Performance Evaluation of Post-Quantum TLS 1.3 on Embedded Systems

George Tasopoulos
Industrial Systems Institute
R.C. ATHENA
Platani, Patra, Greece
tasopoulos@isi.gr

Jinhui Li
Faculty of Information Technology,
Monash University
Clayton, Victoria, Australia
jlii0204@student.monash.edu

Apostolos P. Fournaris
Industrial Systems Institute
R.C. ATHENA
Platani, Patra, Greece
fournaris@isi.gr

Raymond K. Zhao
Faculty of Information Technology,
Monash University
Clayton, Victoria, Australia
raymond.zhao@monash.edu

Amin Sakzad
Faculty of Information Technology,
Monash University
Clayton, Victoria, Australia
amin.sakzad@monash.edu

Ron Steinfeld
Faculty of Information Technology,
Monash University
Clayton, Victoria, Australia
ron.steinfeld@monash.edu

ABSTRACT

Transport Layer Security (TLS) constitutes one of the most widely used protocols for securing Internet communication and has found broad acceptance also in the Internet of Things (IoT) domain. As we progress towards a security environment resistant against quantum computer attacks, TLS needs to be transformed in order to support post-quantum cryptography schemes. However, post-quantum TLS is still not standardized and its overall performance, especially in resource constrained, IoT capable, embedded devices is not well understood. In this paper, we evaluate the time, memory and energy requirements of a post-quantum variant of TLS version 1.3 (PQ TLS 1.3), by integrating the pqm4 library implementations of NIST round 3 post-quantum algorithms Kyber, Saber, Dilithium and Falcon into the popular wolfSSL TLS 1.3 library. In particular, our experiments focus on low end, resource constrained embedded devices manifested in the ARM Cortex-M4 embedded platform NUCLEO-F439ZI (with hardware cryptographic accelerator) and NUCLEO-F429ZI (without hardware cryptographic accelerator) boards. These two boards only provide 180 MHz clock rate, 2 MB Flash Memory and 256 KB SRAM. To the authors' knowledge this is the first thorough time delay, memory usage and energy consumption PQ TLS 1.3 evaluation using the NIST round 3 finalist algorithms for resource constrained embedded systems with and without cryptography hardware acceleration. The paper's results show that the post-quantum signatures Dilithium and Falcon and post-quantum KEMs Kyber and Saber perform in general well in TLS 1.3 on embedded devices in terms of both TLS handshake time and energy consumption. There is no significant difference between the TLS handshake time of Kyber and Saber; However, the handshake time with Falcon is much lower than that with Dilithium. In addition, hardware cryptographic accelerator for symmetric-key primitives improves the performances of TLS handshake time by about 6% on the client side and even by 19% on the server side, on high security levels.

CCS CONCEPTS

• **Computer systems organization → Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability.

KEYWORDS

cryptography, quantum-safe, network security, TLS 1.3, embedded systems

1 INTRODUCTION

In 1994, Peter Shor, published a work [31] that describes an algorithm for quantum computers that can solve the factoring problem and the discrete logarithm problem in polynomial time, the two underlying problems that the current Public Key Infrastructure (PKI) relies mostly upon, thus making many network communications used today vulnerable to future quantum attacks. In recent years, there have been important advances regarding quantum computers. Many companies, like Google and IBM, and organizations, like the University of Science and Technology of China (USTC), have succeeded on making operating quantum computers. Even though, these computers cannot solve real world, significant, computation problems, they pave the way for large scale quantum computers by displaying what is called "quantum supremacy". On 2019, Google displayed quantum supremacy for the first time on an operational quantum computer [11].

In response, in 2017 the National Institute of Standards and Technology of the USA (NIST) initiated a (still ongoing) evaluation and standardization process to select the next generation of industry-standard public key cryptographic primitives. At the time of paper's writing, the competition is in the 3rd round with 7 finalist algorithms, 4 of which are key exchange algorithms and 3 digital signature algorithms. Currently, NIST round 3 candidates can be divided into 3 families: lattice-based, code-based and multivariate. Regarding key exchange there are Kyber, Saber, NTRU, all of them being lattice-based and Classic McEllice being code-based. Regarding digital signatures, there are Dilithium and Falcon, both lattice-based, and Rainbow, being multivariate. Almost all of the candidates, 5 out of 7, use lattice-based cryptography. The implementation of code-based cryptography requires significant memory and sometime high execution times which can be prohibiting adoption reason for resource constrained devices. The recent cryptanalytic results [13], create some uncertainty about multivariate candidates, and moreover, the very large public key size of Rainbow, make the more balanced lattice-based schemes the most promising candidates. Out of the 7 NIST PQC finalist algorithms, we believe four of them, namely Kyber [15], Saber [20] for key exchange and Dilithium [17] and Falcon [21] for authentication (all of which are lattice-based) are the most promising candidates. The above algorithms are chosen for three reasons. As discussed above, the lattice-based schemes are the most promising in terms of security and efficiency, of all the finalist candidates. Classic McEllice and Rainbow require more memory and communication costs than an embedded system typically can afford. NTRU has significantly slower key generation time than Kyber and Saber.

One of the most popular secure communication protocols on the Internet is Transfer Layer Security (TLS). Perhaps over 80% of the Internet traffic is using TLS and for that reason, is one of the most important protocols to evaluate against quantum attack threats. One of the main issues of the community is integrating post-quantum algorithms in TLS and measuring the overheads it introduces with respect to a number of metrics: execution speed, memory requirements, communication size, and code size. Due to the increasing number of constrained devices connecting to the Internet, forming what is known as the Internet-of-Things (IoT), the performance of Post-Quantum TLS (PQ TLS) on these devices is a challenging and important research topic. The time delay, memory, communication and energy requirements of PQ TLS are important considerations for resource constrained devices, and may be critical to their final adoption of PQ TLS. However, prior works on evaluating the resource requirements of PQ TLS for embedded devices [16, 26] suffer from various limitations, namely the restriction to the older version 1.2 of TLS, lack of evaluation of TLS with lattice-based signature schemes [16], lack of evaluation on lower-end, resource constrained, modern embedded devices, restriction to mutually authenticated TLS handshakes [26], and also lack of energy consumption measurements (we refer the reader to Sec. 1.2 for further discussion).

1.1 Our Contributions

In this paper, we address the above-mentioned gaps by studying the execution speed, memory requirements and communication size, of four post-quantum public-key candidates from the NIST PQC 3rd round, on TLS 1.3 coupled with two constrained devices. Specifically, we integrate pqm4 [25] library implementations of the NIST round 3 candidates' signature Dilithium and Falcon and post-quantum key exchange mechanisms (KEM) Kyber and Saber into the wolfSSL [9] implementation of TLS 1.3[1]; And evaluate these PQ algorithms' performance of handshake time delay, memory usage and energy-consumption under a full TLS 1.3 connection between a server and a client over a local Ethernet network[2], assuming only the server authenticates to the client. Our main contributions are as follows:

ARM Cortex-M4 Embedded Platform Integration: We integrate the PQ algorithms library pqm4 [18] to wolfssl for algorithms Kyber, Saber, Dilithium and Falcon. We also use the lwIP library for network connectivity and routing functions for our project. **Embedded Platform Evaluation of Runtime for PQ TLS 1.3**: We measure TLS handshake execution time of both client and server on both embedded platforms, NUCLEO-F439ZI with hardware accelerator and NUCLEO-F429ZI without hardware accelerator. Hardware accelerator is used in symmetric key crypto primitives like encryption and hash used in TLS. The PQ cryptography algorithms themselves, internally use SHA-3 [19] which is implemented in software without acceleration. We found the performance of PQ TLS1.3 with Kyber and Saber is similar; And PQ TSL1.3 with Falcon is less costly than PQ TLS1.3 with Dilithium in terms of execution time on client side while Dilithium is less costly than Falcon on server side.

Embedded Platform Evaluation of Energy for PQ TLS 1.3: We also measure the handshake energy consumption of the client on the embedded platform NUCLEO-F429ZI. We introduced the expansion board X-NUCLEO-LPM01A (can perform static measurement up to 200 mA and dynamic measurement up to 50 mA) to capture the the energy consumption of NUCLEO-F429ZI. Our results show the average current consumption of TLS sessions establishment process is slightly affected by employed cryptography algorithms.

Main Findings Summary. The results of our evaluations lead to the following main findings for our post-quantum TLS protocol versus the baseline quantum-insecure TLS protocol (using ECDHE/ECDSA): the server handshake run-time overheads of the post-quantum protocols are approximately 1.6 times (resp. 5.1 times) with Dilithium (resp. Falcon) on authentication, both for 128-bit quantum security i.e. NIST level 1 and 2.3 times (resp. 10.6 times) for 192-bit quantum security i.e. NIST level 3 Dilithium signature (resp. 256-bit quantum security i.e. NIST level 5 Falcon signature). On the other hand, the client handshake run-time of the post-quantum protocols are approximately 1.25 times for Dilithium authentication (resp. 2.2 times for Falcon authentication) faster for NIST level 1 and 1.1 times slower and 1.5 times faster for NIST level 3 and level 5 respectively for Dilithium and Falcon. The hardware acceleration typically decreased handshake time by about 6% on the client side while on the server-side can decrease it up to 19%. We also found the RAM memory usage overhead to be approximately 33 times for Dilithium (resp. 20 times for Falcon) for NIST level 1.

1.2 Related Work

Prior to the start of NIST competition, there was a bloom of research in the field of post-quantum cryptography. Aside from the post-quantum algorithms that were developed for the competition, other research emerged in recent years (and in particular after the start of NIST standardisation process) focused on prototyping, integrating in popular protocols, measuring the speed, memory requirements and other metrics of these post-quantum algorithms.

