Performance Evaluation of Post-Quantum TLS 1.3 on Resource-Constrained Embedded Systems

George Tasopoulos[1], Jinhui Li[2], Apostolos P. Fournaris[1], Raymond K. Zhao[2], Amin Sakzad[2], and Ron Steinfeld[2]

[1] Industrial Systems Institute/Research Center ATHENA, Patras, Greece,
g.tasop@protonmail.com, fournaris@isi.gr
[2] Faculty of Information Technology, Monash University, Clayton, Australia,
jinhui0018@gmail.com, {raymond.zhao,amin.sakzad,ron.steinfeld}@monash.edu

Abstract. Transport Layer Security (TLS) constitutes one of the most widely used protocols for securing Internet communications and has also found broad acceptance in the Internet of Things (IoT) domain. As we progress toward a security environment resistant to quantum computer attacks, TLS needs to be transformed to support post-quantum cryptography. However, post-quantum TLS is still not standardised, and its overall performance, especially in resource-constrained, IoT-capable, embedded devices, is not well understood. In this paper, we showcase how TLS 1.3 can be transformed into quantum-safe by modifying the TLS 1.3 architecture in order to accommodate the latest Post-Quantum Cryptography (PQC) algorithms from NIST PQC process. Furthermore, we evaluate the execution time, memory, and bandwidth requirements of this proposed post-quantum variant of TLS 1.3 (PQ TLS 1.3). This is facilitated by integrating the *pqm4* and *PQClean* library implementations of almost all PQC algorithms selected for standardisation by the NIST PQC process, as well as the alternatives to be evaluated in a new round (Round 4). The proposed solution and evaluation focuses on the lower end of resource-constrained embedded devices. Thus, the evaluation is performed on the ARM Cortex-M4 embedded platform NUCLEO-F439ZI that provides 180 MHz clock rate, 2 MB Flash Memory, and 256 KB SRAM. To the authors' knowledge, this is the first systematic, thorough, and complete timing, memory usage, and network traffic evaluation of PQ TLS 1.3 for all the NIST PQC process selections and upcoming candidate algorithms, that explicitly targets resource-constrained embedded systems.

1 Introduction

In 1994, Peter Shor described an algorithm on quantum computers [36] that solves the mathematical problems of integer factorisation and discrete logarithm in polynomial time. This alerted the cryptography community, since those hard-to-solve (by traditional computers) mathematical problems constitute the core of the majority of existing public key cryptography solutions in the security world. While the quantum computers existing today may not be powerful enough to

solve real-world problems, they pave the way for large-scale quantum computers in the future [14] that can easily break (e.g. recover the private keys) many existing security protocols and public key cryptography solutions.

In response to this potential new cryptographic reality, several standardisation bodies (eg. NIST, IETF, ETSI) have started working on the transition of cryptography to the post-quantum era [1,13,25]. In 2017, the US National Institute of Standards and Technology (NIST) initiated a still ongoing evaluation and standardisation process to select the next generation of industry-standard public key cryptographic primitives [4]. At the time of this paper's writing, NIST has selected 4 algorithms for standardisation: 3 Digital Signatures and 1 Key Encapsulation Mechanism (KEM). Additionally, NIST recently announced an extra round of evaluation for some of the KEMs that did not get selected (Round 4 of the NIST process) and also a Call for Proposals for Digital Signatures in order to diversify NIST's Digital Signature portfolio [1]. In the meantime, the Internet Engineering Task Force (IETF) is in the process of standardising the integration of post-quantum algorithms in popular security protocols [39].

One of the most prominent security protocols is the Transport Layer Security (TLS) protocol. Due to its wide acceptance and usage in secure communications, it is crucial to make TLS quantum computer safe (resistant against quantum computer cryptanalytic attacks), by integrating post-quantum cryptography (PQC) algorithms into the protocol structure. However, this PQ TLS integration is expected to have a non-trivial overhead on the protocol's performance. There are already several efforts to integrate PQC algorithms in TLS [38] and measure the overhead it introduces, with respect to a number of performance metrics: execution speed, memory requirements, communication size, and code size [37,32,18,31,21,15].

Understanding the PQ TLS performance overhead becomes even more important when dealing with resource-constrained devices and this may be critical to the final adoption of PQ TLS on such devices. However, prior works on evaluating the performance of PQ TLS either focused on high-end devices studying the impact of other factors such as network conditions [37,31,21], or focused on embedded system devices that are not resource-constrained [32,15]. Prior works that evaluated PQ TLS on resource-constrained devices [18], are limited to a small set of PQC algorithms, mainly because their goal was to highlight the feasibility of such an integration rather than provide a complete analysis.

In this paper, we make a thorough and analytic study of the PQC integration in the TLS 1.3 protocol for resource-constrained embedded system devices and their overall performance impact. We perform an exhaustive and systematic analysis of the PQC performance overhead in TLS 1.3 on embedded systems, in terms of execution speed, memory requirements, and communication size, for the majority of post-quantum public-key algorithms, including the NIST selected algorithms for standardisation, the algorithms that progressed to Round 4, as well as some algorithms from NIST PQC Round 3. Namely, we have integrated the KEMs CRYSTALS-Kyber, SABER, NTRU, SIKE, BIKE,

HQC, NTRU LPRime, and FrodoKEM, and the Digital Signature algorithms CRYSTALS-Dilithium, Falcon, SPHINCS+, and Picnic3.

More specifically, by adopting the wolfSSL open-source TLS library, we propose a PQC enhanced TLS 1.3 design. We integrate into our design the above-mentioned PQC algorithms using the *pqm4* [22] and *PQClean* [29] open-source PQC libraries. The design is implemented in a resource-constrained device equipped with an ARM Cortex-M4 microcontroller. We include all the security levels that could fit in the RAM of this constrained evaluation board, and we evaluate the protocol's performance after the integration of all the algorithms selected for standardisation by the NIST PQC process and almost all of the algorithms that proceed to Round 4. Note that we also include in our study some algorithms that have been in Round 3 for completeness [3], including those are promising for a re-submission to the new standardisation Round 4 for Digital Signatures announced by NIST. Also, note that algorithms such as Rainbow and Classic McEliece have not been included in the study due to their large memory requirements, even on their lowest security levels, that prohibits their integration in the TLS 1.3 protocol. In addition, Rainbow's raised security concerns for some specified security parameters are taken into our consideration [16]. A recent work [19], was able to recover the secret key from SIKE in all security levels. Although it is possible that SIKE is now completely broken, we include its measurements for the sake of completeness. Network communication and routing functionality are provided through the lwIP library [22]. We also assume that both peers of the TLS 1.3 handshake, client and server, are mutually authenticated. According to the authors' knowledge, this work constitutes the first comprehensive and all-inclusive study on PQ TLS 1.3 for resource-constrained embedded systems that takes into account the latest results of NIST standardisation efforts (Selected Algorithms 2022 and Round 4 announcements).

