Post-Quantum TLS on Embedded Systems

Integrating and Evaluating Kyber and SPHINCS$^+$ with mbed TLS

Kevin Bürstinghaus-Steinbach
SEW-EURODRIVE
Bruchsal, Germany
kevin@b-steinbach.de

Christoph Krauß[*]
Hochschule Darmstadt
Darmstadt, Germany
christoph.krauss@h-da.de

Ruben Niederhagen
Fraunhofer SIT
Darmstadt, Germany
ruben@polycephaly.org

Michael Schneider
Darmstadt, Germany
michael@bizzel.biz

ABSTRACT

We present our integration of post-quantum cryptography (PQC), more specifically of the post-quantum KEM scheme Kyber for key establishment and the post-quantum signature scheme SPHINCS$^+$, into the embedded TLS library mbed TLS. We measure the performance of these post-quantum primitives on four different embedded platforms with three different ARM processors and an Xtensa LX6 processor. Furthermore, we compare the performance of our experimental PQC cipher suite to a classical TLS variant using elliptic curve cryptography (ECC).

Post-quantum key establishment and signature schemes have been either integrated into TLS or ported to embedded devices before. However, to the best of our knowledge, we are the first to combine TLS, post-quantum schemes, and embedded systems and to measure and evaluate the performance of post-quantum TLS on embedded platforms.

Our results show that post-quantum key establishment with Kyber performs well in TLS on embedded devices compared to ECC variants. The use of SPHINCS$^+$ signatures comes with certain challenges in terms of signature size and signing time, which mainly affects the use of embedded systems as PQC-TLS server but does not necessarily prevent embedded systems to act as PQC-TLS clients.

CCS CONCEPTS

• **Security and privacy → Digital signatures**; • **Networks → Transport protocols**; **Security protocols**; • **Theory of computation → Cryptographic primitives**; **Cryptographic protocols**.

KEYWORDS

PQC; Kyber; SPHINCS$^+$; TLS; embedded systems; mbed TLS

ACM Reference Format:
Kevin Bürstinghaus-Steinbach, Christoph Krauß, Ruben Niederhagen, and Michael Schneider. 2020. Post-Quantum TLS on Embedded Systems: Integrating and Evaluating Kyber and SPHINCS$^+$ with mbed TLS. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security (ASIA CCS '20), October 5–9, 2020, Taipei, Taiwan.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3320269.3384725

1 INTRODUCTION

Recently, the interest in post-quantum cryptography (PQC) has been increasing, not only in academia, but also in industry and the general public. An evidence for this growing interest is the PQC standardization process by NIST with a large number of submissions in the first and second rounds [14]. The fact that stable, large-scale quantum computers can break popular and wide-spread public key primitives (in the following referred to as "classical" cryptography) has reached a growing audience. Companies and research facilities are starting to evaluate, which field of post-quantum cryptography might provide the best candidates to replace current primitives in applications and protocols by post-quantum primitives. TLS as the main communication protocol for the Internet plays a crucial role in this process.

In addition, in our modern world, security needs to be integrated into small embedded devices that surround us in our daily live. The Internet of Things (IoT), home automation, connected driving, industry controllers — there are many use cases where small, embedded devices control sensitive processes or transfer sensitive data. Therefore, it is very important that embedded, low-power devices are secured with state-of-the-art cryptographic protocols and primitives — which soon will be post-quantum primitives.

If it was easy to replace cryptographic algorithms in productive systems, the need for early adoption of post-quantum schemes would be minor. In case that large-scale, stable quantum computers will be built, replacing cryptographic schemes by post-quantum variants could be done fast. Unfortunately, this requires *crypto agility*, which is rarely existing in practice. Systems are running with legacy software that is hard to replace. This is especially true for embedded devices, where software (and cryptographic) updates are even harder to install. Therefore, it is highly required to invest in research for migrating IT systems to post-quantum primitives, especially on embedded systems.

[*] Also with Fraunhofer SIT.

ASIA CCS '20, October 5–9, 2020, Taipei, Taiwan

ACM ISBN 978-1-4503-6750-9/20/10…$15.00
https://doi.org/10.1145/3320269.3384725

In this work, we combine all three fields — PQC, TLS, and embedded devices. We enable an embedded TLS library (mbed TLS) with post-quantum key exchange (based on Kyber in its IND-CCA variant) and post-quantum digital signatures (SPHINCS$^+$). Both Kyber and SPHINCS$^+$ have been submitted to the NIST PQC standardization process and have been selected for the second round, which makes them likely candidates for future standardization. SPHINCS$^+$ is a member of the hash-based family of signature schemes. This family has the reputation of being well trusted, reliable, and secure. However, they have quite large signature sizes, which makes them challenging for embedded systems. By integrating both Kyber as compact key-exchange scheme and SPHINCS$^+$ as reliable but large signature scheme, we show that embedded systems are able to handle the increased cost of PQC schemes.

SPHINCS$^+$ can be instantiated with different hash functions. In our test, we use SHA-256 and SHAKE-256 in SPHINCS$^+$. For comparison with Kyber we use ECDH and for comparison with SPHINCS$^+$ we take ECDSA, which are commonly used classical key-exchange and signature scheme for TLS. In order to test full TLS handshakes, we create the new prototype cipher suite `TLS_KYBER_SPHINCS_-WITH_AES_256_GCM_SHA256`. We then test the selected PQC primitives and the adapted mbed TLS library on four different embedded platforms: For our tests we use a Raspberry PI 3, an ESP32, an industrial field option card, and an LPC (see Section 4). These platforms cover a large range of use cases and applications, which gives a broad view on real-world practicality of our results.

Our modifications to the mbed TLS library and the platform-specific adaptions are available online[1] under Apache 2.0 license.

2 BACKGROUND

In this section, we give a brief overview of Transport Layer Security (TLS). We focus on the TLS handshake to explain the integration of PQC schemes as well as TLS libraries for embedded systems. In addition, we briefly introduce PQC and the chosen PQC schemes Kyber and SPHINCS$^+$.

2.1 Transport Layer Security

Transport Layer Security (TLS) is the de facto standard for secure communication on the Internet. The first version has been published in the year 1994 under the name Secure Sockets Layer (SSL) 1.0 and the newest version is TLS 1.3, which has been finally standardized in the year 2018. In this work, we focus on TLS 1.2 [15] and before, since we use the mbed TLS library which does not support TLS 1.3, yet. However, most of the changes in TLS 1.3 are not relevant for our work, e.g., the deprecation of old ciphers, since anyway we exchange the ciphers to PQC schemes. However, the changes in the handshake protocol (see below) for improving the efficiency would be interesting for future work. In the following, we briefly describe the different TLS protocols and TLS libraries for embedded systems.

2.1.1 TLS Protocols. TLS consists of a record, handshake, change cipher spec, alert, and an application data protocol.

The record protocol is located directly above the transport layer and uses the Transmission Control Protocol (TCP)[2]. It provides bulk data encryption using symmetric cryptographic algorithms such as AES. On this layer, the protection against quantum computers can be easily achieved by using keys of length 256 bits and higher [16]. In addition, the record protocol ensures message integrity and authenticity using either an HMAC or authenticated encryption schemes such as the Galois/Counter Mode (GCM).

Located above the record protocol are the handshake protocol, change cipher spec protocol, alert protocol, and application data protocol. Before sending messages of these protocols, the record protocol does fragmentation, compression, encryption etc. The application data protocol forwards data from the application layer, e.g., HTTPS, to the record layer. The alert protocol is responsible for handling errors. The change cipher spec protocol is used for signaling that the cipher suites, negotiated in the handshake protocol, are now used. The handshake protocol is used for authentication (server only or mutual), negotiation of the used cryptographic primitives, and the establishment of session keys. These session keys are then used for ensuring confidentiality, integrity, and authenticity of the exchanged messages.

Up to TLS 1.2, the handshake protocol can either use RSA or Diffie-Hellman (DH) in the key exchange and RSA and DSA for digital signatures as well as the elliptic curve (EC) variants. Since all of these asymmetric cryptographic schemes are not quantum-computer resistant, we exchange them with the PQC schemes Kyber for key exchange and SPHINCS$^+$ for signatures (see Section 3).