PQ TLS on Embedded Platforms. In the work of Bürstinghaus-Steinbach K. et al. [16], an implementation of TLS with post-quantum algorithms is used to take metrics on embedded devices, but an older version of TLS is used (TLS 1.2) and moreover the signature scheme used is hash-based (SPHINCS+) [12] rather than lattice-based. Another work by Paul et al.[27] introduces a migration strategy towards post-quantum authentication by using post-quantum algorithms in mixed certificate chains and also evaluates post-quantum TLS 1.3 performance on a server, on a PC and on a Raspberry Pi. The

[1]The source code for the embedded platform is available in the public repository: https://gitlab.com/g_tasop/pq-wolfssl-for-embedded

[2]The source code for the PC platform is available in the public repository: https://gitlab.com/g_tasop/pq-wolfssl-for-pc

Raspberry Pi device used in [27] has an ARM Cortex-A53 processor running at 1.2 GHz, a significantly higher end embedded device than the Cortex-M4 at 180 MHz used in our evaluation. Moreover, Paul et al. evaluated only a mutually authenticated TLS 1.3 handshake on both local and remote network connections where both client and server authenticate to each other, whereas our work focuses on server-only authentication (a typical scenario in IoT devices) on a local network.

PQ TLS on Other Platforms. PQClean [6] focuses on the prototyping of post-quantum algorithms. It tries to collect or write standalone implementations of post-quantum algorithms without any external dependencies. Open Quantum Safe (OQS) [33] project, has developed a cryptographic library, named *liboqs*, that focuses on post-quantum algorithms. By leveraging the developed library, this project integrates post-quantum algorithms to popular libraries that implements security protocols, like OpenSSL [5] and OpenSSH [4], making it possible to evaluate or even use these protocols with post-quantum algorithms in real-world scenarios. On the work of Sikeridis D. et al. [32], the OpenSSL and OpenSSH forks of OQS are used to measure the overhead that is introduced with post-quantum integration on these two protocols on realistic network conditions. Both this work and [27] use TLS 1.3 with post-quantum algorithms, but target medium to high resource systems.

Standalone PQ Algorithm Implementations for Embedded Platforms. Regarding post-quantum cryptography on embedded systems, a project named mupq is developed that provides optimized libraries targeting a number of embedded processors or even FPGAs. For example, mupq's library *pqm4* [25] is implementing and collecting code for all the submissions of NIST PQC competition, with a focus on optimizations for the Cortex-M4 series of processors.

To the authors knowledge, this paper constitutes the first systematic and thorough architectural adaptation, implementation and performance evaluation of a Post-Quantum TLS 1.3 (PQ TLS 1.3), based on the popular wolfSSL library, that takes into account the latest NIST PQC competition finalists algorithms and explicitly targets low resource embedded systems.

Structure: In Section 2, we review background knowledge needed for this work. In Section 3, we describe the architecture of the library we use as well as the architectural changes we did to make TLS work with post-quantum algorithms and we discuss the implementation details. In Section 4, we display our measurements from our evaluations and finally in Section 5, we conclude this paper.

2 BACKGROUND

2.1 TLS Protocol

One of the major security protocols that are threatened by a possible quantum attack, is the Transport Layer Security (TLS) protocol. TLS is the most widely used protocol for secure communications on the Internet, making it a de facto security standard. HTTPS for secure website transfer [29], secure connection to mail servers [22] as well as secure Internet access for smartphone apps [28], are some of the many use cases of TLS. TLS version 1.3 [30] has been standardised in 2018 and is currently offering many important changes over the previous version, TLS 1.2. With a focus on strong security, by terminating support for deprecated and potentially dangerous cryptography functions, and on speed, by eliminating a whole round-trip in comparison to the previous version, TLS 1.3 is an important upgrade on the protocol itself. The adoption of TLS 1.3 is in good levels, due to the high centralization of the Internet and to the long period of the drafts evaluation [23]. In fact, the *Internet Society Pulse* reports nearly 60% adoption of TLS 1.3 by the top 1,000 websites globally [10].

TLS is a cryptographic protocol designed to provide secure communication over a computer network. It is typically considered among the application layer, in the Internet protocol stack, as the protocol that provides privacy and data integrity between two parties. TLS consists of two primary components: a *handshake protocol* that authenticates the communicating parties, negotiates cryptographic modes and parameters, and establishes shared keying material to create a secure session (secure communication channel) and a *record protocol* that uses the parameters established by the handshake protocol to protect traffic between the securely communicating peers. The *record protocol*, which is located above the transport layer and uses the Transmission Control Protocol (TCP), will not be directly affected by quantum attacks since it uses symmetric key ciphers and it can be easily modified to be completely quantum safe. This is in contrast to the *handshake protocol*, which makes heavy use of public key cryptography, and hence will be directly affected and threatened by quantum cryptanalytic attacks. TLS 1.3 uses public key cryptography for two main reasons. Key exchange, which mostly uses Diffie-Hellman (DH) over Elliptic Curves (EC) or RSA and Digital Signatures, which use DSA over Elliptic Curves or RSA. For this reason, an integration of quantum resistant public key cryptography algorithms must be made, for both the key exchange and the digital signatures.

On TLS 1.3, the handshake between two parties, a client and a server, begins with the client sending the first TLS 1.3 message, ClientHello, to the server. This message typically contains a client random number, the client TLS version, a list of Cipher Suites and a series of extensions with additional information like: Server Name, Supported Groups, Supported Signature Algorithms, Key Share and the Supported Versions. The "Supported Groups" extension contains all the key exchange methods and the extension "Supported Signature Algorithms" contains all the digital signature algorithms that are supported by the client. In addition, Key Share contains an ephemeral ECDH or RSA public key. TLS 1.3 makes significant use of the extensions field, for example if the negotiated version of TLS is version 1.3, it is indicated on "Supported Version" extension, as the original entry "Supported Version" is set to TLS 1.2, to eliminate compatibility issues with middleboxes. The server then replies with a ServerHello message, which contains a server random number, the selected Cipher Suite using the client's list of Cipher Suites and the server's preferences, the negotiated Protocol Version and the server's Key Share. From now on, every message that is exchanged is encrypted. The server sends the "EncryptedExtensions" message which contains the remaining extensions. Then, the server sends the Server Certificate, containing its digital certificate and the certificate chain up to a root Certificate Authority (CA). Then, the server sends the "CertificateVerify" message which basically, contains a digital signature over a hash of all handshake messages exchanged, starting at ClientHello and up to, but not including, this message itself. Finally, the server sends the "ServerHandshakeFinished" message, indicating that the handshake is complete by the server side.

In the above handshake protocol description that is adopted in this paper's analysis and overall PQ TLS 1.3 evaluation, only the server is authenticated, so the client now has all the necessary information to produce the final master key and can also verify the certificate as well as the signature of the server. The client, finally send the "ClientHandshakeFinished" message that informs the server that the handshake is now complete by the client side.

2.2 TLS Implementations for Embedded Systems and Beyond

When the NIST PQC competition was in the 2nd round, NIST urged the community to implement the post-quantum algorithms on embedded systems and specifically on ARM processors like Cortex-M4 [1]. So, in the summer of 2019, a paper was published that collected and implemented all the post-quantum algorithm implementations from the 2nd round of the competition specifically for Cortex-M4, creating the so-called pqm4 library [25]. These implementations are based on the "Reference Implementations" of the official releases of the algorithms but with a focus on speed and size and using optimized assembly code for the Cortex-M4 processor. As the competition advances, the pqm4 library is regularly updated with new versions of the code or with any changes the algorithms themselves introduce. All these updates are being kept in an open-source git repository available online at [24].

There are numerous implementations of the TLS protocol covering the different needs of computing devices. There are open-source and closed-source implementations, implementations targeting PCs, smartphones and of course, embedded systems. The Open Quantum Safe [33] project aiming on developing quantum-resistant software is focused on developing an open-source library, named liboqs, that implements post-quantum cryptography and is embedded in popular, widely used, TLS libraries like OpenSSL [5], thus providing post-quantum cryptographic protocols. OpenSSL is a very famous implementation of the TLS protocol and is used by the majority of servers that support HTTPS. So the OQS project, created as a fork of the OpenSSL project that is using liboqs, to provide a fully functional TLS implementation using post-quantum cryptography. Unfortunately, OpenSSL targets PCs and servers, making it unsuitable to be ported on resource constrained embedded platforms.

Regarding TLS open-source solutions for embedded systems, the most famous and widely used implementations are: Mbed TLS [3] and wolfSSL [8, 9]. With Mbed TLS lacking support for TLS 1.3, wolfSSL is the only option to be adopted in this paper's research work. WolfSSL is a library that implements the TLS protocol with "traditional" cryptography, without having immediate support for post-quantum cryptography. As of version 4.7.0, wolfSSL offer to the developers the interfaces of including local implementation of the NTRU quantum-resistant algorithm, but lacks support for any other PQC algorithms. In this paper, wolfSSL library was modified so it can use the four selected post-quantum algorithms from NIST PQC 3rd round competition; namely Kyber, Saber, Dilithium and Falcon.

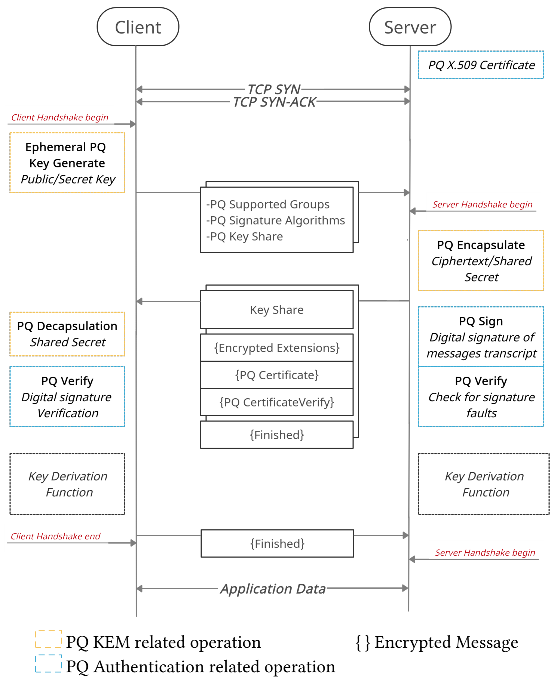

Figure 1: Post-quantum TLS handshake messages

2.3 WolfSSL

WolfSSL [9], the TLS library targeting embedded platforms, generally consists of three major components: *wolfCrypt*, a cryptographic library, *wolfSSL*, the TLS protocol code along with all associated functionality and a set of *utilities*: test programs, benchmarks, and others. More information on these components can be found on Appendix A.

