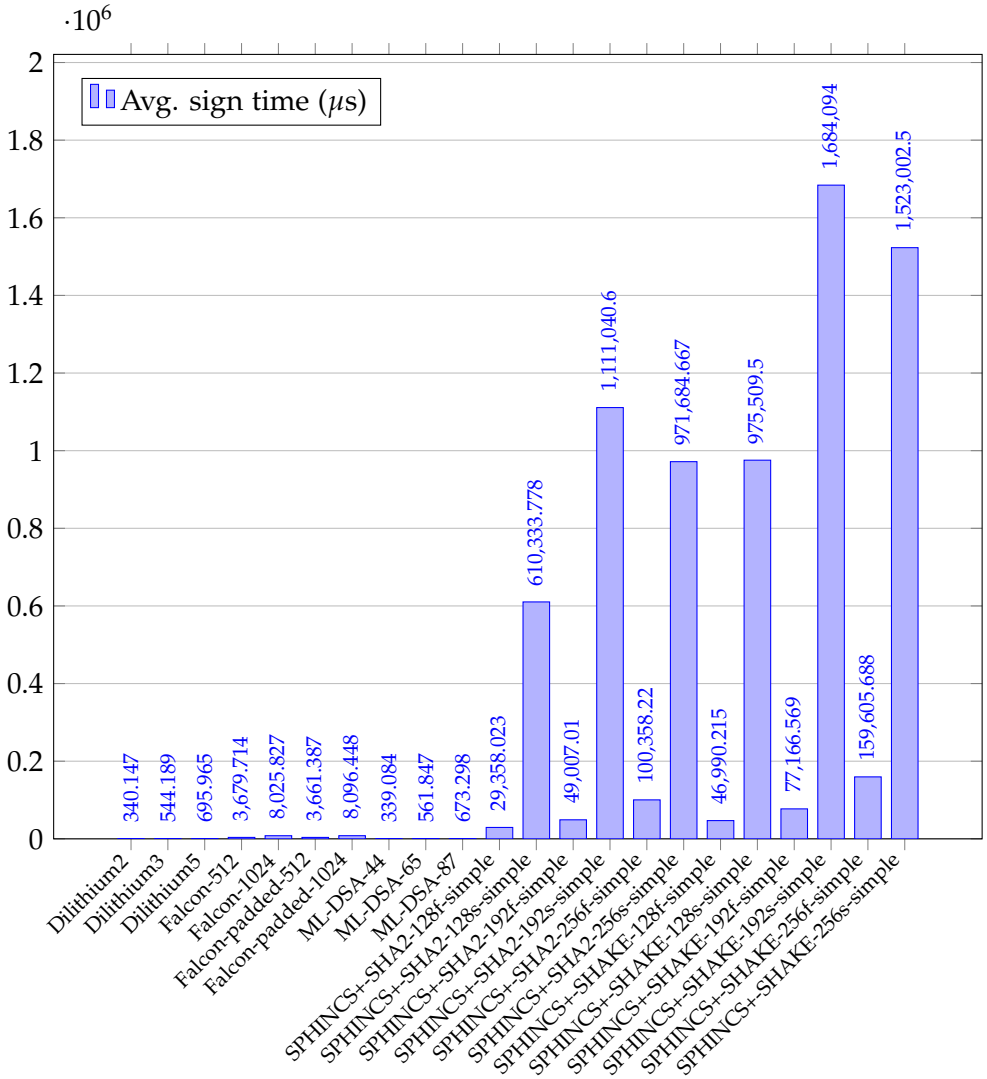
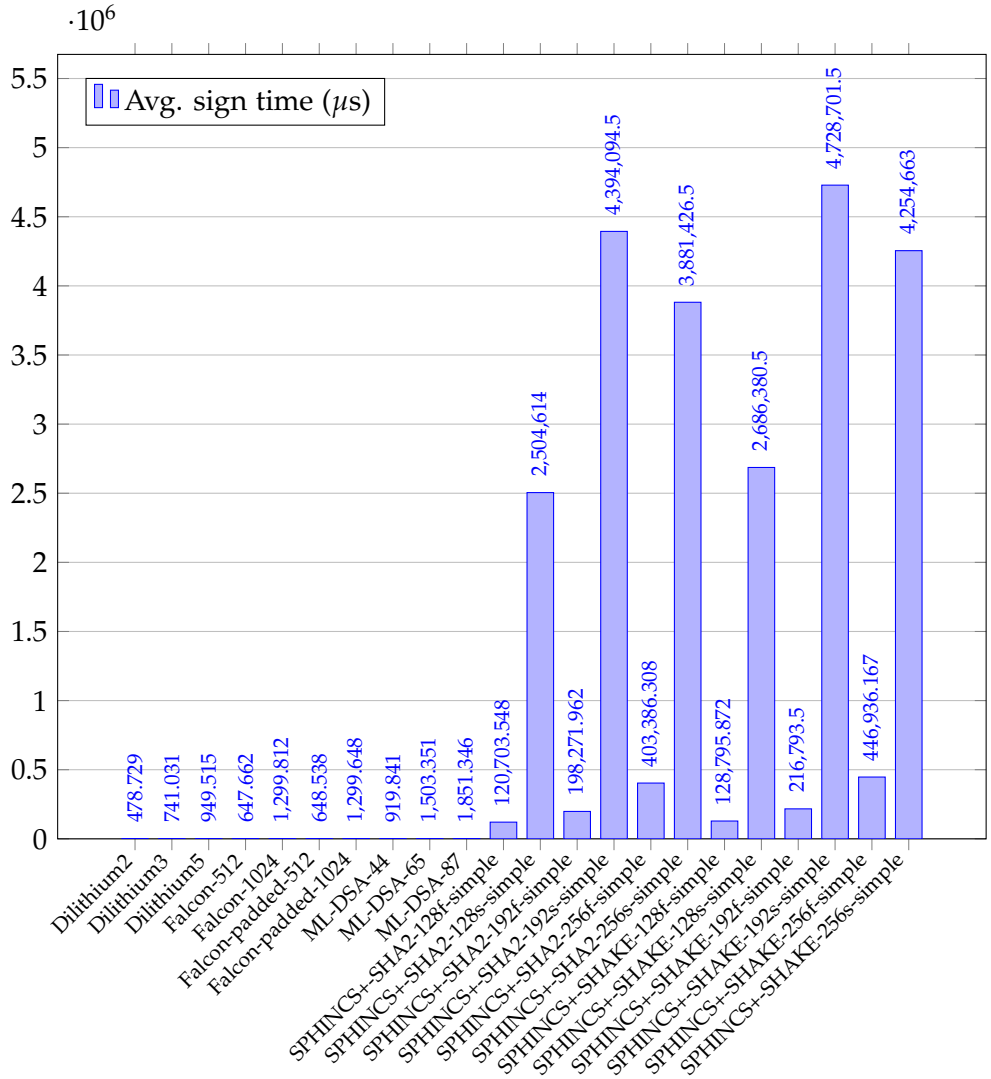