The rest of the paper is organised as follows: In Section 2, we make a review of popular open-source PQC libraries and related research work, and we highlight the research gaps that our paper will fill. In Section 3, background information on TLS 1.3 is provided. In Section 4, the proposed design/architecture of the PQC integreated TLS 1.3 protocol is presented, providing the architectural changes that have been made in order to add PQC capabilities to TLS 1.3, along with various implementation details. In Section 5, the experimental evaluation process, measurements, results, and analysis are provided. Finally, Section 6 concludes the paper.

2 Popular PQC Libraries and Related Research Work

The NIST PQC standardisation process has sparked a bloom of research in the field of post-quantum cryptography. Apart from the PQC algorithms that were developed and submitted to the process, other research that emerged in recent

[3] We refer the reader to the Appendix for the rest of the evaluated algorithms from Round 3.

years focused on prototyping, integration into popular protocols, and measuring the performance overhead that these PQC algorithms will introduce.

Apart from the reference and x86 optimised PQC algorithm implementations offered by the submission team of each candidate PQC algorithm, other software libraries have also been provided by the PQC research community. *PQClean* [29] focuses on the prototyping of PQC algorithms without relying on external software library dependencies, thus offering standalone implementations of PQC algorithms.

Regarding embedded systems, a project named *mupq* has been developed in order to provide optimised PQC libraries targeting a number of embedded system processors or even Field Programmable Gate Arrays (FPGAs). After a call from the NIST PQC process to introduce evaluations on Cortex-M4 processors [5], as an outcome of the work in [28] (which is part of the *mupq* project), the *pqm4* library has been developed. This library includes implementations based on the Reference Implementations of the official releases of the algorithms, but with a focus on speed and size, and uses optimised assembly code for the Cortex-M4 processor. As the NIST PQC process advances, the *pqm4* library is regularly updated with newer versions of the code or with any changes the algorithm specifications introduce. All these updates are kept in an open-source git repository available online at [27].

In regards to TLS design and open-source implementation, the Open Quantum Safe (OQS) [38] project has developed a cryptographic library named *liboqs*, that has collected implementations of PQC algorithms, mostly from *PQClean*. By leveraging the developed *liboqs* library, this project integrated PQC algorithms into popular security protocol libraries, such as OpenSSL [7] and OpenSSH [6], making it possible to use these protocols with PQC algorithms. In the work by Sikeridis et al. [37], the OpenSSL and OpenSSH forks of OQS were used to measure the overhead that was introduced with the post-quantum integration on these two protocols in realistic network conditions. Doring et al. [21] performed similar experiments by using the same library and evaluated more PQC algorithms, but on the same machine without real-world network conditions. Paquin et al. [31] performed benchmarks of PQ TLS by using OQS project with various PQC algorithms, on machines within an emulated network that enabled the authors to experiment with different network parameters. All these works evaluated PQ TLS. However, they focused only on medium or high-resource systems like laptops, PCs, and servers.

However, there exists research works that focused on embedded systems. Bürstinghaus-Steinbach et al. [18] employed an implementation of TLS with PQC algorithms to take measurements on embedded devices. The authors adopted TLS version 1.2 and integrated CRYSTALS-Kyber as a KEM and SPHINCS+ as a Digital Signature. They showed the feasibility of such integration and gathered primitive performance measurements on PQ TLS. Paul et al. [32] introduced a migration strategy towards post-quantum authentication by using PQC algorithms in mixed certificate chains. The authors also evaluated the performance of post-quantum TLS 1.3 on a server, on a PC, and on a Rasp-

berry Pi. Barton et al. [15] used the same Raspberry Pi device to evaluate PQ TLS by using OQS with various PQC algorithms. It should be noted that the Raspberry Pi device used in [32] and [15] is equiped with an ARM Cortex-A53 processor running at 1.2 GHz and with 1 GB of RAM. Although this device is considered as an "embedded device", it belongs to the higher end of such devices with resources capable of running its own compiler and even a full Operating System.

To the authors' knowledge, this paper constitutes the first systematic and thorough architectural adaptation, implementation, and performance evaluation of the PQ TLS 1.3 based on the popular wolfSSL library on a resource-constrained device. Our work integrates the selected algorithms for standardisation by the NIST PQC process, Round 4 candidates, and alternative algorithms in Round 3, explicitly targeting low resource embedded systems.

3 Background

3.1 TLS Protocol

One of the major security protocols that are threatened by a potential 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. Hypertext Transfer Protocol Secure (HTTPS) protocol for secure website transfer [34], secure connection to mail servers [24], as well as secure Internet access for smartphone apps [33], are some of the many use cases of TLS. TLS version 1.3 [35] has been standardised in 2018 and introduces many important changes over the previous version, TLS 1.2.

The adoption of TLS 1.3 is at an adequate level due to the high centralisation of the Internet and the long duration of the draft's evaluation [26]. In fact, the *Internet Society Pulse* reported a nearly 60% adoption of TLS 1.3 by the top $1,000$ websites globally [8].

TLS is a security protocol designed to provide secure communication over a computer network. It is typically considered among the application layer in the Internet protocol suite, providing 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; and a *record protocol* that uses the parameters established by the handshake protocol to protect the traffic between the securely communicating peers. The record protocol is located above the transport layer and uses the Transmission Control Protocol (TCP). Although the symmetric key ciphers used by the record protocol are affected by quantum attacks, specifically by Grover's algorithm [23], they can be easily modified to be quantum-safe by, e.g. doubling the size of the encryption keys. On the other hand, the *handshake protocol*, which makes heavy use of public key cryptography, is directly threatened by quantum cryptanalytic attacks without any simple mitigation. TLS uses public key cryptography for

two main purposes: Key Exchange, which mostly uses Diffie-Hellman (DH) over Elliptic Curves (EC) or RSA, and Digital Signatures, which use Elliptic Curve Digital Signature Algorithm (ECDSA) or RSA. For this reason, the integration of quantum-resistant public key cryptographic algorithms is necessary for both the Key Exchange and the Digital Signatures.