3 ARCHITECTURE AND IMPLEMENTATION

In order for the TLS protocol to operate with post-quantum algorithms, the first action to be performed is to integrate the pqm4 [25] implementations of Kyber, Saber, Dilithium and Falcon in the wolfSSL code. Pqm4 provides implementations of these algorithms that are optimized for our target microcontroller, ARM Cortex-M4. Then, we used lwIP library [18] to build the network layer for our implementation; LwIP library provides the Ethernet connection and routing functions for our project. Apart from the code itself, the PQ algorithms require the usage of some cryptographic primitives. Specifically, all of the adopted algorithms make use of the Keccak primitives, SHA-3 and SHAKE-256 [19]. The provided implementations of these algorithms from the mupq project has been adopted in our work (a simple, C code implementation).

3.1 WolfSSL Post-Quantum Adaptation

In order to make the wolfSSL TLS 1.3 messages on the handshake layer post-quantum compatible, architectural adaptations/adjustments have been made in the wolfSSL library on the Extension "Supported Groups", on the Extension "Signature Algorithms", on supporting

Key Encapsulation Mechanisms and supporting Post-Quantum Digital signatures and Certificates.

In Figure 1, the overall post-quantum adapted handshake message exchanges are presented in detail indicating the PQ operations that are made in each phase of the TLS handshake. One important thing to note, is that wolfSSL, when acting as a server using RSA, immediately after creating a signature, runs the Verify operation to check for signature faults. We mimicked this behaviour on our PQ TLS adaptation. Although it may be redundant, it only adds a minor overhead on our measurements.

3.1.1 Supported Groups. The ClientHello message, the first message that the client sends to initiate the handshake, consists of several fields, one of which is the Extensions field. In this field, the client extends the information provided by the rest of the ClientHello fields and plays a crucial role in TLS 1.3. One of the fields among the Extension field, as shown in Figure 1, is the field *Supported Groups.* In this field, the client sends a list of key exchange algorithms, as encoded identifiers (codepoints), in order of preference, so that the server can select one of them to be used in the handshake. These identifiers are called, Named Groups, and are defined for each supported algorithm by the protocol itself. In order to use post-quantum algorithms, we have introduced our own Named Groups. In order for the wolfSSL library to be inter-operable with other popular libraries, we decided to choose the codepoints that are being used by OQS's fork of OpenSSL [5]. The codepoints for the post-quantum algorithms are shown in Table 1 along with some traditional algorithms' codepoints.

3.1.2 Signature Algorithms. Another useful field on the Extension field, is the "Signature Algorithms" field. In this field, the client provides its preference on the signature algorithms that it supports, regarding the CertificateVerify field. This means that this signature algorithm will be used to sign the transcript of the data exchanged by the server and to be verified by the client. Similar to the extension "Supported Groups", apart from the predefined codepoints for each algorithm we introduce our own codepoints for the post-quantum digital signature algorithms that the PQ TLS wolfSSL can support. The codepoints that have been added are compliant with the OQS's fork of OpenSSL [5], as shown in Table 1 along with with some of the traditional algorithms' codepoints.

3.1.3 Key Encapsulation Mechanism Support/Adaption. All the post-quantum algorithms that participating in the NIST competition are Key Encapsulation Mechanism (KEM) schemes. However, the key exchange method that is used in TLS is the traditional (Elliptic Curve) Diffie-Hellman Key Exchange. In order to adapt the key exchange to the post-quantum environment, in our proposed work the key exchange mechanism of TLS 1.3 is transformed into a KEM scheme through some architectural adaptation. We used the proposition introduced in the Crystals-Kyber key exchange scheme [15] that is also presented below:

Initially, the client generates a key pair and sends the public key to the server with the ClientHello message. The server, using the client's public key, calls the Encapsulation function that produces a Ciphertext, that is sent to the client with the ServerHello message, and a Shared Secret, that the server keeps, as it is the actual shared key. The client, upon receiving the Ciphertext calls the Decapsulation function, together with its Secret Key and produces the same Shared Secret as the server. Now, both the client and the server, share the same key and have completed the key exchange scheme. These exchanged messages are shown in Figure 1 as the Ephemeral PQ Key Generate, PQ Encapsulate and PQ Decapsulate operations.

3.1.4 Digital Certificates Support. Another important object that needs to be modified in order for TLS to work with post-quantum algorithms, is the digital certificate. These are objects that bound an entity, for example a server, with its public key by introducing a signature from a trusted third party. This can occur repeatedly by intermediate third parties, forming what is known as a "chain of certificates". The X.509 [14] is the standard that digital certificates usually follow on protocols like TLS. It contains useful information about the entity, for example: the entity's name, email, web address etc, the issue and expiration date of the certificate, the public key of the owner, the digital signature algorithm code that is used, the digital signature itself, etc.

For the production of these digital certificates using post-quantum cryptographic algorithms, the Open Quantum Safe's fork of OpenSSL [5] was used. Through this library we generated digital certificates using the OpenSSL's API with support for all the post-quantum algorithms that are evaluated in this paper. Our goal is to produce a digital certificate for the server, as it is the only one to authenticate itself. To achieve that, we introduced a base "Certificate Authority" (CA) that can issue other certificates making a chain of trust up until the server. In our paper, this chain is of length two, as the server's certificate is directly signed by the CA. To accomplish this, we created a digital certificate for the CA, which is self-signed and then we produced a digital certificate for the server which is then signed by the CA. Thus, a server certificate is produced, verifiable by our basic CA.

For the sake of simplicity, all the certificates in the chain employ the same signature algorithm each time. This is also the case for both the certificate's signature and the signing operation on the CertificateVerify message. For example, when measuring the performance of Dilithium2, CA's certificate and server's certificate have Dilithium2 signatures and the CertificateVerify message is signed using Dilithium2, as well.

3.2 Brief Experiment setup description

In order for the PQ TLS to be evaluated, we used a series of tools and two embedded systems for WolfSSL to run on, as well as a PC acting as a remote device for the boards to connect to through an Ethernet based network. In the evaluation experiments we perform a series of TLS connections and statistics are gathered using modified Wolfssl benchmark programs. For a thorough description on the used evaluation setup we refer the reader to Appendix B.

4 MEASUREMENTS AND EVALUATION

In this section we discuss the measurements on standalone PQ algorithm performance, comparing them with the traditional algorithms and with each other. Also, in this section the PQ TLS 1.3 evaluation is made and the TLS handshake time measurements are presented and discussed using various combinations of PQ KEM

Table 1: Traditional and Post-quantum Primitives

Algorithm	*NIST Level*	*Codepoint*	*Public Key (bytes)*	*Private Key (bytes)*	*Ciphertext (bytes)*	*Key Generate (ms)*	*Encapsulate (ms)*	*Decapsulate (ms)*
FFDHE [1]	0	0x0100	256	256	256	203.920	204.080 [4]	-
ECDHE [2]	0	0x0017	32	32	32	8.428	17.687 [4]	-
Kyber512	1	0x023A	800	1632	768	9.420	7.877	4.721
LightSaber	1	0x0218	672	1568	736	11.018	7.115	4.067
Kyber768	3	0x023C	1184	2400	1088	12.224	11.412	7.924
Saber	3	0x0219	992	2304	1088	13.650	10.129	6.824
Kyber1024	5	0x023D	1568	3168	1568	16.392	15.797	12.067
FireSaber	5	0x021A	1312	3040	1472	16.779	13.693	10.154
Algorithm	*NIST Level*	*Codepoint*	*Public Key (bytes)*	*Private Key (bytes)*	*Signature (bytes)*	*Key Generate (ms)*	*Sign (ms)*	*Verify (ms)*
RSA [3]	0	0x0285	256	256	256	12.853/450 [5]	448.250	12.500
ECDSA [2]	0	0x0206	32	32	32	8.428	12.305	25.193
Dilithium2	1	0xFEA0	1312	2528	2420	16.508	34.130 (11/172) [6]	13.793
Dilithium3	3	0xFEA3	1952	4000	3293	27.438	55.810 (23/270) [6]	23.669
Falcon512	1	0xFE0B	897	1281	666	1506.429	244.293	3.407
Falcon1024	5	0xFE0E	1793	2305	1280	3404.434	528.484	7.145

[1] 3072-bit, [2] secp256r1 curve, [3] 2048-bit, [4] key agreement time, [5] public / private key generation time, [6] average (min/max) of execution time over 1000 signatures.

and PQ authentication algorithms. Furthermore, the communication size that a PQ TLS handshake needs and the overall memory requirements of different PQ TLS combinations are also discussed in this section. Finally, energy consumption measururements of the TLS handshake using various PQ algorithm combinations are presented.

