SLOVAK UNIVERSITY OF TECHNOLOGY IN BRATISLAVA
FACULTY OF ELECTRICAL ENGINEERING AND INFORMATION TECHNOLOGY

Registration number: FEI-104372-53536

POST-QUANTUM KEY ESTABLISHMENT

DOCTORAL THESIS

2022 **Ing. Peter Špaček**

SLOVAK UNIVERSITY OF TECHNOLOGY IN BRATISLAVA
FACULTY OF ELECTRICAL ENGINEERING AND INFORMATION TECHNOLOGY

Registration number: FEI-104372-53536

POST-QUANTUM KEY ESTABLISHMENT

DOCTORAL THESIS

Study Programme:	Applied Informatics
Field Number:	2511
Study Field:	9.2.9 Applied Informatics
Training Workplace:	Institute of Computer Science and Mathematics
Supervisor:	prof. Ing. Pavol Zajac, PhD.

Bratislava 2022 **Ing. Peter Špaček**

Slovak University of Technology in Bratislava
Institute of Computer Science and Mathematics

Faculty of Electrical Engineering and Information Technology
Academic year: 2021/2022
Reg. No.: FEI-104372-53536

STU FEI

DISSERTATION THESIS TOPIC

Student: **Ing. Peter Špaček**
Student's ID: 53536
Study programme: Applied Informatics
Study field: Computer Science
Thesis supervisor: prof. Ing. Pavol Zajac, PhD.
Head of department: Dr. rer. nat. Martin Drozda
Workplace: Ústav informatiky a matematiky

Topic: **Post-quantum key establishment**

Language of thesis: English

Specification of Assignment:

The aim of the thesis is to design and implement a secure post-quantum protocol for key establishment that should be compatible with TLS protocol design.

Tasks:
1. Identify suitable post-quantum primitives, and evaluate their suitability as a building block for a key establishment protocol.
2. Design a post-quantum protocol for key establishment.
3. Implement the protocol in a secure way.
4. Evaluate the solution and compare it with existing ones.

Selected bibliography:

1. Špaček, P. – Zajac, P. *Implementation of McEliece cryptosystem into TLS.* Diplomová práca. 2017. 38 p.
2. *Post-quantum cryptography.* Cham: Springer, 2017. ISBN 978-3-319-59878-9.
3. Rescorla, E. – Dierks, T. The Transport Layer Security (TLS) Protocol. [online]. 2008. URL: http://tools.ietf.org/pdf/rfc5246.pdf.

Deadline for submission of Dissertation thesis: 31. 05. 2022

Approval of assignment of Dissertation thesis: 12. 07. 2022

Assignment of Dissertation thesis approved by: prof. RNDr. Gabriel Juhás, PhD. – Chairperson of Field of Study Board

ABSTRACT

SLOVAK UNIVERSITY OF TECHNOLOGY IN BRATISLAVA
FACULTY OF ELECTRICAL ENGINEERING AND INFORMATION TECHNOLOGY

Study Programme:	Applied Informatics
Author:	Ing. Peter Špaček
Doctoral thesis:	Post-Quantum key establishment
Supervisor:	prof. Ing. Pavol Zajac, PhD.
Place and year of submission:	Bratislava 2022

This dissertation thesis deals with the problem of secure post-quantum key exchange. It focuses on the TLS communication protocol and its security in the post-quantum world. Quantum computers and algorithms developed for them are able to break the security of currently used mechanisms, which provide confidentiality and integrity of messages. In this work, we present concepts needed to understand the domain and explain how quantum technologies threaten communication security. We describe cryptographic protocols, especially the TLS protocol, on which we focus in this work. We identify the mechanisms that need to be replaced to achieve post-quantum security in TLS key exchange. The NIST standardization process to create a new post-quantum public-key cryptography standard brings together a collection of candidates we can use to address this issue. We design, implement, and test post-quantum key exchange in TLS Handshake using these algorithms. Operational security is also considered. Using the concept of Trusted Execution Environment (TEE) and propose a post-quantum HSM, which serves to separate the unsecured and secured environment. Here we also take into account side-channel resistance. The work analyzes and proposes solutions for several scenarios:

1. The post-quantum key exchange in TLS protocol
2. The post-quantum key exchange in TLS protocol in limited (IoT) devices
3. The post-quantum key exchange in TLS protocol using HSM module
4. Introducing the concept of the PQcube post-quantum HSM module, which includes exchange, administration, and secure key usage.