On TLS 1.3, the handshake between 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 such as 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 extension fields. For example, if the negotiated version of TLS is version 1.3, it is indicated on the "Supported Version" extension, as the original entry "Supported Version" is set to TLS 1.2 to ensure compatibility 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 exchanged message will be encrypted. The server sends the "EncryptedExtensions" message, which contains the remaining extensions. Then, the server sends the "CertificateRequest" message, which states that this session will be mutually authenticated. After that, the server sends the "Certificate" message, 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 exchanged handshake messages, starting with ClientHello and up to, but not including, this message itself. Finally, the server sends the "ServerHandshakeFinished" message, indicating that the handshake is complete from the server side.

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. As the handshake is mutually authenticated, the client now sends its "Certificate" message containing its digital certificate and the certificate chain up to a root CA, or alternatively, a self-signed certificate that is also installed on the server side. Then, the client sends the "CertificateVerify" message with a Digital Signature over a transcript of the messages so far. Finally, the client sends the "ClientHandshakeFinished" message. The handshake is now complete from the client side.

The server can now verify the client's certificate as well as the signature from the client. After that, the handshake is complete, and both peers can start sending and receiving the "Application Data", using symmetric key based authenticated encryption algorithms with the key that derived from the shared secret that both peers have agreed upon during the handshake phase.

3.2 WolfSSL

Regarding open-source TLS solutions for embedded systems, the most famous and widely used implementations are: Mbed TLS [3] and wolfSSL [11,10]. With Mbed TLS lacking support for TLS 1.3, wolfSSL is the only option to be adopted in this paper's research. WolfSSL is a library that implements the TLS protocol with a focus on "classical" cryptography. In more recent versions, wolfSSL has added support for CRYSTALS-Kyber, SABER, NTRU, and Falcon, via an integration with *liboqs* [9]. However, in the time of writing, this integration does not include optimised implementations for the Cortex-M4, except for an experimental setup that uses only Kyber512 from *pqm4* [12]. In our work, the wolfSSL library has been modified in order to support all the ARM Cortex-M4 optimised versions of the PQC algorithms, both KEM and Digital Signature (TLS Authentication).

WolfSSL [11], 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, etc. More information on these components can be found in Appendix A.

4 Proposed Design Approach and Overall PQ TLS 1.3 Architecture Implementation

To develop a complete PQ TLS 1.3 implementation supporting all the NIST PQC algorithms selected for standardisation as well as the upcoming Round 4 candidates, we make appropriate architectural adjustments in the wolfSSL library and the TLS 1.3 standard. Specifically, TLS 1.3 design changes have been made to the two TLS Extensions fields: "Supported Groups" and "Signature Algorithms". In addition, in our work, we make necessary changes to support KEMs as well as post-quantum Digital Signatures and certificates. In Figure 1, the overall PQ TLS 1.3 handshake with the proposed modifications and additions to the standard is presented visually, indicating the PQC operations that are made in each phase of the TLS handshake and the exchanged messages. In the following subsections, the proposed changes are discussed in detail.

4.1 WolfSSL Post-Quantum Adaptation

Supported Groups. The ClientHello message, the first message that the client sends to initiate the handshake, contains the Extensions field. In this field, the client extends the information provided by the rest of the ClientHello fields and it 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 in order of preference as encoded identifiers (codepoints) 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. To use PQC algorithms, new Named

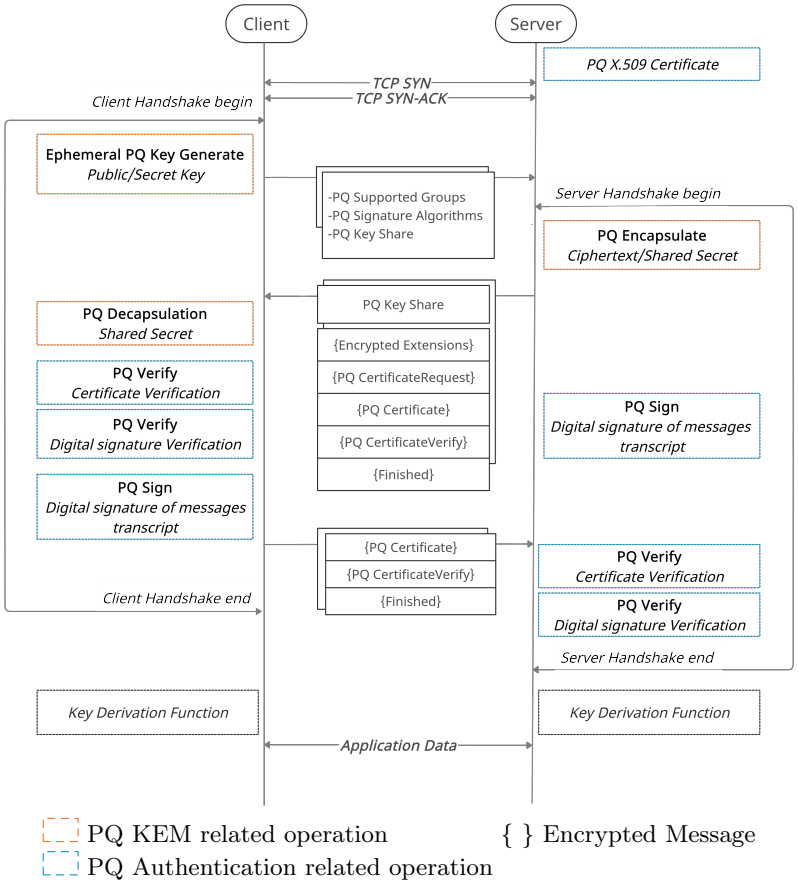

Fig. 1. Post-quantum TLS 1.3 handshake messages.

Groups have been introduced. We have decided to choose the same codepoints as the OQS's fork of OpenSSL [7], to make the wolfSSL library inter-operable with other popular libraries.