4.1 Post-quantum Cryptographic Algorithms

Using the tools provided by the wolfSSL library, the performance in terms of speed for the post-quantum algorithms, Kyber, Saber, Dilithium and Falcon can be measured. In particular, the *wolfcrypt-benchmark* program has been used in order to measure the performance of all the cryptographic functions in wolfCrypt, including public-key algorithms, symmetric algorithms, hash and MAC algorithms etc. Also, by making the necessary changes the *wolfcrypt-benchmark* program functionality has been extended in order to include the performance evaluation of the employed post-quantum algorithms. Note that the *wolfcrypt-benchmark* tool is using the underlying Real-time Clock (RTC) hardware module of the embedded system to measure time with a millisecond accuracy. In addition to the PQ algorithms, we measured the performance of four traditional algorithms, as points of reference. RSA-2048 and ECDSA with curve secp256r1 are used for benchmarking traditional authentication and Finite Field Diffie-Hellman Ephemeral (FFDHE)-3072 and ECDHE with curve secp256r1 for traditional key exchange. It should be noted that every cryptographic algorithm operation has been executed for 10 seconds and the average execution time of all executions is calculated and reported as a result.

In Table 1, the measured execution time on our target platform is presented, including each algorithm's public key and secret key sizes, cipher-text sizes for the KEMs and signature size for the authentication algorithms of both the post-quantum and the traditional algorithms. In Table 1 the claimed security level of each algorithm as defined by NIST is also reported. Level I, means that an attack on that parameter set would require the same or more resources as a key search on AES 128, Level III would require the same or more resource as a key search on AES 192 and Level V would require the same or more resource as a key search on AES 256. Level 0 means that this algorithm offers no quantum security. NIST, also defines security levels II and IV, but none of the pqm4 evaluated algorithms offer a parameter set of that security level. For comparison purposes it must be noted that the security level of RSA 2048 against classical attacks is 112-bits and the classical security level of FFDHE 3072 and curve secp256r1 is 128 bits.

4.1.1 Comparison Between Post-quantum and Traditional Primitives. As it can be observed from Table 1, the Sign speed of all PQ authentication algorithms outperforms RSA's Sign algorithm, except from Falcon1024 that is just 17.90% slower. Also it is clear that ECDSA's Sign operation is faster than both Dilithium and Falcon. In fact Dilithium2's Sign algorithm is ~2.8 times slower than ECDSA's, while the rest of the Sign algorithms are slower by several orders of magnitude, like Falcon1024 that is ~43 times slower. We note that the time required for the Dilithium Sign operation varied heavily, the minimum being 11ms and 23ms and the max being 172 ms and 270 ms for Dilithium2 and Dilithium3 respectively. These measurements were taken after performing 1000 operations and calculating the average time execution delay, displayed in Table 1 along with the minimum and maximum time delay appearing in these 1000 executions (shown in the parenthesis). The variation in execution time for Dilithium is expected since the NIST round 3 Dilithium digital signature implementation in the pqm4 library is not providing constant time execution.

Regarding the Verify algorithm, PQ algorithms perform much better than their Sign operation. It has been observed that the Verify speed of Falcon512 and Falcon1024 are ~3.6 times and ~1.7 times faster than the, already fast, RSA's Verify operation respectively. Also, Dilithium2 offers approximately the same Verify speed as RSA's Verify operation and Dilithium3 is ~1.89 times slower. As for the ECDSA's Verify operation, it is outperformed by both Dilithium, on security levels 1 and 3 and Falcon, on security levels 1 and 5.

Since PQ algorithms offer KEM functionality that differ from the classic key exchange approaches (e.g. using Diffie Hellman schemes), comparisons between traditional key exchange schemes and PQ KEM schemes can not be made directly. For a fair comparison, since the *wolfssl-benchmark* tool is capable of measuring the Key Generation and Key Agreement time separately for the traditional algorithms, we assume that one Encapsulation and one Decapsulation, for the PQ algorithms, is the equivalent measurement of "Key Agreement". Using that rationale, it can be observed that all post-quantum KEMs, on all security levels, are performing better than the traditional FFDHE 3072. It can also be observed that Key Generation time of post-quantum KEMs is comparable with the ECDHE Key Generate, the fastest being Kyber512 Key Generate that is ~12% slower than ECDHE Key Generate and the slowest being FireSaber Key Generate, that is ~99% slower than ECDHE Key Generate. However, it was also observed that the time that these PQ algorithms need for one Encapsulation and one Decapsulation is very fast and comparable to the Key Agreement time of ECDHE, sometimes even faster. We see that ECDHE's Key Agreement time delay is 17.687 ms while the total time of one Encapsulation and one Decapsulation operations on Kyber512, is 12.598 ms, ~29% faster than ECDHE. Even on higher security levels, like Kyber1024, which has the slowest Encapsulation/Decapsulation speed among all the PQ KEMs in our experiment, Key Agreement time is just 57% slower.

From the above measurements and discussion, it can be concluded that the KEM schemes included in our evaluation, have similar performance in term of speed to the traditional key exchange schemes. The most significant PQ primitive computation overhead is introduced by Falcon's Sign time and to a lesser extent by Dilithium's Sign time.

4.1.2 Comparison Between Post-quantum Primitives. Regarding the performance differences between the PQ KEM algorithms, it can be observed that Kyber and Saber have similar Encapsulation and Decapsulation speed, as they are alike in their designs and implementations. It is worth mentioning that Saber scales better in higher security levels. While Kyber and Saber Encapsulation/Decapsulation on security level 1, perform approximately the same in terms of the speed, on security level 3, Saber Encapsulation and Decapsulation are ~11% and ~13% faster than Kyber and on security level 5, Saber Encapsulation and Decapsulation are ~13% and ~16% faster than Kyber, respectively.

Regarding PQ authentication, the performance differences between our evaluated algorithms are more significant. Although Dilithium and Falcon offer the same security level (at security level 1), it is observed that Dilithium's Sign operation is significantly faster than Falcon's, Falcon512's Sign being ~7.2 times slower than Dilithium2's Sign. Simlarly, Falcon1024's Sign is ~9.5 times slower than Dilithium3's Sign at security levels 5 and 3 respectively. On the other hand, Falcon's Verify operation is considerably faster than Dilithium's. Falcon512 verifies ~4 times faster than Dilithium2 and Falcon1024 verifies ~3.3 times faster than Dilithium3. Note that Falcon1024 is on NIST's security level 5 compared to Dilithium3 that is on level 3. Dilithium specifications offer security level 5 support, but unfortunately pqm4 has not included such a Dilithium version in its implementations, probably due to its very large memory requirements. Digital Signature Key generation time delays are also shown on Table 1 but are not discussed further in this paper as they play no active role on the TLS handshake.

4.1.3 Comparison of Key and Signature Sizes. Key and cipher-text sizes, as well as in public key and signature sizes of the post-quantum algorithms are particularly important in a PQ TLS 1.3 implementation, because they are related to the amount of data being transferred over the network when the keys and ciphertexts are exchanged, in the Key Exchange phase and when the certificates and "CertificateVerify" message are being transmitted, later in the protocol. As expected, both KEM and signature schemes have sizes that are much larger (an order of magnitude larger) than those of traditional algorithms. ECDHE keys and ciphertexts are just 32 bytes while the smallest key in PQ KEMs is LightSaber's public key, which has 672 bytes. The same holds for PQ authentication, with ECDSA's public key being just 32 bytes and the smallest public key size in PQ algorithms is Falcon512's which is 897 byte.

It can be observed that Falcon has smaller sizes than Dilithium regarding both public keys and signatures. Falcon512 and Falcon1024 public keys are ~32% and ~8% smaller than Dilithium2 and Dilithium3 public keys, respectively. The results are even better for Falcon regarding signature sizes, with Falcon512 and Falcon1024 signatures being ~3.6 and ~2.6 times smaller than Dilithium2 and Dilithium3 signatures, respectively. In Figure 4, Public Key + Signature size in bytes are presented, for each of the algorithms being evaluated. This size metric, can be used for fair comparisons because it is the defining size of the certificates each algorithm will produce. Secret key sizes are of little importance to PQ TLS 1.3, as the secret key is never transmitted over the network, so we are not evaluating them any further in the paper.

Public key and signature sizes together with the overhead the Sign operation is introducing are the two main drawbacks of the evaluated post-quantum algorithms when compared to the traditional algorithms that are used currently in TLS 1.3.

4.2 Connection Time Delay of PQ TLS 1.3

Having implemented all the architectural changes described in previous sections and using the experimental setup described in subsection 3.2, in this Section we evaluate the TLS connection establishment using post-quantum algorithms on a local Ethernet network, regarding both their key exchange and the authentication. In this subsection we measure the performance of the post-quantum TLS 1.3 protocol and compare it with the performance of the traditional TLS 1.3, as well as to evaluate the PQ TLS 1.3 behaviour using different PQ algorithms. To have a point of reference, *ECDSA+ECDHE* and *RSA+ECDHE* measurements are used as the two baselines for our comparisons.

For the performance evaluation of the post-quantum TLS, we use the *wolfssl-tls-bench* program, provided by the wolfSSL library itself. This program originally simulated a client and a server on the same board, connecting to each other using in-memory transfers. The program was modified in order to benchmark individually either a client or a server that is connected to a remote machine. While *wolfssl-tls-bench* makes a number of connections, it simultaneously takes a series of measurements. More specifically, using a real-time operating system like FreeRTOS, *wolfssl-tls-bench* creates either a client thread or a server thread and it initiates the TLS handshake.