We introduce a set of experiments that test our solutions. We list unique measurements from the use of post-quantum key exchange in the TLS protocol in the scenarios mentioned above. In this work we show that our solutions can be used in the real world, and we prove this using various benchmarks.

Keywords: Post-quantum, TLS, IoT, Protocol, HSM

SÚHRN

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Študijný program:	Aplikovaná informatika
Autor:	Ing. Peter Špaček
Dizertačná práca:	Bezpečná post-kvantová kryptografia
Vedúci záverečnej práce:	prof. Ing. Pavol Zajac, PhD.
Miesto a rok predloženia práce:	Bratislava 2022

Táto dizertačná práca sa zaoberá problematikou bezpečnej post-kvantovej výmeny kľúčov. Bližšie sa zameriava na komunikačný protokol TLS a jeho bezpečnosť v post-kvantovom svete. Kvantové počítače a algoritmy pre ne vyvinuté sú schopné prelomiť bezpečnosť aktuálne využívaných mechanizmov na zabezpečenie dôvernosti a integrity správ. V práci predstavujeme koncepty potrebné na pochopenie problematiky a vysvetľujeme, ako kvantové technológie ohrozujú bezpečnosť komunikácie. Opisujeme používané kryptografické protokoly, a hlavne protokol TLS, na ktorý sa v práci zameriavame. Identifikujeme mechanizmy, ktoré treba vymeniť na dosiahnutie post-kvantovej bezpečnosti pri probléme výmeny kľúčov. Štandardizačný proces NIST na vytvorenie nového post-kvantového štandardu kryptografie s verejným kľúčom priniesol niekoľko vhodných kandidátov, ktorých vieme využiť na riešenie tohoto problému. Navrhujeme, implementujeme a testujeme post-kvantovú výmenu kľúčov v TLS Handshake s využitím práve týchto algoritmov. V práci zohľadňujeme aj operačnú bezpečnosť. Nasledujeme trend Trusted Execution Environment (TEE) a navrhujeme koncept post-kvantového modulu HSM, ktorý slúži na oddelenie nezabezpečeného a zabezpečeného prostredia. Tu zohľadňujeme aj odolnosť voči útokom s využitím postranných kanálov. Práca analyzuje a navrhuje riešenia pre niekoľko scenárov:

1. Post-kvantová výmena kľúčov v TLS protokole
2. Post-kvantová výmena kľúčov v TLS protokole v limitovaných (IoT) zaradeniach
3. Post-kvantová výmena kľúčov v TLS protokole za využitia HSM modulu
4. Predstavenie konceptu post-kvantového HSM modulu PQcube, zahrňajúceho výmenu, správu, aj bezpečné využívanie kľúčov.

Prinášame sadu experimentov, ktoré nami navrhované a implementované riešenia testujú, a prinášame jedinečné merania z využitia post-kvantovej výmeny kľúčov v TLS protokole v našich scenároch. V práci ukazujeme, že nami navrhované riešenia je možné využiť v praxi a toto tvrdenie podkladáme meraniami.

Kľúčové slová: Post-kvantová kryptografia, TLS, HSM, protokol, IoT

Acknowledgments

I would like to express gratitude to my thesis supervisor prof. Ing. Pavol Zajac, PhD. I would like to thank Institute of Computer Science and Mathematics, Faculty of Electrical Engineering and Information Technology and Slovak University of Technology for providing a stimulating environment for me to be able to grow academically and professionally. I would like to thank European Union, NATO and SAIA for support of academic visits and collaboration with other universities. I am deeply grateful to my parents and my wife for support, help and assistance during my studies.

Contents

List of Figures and Tables

List of Abbreviations

AEAD	Authenticated encryption with associated data
AES	Advanced Encryption Standard
API	Application Programming Interface
DH	Diffie-Hellman
ECC	Elliptic Curve Cryptography
ECDSA	Elliptic Curve Digital Signature Algorithm
EdDSA	Edwards-curve Digital Signature Algorithm
GCM	Galois/Counter Mode
HMAC	Hash-based message authentication code
HSM	Hardware security module
IoT	Internet of things
IV	Initialisation Vector
KEM	Key Encapsulation Mechanism
LWE	Learning With Errors
LWR	Learning With Rounding
NIST	National Institute of Standards and Technology
NSS	Network Security Services
NTT	Number theoretic transform
OQS	Open Quantum Safe
OSI	Open Systems Interconnection
PMS	Pre-Master Secret
pq3	Post Quantum Cube
pq3TLS	Post-Quantum Cube TLS
pqlimTLS	Post-Quantum TLS for limited devices
pqTLS	Post-Quantum TLS
PRF	Pseudorandom Function
PSK	Pre-Shared Key
RSA	Rivest–Shamir–Adleman
s2n	Signal to Noise
SDK	Software development kit
SSL	Secure Sockets Layer
TEE	Trusted Execution Environment
TLS	Transport Layer Security