Signature Algorithms. Another TLS 1.3 Extension field of PQC adaptation interest is the "Signature Algorithms". 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", new codepoints are introduced for the post-quantum Digital Signature algorithms of the PQ TLS 1.3 design. The codepoints that have been added are compliant with the OQS's fork of OpenSSL [7].

Key Encapsulation Mechanism Support/Adaption. All the post-quantum encryption algorithms in the NIST PQC are Key Encapsulation Mechanism (KEM) schemes. However, only the traditional (Elliptic Curve) Diffie-Hellman Key Exchange (which is not a KEM) is supported in official TLS 1.3 standard. To transit the Key Exchange to the PQC paradigm, in our work, the Key Exchange mechanism of TLS 1.3 is transformed into a KEM scheme through architectural adaptation. We adopt the proposition introduced in the CRYSTALS-Kyber KEM based Key Exchange scheme [17] 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 calls the Encapsulation function by using the client's public key to produce: a Ciphertext, that will be sent to the client with the ServerHello message; and a Shared Secret that the server keeps since it is the actual shared key. The client, upon receiving the Ciphertext, calls the Decapsulation function by using its Secret Key to produce the same Shared Secret as the server. Now, both the client and the server share the same key that can be passed to a Key Derivation Function to produce the master secret along with any other intermediate secrets that the TLS 1.3 protocol requires. These exchanged messages are shown in Figure 1 as the Ephemeral PQC Key Generate, PQC Encapsulate, and PQC Decapsulate operations, respectively.

Digital Certificates Support. Another important object that needs to be modified in order for TLS to work with PQC algorithms is the digital certificates. These are objects that bound an entity, e.g. a server or a client, with its public key, by introducing a signature from a trusted third party. This can occur repeatedly by intermediate third parties, forming a "chain of certificates". The X.509 [20] is the standard of digital certificates on protocols such as TLS. It usually contains general information about the entity, along with the public key of the owner, the Digital Signature algorithm codepoint, and the Digital Signature itself.

To produce these digital certificates using post-quantum cryptographic algorithms, the OQS's fork of OpenSSL [7] is used. Through the OpenSSL's API, we generate digital certificates with support for all the PQC algorithms that are evaluated in this paper. Our goal is to produce digital certificates both for the server and the client, as they are mutually authenticated. To achieve this, we introduce a base "Certificate Authority" (CA) that can issue other certificates making a chain of trust up until a peer. In our paper, this chain is of length two, as the peer's certificate is directly signed by the CA. To accomplish this, we create a digital certificate for the CA, which is self-signed, and then we produce a digital certificate for the server and a certificate for the client, which both are then signed by the CA. Thus, two certificates are produced, both verifiable by our basic PQC 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, the certificates of CA, server,

and client all have Dilithium2 signatures, and the CertificateVerify message is signed using Dilithium2 as well.

5 Measurements and Evaluation

5.1 Experiment and Measurement Setup

To make the TLS protocol operate with PQC algorithms, we first integrate the *pqm4* [28] implementations of the selected PQC algorithms in our study to the wolfSSL code. Whenever *pqm4* lacks an implementation of an algorithm, we use the corresponding implementation from *PQClean* instead. To realise the network communication of our project, we use the lwIP library [22], a lightweight implementation of the TCP/IP protocol suite, mainly focusing on embedded devices. Almost all of the PQC algorithms in our study require the usage of symmetric cryptographic primitives, specifically the Keccak primitives SHA-3 and SHAKE-256 [30]. *Pqm4* provides optimised implementations of these primitives for the Cortex-M4, and this code has been used in our work. Note that while *PQClean* does not provide any optimised code for the PQC algorithms themselves, we have linked these schemes with the optimised Keccak code from *pqm4*, and as a result we can observe some speed-up.

We use two devices to evaluate the PQ TLS: i) The NUCLEO F439ZI embedded system by STMicroelectronics, with a 32-bit ARM Cortex-M4 microcontroller running at 180 MHz, with 192 KB of usable SRAM (plus 64 KB of CCM RAM that is not utilised) and 2 MB of Flash memory; and ii) A PC running Ubuntu 20.04 in x86_64 architecture, equipped with an Intel i7-1165G7 with 8 cores running at 2.8 GHz. Both devices are connected to the same access point, through the Ethernet interface with a mean Round-Trip Time (RTT) of 0.493 ms. Using a modified version of the wolfSSL TLS benchmark program, we evaluate a series of TLS connections between these two devices and gather measurements and statistics.

As discussed in [37], changing the TCP time window parameter (TCP_WND) of the underlying TCP implementation will impact the performance of the TLS handshake. We experiment with different values of the TCP_WND parameter, in multiples of TCP_MSS (maximum segment size of each packet). We observe that the handshake is particularly slower for some TCP_WND values than the others. Because the network analysis is not in the scope of this work, we choose TCP_WND $= 2 \times$ TCP_MSS, where the TLS handshake performs well in our local Ethernet network without further investigation. Contrary to [37], where the authors stated that one could get better performance with bigger TCP_WND, we find that the above described parameter choice performs better in the context of our local network with minimal latency and almost no packet loss.

In this section, we first compare the performance between different standalone PQC algorithms and with the classical public key cryptography algorithms of traditional TLS 1.3. Afterwards, we discuss the PQ TLS 1.3 handshake measurements using various combinations of PQC authentications and PQC KEMs.

Furthermore, we discuss the communication size (in terms of exchanged amount of bytes) of a PQ TLS 1.3 handshake and the overall memory requirements of different PQ TLS 1.3 combinations.

5.2 Post-quantum Cryptographic Algorithms Measurements

For the sake of completeness, we measure the standalone performance of the adopted PQC algorithms by using a modified version of the wolfSSL's benchmark program, *wolfcrypt-benchmark*. This tool uses the Real-time Clock (RTC) hardware module of the embedded system to measure time with millisecond precision. We use the implementations from *pqm4* for CRYSTALS-Dilithium, Falcon, CRYSTALS-Kyber, SABER, NTRU, NTRU LPRime, FrodoKEM, SIKE, BIKE, and Picnic3, and we use the implementations from *PQClean* for SPHINCS+ and HQC. In addition to the PQC algorithms, we measure the performance of four classical algorithms: RSA-2048 and ECDSA with curve secp256r1 are used for benchmarking classical authentication, and Finite Fields Diffie-Hellman Ephemeral (FFDHE)-3072 and ECDHE with curve secp256r1 are used for benchmarking classical Key Exchange, respectively. Every cryptographic algorithm operation has been executed for 10 seconds and the average execution time of all executions within 10 seconds is calculated and reported as the result.