Table 2: PQ TLS 1.3 Handshake Time

Dig. Sign. Alg.	*Key Exch. Alg.*	*Stack Usage (bytes)*	*.bss Usage (bytes)*	*Comm. Size (bytes)*	*With HW acc. (ms)*		*Without HW acc. (ms)*		*Notation*
					client	*server*	*client*	*server*	
RSA [1]	ECDHE [2]	2368	0	2069	71.839	508.83	75.667	549.320	*RSA+ECDHE*
ECDSA [2]	ECDHE [2]	2368	0	1312	94.419	53.917	98.538	57.313	*ECDSA+ECDHE*
Dilithium2	Kyber512	52 472	0	8319	74.220	83.184	79.511	90.961	*Dil1+Kyb1*
Dilithium2	LightSaber	52 472	0	8159	75.224	82.426	79.652	90.929	*Dil1+Sab1*
Falcon512	Kyber512	3800	39 936	4365	42.964	274.382	43.809	277.910	*Falc1+Kyb1*
Falcon512	LightSaber	7968	39 936	4199	43.721	273.392	43.212	276.405	*Falc1+Sab1*
Dilithium3	Kyber768	80 120	0	11 409	107.992	125.071	115.538	140.414	*Dil3+Kyb3*
Dilithium3	Saber	80 120	0	11 217	108.143	125.055	114.684	140.224	*Dil3+Sab3*
Falcon1024	Kyber768	4304	79 872	7191	63.822	574.385	66.455	582.436	*Falc5+Kyb3*
Falcon1024	Saber	8992	79 872	6996	63.692	572.936	65.508	583.051	*Falc5+Sab3*
Dilithium3	Kyber1024	80 120	0	12 273	117.579	129.98	125.357	153.881	*Dil3+Kyb5*
Dilithium3	FireSaber	80 120	0	11 921	115.403	124.858	121.735	148.723	*Dil3+Sab5*
Falcon1024	Kyber1024	4816	79 872	8052	73.213	578.714	68.200	587.333	*Falc5+Kyb5*
Falcon1024	FireSaber	10 016	79 872	7702	70.703	576.648	65.211	585.846	*Falc5+Sab5*

[1] 2048-bit, [2] secp256r1 curve

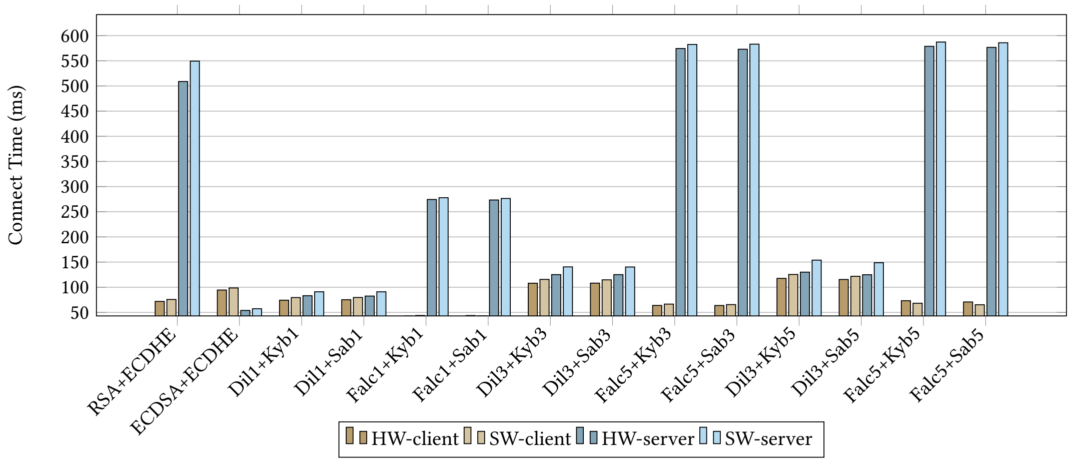

Figure 2: Average handshake time with TLS_AES_256_GCM_SHA384

On another machine, in our case a PC on the same local network, the same *wolfssl-tls-bench* program is executed as the other end of the TLS connection and the TLS 1.3 handshake is initiated. After the handshake is complete the benchmarked client and server exchange a number of bytes, the "application data". When they reach the data exchange limit (defined by the evaluator), both the server and the client, terminate the TLS connection and the underlying TCP socket. Then, the *wolfssl-tls-bench* program re-starts the TCP connection, the TLS handshake and re-exchanges "application data" etc. Connection benchmarking lasts for a specified number of seconds by the evaluator. During this time frame the benchmark program collects statistics and after the end of this time frame all connections are closed and statistics are printed including the number of handshakes during the time frame, the amount of application data that was exchanged, the time spent in handshaking etc. In the paper's benchmarking, the tests are running for 50 seconds and the "application data" that are exchanged in each session, are 4096 bytes long. As we are interested only in the TLS handshake metrics, we focus on the "Average Connection Time", that is the average time spent on handshaking of all the connections that were established at the specified time frame. We also, have performed tests for all the available Cipher Suites of the application data in TLS transmissions, that describe the symmetric encryption, the hash and MAC algorithms that have been used. However, as the statistics variations of the different Cipher Suites in the TLS record layer are trivial and unrelated to the PQ algorithms evaluation, in the paper, we display the statistics when using only the *TLS13-AES256-GCM-SHA384* Cipher Suite.

It should be noted that for simplicity, the client and the server have been configured to agree upon the public-key algorithms without an extra round-trip. In other words, the key share that the client sends with "ClientHello" message contains the key of a valid PQ key exchange group that the server can support and it is on the top of the list of preferences. The Public key algorithms (PQ or traditional) are chosen at compile-time. In Table 2, the experiments'

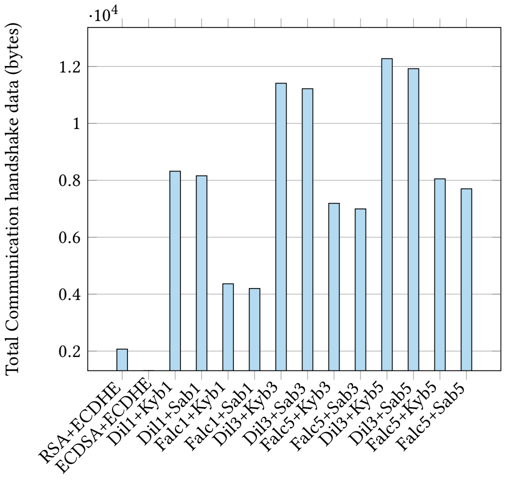

Figure 3: Communication sizes

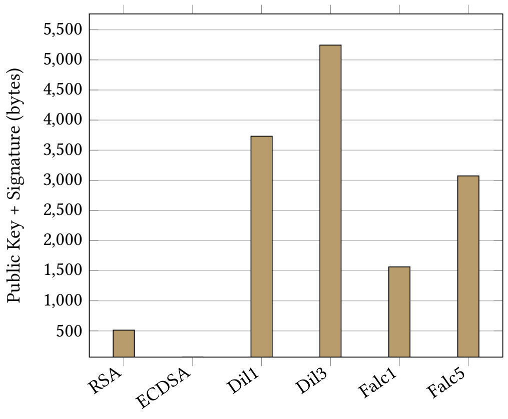

Figure 4: Public Key + Signature Size in bytes

measurement/results on NULCEO-F439ZI and on NUCLEO-F429ZI, are presented in detail.

4.2.1 Comparison Between Post-quantum and Traditional Algorithms. As shown in Figure 1 the client and server, perform different operations in TLS 1.3. Regarding the Key Exchange phase, the client performs a Key Generation and a Decapsulation operation while the server performs an Encapsulation. Regarding the Authentication phase, and specifically the experimental setup we use where the client is not authenticated and the certificate chain has depth 2, the clients needs to perform two Verify operations, one for the Certificate and one for the CertificateVerify message. The server, on the other hand, needs to perform a Sign operation and a Verify operation, to verify that its signature was created correctly. If we add to that, the high disparity between the execution times of the Sign and Verify operations, on our evaluated algorithms and especially for Falcon, we can see that the overall performance significantly differs when the board is acting as a client or as a server. We observe the same results, for the traditional algorithms too, as both ECDSA and RSA have disparities in Sign and Verify execution times, with RSA having a larger gap.

When comparing the handshake time of the client, between the traditional and post-quantum algorithms on security level 1 we see that *Dil1+Kyb1* or *Dil1+Sab1* is on the same level as *RSA+ECDHE* and outperforms *ECDSA+ECDHE.* (will add percentages later). As we move to the higher security levels of the PQ algorithms we see an increase on the average handshake time, mostly due to the costly Dilithium3 authentication operations, but still the handshake time is on acceptable levels with a maximum increase of 64% from *RSA+ECDHE* to *Dil3+Kyb5.* Comparing the results of Falcon, though, we see that all the combinations of different KEMs with Falcon512 or Falcon1024 outperforms *RSA+ECDHE* and *ECDSA+ECDHE*, except for *Falc5+Kyb5* that is just 2% slower.

Regarding the server, our results show a different picture. Although *RSA+ECDHE* has very slow handshake time, *ECDSA+ECDHE* is extremely fast. Both *Dil1+Kyber1* and *Dil1+Sab1* display around 54% increase of handshake time from *ECDSA+ECDHE* on the same security level (128-bit). On higher security levels the difference is more significant and *Dil3+Kyb5* is 2.4 times slower than *ECDSA+ECDHE*, although they operate at different security levels. Falcon, on the other hand, is performing much worse as a server, because of the very slow Sign operation. We see that even on the same traditional security level as *ECDSA+ECDHE*, *Falc1+Kyb1* and *Falc1+Sab1* performs 5 times slower. As we get to the higher security levels, Falcon1024 with all the KEMs takes almost double the time than Falcon512 with all the KEMs. Note that Falcon1024 is operating at NIST security level 5, in contrast to Dilithium3 that operates at NIST security level 3.

4.2.2 Comparison Between Post-quantum Algorithms. Regarding the time overhead that the two different KEMs (Kyber and Saber) are introducing, it can be observed that on security level 1, the TLS handshake time delay difference among combinations with different PQ signatures is minor, <2% on all combinations. Similarly, on security level 3, this difference is still minor, also less that 2%. On security level 5, the digital signatures combinations that use FireSaber as KEM, perform slightly better that other setups, but still approximately the time delay difference varies between 2% and 4%. It can also be observed that in the different PQ combinations used in PQ TLS 1.3 the main factor of speed overhead, is PQ authentication, namely the performance differences between Falcon and Dilithium. It can be observed that Falcon performs better when the board acts as a client and Dilithium performs better when the board is acting as a server. Comparing the client combinations that includes Dilithium with the ones that includes Falcon, on security level 1, it can be seen that *Falc1+Kyb1* and *Falc1+Sab1*, are ~72% faster than *Dil1+Kyb1* and *Dil1+Sab1.* Similarly, on security level 3 and 5, combinations with Falcon are ~41% and ~38% faster than the ones with Dilithium.