TPM Trusted Platform Module

Introduction

New discoveries in the field of quantum technologies [1], [2], [3], [4] show that quantum computers are becoming reality. Quantum computers and application-specific quantum hardware pose new threats to currently used cryptography [5], [6]. Whoever has quantum technology can get access to state secrets, business-critical information, financial transactions, and read private messages. In order to prevent this from happening, we need to replace cryptographic algorithms that we use today with more secure, post-quantum ones.

In November 2017, NIST collected 69 "complete and proper" papers in the post-quantum standardization process [7]. At the beginning of 2019, second-round candidates for the NIST competition were published. At the time of writing this thesis, the finalists of the third round are already known, and we are getting ready for a new public-key cryptography standard.

In this work, we analyze the aspects of TLS protocol in a post-quantum setting and propose our solution. The most common protocol responsible for security on the Internet is TLS. In August 2018, TLS version 1.3 was published. It is the most secure transport layer protocol so far [8]. Nevertheless, it was not designed with a quantum computer in mind.

The focus of this thesis is to examine and evaluate the possibilities of making TLS key exchange secure in the post-quantum world.

This means we aim to design, implement and evaluate a post-quantum handshake protocol compatible with TLS. We prototype Hardware Security Module for post-quantum public-key algorithms and use such a solution in the TLS setting. We should also have in mind that the delays caused by Hardware Security Module in the post-quantum TLS setting should be still tolerable.

Research objectives

We can identify several objectives connected to the aim of our work. These are subgoals we will follow when elaborating this research work:

- We need to review the details of the TLS protocol, its mechanisms, and the structure of its messages. We will identify aspects that can be reused and aspects that need to be replaced.
- We will search for possible replacement mechanisms for key exchange that should resist quantum computers.
- We will collect suitable TLS implementations for our experiments and choose the best one for the practical aspect of our research.
- We will design and implement a post-quantum key exchange mechanism in the TLS context.
- In designing a new TLS-like protocol, we will consider the growing world of IoT devices. This should be projected into key exchange choices, as well as the architecture of the new protocol.
- We will find available methods for ensuring operational security and implement the chosen solution.
- We will test our solution with a set of experiments to find whether our solution is suitable for use in practice.

Directly from our objectives, a few important questions have emerged. First is the question of operational security. We need to consider not only the cryptoanalysts breaking ciphers, but also the "line of least resistance" attackers. This includes malware obtaining secret keys, monitoring cryptographic operations to leak some side-channel information, etc.

Quantum-secure algorithms themselves require more memory space, more computational power, or/and more time. Symmetric cryptography needs longer keys and blocks, public-key cryptography needs to be changed completely, and may have different challenges, e.g. enc/dec time ratio. Also, changes are required on the protocol level to achieve our goal. This may result in a delays in communication establishment, a need for memory space for the keys, and overall delays in communication. So, another important aspect is the trade-off or the "cost" of post-quantum secure TLS protocol.

Overview of the thesis

The first chapter presents the preliminaries. We describe post-quantum security, we review the details of the TLS protocol, its mechanisms, and the structure of its messages. We present operational security and similar research. We describe TLS implementations. We found possible replacement mechanisms for key exchange that should resist quantum computers in the NIST post-quantum standardization process and explained related concepts.

The post-quantum level of security in TLS requires design changes. We present those changes in the second chapter, together with possible candidates for key exchange algorithms. We identify parts of TLS that can be reused and parts that need to be replaced. We also consider limited devices and the growing world of IoT. We introduce SEcube and our concept of post-quantum HSM to provide a higher level of operational security. Also, we add an option to use of protected implementation of Kyber.

We chose the s2n implementation of TLS and modify it to support our experiments. In the third chapter, we mention some of the implementation challenges that we encountered to explain the artifact of our design research. We also describe the building blocks and sources for the implementation of our study.

In the chapter four, we test and evaluate all components and steps of the post-quantum TLS key agreement process. We show the successful post-quantum decryption and key agreement on the host and HSM use. We also show successful symmetric cryptography use in TLS Record protocol after post-quantum key exchange.