The client initiates a TLS connection by sending a `ClientHello` message, which contains the supported TLS version, the supported cipher suites, the supported compression algorithms, a client random, and optional data such as session ID or session ticket for session resumption. A cipher suite describes the used cryptographic algorithms for key exchange, signatures, encryption, message authentication, and the pseudorandom function (PRF) used to generate keying material. For example, the cipher suite `TLS_ECDHE_ECDSA_-WITH_AES_256_GCM_SHA256` uses ECDHE for key exchange, ECDSA for digital signatures, and AES-256 in GCM block-cipher mode for data encryption as well as message integrity and authenticity. The SHA-256 hash function is used in the PRF for calculating keying material.

The server answers with a `ServerHello` message which contains the chosen TLS version, chosen cipher suite, chosen compression algorithm, and a server random. He also sends a `Certificate` message containing the certificate with the public key of the server as well as the entire certificate chain up to a root certification authority (CA). Currently, the CAs usually use RSA for the signatures in the certificates. RSA needs to be replaced with a PQC scheme to be resistant against quantum computer attacks. The `ServerKeyExchange` message is sent only in case DH is used for key exchange. In this work, we are going to focus on cipher suites that are using an ephemeral key exchange without the use of long-term keys on the side of either the server or the client. In this case, the message contains the ephemeral public DH key of the server and a signature over the public DH key, client random, and

[1] https://github.com/kbuersti/mbedtls

[2] Datagram Transport Layer Security (DTLS) enables the use of User Datagram Protocol (UDP) instead of TCP.

server random. We replace DH with the PQC scheme Kyber and use this message to send an ephemeral Kyber public key from the server to the client. These data fields and the random values are signed using SPHINCS$^+$. In case client authentication is required, the server also sends a `CertificateRequest` message. With the `ServerHelloDone` message the server gives the client the signal to continue the handshake.

The client sends the client `Certificate` message in case mutual authentication is required, i.e., the server has sent a `Certificate-Request` message. Again, PQC schemes have to be used for signatures in the certificate to be resistant against quantum computer attacks. In any case, the client sends a `ClientKeyExchange` message. If RSA is used, the message contains a pre-master-secret, generated by the client, and is encrypted with the public key of the server. When using DH, it contains the public DH key of the client. In our work, we use this message to send a shared secret encrypted as Kyber ciphertext from the client to the server. In case the client needs to authenticate itself (and has sent a client `Certificate` message), the client also sends a `CertificateVerify` message. Basically, this message contains a digital signature over all handshake messages sent or received, starting at `ClientHello` and up to, but not including, this message. With the `ChangeCipherSpec` protocol message, the client indicates to switch to the negotiated algorithms. The `Finished` message completes the handshake and contains a message authentication code (MAC) over all handshake messages sent or received, starting at `ClientHello` and up to, but not including, this message.

The server answers also with a `ChangeCipherSpec` protocol message and a `Finished` message. After that, application data can be securely exchanged.

2.1.2 TLS Libraries. The TLS protocol is already implemented in various libraries. Very popular open source libraries are for example OpenSSL[3] and LibreSSL[4]. However, these libraries both are not suitable for embedded systems since they have not been developed for highly resource constraint environments. Nevertheless, there are two popular libraries for embedded systems that are easily portable to new platforms and have low resource requirements: mbed TLS[5] and wolfSSL[6].

The mbed TLS library is fully open source under the Apache License 2.0, whereas wolfSSL is available under two licenses. It is open source under the GPLv2 license for non-commercial use and has to be licensed by businesses for commercial use. Because of the flexible Apache license and the embedded oriented design, we chose mbed TLS for our implementation.

2.2 Post-Quantum Cryptography

Post-quantum cryptography (PQC) refers to (asymmetric) cryptographic algorithms that are resistant against attacks using a quantum computer. Running Shor's algorithm [33, 34] on a quantum computer, all currently popular asymmetric algorithms based on the integer factorization problem (e.g., RSA), the discrete logarithm problem (e.g., DH), or the elliptic-curve discrete logarithm problem (e.g., ECDH) can be broken in polynomial time. TLS makes heavy use of RSA, DH, and the EC variants in the handshake protocol for signing and key exchange. To be resistant against quantum computers, these algorithms have to be exchanged by PQC schemes.

Currently, PQC schemes can be divided into five families: code-based, lattice-based, hash-based, multivariate, and supersingular elliptic-curve isogeny cryptography. Most mature are hash-based schemes for digital signatures. They have been first introduced by Lamport [23] as well as Merkle and Winternitz [26] in 1979 and further improved over the last decades. They have been thoroughly evaluated and provide a high security level. Lattice-based cryptography has been introduced by Ajtai [2] in 1996. It has the advantage of providing very efficient schemes for key encapsulation compared to other PQC families. However, its security is not as well understood as that of hash-based schemes, yet.

In order to evaluate these two promising PQC families on embedded devices, in this work, we use the lattice-based scheme Kyber for key exchange and SPHINCS$^+$ as signature scheme. Both are part of the ongoing NIST post-quantum standardization process and are promising candidates for future standard schemes. We selected security parameter sets for the PQC schemes that have a NIST security level of 1 corresponding to AES-128 in order to achieve a fairer comparison with classical schemes at 128-bit (classical) security. Our evaluation is based on the round-one versions of the scheme.

2.2.1 Kyber. Bos et al. proposed Kyber [6, 9], a lattice-based IND-CCA (INDistinguishability under adaptive Chosen Ciphertext Attack) key encapsulation mechanism (KEM) which uses similar concecpts as NewHope [3]. Lattice-based cryptography is one of the most promising candidates for NIST post-quantum standardization, also most submissions are based on lattices. Kyber uses module-lattices because they provide an efficient trade-off between security and performance. It was also designed with embedded devices in mind, e.g., using precomputed tables of powers for a number theoretic transform. Since NewHope was already tested on embedded and non-embedded platforms [1] including the Google Chrome browser [12] it seems reasonable to test Kyber, which is based on NewHope, also on embedded platforms. In their submission to the NISTs post-quantum standardization process Bos et al. defined also an unauthenticated key exchange protocol with Kyber.

The main parameters of Kyber are the degree n of the polynomial ring, a prime q that defines the underlying ring structure, a positive integer η used for a binomial distribution, and an integer k such that $k \cdot n$ is the dimension of the corresponding LWE problem. The different security levels Kyber512, Kyber768, and Kyber1024 are achieved by varying k and η. In this work, we focus on Kyber512 at NIST PQC security level 1. For this security level, the public key has a size of 736 B and the cipher text of 800 B.

Kyber is defined as a key encapsulation mechanism (KEM). An ephemeral key exchange (KEX) scheme can easily be obtained by creating a new ephemeral public key for each key exchange and sending it to the communication partner. The other party creates a random secret key, uses the ephemeral public key to encapsulate the secret key, and sends the encapsulated key back. Finally, the first party decapsulates the ephemeral secret key, which gives both parties a shared secret key.

[3] https://www.openssl.org/
[4] https://www.libressl.org/
[5] https://tls.mbed.org/
[6] https://www.wolfssl.com/

2.2.2 *SPHINCS$^+$.*

Bernstein et al. proposed with SPHINCS$^+$ a stateless hash-based signature scheme for the NIST post-quantum standardization process [7] that comes with strong security proofs with minimal security assumptions. The scheme SPHINCS$^+$ is based on SPHINCS [8] and for the predecessor Hülsing et al. already showed a fast implementation for an embedded ARM board [21].

SPHINCS$^+$ uses a hierarchical structure of Merkle hash trees with one-time and few-time signature schemes at their leafs. The public key is the root of the top Merkle tree. The leafs of the inner Merkle trees are one-time signature schemes that are used to sign the roots of the Merkle trees on the next lower level. The leafs on the lowest Merkle trees are few-times signature schemes that are used to sign the actual message digests. For *key generation*, only the top Merkle tree needs to be computed in order to obtain its root as public key. For *signing*, a Merkle tree on the lowest level and a few-time signature at its leafs is chosen deterministically (based on the message digest) and the verification path from the selected leaf node all the way through the hierarchical tree structure to the root node of the highest level is computed. For *verification*, the root node of the highest Merkle tree is recomputed from the message digest using the verification path and verified with the public key. The most expensive operation is the computation of a signature; key generation and verification are much cheaper.