On the other hand, when KEM combinations with Falcon perform as a server, we see that on security level 1, *Falc1+Kyb1* and *Falc1+Sab1* are ~3.3 times slower than *Dil1+Kyb1* and *Dil1+Sab1.* On security level 3, server KEM combinations with Falcon are ~4.6 times slower than KEM combinations with Dilithium and on the highest security level 5 the KEM combinations with Falcon are ~4.5 times slower than the KEM combination with Dilithium.

4.3 Communication Sizes

As stated in subsection 4.1, the sizes of public keys, ciphertexts, certificates and signatures play an important role on evaluating TLS performance. With the introduction of PQ algorithms these sizes increased largely. We collected the number of bytes each peer is sending and receiving in a TLS handshake (communication sizes) with all the evaluated algorithm combinations and the outcome is presented on Table 2. Note that the size overhead introduced by TCP headers and other lower layer protocols is not considered in the measurements and we include only the sizes (in bytes) of the messages that the TLS1.3 sends or receives between the client and the server. It can be observed from Figure 3, where the above measurements are shown visually, that the communication size of Dilithium for all combinations is larger than the communication size of Falcon combinations when using the same KEMs on the same security levels (approximately 1.5 times larger). The combination *Falc1+Sab1* has the smallest communication size with 4199 bytes while *Dil3+Kyb3* has the largest communication size with 12273 bytes among the all the compared combinations. It can also be noted, that the communication data sizes of combinations with Kyber are larger than the ones with Saber when using the same signature algorithms on the same security levels, although their difference is small.

On security level 1, using the *Dil1+Kyb1* TLS 1.3 has 160 more bytes overhead than the *Dil1+Sab1* combination. On security level 3 the communication data size of TLS 1.3 using *Dil3+Kyb3* is 192 bytes more than TLS 1.3 with *Dil3+Sab3* and on security level 5, the communication size of TLS 1.3 with *Dil3+Kyb5* is 352 bytes more than TLS 1.3 with *Dil3+Sab5*. When using TLS 1.3 with traditional public key cryptography algorithms, the TLS 1.3 with *RSA+ECDSA* is communicating a total of 2069 bytes and TLS 1.3 with *ECDHE+ECDSA* just 1312 bytes. Compared to the PQ combinations, it can be observed that the communication size of TLS 1.3 with *Falc1+Sab1*, is already 3.2 times larger than the communication size with *ECDHE+ECDSA* and as security level increases this gap widens, with TLS 1.3 using *Dil3+Kyb3* having the overhead of an order of magnitude more bytes.

4.4 Memory Requirements

When compiling the code and flushing the binary file to the NUCLEO board, the STM32Cube IDE provides valuable information about the code size, the RAM usage, the stack requirements of each function etc. The code size is typically divided into three regions. *text*, that is the bytes that will load in the Flash memory, *data*, that is used for the C program's initialized data and *.bss* that is used for uninitialized data. Given that both the evaluation boards have a 2 MB flash memory, the *text* memory size in all algorithm combinations never superseded 20% of the total flash size. On the other hand, data and .bss are regions that are stored in RAM which is very constrained in embedded systems thus becoming the focus of our study. Falcon makes heavy use of the .bss region, as it can be seen in pqm4's benchmark table [7]. Dilithium, although not using the .bss region at all, it requires a substantial amount of static memory. The Stack memory requirements of each TLS 1.3 thread (client or server), together with .bss region, give us the total static memory allocations that our implementations require and they are

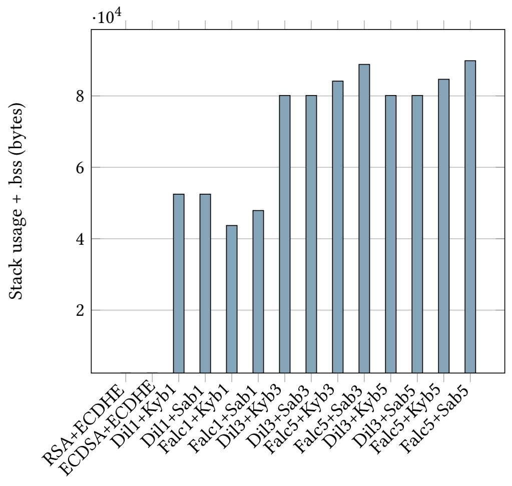

Figure 5: Stack + .bss usage by client or server thread in bytes

displayed for each PQ algorithm combination on Figure 5. Note that the evaluation boards, and typically any embedded system on the low end of resource constrained devices offer just a few kilobytes of RAM memory. Given that the evaluation boards have 192 KB of usable SRAM in the paper (we exclude the 64 KB of core coupled memory), the 80120 bytes of stack memory that Dilithium requires in the TLS1.3 implementation, is consuming 40% of the total available memory. TLS 1.3 with *Falc1+Kyb1* and *Falc1+Sab1* has smaller memory requirements using up 22% and 24% of total memory respectively, while on higher security levels TLS 1.3 with Falcon1024 combinations require substantially more memory, namely *Falc5+Kyb5* requiring 84656 bytes which is 43% of total memory. It should be noted that at the time of the paper's writing, the pqm4 ARM Cortex-M4 adapted implementations of Dilithium, was not memory optimised. The NIST PQC competition round 2 pqm4 implementations of the relevant algorithms have been memory-usage optimized which performed better in terms of memory usage compared to their round 3 counterparts. It can be assumed that in the future, pqm4 development team will optimise the NIST round 3 implementations too, thus bringing the memory usage of PQ authentication using Dilithium closer to traditional algorithms.

4.5 Energy Consumption of PQ TLS 1.3

IoT devices are generally sensitive to energy consumption. Therefore, it is critical to capture and analyse the energy consumption of post-quantum algorithms during the TLS 1.3 session establishment and secure data transmission. Given that in most realistic embedded devices usage scenarios the embedded system acts as a client, connected to a powerful server, in this paper's energy consumption evaluation we assume that the embedded system acts as a client. The energy consumption of TLS1.3 session is measured for different combinations of post-quantum algorithms Dilithium-Kyber, Dilithium-Saber, Falcon-Kyber, and Falcon-Saber at different security levels. Also, similar to the previous sections' performance measurements, the energy consumptions of two popular traditional

Table 3: PQ TLS 1.3 Client Nucleo-F429ZI Energy Consumption (no Hardware Cryptographic Acceleration)

Dig. Sign.Alg.	*Key Exch. Alg.*	*TLS13-AES256-GCM-SHA384*	
		Current(uA)	*Energy(uJ)*
ECDSA (secp256r1)	ECDHE (secp256r1)	2317.836	753.703
RSA 2048	ECDHE (secp256r1)	2156.809	538.558
Dilithium2	Kyber512	2142.380	562.131
Dilithium2	LightSaber	2090.763	549.560
Falcon512	Kyber512	2232.228	322.713
Falcon512	LightSaber	2148.399	306.361
Dilithium3	Kyber768	2119.726	808.199
Dilithium3	Saber	2160.082	817.499
Falcon1024	Kyber768	2177.889	477.614
Falcon1024	Saber	2190.953	473.632
Dilithium3	Kyber1024	2138.605	884.694
Dilithium3	FireSaber	2175.797	874.073
Falcon1024	Kyber1024	2186.601	492.116
Falcon1024	FireSaber	2143.990	461.379

algorithms *ECDSA+ECDHE* and *RSA+ECDHE* have been measured, acting as points of reference to the post-quantum TLS 1.3 measurements.The measurements of current and energy consumption are made using an actual network environment (the board is connected to a server PC through a router with Ethernet network cable), which includes the energy consumption of network communication. To be consistent with the other performance measurements (time, memory usage etc) we only focus on the TLS13-AES256-GCM-SHA384 Cipher Suite (For more information please see Appendix B).

4.5.1 Comparison Between Post-quantum and Traditional Algorithms. As can be seen in Table 3, there is small difference between the average current consumption of traditional algorithms and post-quantum algorithms. It can be observed that the average current consumption of TLS 1.3 handshake with *RSA+ECDHE* is slightly lower than that of PQ algorithms , and the average current consumption of *ECDSA+ECDHE* is slightly higher than the one of PQ algorithms. For instance, under TLS-AES128-GCM-SHA384 the average current consumption of *RSA+ECDHE* is 2156.809 uA, *ECDSA+ECDHE* is 2317.836 uA and the average current consumption among all PQ algorithms is 2158.951 uA.

4.5.2 Comparison Between Post-quantum Algorithms. In Table 3, it can be observed that the current consumption of a specific PQ algorithm at higher security level has no significant difference with the current consumption of the same algorithm at lower security level. For example, the current consumption of *Dil1+Kyb1*, *Dil3+Kyb3* and *Dil3+Kyb5* are 2149.552 uA, 2127.079 uA and 2151.783 uA respectively.

5 COMPARISON WITH OTHER WORKS

In order to fully evaluate our described embedded systems PQ TLS 1.3 design and implementation, in this section, we provide comparisons with other relevant works. Note that there is no one-to-one match between our proposed work and any other relevant papers on PQ TLS ie. using low end resource constrained embedded systems for TLS version 1.3, with round 3 NIST finalists and with ARM cortex M4 optimized PQ algorithm implementations (though the pqm4 library). In Table 4, comparisons are provided between the proposed work and the work of Paul et al. [27] and Burstinghaus et al. [16] that bare some similarities to the proposed work and up to a point supplement to our design.