Figure 4.10: SIG algorithms sign time benchmark (ARM)



For the signing operation, SPHINCS+ and Falcon reported the slowest times again, being many times slower than others. ML-DSA-44 seems to be the fastest on x86, but that is not the case on ARM. On the ARM platform, Falcon-512 was faster than both ML-DSA-44 and Dilithium3. Dilithium2 reported being 40% slower on ARM compared to x86, while ML-DSA-44 was 170% slower on ARM. However, Falcon-512 surprised us this time by being 460% faster on ARM compared to x86. This is an interesting observation, and the CPU cycles in Table 4.8 and 4.9 can confirm that Falcon was indeed faster on ARM.

Figure 4.11: SIG algorithms verify time benchmark (x86)

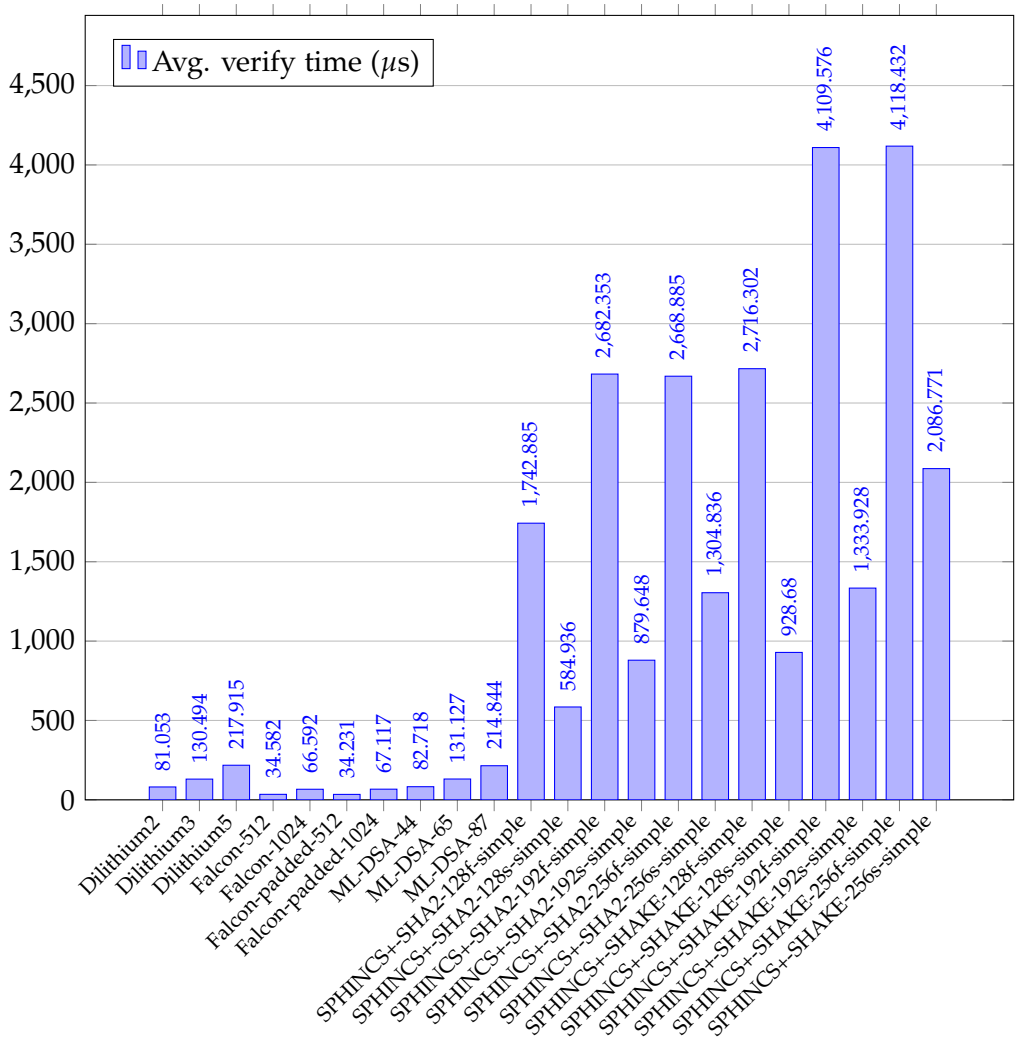
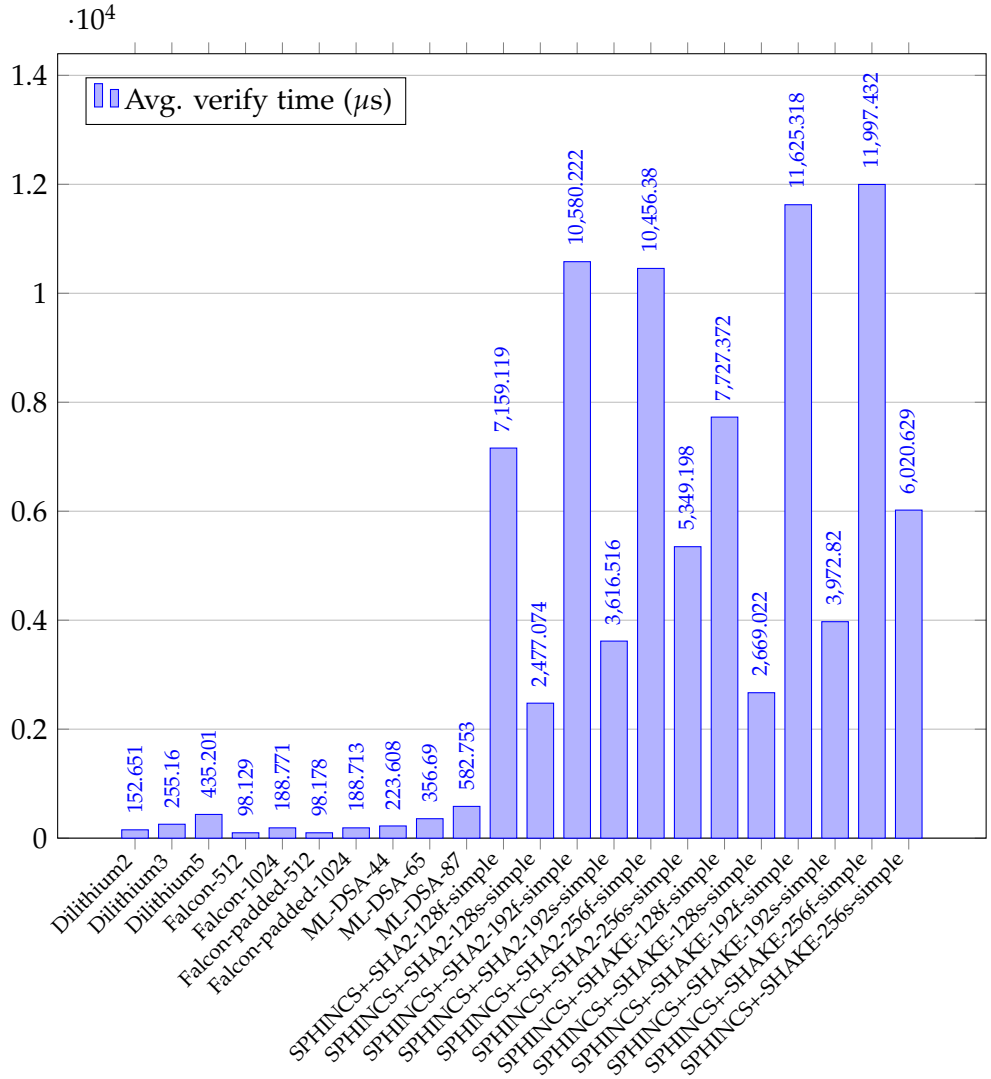