The proposal of SPHINCS$^+$ [7] contains 18 different parameter sets based on three different arguments: hash function (SHAKE-256, SHA-256, and Haraka), security level (NIST level 1, 3, and 5), and trade-off between signature size (s) and speed (f). We chose two variants of SPHINCS$^+$-128f at NIST security level 1 and parameter-optimized for speed with the hash functions SHAKE-256 and SHA-256 for our post-quantum TLS library. We focused on SHAKE-256 and SHA-256 as hash functions, because they are already well studied in contrast to Haraka. The size of the public key is only 32 B; however, one signature requires about 17 kB.

2.3 Related Work

As far as we know, we are the first to implement two promising post-quantum standardization candidates into an embedded TLS library, to perform a complete post-quantum handshake, and to evaluate the performance on multiple microcontroller boards. Nevertheless, there has been a lot of research in both directions — post-quantum TLS (PQ-TLS) and PQC on embedded devices.

2.3.1 *PQ-TLS.*

Chang et al. implemented a complete post-quantum handshake in the Polar SSL library using a lattice-based key exchange and a multivariate signature scheme; they tested their performance on a Intel desktop CPU [13]. Google used lattice-based post-quantum key exchange (NewHope) on top of standard elliptic curve-based (X25519) key exchange and tested TLS handshakes between Chrome browser and Google webservers [12]. In [10], Ring-LWE based key agreement was integrated into OpenSSL and tested on classical hardware. Open Quantum Safe[7] provides a library with implementations of PQC primitives and prototype integrations into OpenSSL. It runs on ARM Cortex A8 and Raspberry Pi. The pqm4 project [8] contains implementations of PQ key-encapsulation mechanisms and PQ signature schemes targeting the ARM Cortex-M4 family of microcontrollers. The work of [29] emulates real network conditions and evaluates the impact of various post-quantum primitives (including key exchange and signatures) on TLS connection establishment performance. In [35] the authors integrate and test the impact of PQ signature algorithms on TLS 1.3 under realistic network conditions.

2.3.2 *PQC on Embedded Devices.*

There are multiple implementations of single post-quantum primitives on certain embedded devices. All those highly targeted implementations were not integrated into a TLS library. Hülsing et al. implemented the SPHINCS signature scheme on an embedded microprocessor [21] and Howe et al. also implemented a lattice-based standardization candidate on an FPGA and microcontroller devices [20]. In [17] the authors test and compare the lattice-based signature schemes GLP, BLISS, and Dilithium on ARM Cortex-M4 microcontrollers. The GLP scheme has been presented in an optimized AVX implementation in [18]. BLISS and its improvement BLISS-B have microcontroller implementations on AVR [25] and ARM Cortex-M4 [28] [36]. The works by Kuo et al. [22] and Oder and Güneysu [27] implement the FrodoKEM scheme on FPGAs and Alkim et al. [4] present a microcontroller implementation. [30] implements Ring-LWE encryption and BLISS on an 8-bit Atmel ATxmega128 microcontroller. [20] implements the FrodoKEM key encapsulation mechanism on a low-cost FPGA and microcontroller devices. [19] presents an implementation of the standard lattice-based encryption scheme, proposed by Lindner and Peikert. In [11], the authors report performance measurements of an optimized software implementation of Kyber on a Cortex-M4 processor. The EU Horizon 2020 project SAFEcrypto also investigates the use of (standard) lattice-based cryptography in hardware, specifically for conservative use cases such as satellite communications.

3 INTEGRATION OF PQC INTO THE MBED TLS LIBRARY

In order to perform a post-quantum TLS handshake, we integrated the reference implementations of the PQC algorithms Kyber and SPHINCS$^+$ (see Section 2.2) into the mbed TLS library. The mbed TLS library is written in C and can be divided into three parts: cryptographic primitives, TLS protocol, and tools as described in the following paragraphs.

3.1 Cryptographic Primitives

The mbed TLS library capsules cryptographic algorithms into modules with loosely-coupled interfaces. These modules can be grouped into symmetric encryption algorithms with mode of operation, hash functions, random number generators, and public key algorithms. We encapsulated Kyber and SPHINCS$^+$ each in its own module using the reference code of the algorithms as base for the implementation. They both require the hash functions SHA-256 and SHAKE-256. Since SHA-256 is already part of the mbed TLS library, we simply changed the corresponding function calls in the reference implementation of SPHINCS$^+$ from OpenSSL to the mbed TLS counterparts. SHAKE-256 is not part of the mbed TLS library and we kept the source code that was provided with the reference implementations.

[7] https://openquantumsafe.org/

[8] https://github.com/mupq/pqm4/

The reference implementation of SPHINCS$^+$ only offers to choose parameters like the hash function at compile time. As described in Section 2.2, we support two variants of SPHINCS$^+$, which need to be selected at runtime based on the corresponding server public key. The reference implementation already prepared a defined set of function calls to the hash functions. Following the "lightweight interfaces" model of the mbed TLS library, we added a data structure `sphincs_md_info_t` for parameters and function calls of the respective hash function. This structure allows to choose the hash function dynamically, e.g., depending on the SPHINCS$^+$ key context.

3.2 TLS Protocol

For our evaluation, we focus on TLS connections with server authentication only. For our scenario where client and server agree on the Kyber-SPHINCS$^+$ post-quantum cipher suite, there are three messages of particular interest:

- `Certificate`: The certificate message contains a SPHINCS$^+$ signature. This allows the client to extract all necessary information about server's SPHINCS$^+$ public key. Therefore, we defined an ASN1-based structure for SPHINCS$^+$ to send X509 certificates.
- `ServerKeyExchange`: To send a key exchange message, the server performs two steps: First, a new ephemeral Kyber key pair is generated and the public key is pasted into the key-exchange message. Then the key exchange data is signed with the server's SPHINCS$^+$ private key and the signature is added to the key-exchange message, which then is sent to the client.
- `ClientKeyExchange`: After receiving the server's key exchange message, the client verifies the SPHINCS$^+$ signature and generates its own client Kyber key-exchange response. In our test scenario, the client key exchange is unauthenticated, but client authentication can be easily implemented since the SPHINCS$^+$ signature scheme is fully integrated into the TLS library.

3.2.1 Cipher Suites. For the use of Kyber and SPHINCS$^+$, we introduced the new prototype cipher suite `TLS_KYBER_SPHINCS_WITH_AES_256_GCM_SHA256` to use the post-quantum algorithms during a TLS handshake. It uses AES-256 in GCM mode for bulk encryption and SHA-256 for message authentication codes.

The Kyber key exchange is integrated similar to the ECDHE key exchange. The server generates a new Kyber key-pair and the other parameters using the `make_params` function and parses the client response in the `read_public` function. On the other end the client reads the public key information through the function `read_params` and generates its own payload with `make_public`. They both derive the shared secret with `calc_secret`.

The SPHINCS$^+$ signature scheme is integrated similar to ECDSA. The server generates a new SPHINCS$^+$ signature with the `write_signature` function and the client verifies the signature with the `read_signature` function.

3.2.2 Certificates. Since the cipher suites do not define which variant of SPHINCS$^+$ is used, the certificate needs to provide the necessary information. We are using two new OIDs in the certificate for SPHINCS$^+$, one for the SHA-256 variant and one for the SHAKE-256 variant. Further SPHINCS$^+$ parameters currently are still defined during compile time; if other variants are required, e.g., parameter sets with higher security level, changes to the implementation are necessary.