Paul et al. [27] approach aims at higher end embedded system devices and PCs. The closest match to our proposal is the results on a Raspberry Pi 3 Model B (RPi3), equipped with an ARM Cortex-A53 quad-core processor running at 1.2 GHz with 1 GB of RAM. The main focus of [27] is on the feasibility of a novel technique to integrate PQ authentication in certificate chains, thus not focusing on optimising performance. It should be pointed out that in [27] certificate chains of depth 3 are used, compared to ours work where depth 2 certificates are used. Also, Paul et al. is using mixed certificate chains, i.e. chains with different signature algorithms in each certificate, and regular certificate chains, i.e. chains with the same signature algorithm. Although in [27] the primary focus is not optimising the performance, the authors provide handshake measurements using the same TLS version as our work, TLS 1.3, and using the same measurement setup, an embedded system connected to a local network via the Ethernet interface, establishing TLS connections to a PC. In [27] tests with remote connections are also performed to include network latency measurements. However, a major difference from our work, is that Paul et al. evaluates a *mutually authenticated* TLS 1.3 handshake, where both client and server authenticate themselves, whereas in our work server based authentication is evaluated.

As seen in Table 4 our work is compared with the [27] TLS1.3 implementations using Kyber512 for KEM and Dilithium2, Falcon512 and SPHINCS+128f on authentication. It can be noted that all algorithm comparison measurements in [27] are on NIST security level 1.

The work by Burstighaus et al. [16] , on the other hand is focused on resource constrained devices with the most powerful of them being a Raspberry Pi (RPi3), (the same device as in [27]). Apart from that the authors in [16] also evaluated PQ TLS on an ESP32 device equipped with an Xtensa dual-core 32-bit LX6 processor operating at 240 MHz with 520 KB of SDRAM and 16 MB of flash memory, as well as two other devices, too constrained to be fairly compared to our work. However, in contrast to [27] and our work, [16] is focused on TLS 1.2 and not TLS1.3. Regarding the PQ algorithms, Burstighaus et al. evaluated Kyber512 for KEM and SPHINCS+128f for authentication, both algorithms being at NIST security level 1. Similarly to [27] and our work, Burstighaus et al. evaluated TLS with traditional algorithms, specifically ECDHE with ECDSA, both with curve secp256r1. Also, it must be noted that Burstighaus et al. provide results on handshake routines and cryptographic primitives and not on the network stack and network response times

From Table 4 it can be observed that the TLS handshake time in all the systems, except for the ESP32, are in the same order of magnitude. Regarding *Dil1+Kyb1* we see that the measurements of [27] on Raspberry Pi performs 2.5 times faster than the NUCLEO-F439ZI when acting as a client and 3 times faster than the NUCLEO-F439ZI when acting as a server. As for the *Falc1+Kyb1*, the NUCLEO-F439ZI even outperforms the Raspberry Pi by 32% when acting as a client but is 4.4 times slower than Raspberry Pi when acting as a server. Regarding the traditional TLS using *ECDHE+ECDSA*, that all the works have measurements for, we can see that the NUCLEO-F439ZI when acting as client is 3.2 times slower than

Table 4: PQ TLS 1.3 Comparison with other works; rounded for better comparison

Dig. Sign. Alg.	*Key Exch. Alg.*	*RPi3 [27] (ms)*		*RPi3 [16] (ms)*		*NUCLEO-F439ZI (ms)*		*ESP32 [16] (ms)*	
		client	*server*	*client*	*server*	*client*	*server*	*client*	*server*
ECDSA [1]	ECDHE [1]	29	28	49	43	94	54	1100	890
Dilithium2	Kyber512	29	28			74	83		
SPHINCS+128f	Kyber512	280	290	67	840			970	23 000
Falcon512	Kyber512	63	62			43	274		

[1] secp256r1

the Raspberry Pi implementation of [27] and 2 times slower than the Raspberry Pi implementation of [16]. The measurements on the ESP32 are much slower than the results of our work, namely *ECDHE+ECDSA* being 12 times slower as a client and 16 times slower as a server compared to ours. The difference on the TLS performance of the Raspberry Pi from these works might be due to the different versions of the algorithms being used. As noted in [27] and [16], the reference PQ algorithms implementations have been used.

6 CONCLUSION

In this paper, we discussed the importance of adopting a quantum-safe version of TLS, targeting embedded systems in response to the imminent threats of a possible quantum attack. We integrated post-quantum algorithms into the popular WolfSSL library, implementing TLS optimized for embedded systems, and measured the time delay, memory, communication and energy consumption overhead the introduced PQ algorithms. We used Dilithium, and Falcon for the authentication and Kyber and Saber for Key Encapsulation (KEM), all being promising lattice-based PQ algorithms. We used two boards, both equipped with an ARM Cortex-M4, and remote machine, a PC, for each board to connect to, through the Ethernet interface.

Our results show that the largest impact on performance is introduced by the authentication algorithms, as the KEM algorithms provide similar performance to traditional algorithms. Also, the results show that when the board acts as a client, Falcon performs much better than Dilithium, while the opposite is happening when the boards acts as a server. Regarding memory usage, it is shown that the current implementations, specifically Dilithium's pqm4 implementation, is not memory optimized, thus introducing a large memory overhead. Another significant finding, is that the communication size, meaning the total bytes a peer has to exchange during a TLS handshake, is considerably increased when PQ algorithms are used compared to traditional TLS 1.3. Finally, regarding the energy consumption evaluation, it is shown that the average current consumption is independent of the different PQ combinations, as well as the different security levels since the current consumption is probably dominated by the communication transmission cost.

ACKNOWLEDGMENTS

In this paper, G. Tasopoulos and A. Fournaris has received funding from the European Union's Horizon 2020 research and innovation program CONCORDIA under grant agreement No 830927 and from the European Union's Horizon 2020 research and innovation programme ENERMAN under grant agreement No 958478.

REFERENCES

[1] [n.d.]. Focus on Cortex-M4. https://csrc.nist.gov/CSRC/media/Presentations/the-2nd-round-of-the-nist-pqc-standardization-proc/images-media/moody-opening-remarks.pdf. Accessed: 26-1-2021.
[2] [n.d.]. FreeRTOS. https://www.freertos.org. Accessed: 17-11-2021.
[3] [n.d.]. mbedTLS, Library. https://github.com/ARMmbed/mbedtls. Accessed: 26-1-2021.
[4] [n.d.]. OQS-OpenSSH-fork. https://github.com/open-quantum-safe/openssh. Accessed: 4-2-2021.
[5] [n.d.]. OQS-OpenSSL-fork. https://github.com/open-quantum-safe/openssl. Accessed: 4-2-2021.
[6] [n.d.]. PQClean. https://github.com/PQClean/PQClean. Accessed: 6-7-2021.
[7] [n.d.]. Pqm4's benchmark.md table in Github. https://github.com/mupq/pqm4/blob/master/benchmarks.md. Accessed: 17-11-2021.
[8] [n.d.]. WolfSSL-github-repoitory. https://github.com/wolfSSL/wolfssl. Accessed: 17-11-2021.
[9] [n.d.]. wolfSSL, Library. https://www.wolfssl.com/. Accessed: 17-11-2021.
[10] 2021. TLS 1.3 adoption according to the Internet Society Pulse. https://pulse.internetsociety.org/technologies. Accessed: 6-7-2021.
[11] Frank et al. Arute. 2019. Quantum supremacy using a programmable superconducting processor. *Nature* 574, 7779 (01 Oct 2019), 505–510. https://doi.org/10.1038/s41586-019-1666-5
[12] Jean-Philippe Aumasson, Daniel J Bernstein, Christoph Dobraunig, Maria Eichlseder, Scott Fluhrer, Stefan-Lukas Gazdag, Andreas Hülsing, Panos Kampanakis, Stefan Kölbl, Tanja Lange, et al. 2019. SPHINCS. (2019).
[13] Ward Beullens. 2021. Improved Cryptanalysis of UOV and Rainbow. In *EUROCRYPT (1) (Lecture Notes in Computer Science, Vol. 12696)*. Springer, 348–373.
[14] Sharon Boeyen, Stefan Santesson, Tim Polk, Russ Housley, Stephen Farrell, and David Cooper. 2008. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280. https://doi.org/10.17487/RFC5280
[15] Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. 2018. CRYSTALS-Kyber: a CCA-secure module-lattice-based KEM. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 353–367.
[16] Kevin Bürstinghaus-Steinbach, Christoph Krauß, Ruben Niederhagen, and Michael Schneider. 2020. Post-Quantum TLS on Embedded Systems. Cryptology ePrint Archive, Report 2020/308. https://eprint.iacr.org/2020/308.
[17] Léo Ducas, Eike Kiltz, Tancrede Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. 2018. Crystals-dilithium: A lattice-based digital signature scheme. *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2018), 238–268.
[18] Adam Dunkels. 2001. Design and Implementation of the lwIP.
[19] Morris Dworkin. 2015. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. https://doi.org/10.6028/NIST.FIPS.202
[20] Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, and Frederik Vercauteren. 2018. Saber: Module-LWR based key exchange, CPA-secure encryption and CCA-secure KEM. In *International Conference on Cryptology in Africa*. Springer, 282–305.
[21] P. A. Fouque, J. Hoffstein, P. Kirchner, V. Lyubashevsky, T. Pornin, T. Prest, T. Ricosset, G. Seiler, W. Whyte, and Z. Zhang. 2020. FALCON: Fast-Fourier lattice-based compact signatures over NTRU. Submission to the NIST's post-quantum cryptography standardization process. https://falcon-sign.info/falcon.pdf.
[22] Paul E. Hoffman. 2002. SMTP Service Extension for Secure SMTP over Transport Layer Security. RFC 3207. https://doi.org/10.17487/RFC3207
[23] Ralph Holz, Jens Hiller, Johanna Amann, Abbas Razaghpanah, Thomas Jost, Narseo Vallina-Rodriguez, and Oliver Hohlfeld. 2020. Tracking the Deployment of TLS 1.3 on the Web: A Story of Experimentation and Centralization. *SIGCOMM*

Comput. Commun. Rev. 50, 3 (July 2020), 3–15. https://doi.org/10.1145/3411740.3411742

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

[25] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. 2019. pqm4: Testing and Benchmarking NIST PQC on ARM Cortex-M4. Cryptology ePrint Archive, Report 2019/844. https://eprint.iacr.org/2019/844.