Figure 4.12: SIG algorithms verify time benchmark (ARM)



In the final step of verifying the signature, SPHINCS+ remains the slowest among all candidates. Despite the last results, Falcon-512 was the fastest this time on both x86 and ARM. Dilithium and ML-DSA had slower times than Falcon, but they were decent nevertheless. On ARM, Falcon-512 was 183% slower than his counterpart on x86, while Dilithium2 was 80% slower on ARM.

Algorithm	Parameter set	Keypair CPU cycles	Sign CPU cycles	Verify CPU cycles
Dilithium	Dilithium2	236384	1086796	258908
	Dilithium3	435570	1738788	416896
	Dilithium5	675188	2223749	696235
Falcon	Falcon-512	42107493	11758069	110405
	Falcon-1024	114267037	25646030	212700
	Falcon-padded-512	42014901	11699576	109287
	Falcon-padded-1024	118934780	25871634	214377
ML-DSA	ML-DSA-44	245223	1083414	264227
	ML-DSA-65	442196	1795228	418903
	ML-DSA-87	669287	2151338	686410
SPHINCS+	SPHINCS+-SHA2-128f-simple	4030823	93812521	5569170
	SPHINCS+-SHA2-128s-simple	256248622	1950327295	1869033
	SPHINCS+-SHA2-192f-simple	5952183	156600631	8571166
	SPHINCS+-SHA2-192s-simple	379520204	3550344513	2810752
	SPHINCS+-SHA2-256f-simple	15782472	320694164	8528107
	SPHINCS+-SHA2-256s-simple	248337948	3105029868	4169411
	SPHINCS+-SHAKE-128f-simple	6426390	150155947	8679619
	SPHINCS+-SHAKE-128s-simple	411612445	3117250934	2967413
	SPHINCS+-SHAKE-192f-simple	9530658	246584707	13131736
	SPHINCS+-SHAKE-192s-simple	595016155	1086575988	4262350
	SPHINCS+-SHAKE-256f-simple	25274532	510019463	13160020
	SPHINCS+-SHAKE-256s-simple	407269028	571805159	6667983