3.2.3 Handshake Fragmentation. SPHINCS$^+$ signatures have a size of up to 50 kB, which exceeds the maximum single record size of 2^{14} bytes, i.e., 16 kB. Therefore, handshake messages containing SPHINCS$^+$ signatures (e.g., `Certificate` and `ServerKeyExchange`) need to be fragmented. At the time of writing, mbed TLS does not support this feature for TLS (but only for DTLS). Thus, we added an additional record layer where messages exceeding the maximum content length are disassembled on send and reassembled on receive.

During the send process, the message length is checked whether it is greater than `MAX_CONTENT_LENGTH`. This length cannot be greater than 2^{14} bytes. If the message length is larger, it is split into fragments of size `MAX_CONTENT_LENGTH`. Every fragment contains the standard TLS record header with a version, type, and length field. The first fragment also contains the handshake header with the message type and the handshake length after the record header. The rest of the fragment is the handshake payload. The first fragment looks like a normal message except that the record length field does not match the handshake length field. All following fragments do not contain a handshake header, but only the payload after the record header.

On receipt of a handshake message, the record layer checks if it is only a fragment by comparing the record length and handshake length. If the handshake length is larger, more fragments need to be received in order to reassemble the complete handshake message. The fragments are stored in a temporary buffer until the stored size matches the handshake length.

3.3 Tools

Running and measuring a complete post-quantum handshake also requires some utilities in addition to the TLS library (i.e. for certificate generation as well as client and server testing applications). Therefore, we added some extensions to existing tools accompanying the mbed TLS library and to its configuration.

3.3.1 Configuration. All changes we made to the library can be controlled through the `config.h` file at compile time. Each module of the cryptographic primitives can be activated or deactivated separately. The defines are `MBEDTLS_SPHINCS_C` and `MBEDTLS_KYBER_C`. The SPHINCS$^+$ and Kyber cipher suite must be activated by setting `MBEDTLS_KEY_EXCHANGE_KYBER_SPHINCS_ENABLED`. Finally, since SPHINCS$^+$ signatures exceed the single record size, handshake fragmentation needs to be enabled using `MBEDTLS_SSL_HS_FRAGMENTATION` and the maximum message size must be set to 2^{14} bytes using `MBEDTLS_SSL_MAX_SIZE`.

3.3.2 Test Programs. We modified and added some tools to enable PQC key generation, client and server tests, and performance measurements with our modification.

- *Key Generation:* We enabled the `gen_key` tool to generate keys for Kyber and SPHINCS$^+$. The tool exports the SPHINCS$^+$

keys encoded in the newly defined ASN1-based structure for SPHINCS$^+$.

- *Server/Client:* The library already has a test server and client. We extended these programs to allow various performance measurements on the target platforms.
- *Benchmark:* We added a benchmark for SPHINCS$^+$ and Kyber to the existing benchmark structure.

4 TARGET PLATFORMS

The mbed TLS library has been designed to be able to run on various platforms. In order to make it easy for developers to port the library to a different environment, mbed TLS uses a well-documented platform layer. The platform layer is structured into several parts that interact with the target platform environment: networking, timing, entropy sources, hardware acceleration, file system access, real-time clocks, and diagnostics [31].

Since the main goal of mbed TLS library is to provide TLS connections, it requires a connection to a TCP stack. The library is based on a Berkeley-socket like interface, which provides blocking and non-blocking calls optionally with timeouts. The mbed TLS library provides functionality for secret key generation and uses nonces in the TLS handshake, all of which are generated as random bit sequences. Within the mbed TLS library, entropy sources are separated in weak and strong sources. For security-relevant purposes the library needs to be provided with a strong source. Some operations in the library require correct time information, e.g., when checking the validity of X509-certificates. For performance measurements, it is sufficient to measure a time interval. However, timing information is not a necessary functionality for the library to work.

In order to measure the performance on embedded platforms, we ported the mbed TLS library with our PQC adaptions to four different platforms. Each platform represents a specific group of embedded devices: We chose the Raspberry Pi 3B+ as small computer, the ESP32-PICO board with a popular IoT chip, a fieldbus option card for industrial computers, and a very resource constraint LPC platform. An overview of the platforms is shown in Table 1.

4.1 Raspberry Pi 3 Model B+

The Raspberry Pi is a small and affordable single-board computer developed by the Raspberry Pi Foundation for educational purposes[9]. It is a popular prototyping platform and core of many smart home projects. The platform has an ARM Cortex-A53 quad-core processor running at 1.4 GHz and 1024 MB of RAM. To store operating system and software, the Raspberry Pi needs to be equipped with an SD-Card. The Raspberry Pi provides the most resources within the platforms we are investigating. It is capable of running multi-process operating system, e.g., the Debian-based Linux distribution Raspbian. Network connectivity is provided with a gigabit Ethernet interface.

Port. There are no further requirements for using the mbed TLS library on the Raspberry Pi. It can be used directly, e.g., with the Raspbian operating system. The mbed TLS library can directly access all required features like network connections and timers by default. The Raspberry Pi 3B+ does not provide any hardware acceleration for cryptographic operations, but mbed TLS provides highly optimized assembler code for ARM platforms. We only added a kernel extension to access the CPU cycle counter for timing measurements [5].

4.2 ESP32-PICO-KIT V4

The ESP32-PICO-KIT V4 platform has an ESP32 microcontroller that integrates Wi-Fi and Bluetooth 4.2 solutions on a single chip. Due to its wireless networking features, this platform is popular in the IoT developer community. The core of the microcontroller is a Xtensa dual-core 32-bit LX6 processor operating at 240MHz. The PICO-KIT provides 520 kB of SDRAM and 16 MB of flash memory to store and run the software. Time measurements can be performed using a 32-bit hardware timer with microsecond resolution and using a CPU cycle counter. Some PQC operations, e.g., the SPHINCS$^+$ signing operation, require that many CPU cycles that we had to integrate cycle-count overflow detection.

Port. The official development framework for the ESP32 platform is the ESP-IDF[10]. A port of the mbed TLS library is included as component in ESP-IDF. The port includes the integration of the network layer as well as a hardware entropy source. In addition to that, the ESP32 has hardware acceleration for RSA, ECC, AES, and SHA, which can be activated in the mbed TLS configuration. We kept those parts, patched the configuration files with our PQC modifications and replaced the rest of the library with our test library. There are no timing functions provided for the library, but the hardware timer as well as the cycle counter can be accessed through the ESP32 kernel.

4.3 Fieldbus Option Card

Millions of automated industrial systems in factory plants worldwide are controlled through programmable logic controllers (PLCs) with fieldbus communication. Fieldbus connectivity is often provided by option cards. We performed tests using a fieldbus option card (FOC) that has an ARM966E-S processor, a system clock running at 100MHz, an SDRAM of 650kB, and flash storage of 8MB. There is a real-time clock with nanosecond resolution to support high real-time constrains. The card was connected via a fast Ethernet interface.

Port. The mbed TLS library had not been ported to this platform before and the network layer works in a different way than required by the mbed TLS platform interface. The mbed TLS library was designed for a Berkeley socket interface performing synchronous blocking or non-blocking calls to the network layer. However, the firmware of the FOC uses mainly asynchronous calls to the network stack such that the task never blocks the real-time system. In addition to that, a proprietary interface to the network functionality is used. Therefore, we developed two adaption layers: An application adaption layer within the firmware in order to translate asynchronous calls from the proprietary interface to the mbed TLS library interface and a network adoption layer within mbed TLS to connect the mbed TLS library to the proprietary network stack.

[9] https://www.raspberrypi.org/

[10] https://github.com/espressif/esp-idf

Table 1: Processor and peripheral specifications of the evaluated platforms.

	RPi3	ESP32	FOC	LPC
Platform	Raspberry Pi 3 Model B+	ESP32-PICO-KIT V4	Fieldbus Option Card	LPC11U68 LPCXpresso
CPU	ARM Cortex-A53	Xtensa LX6	ARM966E-S	ARM Cortex-M0+
Clock	1400 MHz	240 MHz	100 MHz	50 MHz
RAM	1024 MB	520 kB	650 kB	32 kB
Flash	SD-Card	16 MB	8 MB	256 kB
Network	Ethernet/Wi-Fi	Wi-Fi	Ethernet	–

The application adaption layer is designed to connect calls to the mbed TLS library, e.g., `mbedtls_send` or `mbedtls_receive`, to the application running on the PLC using a transparent stack adaption. The application is not aware if the socket it is using is in fact a TLS socket provided by the application adaption layer or just a regular socket. For example, we implemented a synchronous non-blocking pull mechanism for asynchronous receive operations using a separate task.