[26] Sebastian Paul, Yulia Kuzovkova, Norman Lahr, and Ruben Niederhagen. 2021. Mixed Certificate Chains for the Transition to Post-Quantum Authentication in TLS 1.3. Cryptology ePrint Archive, Report 2021/1447. https://ia.cr/2021/1447.

[27] Sebastian Paul, Yulia Kuzovkova, Norman Lahr, and Ruben Niederhagen. 2021. Mixed Certificate Chains for the Transition to Post-Quantum Authentication in TLS 1.3. Cryptology ePrint Archive, Report 2021/1447. https://ia.cr/2021/1447.

[28] Abbas Razaghpanah, Arian Akhavan Niaki, Narseo Vallina-Rodriguez, Srikanth Sundaresan, Johanna Amann, and Phillipa Gill. 2017. Studying TLS Usage in Android Apps. In *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies* (Incheon, Republic of Korea) *(CoNEXT '17)*. Association for Computing Machinery, New York, NY, USA, 350–362. https://doi.org/10.1145/3143361.3143400

[29] Eric Rescorla. 2000. HTTP Over TLS. RFC 2818. https://doi.org/10.17487/RFC2818

[30] Eric Rescorla. 2018. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446. https://doi.org/10.17487/RFC8446

[31] Peter W. Shor. 1994. Algorithms for Quantum Computation: Discrete Logarithms and Factoring. In *FOCS*. IEEE Computer Society, 124–134.

[32] Dimitrios Sikeridis, Panos Kampanakis, and Michael Devetsikiotis. 2020. Assessing the Overhead of Post-Quantum Cryptography in TLS 1.3 and SSH. https://doi.org/10.1145/3386367.3431305

[33] Douglas Stebila and Michele Mosca. 2016. Post-quantum Key Exchange for the Internet and the Open Quantum Safe Project. In *SAC (Lecture Notes in Computer Science, Vol. 10532)*. Springer, 14–37.

A WOLFSSL LIBRARY MAIN COMPONENTS

Wolfssl consists of the following main components: *WolfCrypt*: This component includes all the traditional cryptographic algorithms, public-key and symmetric-key, as well as hash algorithms, MAC algorithms and programs that handle certificate and key files. It provides optimised code for a series of architectures as well as hardware support for selected platforms.

WolfSSL: This component includes all the protocol related codes, that implement the TLS protocol itself as well as other protocols, like Datagram Transport Layer Security (DTLS). It includes all the settings and preferences of the TLS protocol and the interfaces to communicate either to a lower level protocol, like TCP or to a higher level, like an operating systems for example FreeRTOS [2].

Utilities: This component includes all non-essential utilities like benchmarks or test programs verifying the correct functionality of the wolfSSL library. In this paper, some of these benchmark programs have been used in order to measure the performance of PQ cryptographic algorithms or the TLS protocol itself. We particularly used two toolsets as follows:

- *Wolfcrypt-benchmark* is a benchmark tool used to measure the performance of all the enabled cryptographic algorithms and provides relevant statistics. This program has been used in the paper as basis for taking time measurement on post-quantum algorithms in order to compare them with the traditional ones.
- *Wolfssl-tls-bench* is another benchmarking tool that measures and provides a series of metrics regarding TLS sessions. It can either make use of an operating system, like FreeRTOS, and simulate a server and a client connecting through TLS on the same machine or it can be run on different machines, one being the server and the other being the client to provide a realistic benchmark scenario. The tool repeatedly establishes TLS sessions (by running the TLS handshake), exchanges data for a given time period and then it provides statistics about the established connections eg. the average time spent on handshaking, the size of exchanged data etc. This program has been used in this paper in order to measure the performance of TLS protocol while using post-quantum algorithms and to compare the results with the TLS protocol using traditional algorithms.

B EXPERIMENT SETUP

B.1 Embedded Systems

The selection of the microcontroller was made in line with the NIST competition requirements to focus on ARM Cortex-M4 based embedded devices. We choose two embedded systems by STMicroeletronics with the ARM Cortex-M4 microcontroller, NUCLEO-F439ZI and NUCLEO-F429ZI. Both systems have a 32bit ARM Cortex-M4 processor running at 180 MHz with a SRAM of 256 KB and 2 MB of Flash memory to store the program code. The SRAM is divided into two sections, a 64 KB Core Coupled Memory (CCM) SRAM and a 192 KB regular SRAM. The first section might be faster, but its small size made we work entirely on the second one, the 192 KB regular SRAM. Both boards are also equipped with an Ethernet interface and with a true hardware random generator. However, the NUCLEO-F439ZI is equipped with a hardware cryptographic acceleration cell while the NUCLEO-F429ZI does not support such hardware acceleration. Hardware acceleration in the first board is supported for AES 128, 192, 256, Triple DES, HASH (MD5, SHA-1, SHA-2), and HMAC.

The selection of these two boards allows us to evaluate the post-quantum TLS's performance on ARM Cortex-M4 processor in two scenarios, TLS 1.3 on embedded processors with hardware accelerated cryptography operations and TLS 1.3 on embedded processors without hardware accelerated cryptography operations. Thus, by comparing the results from the two boards, we can measure the acceleration, in terms of time spent on handshake, that these hardware components can provide. We should also note that even though the PQ algorithms does not make use of these hardware accelerated primitives directly, as they use the Keccak primitives, SHA-3 and SHAKE-256 [19], the handshake itself makes use of the hardware accelerated primitives, thus the hardware cryptographic accelerator can boost performance of the PQ TLS handshake.

As discussed on the work of Sikeridis et al. [32], changing the TCP_WND parameter of the underlying TCP implementation, impacts the performance of the TLS handshake. We experimented with different values of the TCP_WND parameter, usually in multiples of TCP_MSS that is the maximum segment size of each packet, and we observed that on some values, the handshake was particularly slower than the others. Because the network analysis is not on the scope of this work, we choose a value that the TLS handshake performed well, namely TCP_WND = $2 \times$ TCP_MSS without further evaluation.

B.2 PC as a Remote End Device

To conduct real TLS connection experiments, an end device beyond the embedded system boards is needed. Such device, acts as a server when the boards are configured as a client and acts as a client when the boards are configured as a server. A PC connected to the local network that is running Ubuntu 20.04 in x86_64 architecture and is equipped with an Intel i7-1165G7 with 8 cores running at 2.8 GHz is setup in our experiment to play this role. This machine is connected through the Ethernet interface to the same access point as the board, with a mean Round-Trip Time (RTT) of 0.493 ms. As this paper focuses on the embedded system's performance, we are not taking measurements of the PC's TLS performance. Without loss of generality, we consider the embedded system's other communication end of the TLS connection as a powerful machine that has very high performance (trivial TLS time delay) and thus does not impact the embedded system measurements.

B.3 Development Framework

All embedded system code development was done using the official IDE development framework for the NUCLEO boards, the STM32CUBE IDE. This IDE made the porting of wolfSSL on the embedded system simple, using the official supported wolfSSL libraries. All post-quantum algorithm integration and TLS modifications in the embedded wolfSSL vanilla version were made using the STM32CUBE IDE. Regarding the PC wolfSSL, Eclipse CDT was used along with the wolfSSL's git release version 4.7.0 that was also modified to support PQ TLS 1.3.

B.4 Memory Management

In resource constrained embedded systems, memory resources are limited, thus appropriate memory management is crucial. In this paper's experiments, the ST NUCLEO embedded systems use FreeRTOS [2] as a real-time operating system, so the memory is managed by this OS. FreeRTOS is using the Stack and Heap scheme thus, every newly created thread (task) is given a region in memory as Stack memory and this regions becomes unavailable for the rest of the embedded system's memory, serving as Heap memory for dynamic allocations. As more threads are created and the total Stack size is increased, the available Heap memory is reduced. In the proposed wolfSSL implementation, we determine at compile time, how much of the embedded system memory will be assigned to the OS in order to manage threads, making this the Total Heap that is available to the threads on run time. Also at compile time, we determine how much Stack memory each of the created threads will use. This Stack memory, of course is subtracted from the total available Heap memory, and is managed by each created thread. Necessary trade-offs need to be made between the available Heap memory and the required Stack memory of the threads, so that every thread has sufficient memory to function properly and still allow the OS enough memory space for dynamic allocation. In our experiments, the amount of available memory assigned to the OS to manage threads and the exact Stack usage, in pair with the size of .bss region is reported on Table 2 and it is presented in Figure 5. Regarding the PC device, memory management is not an issue, as it has plenty of available memory, that is managed by the OS without any direct intervention by us.

B.5 Energy Consumption Measurements setup

The measurements were made using the X-NUCLEO-LPM01A expansion board and the STM32CubeMonitor-Power software offered by ST. Table 3 illustrates the average current and total energy consumption of various PQ algorithms with different security levels during TLS 1.3 session establishment. The column "Current" in Table 3 corresponds to the average current (measured in micro-Ampere) consumed during the handshake process, captured from STM32CubeMonitor-PWR by selecting the time frame of a TLS handshake. The column "Energy" in Table 3 corresponds to the total energy consumption (measured in micro-Joule) per TLS handshake, which is calculated by multiplying the average current consumption, supply voltage and handshake time