Table 4.8: SIG algorithms CPU cycles for all operations (x86)

Algorithm	Parameter set	Keypair CPU cycles	Sign CPU cycles	Verify CPU cycles
CRYSTALS-Dilithium	Dilithium2	164494	478622	152562
	Dilithium3	299681	740926	255054
	Dilithium5	452431	949419	435102
Falcon	Falcon-512	19610173	647564	98042
	Falcon-1024	54457292	1299719	188683
	Falcon-padded-512	19331859	648432	98084
	Falcon-padded-1024	51403975	1299541	188617
ML-DSA	ML-DSA-44	214270	919728	223506
	ML-DSA-65	372850	1503247	356602
	ML-DSA-87	564798	1851245	582670
SPHINCS+	SPHINCS+-SHA2-128f-simple	5175000	120703506	7159026
	SPHINCS+-SHA2-128s-simple	331109920	2504613760	2476982
	SPHINCS+-SHA2-192f-simple	7608534	198271744	10580105
	SPHINCS+-SHA2-192s-simple	480753943	4394094720	3616431
	SPHINCS+-SHA2-256f-simple	19765937	403386092	10456295
	SPHINCS+-SHA2-256s-simple	316159584	3881426048	5349109
	SPHINCS+-SHAKE-128f-simple	5511175	128795687	7727273
	SPHINCS+-SHAKE-128s-simple	353361835	2686380160	2668930
	SPHINCS+-SHAKE-192f-simple	8356661	216793344	11625206
	SPHINCS+-SHAKE-192s-simple	525339622	4728701312	3972646
	SPHINCS+-SHAKE-256f-simple	22208146	446936000	11997354
	SPHINCS+-SHAKE-256s-simple	356430916	4254662784	6020513

Table 4.9: SIG algorithms CPU cycles for all operations (ARM)

The CPU cycles required for running all operations are also provided for completeness. While CPU cycles might not be very accurate in regards to how many low-level operations the CPU executes within one cycle, they can provide additional profiling data and reassurance that our runtime times above are indeed correctly measured.

4.3 Result analysis and considerations

This section highlights real-world use cases for each algorithm participating in the NIST PQC standardization process. Taking the mathematical models into consideration, lattice-based algorithms are well-suited for KEMs. They offer relatively small key sizes and efficient computations, making them promising candidates for general-purpose use. Code-based algorithms are particularly suitable for applications that require fast encryption and decryption times, such as secure communication protocols (TLS/SSL). They have larger public key sizes compared to other PQC algorithms but they also offer fast performance in exchange. Multivariate polynomial algorithms are mostly fit for digital signature schemes. They provide fast signature generation and verification times, making them useful for applications that require frequent signing operations, such as authentication protocols or digital certificates. Hash-based algorithms are primarily used for digital signature schemes. They provide long-term resistance against quantum attacks and are appropriate for applications that do not require frequent key updates. Finally, isogeny-based algorithms are relatively new and are considered a promising candidate for key exchange protocols and are particularly attractive for applications that require small key sizes and efficient computations.

Going through our empirical measurements, we can provide insights into the performance characteristics and practicality. Key sizes, ciphertext sizes, and signature sizes are also significant factors that need to be taken into account in PQC, as they directly affect storage requirements, transmission bandwidth, and computational efficiency.

After inspecting the results for the KEM algorithms, ML-KEM and Kyber have had the fastest computation times overall among all algorithms, operations, and platforms. ML-KEM is an algorithm that has evolved from Kyber, so they have the same key lengths. However, ML-KEM performed best in the decapsulation step, while Kyber performed best in the key generation and encapsulation steps. Both exhibit fast computation times compared to other candidates, thus they would be suitable for high-speed communications in scenarios such as satellite communication or high-frequency trading systems where low latency is very important. McEliece reported fast encapsulation times but it has performed poorly in key generation and decapsulation scenarios. McEliece has bigger key sizes, but overall smaller ciphertext sizes, which could be interesting when thinking about applications of PQC on low-resource embedded devices. The NTRU Prime algorithm offers average performance in all situations, with relatively small key sizes compared to other PQC schemes, making it attractive for applications with limited storage or bandwidth. BIKE performed poorly in all three situations, exhibiting very slow times, especially in

the key generation and decapsulation phases. The key sizes for BIKE are also big, and the ciphertext is among the biggest ones, along with FrodoKEM and HQC. HQC reported reasonable performance during the key generation step but slow execution times in the encapsulation phase. FrodoKEM has performed well during key generation and decapsulation steps but reported slower times during encapsulation.