The network adaption layer provides the connection between mbed TLS and the proprietary network stack of the firmware. We designed the network adaption layer to manage a specific amount of virtual TCP sockets and connect them when needed to calls from the mbed TLS library. To maintain a logical connection between the TLS socket from the application layer to the network socket, we use an identifier in the `mbedtls_net_context`. When opening a TLS socket in the application adaption layer, all TLS contexts are initialized and a socket identifier from the network layer is received, which is then registered in the `mbedtls_net_context`.

We are using the real-time clock on the platform for timing measurements, which is not capable of holding the date and time but provides only relative timing information. For entropy generation, we only have a weak source based on time and network statistics on this platform. Since the platform has no file system, we store the server certificate as byte array in the code.

4.4 LPC11U68 LPCXpresso

The LPC11U68 LPCXpresso development board provides a 32-bit ARM Cortex-M0+ running at 50 MHz. The platform has 32 kB of RAM and 256 kB of flash memory. Since this board has no network connectivity, we do not evaluate the performance of the complete post-quantum handshake but we developed a benchmark program that measures the performance of SPHINCS$^+$ and Kyber. We measured the runtime using the real-time clock of the LPC11U68 CPU with millisecond resolution.

5 EVALUATION

We created a test setup in order to test the performance of the PQC primitives and the entire TLS handshake on the different target platforms. The center of the test setup is a desktop PC with Windows 10 64-bit and an Intel Xeon E3-1231v3 running on 3.40 GHz with 16 GB RAM. When measuring the TLS handshake performance, the desktop PC performs the remote role of client or server depending on the use case. To get practical results, the boards are connected to the desktop PC according to their network capabilities. The FOC and the Raspberry Pi are connected via Ethernet, while the ESP32 is connected via WiFi. The LPC11U68 has no network connectivity and therefore, we do not provide performance data for the entire handshake operation but only for the cryptographic primitives.

For timing measurements, we used the respective timing resources of the target platform. Since not all of the platforms provide cycle-accurate timing measurements, we converted the measurements of all platforms from their respective time domain to milliseconds for comparison. We measured the handshake performance at the level of the handshake state machine. Therefore, our measurements do not include network round-trip time, response time of the remote machine, and network-stack overhead but only the time of the handshake routines and the cryptographic primitives. In addition, we measured the performance of the cryptographic primitives directly using stand-alone benchmarking test benches. For a fair comparison, we measured each test case multiple times and report the minimum of the measurements.

5.1 Evaluation of the Cryptographic Primitives

First, we evaluate the performance of the SPHINCS$^+$ signature algorithm compared to ECDSA and of the Kyber key exchange compared to ECDHE on the embedded target platforms (both ECDSA and ECDHE using the curve SECP256R1).

5.1.1 Signature Algorithm. We report the performance of the signature operations in Table 2. The key generation of SPHINCS$^+$-128f using SHA-256 is only slightly slower than that of ECDSA for most platforms; only on the LPC and the FOC, the difference is significant. On both the LPC and the FOC, key generation of SPHINCS$^+$ SHA-256 requires several seconds (6 s and 12 s respectively as opposed to around 3 s for ECDSA). The SHAKE-256 variant however is much slower than ECDSA, reaching up to several seconds on the ESP32 (2 s), the FOC (17.3 s), and the LPC (44 s). Key generation for the signing key is typically only required very rarely and is not included in the handshake performance measurements.

Signing with SPHINCS$^+$ is more expensive than verifying a signature. This can be shown by the number of calls to the hash function. We used a counter in the implementations of the hash functions to document the number of calls. The runtime of the signature algorithm is highly correlated with this number. The number of calls depends on the chosen parameter set for the signature algorithm. For our parameter sets, the total number of calls to the hash function is for SPHINCS$^+$ using SHA-256 12,000 calls for verification compared to 280,000 calls for signing. Using SHAKE-256, it requires 11,500 calls for verification and 260,000 for signing.

Table 2: Runtime of SPHINCS$^+$ (SPHINCS$^+$-128f) and ECDSA (SECP256R1) operations (rounded to two significant figures).

Method	RPi3	ESP32	FOC	LPC
Key Generation				
SPHINCS$^+$				
SHA-256	26 ms	710 ms	6,000 ms	12,000 ms
SHAKE-256	200 ms	2,100 ms	17,000 ms	45,000 ms
ECDSA	28 ms	260 ms	3,700 ms	2,800 ms
Signing				
SPHINCS$^+$				
SHA-256	840 ms	22,000 ms	51,000 ms	380,000 ms
SHAKE-256	5,100 ms	64,000 ms	200,000 ms	1,300,000 ms
ECDSA	15 ms	290 ms	1,500 ms	1,100 ms
Verification				
SPHINCS$^+$				
SHA-256	66 ms	950 ms	2,200 ms	16,000 ms
SHAKE-256	240 ms	2,800 ms	8,900 ms	60,000 ms
ECDSA	25 ms	580 ms	2,900 ms	4,100 ms

On the Raspberry Pi, signing performance with slightly less than one second for the SHA-256 variant is much higher than the 15 ms for ECDSA. The SHAKE-256 variant even requires over 5 s. On the other platforms, signing performance degrades even more severely. Most notably signing using the SHAKE-256 variant of SPHINCS$^+$ take almost 22 min on the LPC as opposed to "only" 1 s for ECDSA. The signing operation is typically used by the server in a TLS handshake scenario.

Since the Raspberry Pi can take full advantage from its powerful CPU and the optimized SHA-256 implementation, the SPHINCS$^+$ verify operation only needs 66 ms for the SHA-256 variant and 239 ms for SHAKE-256. The ESP32 needs 0.951 s for verification with the SHA-256 variant and with the SHAKE-256 variant 2.774 s. The FOC is even slower and needs for the same operation 2.244 s with SHA-256 and 8.879 s with SHAKE-256. The LPC provides the lowest performance with 375 s (SHA-256) and 1313 s (SHAKE-256).

The runtime for the SPHINCS$^+$ operations key generation, signing, and verification compared to ECDSA is also shown in Table 2. In particular the signing operation takes significantly longer using the SPHINCS$^+$ variants than a classical ECDSA signature operation. This shows that using SPHINCS$^+$ as post-quantum secure signature scheme comes at much higher cost than using ECDSA. However, if a platform-optimized implementation or hardware support for the hash function (SHA-256 or SHAKE-256) is available, this negative performance impact probably can be ameliorated.

5.1.2 Key Exchange. In contrast to the SPHINCS$^+$ signatures, the Kyber key exchange with parameter set Kyber512 does not stress the computationally boundaries of the target platforms. We report our measurements in Table 3. The three operations key generation, encryption, and decryption can each be done in about 1 ms. The differences become more significant on the ESP32 were a key generation needs 12 ms, encryption 16 ms, and a decryption 18 ms. This trend is also visible in the measurements from the FOC with 51 ms for the key generation, 73 ms for the encryption, and 83 ms for decryption. Even on the LPC with a Cortex M0, all operations can be finished in at most 300 ms each.

Table 3: Runtime of Kyber (Kyber512) and ECDHE (SECP256R1) operations (rounded to two significant figures).

Method	RPi3	ESP32	FOC	LPC
Key Generation				
Kyber	0.79 ms	12 ms	51 ms	220 ms
ECDHE	14 ms	290 ms	1,400 ms	2,900 ms
Encryption				
Kyber	1.1 ms	16 ms	73 ms	300 ms
ECDHE	12 ms	250 ms	2,800 ms	990 ms
Decryption				
Kyber	1.1 ms	18 ms	83 ms	300 ms
ECDHE	14 ms	290 ms	1,400 ms	3,000 ms

Compared to the ECDHE key exchange, Kyber performs better in every operation on every platform. Especially on the FOC the difference is significant. Each ECDHE operation is at least one order and up to two orders of magnitude slower than a Kyber operation. The ESP32 has hardware acceleration for ECC operations that reduces the runtime for ECDHE operations, but the runtime of Kyber it still lower by one order of magnitude. The results of Botros et. al. in [11] indicate that even further improvements are possible with platform specific optimizations for Kyber.