In Table 1, the measured execution time on our target platform is presented, along with each algorithm's public key size and ciphertext/signature sizes. The claimed NIST security level of each algorithm is also presented. Level 1/3/5 means that an attack on that parameter set would require the same or more resources as a key search on AES 128/192/256, respectively. Level 2/4 means that an attack would require the same or more resources as a collision search on SHA-256/384, respectively. On the other hand, Level 0 means that this algorithm offers no quantum security. Note 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, respectively.

Comparison of Key Encapsulation Schemes. From Table 1, we can see that Kyber, the only KEM that is selected for standardisation by NIST so far, offers the best overall performance compared to all the other schemes. It has medium-sized keys and resulting ciphertexts as well as offering excellent performance in terms of execution speed.[4] It outperforms the classical FFDHE by an order of magnitude and even outperforms ECDHE by several milliseconds.

Among NIST PQC Round 4 candidates, HQC, although much slower than Kyber, offers good performance in terms of execution time but has the largest sizes compared to all the other schemes. BIKE has smaller sizes but slower execution time than HQC.

[4] The "execution speed" is calculated by the sum of "Key generation", "Encapsulation", and "Decapsulation" time for the PQC schemes and by using the "Agreement Time" for the classical algorithms.

Table 1. Summary of Traditional and Post-quantum Primitives.

KEM Algorithm	*NIST level*	*Public Key (bytes)*	*Ciphertext (bytes)*	*Key Generate (ms)*	*Encapsulate (ms)*	*Decapsulate (ms)*	*Notation*
FFDHE [1]	0	256	256	203.920	204.080 [4]	-	*FFDHE*
ECDHE [2]	0	32	32	8.428	17.687 [4]	-	*ECDHE*
Kyber512	1	800	768	8.133	6.239	3.419	*Kyb1*
BIKE-L1	1	1541	1573	200.620	25.969	411.308	*Bike1*
HQC128	1	2249	4481	30.202	50.682	72.775	*Hqc1*
SIKEp434	1	330	346	309.235	498.227	530.200	*Sike1*
SIKEp503	2	378	402	423.708	690.188	733.071	*Sike2*
SIKEp610	3	462	486	732.786	1341.125	1346.500	*Sike3*
Kyber768	3	1184	1088	12.224	11.412	7.924	*Kyb3*
SIKEp751	5	564	596	1262.750	2036.000	2183.000	*Sike5*
Kyber1024	5	1568	1568	12.918	11.623	8.539	*Kyb5*
Auth. Algorithm	***NIST level***	***Public Key (bytes)***	***Signature (bytes)***	***Key Generate (ms)***	***Sign (ms)***	***Verify (ms)***	***Notation***
RSA [3]	0	256	256	12.853/450[5]	448.250	12.500	*RSA*
ECDSA [2]	0	32	32	8.428	12.305	25.193	*ECDSA*
Sphincs128s	1	32	7856	8674.000	66 239.000	61.588	*Sphi1s*
Sphincs128f	1	32	17088	137.750	3361.000	190.167	*Sphi1f*
Falcon512	1	897	666	1266.667	243.881	3.275	*Falc1*
Dilithium2	2	1312	2420	12.063	25.404 [6]	9.569	*Dil2*
Dilithium3	3	1952	3293	19.438	39.309 [7]	16.244	*Dil3*
Falcon1024	5	1793	1280	4802.667	527.789	6.852	*Falc5*

[1] 3072-bit, [2] secp256r1 curve, [3] 2048-bit, [4] key agreement time, [5] public / private key generation time, [6] (11/114) (min/max), [7] (15/138) (min/max) of execution time over 1000 signatures

Finally, SIKE has the slowest execution time among all the schemes but offers the smallest public key and ciphertext sizes (even smaller than Kyber). This fact makes SIKE an interesting candidate that only requires a very small amount of network traffic. The execution speed can hopefully be improved by either novel code optimisation techniques or by hardware accelerations in the future. However, a recent work [19], has successfully recovered the secret key from SIKE in all security levels in practice, thus bringing concerns of its overall security.

Comparison of Authentication Schemes. It can be observed that Dilithium offers the most balanced performance compared to all the other Digital Signature schemes selected for standardisation. While having a medium-sized public key and signature, it offers excellent performance in terms of execution time. Both Dilithium and Falcon outperform RSA in all operations at security level 1, and they outperform even the ECDSA in the "Verify" operation. However, both are slower on the "Sign" operation than ECDSA, with Falcon being slower by an order of magnitude. Note that we do not compare the "Key Generation" operation as it is never performed online in a TLS handshake.

The SPHINCS+ offers 2 variants: The "small" variant has small signatures but very slow execution times and the "fast" variant has faster execution times but extremely large sizes. The signature size of the "fast" variant at NIST security

level 1 is very large (over 17 KB) and even exceeds the *Max_record_size* of the TLS protocol itself. On the other hand, the "small" variant needs over 1 minute of execution time for the "Sign" operation.

The most time consuming operation is the "Sign" operation in all the PQC authentication schemes. As the scenario in our evaluation is mutual authentication, the employed embedded system boards have to use the expensive "Sign" operation when acting both as a server and as a client. This would result in large overhead in execution time of the PQ TLS 1.3 handshake, mainly due to the PQC authentication schemes. On the other hand, in the scenarios with server-only authentication, there would be a significant improvement in execution time using PQC authentications when the embedded system board acts as a client, especially when Falcon is used.

5.3 Post-quantum TLS 1.3 Measurements

After implementing all the architectural changes described in the previous section, in this subsection, we measure the performance of the post-quantum TLS 1.3 protocol with different PQC algorithm combinations and compare this with the performance of the classical TLS 1.3. PQ TLS 1.3 connections are established on a local Ethernet network, following the experimental setup of subsection 5.1.