By looking at the results for the SIG algorithms, we can notice that SPHINCS+ has been consistently the slowest algorithm within all tests. One advantage that SPHINCS+ could have is that it uses very small key sizes, which could be favorable over performance in specific scenarios where small storage size is more important than speed. ML-DSA and Dilithium were similar in execution times, both exhibiting very fast execution times in all three scenarios. They seem to be well-suited for applications such as software updates or code signing. Their efficient signature generation and verification make it suitable for authenticating firmware updates in embedded systems or verifying the integrity of software packages distributed over package repositories. For keypair generation, Dilithium2 and ML-DSA-44 performed very similar. On ARM, Dilithium was slightly faster than the ARM version of ML-DSA. Falcon has performed extremely well in verifying the signatures, ending up in the first place as the fastest candidate on both x86 and ARM. Unfortunately, it did not perform that well within the keypair generation, having the slowest execution times. However, Falcon-512 surprised us by being almost five times faster on ARM compared to x86 on signing operations. Falcon also seemed to have reasonably small signature sizes and decent key sizes.

Chapter 5

Conclusions

This study presented a thorough review of the latest PQC algorithms enrolled in the NIST PQC standardization contest in terms of mathematical models, key generation, encapsulation, decapsulation, key sizes, operation runtime, and real-world usage. Moreover, we strived to provide a more complete performance analysis of NIST PQC algorithms on ARM, for both KEM and SIG algorithms. We compared results and metrics gathered from benchmarking all available algorithms to get a full view of the picture. Lastly, we compared these results, discussing each algorithm's strengths, drawbacks, and potential use cases.

The results of this project contribute to the ongoing efforts in standardizing post-quantum encryption algorithms and give useful recommendations for researchers and developers. All factors must be taken into consideration when choosing an algorithm for secure communication. As quantum computing advances, it is critical to monitor and adjust cryptographic systems to ensure the secrecy and integrity of sensitive data. Future research might build on this work by investigating additional post-quantum encryption methods, evaluating their performance on various operating systems and platforms, and addressing real-world deployment circumstances. The effective development and implementation of post-quantum cryptographic algorithms will assure the long-term security of communications in the age of quantum computing, protecting sensitive data from new emerging threats.

Bibliography

[1] Wikipedia. *IBM Osprey quantum computer with 433 qubits*. Accessed: 25-05-2024. URL: https://en.wikipedia.org/wiki/IBM_Osprey.

[2] Wikipedia. *Shor's algorithm*. Accessed: 25-05-2024. URL: https://en.wikipedia.org/wiki/Shor%27s_algorithm.

[3] Wikipedia. *Grover's algorithm*. Accessed: 25-05-2024. URL: https://en.wikipedia.org/wiki/Grover%27s_algorithm.

[4] Dimitrios Sikeridis, Panos Kampanakis, and Michael Devetsikiotis. "Post-Quantum Authentication in TLS 1.3: A Performance Study". In: *IACR Cryptol. ePrint Arch.* 2020 (2020), p. 71. URL: https://api.semanticscholar.org/CorpusID:210877381.

[5] Manohar Raavi et al. "Security Comparisons and Performance Analyses of Post-quantum Signature Algorithms". eng. In: *Applied Cryptography and Network Security*. Vol. 12727. Cham: Springer International Publishing, 2021, pp. 424–447. ISBN: 303078374X.

[6] NIST. *PQC*. Accessed: 15-05-2024. URL: https://csrc.nist.gov/projects/post-quantum-cryptography.

[7] NIST. *PQC additional SIG candidates*. Accessed: 15-05-2024. URL: https://csrc.nist.gov/news/2023/additional-pqc-digital-signature-candidates.

[8] NIST. *PQC standardized candidates and Round 4*. Accessed: 15-05-2024. URL: https://csrc.nist.gov/News/2022/pqc-candidates-to-be-standardized-and-round-4.