5.2 Evaluation of the TLS Handshake

We measured performance indicators for our PQC cipher suite `TLS_KYBER_SPHINCS_WITH_AES_256_GCM_SHA256` compared to a classical handshake based on elliptic curve cryptography using `TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA256`. We investigate six different use cases: Every platform is tested as server and as client (with server authentication only) and we performed measurements for both SPHINCS$^+$ variants and the classical cipher suite as reference. The classical reference cipher suite consists of ephemeral Diffie-Hellman and DSA both using the elliptic curve variant with the SECP256R1 curve, which provides a 128-bit (classical) security level. As mentioned above, we do not measure a complete TLS key exchange on the LPC, because it does not have a network interface, but only on the Raspberry Pi, the ESP32, and the FOC.

5.2.1 Handshake Message Sizes. Table 4 shows the impact of the chosen PQC schemes on the size of the different TLS messages that are exchanged during the handshake. Due to the large size of SPHINCS$^+$ signatures, the size of both `ServerCertificate` and `ServerKeyExchange` is increased by two orders of magnitude. The `ClientKeyExchange` mainly contains the key exchange data from the client, i.e., a Kyber cipher text, which results in a more moderate increment of the message size by a factor of more than 10.

5.2.2 Runtime. We measure the runtime of each handshake for all use cases. As mentioned before, we only measure the handshake routines and cryptographic primitives but not the network stack and network response times. Since client and server process the

Table 4: Comparison of handshake message sizes between the classical and the post-quantum cipher suite.

Type	Server- Certificate	Server- KeyExchange	Client- KeyExchange
PQC	17, 330 B	17, 780 B	800 B
Classical	553 B	144 B	60 B

exchanged messages differently, there are two kinds of data sets. The cryptographic workload for the server is the signature generation with SPHINCS$^+$, the generation of an ephemeral Kyber keypair, and the decryption of the Kyber key-exchange data. On the other side, the client needs to verify a SPHINCS$^+$ signature and encrypts the key-exchange data with Kyber.

Depending on the platform, the measurements are more or less reliable and also differ in granularity. While the FOC provides nanosecond resolution and a reliable industrial grade real-time clock, the Raspberry Pi uses a cycle counter, and the ESP32 uses an external clock that has a resolution of microseconds.

The performance results of the entire TLS handshake comparing the PQC-version to the current version using classical cryptography are shown in Table 5 and explained in the following.

Raspberry Pi. A classical client handshake takes a minimum of 49 ms on a Raspberry Pi 3B+. The SHA-256 variant of KYBER_-SPHINCS+ needs 67 ms and thus is about 1.3 times slower. The SHAKE-256 variant has an about 4.9 times smaller runtime than the classical variant. The difference becomes even more significant for the server. A 43 ms classical server handshake runtime increases by almost 20 times (SHA-256) and even by about 120 times (SHAKE-256) when performed with post-quantum primitives.

ESP32. Performing a classical client handshake on a ESP32 takes a minimum of 1134 ms, while the SHA-256 variant of KYBER_-SPHINCS+ needs 974 ms, so the post-quantum handshake performs about 1.2 times faster. The SHAKE-256 variant needs about 2.5 times more runtime than the classical variant. While at least one post-quantum handshake variant on the client is faster than the classical handshake, on the server side the classical variant completely takes over. This is because the ESP32 provides accelerators for big integer arithmetic. The port of mbed TLS makes use of this for the ECDHE and ECDSA computations, while the hash functions are computed entirely in software. Even though the ESP32 has an accelerator for the SHA-256 algorithm, mbed TLS does not take advantage of it for computing the post-quantum signatures. A 892 ms classical server handshake runtime increases by about 25 times (SHA-256) and even 72 times (SHAKE-256) when performed with PQC primitives.

FOC. Performing a classical client handshake on the fieldbus option card takes a minimum of 5743 ms, while the SHA-256 variant of KYBER_SPHINCS+ needs only 2349 ms and therefore is about 2.4 times faster than the classical variant. The SHAKE-256 variant has an about 1.6 times longer runtime than the classical variant. While one of the post-quantum handshake variants on the client is faster than the classical handshake, on the server side the cost for SPHINCS$^+$ signatures dominates the runtimes. The SHA-256 post-quantum handshake is almost 12 times slower than the classical server handshake runtime of 4401 ms and the SHAKE-256 variant of the post-quantum handshake is even about 46 times slower. Here, the distribution between server and client workload for the post-quantum algorithms is very unbalanced, while the classical algorithm has similar runtimes for client and server.

Table 5: Comparison of handshake runtime for different cipher suites (rounded to two significant figures).

Cipher Suite	RPi3	ESP32	FOC
Server			
KYBER-SPHINCS+-SHA-256	840 ms	23,000 ms	52,000 ms
KYBER-SPHINCS+-SHAKE-256	5,100 ms	64,000 ms	200,000 ms
ECDHE-ECDSA	43 ms	890 ms	4,400 ms
Client			
KYBER-SPHINCS+-SHA-256	67 ms	970 ms	2,300 ms
KYBER-SPHINCS+-SHAKE-256	240 ms	2,800 ms	9,000 ms
ECDHE-ECDSA	49 ms	1,100 ms	5,700 ms

The runtime measurements we are reporting in Tables 2 and 3 for the cryptographic primitives and in Table 5 for the TLS handshake are the optimal values achieved in several independent test runs and are not correlated to each other. We also measured both the runtime of the cryptographic primitives and of the corresponding TLS handshake in additional tests. In most cases, the cryptographic primitives take over 98% of the runtime of the entire handshake, i.e., the overhead within the TLS library for parsing data etc. is marginal. However, for the classical cipersuite with ECDHE and ECDSA on the ARM-platforms (Raspberry PI and FOC), the cryptographic operations take only about 65% of the overall handshake — due to the optimized ECC implementations in the mbed TLS library for ARM processors.

5.2.3 Code Size. The size of the mbed TLS dynamic library is not a good indicator for the actual code size in order to achieve a fair assessment of the PQC overhead, because it contains code that is not required by the use cases and thus might lead to and underestimation of the impact of the PQC primitives. Instead, the actual code sizes of a server and a client are more meaningful. The map files of these programs include the parts of the mbed TLS library that are actually used, which is a more practically relevant definition for the code size. We evaluated the following sections of the client and server binaries:

- *DATA* holds initialized static variables. For these variables, space is reserved on the read-write section of the flash memory that stores the constant initialization values. During runtime, this section may be copied to RAM.
- *BSS* contains uninitialized static variables. Depending on the platform, these values are initialized differently on startup. The value of this variable will always be in RAM since there is no constant for initialization. Generally, RAM is a very constrained resource on embedded devices.
- *TEXT* contains the executable instructions and constant values. This section is stored in flash memory.

Table 6: Code sizes of client and server.