We use the *wolfssl-tls-bench* program provided by the wolfSSL library to evaluate the performance of the post-quantum TLS. We make necessary changes in order to run this program, and collect statistics and measurements for our PQ TLS. In this work, the benchmark for each PQC algorithm combination runs for 50 seconds. We select the Cipher Suite *TLS13-AES256-GCM-SHA384* as the symmetric primitives, in line with previous works [32,31]. Note that for simplicity, the client and the server in our experiment have been configured to agree upon the public-key algorithms immediately, without any extra round-trip and without pre-shared key assumption.

In Table 2, the experiment's measurements/results, along with the static memory consumption and the communication byte sizes of the handshake, are presented in detail.

TLS 1.3 Handshake Time Measurements. The measurements collected from the client differ from the measurements collected from the server. Although the TLS 1.3 handshake authentication related operations are similar between the two peers (1 "Sign" and 2 "Verify" operations for each), this is not the case for the KEM related operations: the client executes 1 "Key Generate" and 1 "Decapsulate" operation, while the server executes 1 "Encapsulate" operation. This leads to asymmetric execution time of the TLS handshake when the board acts as a client and as a server, especially for schemes where these operations are slow. This is most obvious in the measurements with BIKE, where the execution time of its operations is highly divergent: very fast "Encapsualte", but very slow "Key Generate" and "Decapsualte". This makes the client handshake time ∼5.6 times slower than the server. The *Dil2-Hqc1* client handshake time is ∼1.36

Table 2. PQ TLS 1.3 Handshake Measurements.

Notation	*Static Usage (bytes)*	*.bss Usage (bytes)*	*Communication Sizes (bytes)*	*Avg Handshake Time (ms)*	
				client	*server*
Selected Algorithms for Standardisation					
Dil2-Kyb1	49 648	0	14 748	96.318	91.062
Falc1-Kyb1	3680	39 936	6833	288.305	285.951
Dil3-Kyb3	69 072	0	20 224	157.126	153.492
Falc5-Kyb3	4200	79 872	11 789	594.495	589.058
Dil3-Kyb5	69 104	0	21 088	165.590	152.537
Falc5-Kyb5	4712	79 872	12 647	601.827	592.302
Sph1s-Kyb1	800	0	33 892	66 977.000	66 776.000
4rth Round Algorithms					
Dil2-Bike1	81 528	49	16 292	690.000	121.756
Dil2-Hqc1	71 672	0	19 910	198.603	145.989
Dil2-Sike1	49 648	0	13 858	886.359	566.125
Dil2-Sike2	49 648	0	13 962	1196.510	760.265
Dil2-Sike3	49 648	0	14 130	2089.690	1416.368
Dil2-Sike5	49 648	0	14 342	3403.222	2149.923
Dil3-Sike3	69 232	0	18 902	2246.077	1529.143
Dil3-Sike5	69 104	0	19 114	3450.167	2170.840
Traditional Algorithms					
RSA-ECDHE	2368	0	3742	540.220	538.158
ECDSA-ECDHE	2368	0	2353	109.171	106.927

times slower than the server. For rest of the PQC algorithm combinations, this difference is generally smaller, on an average of 6.25 ms.

Given that TLS 1.3 mutual authentication is used, KEM combinations with Dilithium perform much better than KEM combinations with Falcon. This is due to the fact that Falcon has extremely fast "Verify" but slow "Sign" operations. We also see that *Sph1s-Kyb1* is extremely slow, with over 66 seconds of run-time. Among the Round 4 algorithms, we see that *Dil2-Hqc1* performs extremely well, faster than *Falc1+Kyb1* and only ∼2 times slower than *Dil2+Kyb1*. *Dil2-Bike1* as a client is ∼3.5 times slower than *Dil2-Hqc1*, but outperform it as a server.

Finally, all SIKE combinations have considerable execution time overhead (leading to execution time of a few seconds) compared to Digital Signature combinations with HQC or BIKE.[5]

Communication Sizes. The "Communication Size" is the byte size of all the messages exchanged during the TLS 1.3 handshake, which is the sum of all the bytes that a peer has sent plus the sum of all the bytes that the peer has received. We only consider the byte size of the TLS 1.3 messages, and the overhead

[5] This should be considered with caution since there is evidence that SIKE is no longer cryptographically secure in its current version.

from the TCP headers and lower network level protocols is not included in the measurements. Communication size measurements provide a clear picture of the network traffic that is involved during a TLS 1.3 handshake with a specific PQC combination. For example, although KEM combinations with Dilithium generally perform much better in terms of speed compared to KEM combinations with Falcon, it can be observed that *Dil2+Kyb1* requires more than twice the bandwidth than *Falc1+Kyb1*. The same applies to higher security levels. In addition, the network traffic of the "fast" variant of SPHINCS+ is very high, almost 2.3 times compared to *Dil2+Kyb1* and almost 5 times compared to *Falc1+Kyb1*.

Regarding the Round 4 algorithms, *Dil2-Sike1* has the lowest communication sizes among all the combinations with Dilithium2. For example, *Dil2+Kyb1* uses ∼6.5% more bandwidth than *Dil2-Sike1*. However, given the current insecure status of SIKE, this result cannot be considered useful in practice. Note that the communication sizes are dominated by the Authentication introduced overhead. This has more significant impact in our mutual authentication scenario. As demonstrated in Table 1, each peer must send and receive a certificate or a whole certificate chain, and one Digital Signature. Nevertheless, larger KEM sizes affect the "Communication Sizes", as we can see that *Dil2-Bike1* and *Dil2-Hqc1* use ∼10.5% and ∼35% more bandwidth than *Dil2-Kyb1*, respectively, even though they all use Dilithium2 for authentication.

Compared to the classical TLS, PQ TLS 1.3 introduces a larger overhead in terms of network traffic due to the excessive communication size. *Dil2+Kyb1* uses 6.26 times more bandwidth than ECDSA+ECDHE, while *Falc1+Kyb1*, having a lower overhead, consumes only 2.9 times more bandwidth than ECDSA+ECDHE.