[9] NIST. *PQC Round 1 of selecting additional SIG algorithms*. Accessed: 21-05-2024. URL: https://csrc.nist.gov/Projects/pqc-dig-sig/round-1-additional-signatures.

[10] NIST. *SIKE discontinued notice*. Accessed: 26-05-2024. URL: https://csrc.nist.gov/csrc/media/Projects/post-quantum-cryptography/documents/round-4/submissions/sike-team-note-insecure.pdf.

[11] NIST. *CACM - David Geer*. Accessed: 21-05-2024. URL: https://cacm.acm.org/news/nist-post-quantum-cryptography-candidate-cracked/.

[12] Wouter Castryck and Thomas Decru. "An Efficient Key Recovery Attack on SIDH". eng. In: *Advances in Cryptology – EUROCRYPT 2023*. Lecture Notes in Computer Science. Cham: Springer Nature Switzerland, pp. 423–447. ISBN: 9783031305887.

[13] SIKE. *SIKE PQC algorithm*. Accessed: 26-05-2024. URL: https://sike.org/.

[14] Ward Beullens. "Breaking Rainbow Takes a Weekend on a Laptop". eng. In: *Advances in Cryptology – CRYPTO 2022*. Cham: Springer Nature Switzerland, pp. 464–479. ISBN: 9783031159787.

[15] Sean Zakrajsek. "Performance Analysis of NIST Round 2 Post-Quantum Cryptography. Public-key Encryption, Decryption and Key-establishment algorithms on ARMv8 IoT Devices using SUPERCOP". eng. In: 2020. URL: https://repository.stcloudstate.edu/cgi/viewcontent.cgi?article=1139&context=msia_etds.

[16] Thomas E. Carroll, Addy Moran, and Lindsey Redington. "Exploring the Adoption Challenges of Post-Quantum Cryptography in EV Charging Infrastructure". eng. In: 2024. URL: https://www.pnnl.gov/main/publications/external/technical_reports/PNNL-35760.pdf.

[17] Kathryn Hines et al. "Post-Quantum Cipher Power Analysis in Lightweight Devices". In: *Proceedings of the 15th ACM Conference on Security and Privacy in Wireless and Mobile Networks*. WiSec '22. San Antonio, TX, USA: Association for Computing Machinery, 2022, pp. 282–284. ISBN: 9781450392167. DOI: 10.1145/3507657.3529652. URL: https://doi.org/10.1145/3507657.3529652.

[18] Basel Halak et al. "Evaluation of Performance, Energy, and Computation Costs of Quantum-Attack Resilient Encryption Algorithms for Embedded Devices". In: *IEEE Access* PP (Jan. 2024), pp. 1–1. DOI: 10.1109/ACCESS.2024.3350775.

[19] Open Quantum Safe. *liboqs*. Accessed: 15-05-2024. URL: https://openquantumsafe.org/liboqs.

[20] BIKE. *BIKE PQC algorithm*. Accessed: 26-05-2024. URL: https://bikesuite.org/.

[21] McEliece. *Classic McEliece PQC algorithm*. Accessed: 26-05-2024. URL: https://classic.mceliece.org/.

[22] FrodoKEM. *FrodoKEM PQC algorithm*. Accessed: 26-05-2024. URL: https://frodokem.org/.

[23] HQC. *HQC PQC algorithm*. Accessed: 26-05-2024. URL: https://pqc-hqc.org/.

[24] CRYSTALS. *Kyber/Dilithium PQC algorithms*. Accessed: 26-05-2024. URL: https://pq-crystals.org/.

[25] ML-KEM. *ML-KEM PQC algorithm*. Accessed: 26-05-2024. URL: https://csrc.nist.gov/pubs/fips/203/ipd.

[26] NTRU. *NTRU Prime PQC algorithm*. Accessed: 26-05-2024. URL: https://ntruprime.cr.yp.to/.

[27] Falcon. *Falcon PQC algorithm*. Accessed: 26-05-2024. URL: https://falcon-sign.info/.

[28] ML-DSA. *ML-DSA PQC algorithm*. Accessed: 26-05-2024. URL: https://csrc.nist.gov/pubs/fips/204/ipd.

[29] SPHINCS+. *SPHINCS+ PQC algorithm*. Accessed: 26-05-2024. URL: https://sphincs.org/.