	TEXT		BSS		DATA	
Cipher Suite	Client	Server	Client	Server	Client	Server
			RPi3			
KYBER-SPHINCS+-SHA-256	91, 900 B	114, 340 B	8, 812 B	8, 820 B	5, 811 B	63, 505 B
KYBER-SPHINCS+-SHAKE256	91, 900 B	114, 340 B	8, 812 B	8, 820 B	5, 811 B	63, 505 B
ECDHE-ECDSA	93, 492 B	94, 336 B	8, 824 B	8, 824 B	8, 170 B	12, 541 B
			ESP32			
KYBER-SPHINCS+-SHA-256	61, 682 B	64, 602 B	32 B	32 B	8, 803 B	33, 175 B
KYBER-SPHINCS+-SHAKE256	61, 682 B	64, 602 B	32 B	32 B	8, 803 B	33, 175 B
ECDHE-ECDSA	46, 463 B	49, 346 B	36 B	36 B	7, 225 B	8, 509 B
			FOC			
KYBER-SPHINCS+-SHA-256	62, 976 B	64, 840 B	8, 764 B	8, 764 B	7, 897 B	31, 894 B
KYBER-SPHINCS+-SHAKE256	62, 976 B	64, 840 B	8, 764 B	8, 764 B	7, 897 B	31, 894 B
ECDHE-ECDSA	66, 056 B	67, 312 B	8, 770 B	8, 770 B	6, 564 B	7, 676 B

To ensure that the memory footprint is as small as possible, we optimized mbed TLS specifically for the test cases by using the mbed TLS configuration file `config.h` to disable as many features of the library as possible and by using compiler settings for size-optimized code. Depending on the test case or platform, we used different sets of configurations, but some options in the configuration are mandatory for every test setting:

- *Base system:* These are basic library functions, e.g., public key algorithms, that are needed by most of the modules.
- *TLS:* All test cases are using TLS version 1.2.
- *Cipher suites:* All tested cipher suites have some algorithms in common, e.g., AES-256 in GCM mode for bulk encryption and SHA-256 as hash function for the MACs.
- *X509:* Certificates are used in all handshakes. All variants of SPHINCS$^+$ and ECDSA use X509 certificates for the public key of the server.

Further options depend on the test case. For the classical handshake we used the cipher suite `TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA256` with the elliptic curve SECP256R1 for ECDHE key exchange and ECDSA signatures. We used our PQC cipher suite `TLS_KYBER_SPHINCS_WITH_AES_256_GCM_SHA256` for the post-quantum handshake with Kyber key exchange and SPHINCS$^+$ signatures with SHA-256 or SHAKE-256 as hash functions. In addition to that, we enabled handshake fragmentation for the post-quantum handshake, because of the large message sizes.

According to the mbed TLS developers it is possible to reduce the combined size in ROM and RAM to only 30 kB [24]. Our implementation is not that tiny, but with a complete code size minimum of 54 kB for a classical handshake and 70 kB for a post-quantum client is quite close. For the server, the minimum is 60 kB with the classical handshake and the smallest post-quantum server has a size of 98 kB, because of the storage requirements for certificate handling. While an ECDSA server certificate can be stored within 1 kB, the SPHINCS$^+$ certificate takes 25 kB. Independent of the public key algorithms the BSS section is nearly empty on the ESP32 and about 9 kB large on all other platforms. This is because of the hardware acceleration for AES on the ESP32. On the other platforms, the AES implementation is responsible for nearly the complete BSS size. The post-quantum variants have always the exact same code size. The only difference between these two is the use of the hash function in SPHINCS$^+$. However, it is not possible to save the code size of the unused hash function, because the SHAKE-256 implementation is also used by Kyber and the SHA-256 implementation is also used in other parts of the TLS protocol. The overall differences between classical and post-quantum handshake are not significant, except for the server certificate.

In Table 6 it can be seen that the client code sizes per platform are quite equal for FOC and Raspberry Pi, the ESP32 uses its hardware accelerator for big number operations, which saves some code size for the classical handshake. On the server there are more significant differences, but Table 6 shows that at least on the FOC the code sizes for post-quantum and classical server are equal when adjusted by the certificate size.

5.2.4 RAM Usage. To measure peak stack usage, we filled the entire available stack space with a specific pattern (i.e., `0xAA`) at the beginning. After performing the test run, we checked the occurrence of this pattern on the stack as indication on stack usage. In addition, we used map-file information.

The full TLS handshake can be performed with a stack peak of only about 20kB. While this is really small, the library dynamically allocates buffers during runtime. An analysis of the allocations showed the potentially smallest practical memory footprint.

Most notable allocations are the incoming and outgoing message buffers as well as a temporary buffer for the handshake fragmentation. The rest are mostly small buffers and contexts. The minimum size of the message buffers is the size of the biggest handshake message. These messages turned out to be messages that contain a SPHINCS$^+$ signature, `Certificate`, and `ServerKeyExchange`. A single self-signed X509 certificate with a SPHINCS$^+$ signature and public key is about 17kB big and a Kyber key exchange data with a SPHINCS$^+$ signature have a size of about 17.8kB. From these values, a safe message buffer size of 20kB can be derived.

6 CONCLUSION

In this paper, we described our integration of the post-quantum KEM scheme Kyber (for key exchange) and the post-quantum signature scheme SPHINCS$^+$ into the embedded TLS library embed TLS. First, we measured the performance of the PQC primitives on four embedded platforms, a Raspberry Pi, an ESP32, a fieldbus option card, and a heavily resource restricted LPC. Then, we evaluated the performance of a PQC-TLS variant using Kyber and SPHINCS$^+$ on three embedded platforms (excluding the LPC due to its lack of network interfaces) and compared the performance to classical TLS with corresponding ECC primitives.

We measured the required time for performing the PQC primitives and their ECC counterparts on the different platforms. Kyber performs on all platforms better than the ECDHE key exchange. However, as expected, SPHINCS$^+$ is in general slower than ECDSA. Especially signing takes significantly longer using SPHINCS$^+$ in both tested variants with SHAKE-256 and with SHA-256. Since there is optimized source code of SHA-256 for ARM platforms, the SHA-256 variant performed significantly better than the SHAKE-256 on all platforms that we tested. However, we expect that by integrating hardware support for calculating the hash function (SHAKE-256 or SHA-256) the performance can be significantly improved. Thus, for enabling current embedded systems to efficiently calculate digital signatures using hashed-based PQC schemes, appropriate hardware acceleration for the required hash functions should be integrated.

We also performed measurements of the TLS handshake, which showed that the most of the time is consumed by performing the PQC schemes. The size of all related handshake messages increases significantly in particular if SPHINCS$^+$ signatures need to be transmitted. In addition, we measured code and RAM size and showed that the integration of PQC schemes on embedded devices is feasible with relatively low overhead.

The cost of computing signatures with SPHINCS$^+$ poses the biggest obstacle for using the selected embedded platforms as a TLS server. However, the performance of the PQC-TLS client is similar for the PQC variant as for the ECC variant. Thus, even without dedicated hardware acceleration, PQC can be used in embed TLS as of today in scenarios where the embedded device has the role of the TLS client. For the use of PQC in an embedded TLS server, we recommend the addition of hardware accelerators in order to speed up the SPHINCS$^+$ signing computations or to use a more resource friendly PQC signature primitive.

ACKNOWLEDGMENTS

The work presented in this paper has been partly funded by the German Federal Ministry of Education and Research (BMBF) under the project “QuantumRISC” (ID 16KIS1033K) [32].

REFERENCES

[1] Infineon Technologies AG. 2017. *Ready for tomorrow: Infineon demonstrates first post-quantum cryptography on a contactless security chip.* https://www.infineon.com/cms/en/about-infineon/press/press-releases/2017/INFCCS201705-056.html

[2] Miklós Ajtai. 1996. Generating Hard Instances of Lattice Problems (Extended Abstract). In *ACM Symposium on Theory of Computing — STOC '96*. ACM, 99–108.

[3] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. 2016. Post-quantum Key Exchange – A New Hope. In *USENIX Security Symposium — USENIX Security 2016*. USENIX Association, 327–343.

[4] Erdem Alkim, Philipp Jakubeit, and Peter Schwabe. 2016. NewHope on ARM Cortex-M. In *Security, Privacy, and Applied Cryptography Engineering — SPACE 2016 (LNCS)*, Claude Carlet, M. Anwar Hasan, and Vishal Saraswat (Eds.), Vol. 10076. Springer, 332–349.