Memory Requirements. We use the tools provided by the STM32Cube IDE in order to perform an analysis on the static memory usage of our PQ TLS 1.3 implementation. The tool provides information about the code size, the RAM usage, and the stack requirements of each function etc. The *text* segment of the code size that includes the executable code and is stored in the Flash memory, never exceeds 30% of our total 2MB Flash memory. However, the *data* and *.bss* segments play a crucial role on the PQC algorithm integration, as they are stored in the 192 KB SRAM of the embedded system evaluation board. Algorithms introducing large artifact sizes or having an implementation that requires a lot of Stack memory, may eventually be impossible to be integrated in a memory-constrained device. For example, Rainbow has a public key of size 157KB, that alone consumes 80% of the total available memory, making it impossible to be evaluated in our board. Likewise, Classic McEliece is not included in *pqm4* in the first place for the same reason [28]. Also, Dilithium at security level 5 is not included in *pqm4*, because the memory requirements are too large. In Table 2, we show the Static memory or *data* region required by each KEM-Digital Signature combination as well as the respective *.bss* region, as reported by STM32Cube IDE tool.

Given that the employed embedded system evaluation board has 192 KB usable SRAM (excluding the 64 KB Core Coupled Memory), *Dil2+Kyb1* consumes

∼25% of the total available memory, while on higher security levels, *Dil3+Kyb5* uses ∼35%. Combinations with Falcon generally use less memory: *Falc1+Kyb1* consumes ∼ 22% of the total available memory and *Falc5+Kyb5* uses 43%. On the other hand, *Sph1s-Kyb1* uses merely 800 bytes of RAM.

Regarding the Round 4 algorithms, both BIKE and HQC consume a significant amount of memory. *Dil2-Hqc1* and *Dil2-Bike1* consume 41% and 36% of the total memory of the board, which are 44% and 64% more memory than *Dil2-Kyb1*, respectively.

Note that the PQC algorithm implementations do not make any use of dynamically allocated memory internally, but the rest of the program's components (lwIP, FreeRTOS, wolfSSL) do. The memory allocations of public key, private key, ciphertext, and signatures are also made dynamically by wolfSSL. In addition, at the time of this paper's writing, some of the *pqm4* implementations were not memory-usage optimised. It is likely that stack-usage optimised versions of the PQC algorithms selected for standardisation will be developed in the future, which would hopefully reduce significantly their memory footprint and thus make it possible to use them on memory-constrained devices.

6 Conclusion

In this paper, the importance of adopting a quantum-safe version of TLS 1.3 targeting embedded systems, in response to the imminent threats of a possible quantum attack, is discussed. In the paper, we provide the necessary architectural/design adaptations to the TLS 1.3 standard in order to make it quantum safe and we provide a systematic PQ TLS 1.3 design that can support a broad range of PQC algorithms, including the NIST PQC selected algorithms for standardisation, Digital Signatures that NIST considered for further evaluation (on a new round), as well as the alternative PQC algotithms that NIST deemed worthy to move to Round 4 of the NIST PQC process. We use an embedded system board equipped with an ARM Cortex-M4 and a remote PC with an Intel x86_64 chip in our experiment. We have connected our devices to the same access point through the Ethernet interface and performed a series of mutually authenticated PQ TLS 1.3 connections for benchmarking. Our design implementation is based on the wolfSSL TLS library for embedded systems, on which the PQC algorithms are seamlessly integrated. To evaluate the PQ TLS 1.3 performance on embedded systems, performance of the developed PQ TLS 1.3 implementation is measured in terms of the execution timing, memory footprint, and communication size overhead that the PQC algorithms will introduce. This work, to the authors' knowledge, constitute the first complete work on PQ TLS 1.3 for resource-constrained embedded systems that takes in account all the selected algorithms for standardisation from the NIST PQC process, almost all Round 4 algorithms, and additionally some alternative algorithms from Round 3.

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. Announcing pqc candidates to be standardized, plus fourth round candidates. https://csrc.nist.gov/News/2022/pqc-candidates-to-be-standardized-and-round-4, accessed: 29-07-2022
2. FreeRTOS. https://www.freertos.org, accessed: 29-07-2022
3. mbedTLS Library. https://github.com/ARMmbed/mbedtls, accessed: 29-07-2022
4. NIST call for submissions. https://www.nist.gov/news-events/news/2016/12/nist-asks-public-help-future-proof-electronic-information, accessed: 29-07-2022
5. NIST urge to 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, Slide 22, Accessed: 29-07-2022
6. OQS OpenSSH fork repository. https://github.com/open-quantum-safe/openssh, accessed: 29-07-2022
7. OQS OpenSSL fork repository. https://github.com/open-quantum-safe/openssl, accessed: 29-07-2022
8. TLS 1.3 adoption according to the Internet Society Pulse. https://pulse.internetsociety.org/technologies, accessed: 29-07-2022
9. WolfSSL Changelog. https://www.wolfssl.com/docs/wolfssl-changelog/, accessed: 29-07-2022
10. WolfSSL github repository. https://github.com/wolfSSL/wolfssl, accessed: 29-07-2022
11. wolfSSL Library. https://www.wolfssl.com/, accessed: 29-07-2022
12. WolfSSL PQ key establishment in Cortex-M4. https://www.wolfssl.com/post-quantum-tls-1-3-key-establishment-comes-stm32-cortex-m4/, accessed: 29-07-2022
13. Quantum-safe cryptography (QSC). https://www.etsi.org/technologies/quantum-safe-cryptography (2020), accessed: 29-07-2022
14. Arute, F., Arya, K., Babbush, R., Bacon, D., Bardin, J.C., Barends, R., Biswas, R., Boixo, S., Brandao, F.G., Buell, D.A., et al.: Quantum supremacy using a programmable superconducting processor. Nature **574**(7779), 505–510 (2019)
15. Barton, J., Buchanan, W.J., Pitropakis, N., Sayeed, S., Abramson, W.: Post quantum cryptography analysis of TLS tunneling on a constrained device. In: ICISSP. pp. 551–561. SCITEPRESS (2022)
16. Beullens, W.: Breaking rainbow takes a weekend on a laptop. IACR Cryptol. ePrint Arch. p. 214 (2022)
17. Bos, J.W., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J.M., Schwabe, P., Seiler, G., Stehlé, D.: CRYSTALS - kyber: A cca-secure module-lattice-based KEM. In: EuroS&P. pp. 353–367. IEEE (2018)
18. Bürstinghaus-Steinbach, K., Krauß, C., Niederhagen, R., Schneider, M.: Post-quantum TLS on embedded systems: Integrating and evaluating kyber and SPHINCS+ with mbed TLS. In: AsiaCCS. pp. 841–852. ACM (2020)
19. Castryck, W., Decru, T.: An efficient key recovery attack on sidh (preliminary version). Cryptology ePrint Archive, Paper 2022/975 (2022), https://eprint.iacr.org/2022/975, https://eprint.iacr.org/2022/975

20. Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., Polk, W.: Internet x. 509 public key infrastructure certificate and certificate revocation list (crl) profile. Tech. rep. (2008)
21. Döring, R., Geitz, M.: Post-quantum cryptography in use: Empirical analysis of the TLS handshake performance. In: NOMS. pp. 1–5. IEEE (2022)
22. Dunkels, A.: Design and implementation of the lwip tcp/ip stack. Swedish Institute of Computer Science **2**(77) (2001)
23. Grover, L.K.: A fast quantum mechanical algorithm for database search. In: STOC. pp. 212–219. ACM (1996)
24. Hoffman, P.: Smtp service extension for secure smtp over transport layer security. Tech. rep. (2002)
25. Hoffman, P.E.: The transition from classical to post-quantum cryptography. Internet Engineering Task Force, Internet-Draft drafthoffman-c2pq-05 (2019)
26. Holz, R., Hiller, J., Amann, J., Razaghpanah, A., Jost, T., Vallina-Rodriguez, N., Hohlfeld, O.: Tracking the deployment of TLS 1.3 on the web: a story of experimentation and centralization. Comput. Commun. Rev. **50**(3), 3–15 (2020)
27. Kannwischer, M.J., Rijneveld, J., Schwabe, P., Stoffelen, K.: PQM4: Post-quantum crypto library for the ARM Cortex-M4, https://github.com/mupq/pqm4
28. Kannwischer, M.J., Rijneveld, J., Schwabe, P., Stoffelen, K.: pqm4: Testing and benchmarking NIST PQC on ARM cortex-m4. IACR Cryptol. ePrint Arch. p. 844 (2019)
29. Kannwischer, M.J., Schwabe, P., Stebila, D., Wiggers, T.: Improving software quality in cryptography standardization projects. In: EuroS&P Workshops. pp. 19–30. IEEE (2022)
30. NIST: SHA-3 standard: Permutation-based hash and extendable-output functions. https://doi.org/10.6028/NIST.FIPS.202 (2015)
31. Paquin, C., Stebila, D., Tamvada, G.: Benchmarking post-quantum cryptography in TLS. In: PQCrypto. Lecture Notes in Computer Science, vol. 12100, pp. 72–91. Springer (2020)
32. Paul, S., Kuzovkova, Y., Lahr, N., Niederhagen, R.: Mixed certificate chains for the transition to post-quantum authentication in TLS 1.3. In: AsiaCCS. pp. 727–740. ACM (2022)
33. Razaghpanah, A., Niaki, A.A., Vallina-Rodriguez, N., Sundaresan, S., Amann, J., Gill, P.: Studying TLS usage in android apps. In: ANRW. p. 5. ACM (2018)
34. Rescorla, E.: Http over tls. Tech. rep. (2000)
35. Rescorla, E.: The transport layer security (tls) protocol version 1.3. Tech. rep. (2018)
36. Shor, P.W.: Algorithms for quantum computation: Discrete logarithms and factoring. In: FOCS. pp. 124–134. IEEE Computer Society (1994)
37. Sikeridis, D., Kampanakis, P., Devetsikiotis, M.: Assessing the overhead of post-quantum cryptography in TLS 1.3 and SSH. In: CoNEXT. pp. 149–156. ACM (2020)
38. Stebila, D., Mosca, M.: Post-quantum key exchange for the internet and the open quantum safe project. In: SAC. Lecture Notes in Computer Science, vol. 10532, pp. 14–37. Springer (2016)
39. Steblia, D., Fluhrer, S., Gueron, S.: Hybrid key exchange in tls 1.3. Internet Engineering Task Force, Internet-Draft draft-ietf-tls-hybrid-design-01 (2020)

Table 3. Summary of Other Post-quantum Primitives.

Algorithm	*NIST level*	*Public Key (bytes)*	*Ciphertext (bytes)*	*Key Generate (ms)*	*Encapsulate (ms)*	*Decapsulate (ms)*
LightSaber	1	672	736	10.564	6.555	3.960
NTRU	1	699	699	31.093	11.990	3.650
Frodo640	1	9,616	9,720	474.682	473.091	469.545
NTRULPR-761	2	1,039	1,167	18.116	11.284	8.818
Algorithm	*NIST level*	*Public Key (bytes)*	*Ciphertext (bytes)*	*Key Generate (ms)*	*Sign (ms)*	*Verify (ms)*
Picnic3	1	35	14,612	5.922	2,188.400	1,355.000

Table 4. PQ TLS 1.3 Handshake Measurements with Other Post-quantum Primitives.

Notation	*Static Usage (bytes)*	*.bss Usage (bytes)*	*Communication Sizes (bytes)*	*Avg Handshake Time (ms)*	
				client	*server*
			Extra Measurements		
Pic1-Kyb1	87 000	13	52 445	5022.167	5027.083
Dil2-Ntru1	50 000	0	14 578	126.978	104.136
Dil2-Frod1	73 000	0	32 491	1194.501	687.865
Dil2-Ntrulpr1	49 648	0	15 388	125.351	123.697
Dil2-Sab1	49 648	0	14 588	109.429	98.401

A Main Components of wolfSSL Library

WolfSSL consists of the following main components:

WolfCrypt. This component includes all the classical public-key and symmetric-key cryptographic algorithms, as well as hash algorithms, MAC algorithms, and programs handling certificate and key files. It provides optimised code for various 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 such as 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 such as TCP, or to a higher level such as an operating systems e.g. FreeRTOS [2].

Utilities. This component includes all non-essential utilities such as benchmarks or programs for testing purpose to verify 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 use the following two tools:

– *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 measurements on PQC algorithms in order to compare them with the classical algorithms.

– *Wolfssl-tls-bench* is another benchmarking tool that measures and provides a series of metrics regarding TLS sessions. It can make use of an operating system, like FreeRTOS, and simulate a server and a client connecting through TLS on the same device. Alternatively, it can be run on different devices, 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 e.g. 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 PQC algorithms and to compare the results with the TLS protocol using classical algorithms.