[5] Matthew Arcus. 2018. *Using the Cycle Counter Registers on the Raspberry Pi 3.* https://matthewarcus.wordpress.com/2018/01/27/using-the-cycle-counter-registers-on-the-raspberry-pi-3/

[6] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. 2017. *CRYSTALS-Kyber — Submission to the NIST post-quantum project.*

[7] Daniel J. Bernstein, Christoph Dobraunig, Maria Eichlseder, Scott Fluhrer, Stefan-Lukas Gazdag, Andreas Hülsing, Panos Kampanakis, Stefan Kölbl, Tanja Lange, Martin M. Lauridsen, Florian Mendel, Ruben Niederhagen, Christian Rechberger, Joost Rijneveld, and Peter Schwabe. 2017. *SPHINCS+ — Submission to the NIST post-quantum project.*

[8] Daniel J. Bernstein, Daira Hopwood, Andreas Hülsing, Tanja Lange, Ruben Niederhagen, Louiza Papachristodoulou, Michael Schneider, Peter Schwabe, and Zooko Wilcox-O'Hearn. 2015. SPHINCS: Practical Stateless Hash-Based Signatures. In *Advances in Cryptology — EUROCRYPT 2015 (LNCS)*, Elisabeth Oswald and Marc Fischlin (Eds.), Vol. 9056. Springer, 368–397.

[9] 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 *IEEE European Symposium on Security and Privacy — EuroS&P 2018*. IEEE, 353–367.

[10] Joppe W. Bos, Craig Costello, Michael Naehrig, and Douglas Stebila. 2015. Post-Quantum Key Exchange for the TLS Protocol from the Ring Learning with Errors Problem. In *IEEE Symposium on Security and Privacy — SP 2015*. IEEE, 553–570.

[11] Leon Botros, Matthias J. Kannwischer, and Peter Schwabe. 2019. Memory-Efficient High-Speed Implementation of Kyber on Cortex-M4. In *Progress in Cryptology — AFRICACRYPT 2019 (LNCS)*, Johannes Buchmann, Abderrahmane Nitaj, and Tajje-eddine Rachidi (Eds.), Vol. 11627. Springer, 209–228.

[12] Matt Braithwaite. 2016. *Experimenting with Post-Quantum Cryptography.* https://security.googleblog.com/2016/07/experimenting-with-post-quantum.html

[13] Yun-An Chang, Ming-Shing Chen, Jong-Shian Wu, and Bo-Yin Yang. 2014. Postquantum SSL/TLS for Embedded Systems. In *Service-Oriented Computing and Applications — SOCA 2014*. IEEE Computer Society, 266–270.

[14] NIST CSRC. 2017. *Post-Quantum Cryptography Round 1 Submissions.* https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Round-1-Submissions

[15] T. Dierks and E. Rescorla. 2008. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246.

[16] Markus Grassl, Brandon Langenberg, Martin Roetteler, and Rainer Steinwandt. 2016. Applying Grover's Algorithm to AES: Quantum Resource Estimates. In *Post-Quantum Cryptography — PQCrypto 2016 (LNCS)*, Tsuyoshi Takagi (Ed.), Vol. 9606. Springer, 29–43.

[17] Tim Güneysu, Markus Krausz, Tobias Oder, and Julian Speith. 2018. Evaluation of Lattice-Based Signature Schemes in Embedded Systems. In *IEEE International Conference on Electronics, Circuits and Systems — ICECS 2018*. IEEE, 385–388.

[18] Tim Güneysu, Tobias Oder, Thomas Pöppelmann, and Peter Schwabe. 2013. Software Speed Records for Lattice-Based Signatures. In *Post-Quantum Cryptography — PQCrypto 2013 (LNCS)*, Philippe Gaborit (Ed.), Vol. 7932. Springer, 67–82.

[19] James Howe, Ciara Moore, Máire O'Neill, Francesco Regazzoni, Tim Güneysu, and K. Beeden. 2016. Standard Lattices in Hardware. In *Design Automation Conference — DAC 2016*. ACM, 162:1–162:6.

[20] James Howe, Tobias Oder, Markus Krausz, and Tim Güneysu. 2018. Standard Lattice-Based Key Encapsulation on Embedded Devices. *IACR Transactions on Cryptographic Hardware and Embedded Systems — TCHES* 2018, 3 (Aug. 2018), 372–393.

[21] Andreas Hülsing, Joost Rijneveld, and Peter Schwabe. 2016. ARMed SPHINCS – Computing a 41 KB Signature in 16 KB of RAM. In *Public-Key Cryptography — PKC 2016 (LNCS)*, Chen-Mou Cheng, Kai-Min Chung, Giuseppe Persiano, and Bo-Yin Yang (Eds.), Vol. 9614. Springer, 446–470.

[22] Po-Chun Kuo, Wen-Ding Li, Yu-Wei Chen, Yuan-Che Hsu, Bo-Yuan Peng, Chen-Mou Cheng, and Bo-Yin Yang. 2017. High Performance Post-Quantum Key Exchange on FPGAs. Cryptology ePrint Archive, Report 2017/690.

[23] Leslie Lamport. 1979. *Constructing digital signatures from a one way function.* Technical Report SRI-CSL-98. SRI International Computer Science Laboratory.

[24] ARM Limited. [n.d.]. *Tiny SSL Library.* https://tls.mbed.org/tiny-ssl-library

[25] Zhe Liu, Thomas Pöppelmann, Tobias Oder, Hwajeong Seo, Sujoy Sinha Roy, Tim Güneysu, Johann Großschädl, Howon Kim, and Ingrid Verbauwhede. 2017. High-Performance Ideal Lattice-Based Cryptography on 8-Bit AVR Microcontrollers. *ACM Trans. Embedded Comput. Syst.* 16, 4 (2017), 117:1–117:24.

[26] Ralph C. Merkle. 1990. A Certified Digital Signature. In *Advances in Cryptology — CRYPTO' 89 Proceedings*, Gilles Brassard (Ed.). Springer New York, New York, NY, 218–238.

[27] Tobias Oder and Tim Güneysu. 2019. Implementing the NewHope-Simple Key Exchange on Low-Cost FPGAs. In *Progress in Cryptology — LATINCRYPT 2017 (LNCS)*, Tanja Lange Orr and Dunkelman (Eds.), Vol. 11368. Springer, 128–142.

[28] Tobias Oder, Thomas Pöppelmann, and Tim Güneysu. 2014. Beyond ECDSA and RSA: Lattice-based Digital Signatures on Constrained Devices. In *Design Automation Conference — DAC 2014*. ACM, 110:1–110:6.

[29] Christian Paquin, Douglas Stebila, and Goutam Tamvada. 2020. Benchmarking Post-Quantum Cryptography in TLS. Cryptology ePrint Archive, Report 2019/1447. In *Post-Quantum Cryptography — PQCrypto 2020 (LNCS)*, Jintai Ding and Jean-Pierre Tillich (Eds.). Springer.

[30] Thomas Pöppelmann, Tobias Oder, and Tim Güneysu. 2015. High-Performance Ideal Lattice-Based Cryptography on 8-Bit ATxmega Microcontrollers. In *Progress in Cryptology — LATINCRYPT 2015 (LNCS)*, Vol. 9230. Springer, 346–365.

[31] Manuel Pégourié-Gonnard. 2017. *Porting mbed TLS to a new environment or OS*. https://tls.mbed.org/kb/how-to/how-do-i-port-mbed-tls-to-a-new-environment-OS

[32] QuantumRISC. 2020. *QuantumRISC — Next Generation Cryptography for Embedded Systems*. https://www.quantumrisc.org/

[33] Peter W. Shor. 1994. Algorithms for quantum computation: discrete logarithms and factoring. In *Foundations of Computer Science*. IEEE, 124–134.

[34] Peter W. Shor. 1999. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *SIAM Rev.* 41, 2 (1999), 303–332.

[35] Dimitrios Sikeridis, Panos Kampanakis, and Michael Devetsikiotis. 2020. Post-Quantum Authentication in TLS 1.3: A Performance Study. Cryptology ePrint Archive, Report 2020/071. In *Network and Distributed System Security Symposium — NDSS 2020*. The Internet Society.

[36] Julian Speith, Tobias Oder, Marcel Kneib, and Tim Güneysu. 2018. A lattice-based AKE on ARM Cortex-M4. In *BalkanCryptSec 2018*. https://www.emsec.ruhr-uni-bochum.de/research/publications/ake-m4/