1 Preliminaries

This section presents the preliminaries required to understand our work. First, we present state of the art of post-quantum security, and how are quantum machines threatening currently used cryptography. Them, we focus on Transport Layer Security (TLS) cryptographic protocol, a widely used cryptographic protocol for key exchange. Because we want to explore the possibilities of a post-quantum secure key agreement, we need to understand the details and mechanisms used in TLS now to find new and more secure ways to exchange keys. We also take a quick look at the situation in IoT devices to see that it is also essential to find post-quantum alternatives to cryptographic algorithms and mechanisms in limited devices. Because we choose the TLS protocol for our research, we provide an overview of current TLS implementations. Next, we explain the basic principles of post-quantum public-key algorithms and NIST standardization process for post-quantum public-key standards. We emphasize the importance of practical security, and we describe the concept of running the code in a trusted environment. We present hardware security modules, and finally, research similar to this thesis.

1.1 Security in the post-quantum world

Since the first ideas about quantum computing in the 1980s, scientists have been researching new ways to use discoveries in physics for computing [1]. The excitement grew even further in 1994, when Shor published an algorithm for integer factorization. At the moment, we live on the edge of a new era. New results in the development of quantum computers [2], [3], [4] will have severe consequences. With more computing power than the adversaries have, new threads appear as well. Algorithms currently used to secure information, especially for key exchange and digital signatures, are vulnerable to new types of attacks that emerged with the development of quantum computers.

It is not important for this work to understand how a quantum computer works. We can perceive it as a black box that is able to run specific algorithms. Two notable publications impacted the world of cryptography so much that cryptographers started to consider new replacements for currently used solutions:

1. **Shor's algorithm**, published in 1994 by Peter Shor [9], is a quantum computer algorithm for prime factorization and discrete logarithms in polynomial time.

2. Lov Grover published a database search algorithm in 1996 [10]. One interesting consequence is that **Grover's algorithm** is able to find the n-bit key with complexity $\sqrt{2^n}$ iterations [11]. Note, that a conventional computer would need 2^{n-1} iteration to find the same key.

1.1.1 Impact on asymmetric cryptography

Most used public-key algorithms for key exchange or digital signatures are vulnerable to quantum machines [5], [6]. Based on the integer factorization problem, the RSA cipher is the apparent victim of Shor's algorithm. Other commonly used public ciphers, Diffie-Hellman or its variant based on the elliptic curves over finite fields (ECDH). As mentioned in [12], Shor's algorithm can also be used for computing discrete logarithms. And because solving Diffie-Hellman problem can be achieved by solving discrete logarithms [13], Diffie-Hellman is also not suitable for post-quantum usage. Proos and Zalka [14] have shown that breaking cryptography based on elliptic curves is more straightforward than breaking RSA.

1.1.2 Impact on symmetric cryptography

Symmetric ciphers are not entirely broken with Grover's algorithm. But the square root speed up of brute-force attacks requires changing what is considered "secure". The Advanced Encryption Standard *(AES)* [15] is widely used for providing data confidentiality. With Grover's algorithm in mind, the security level of *AES-128* is lowered to 64-bit ($\sqrt{2^{128}} = 2^{64}$). That means that AES settings with 128 bits or lower key length will no longer be secure, and AES needs to be used with 192 or 256 bits for key sizes [16].

Applying Grover's algorithm on DES, however, brings its 56-bit security to only 185 iterations. 3DES is also not secure enough for the quantum world. Its 112-bit key is lowered to 56-bit security, and that is not considered secure [17].

1.1.3 Impact on hash functions

Hash functions suffers from the same consequences from a quantum computer as symmetric cryptography. Grover's algorithm can be used to find a collision using square root speedup. Brassard et al.[18] showed that by creating a table of size $\sqrt[3]{2^n}$ together with using Grover's algorithm, he can reduce the security level of hash functions three times. That means that for hash functions we need at least 224-bit variants.

1.2 Transport Layer Security

TLS (formerly SSL) is probably the most important security protocol. TLS in version 1.3 is the newest protocol designed to secure transport layer communication. It is considered the most secure and recommended to be used in internet communication [19]. It is located between the application and transport layer of the OSI model. It enables secure endpoint communication, communicating party authentication, data integrity, and confidentiality. TLS can be perceived as a tunnel where only endpoints can access data using cryptography. Also, an adversary cannot modify data without detection after establishment. If this happens, TLS will find it and refuse the message. The server is always authenticated, and the client is authenticated only optionally. As stated in [20], TLS uses two methods for authentication: symmetric pre-shared key (PSK) or asymmetric cryptography (RSA [21], ECDSA [22] or Edwards-curve Digital Signature Algorithm (EdDSA) [23]).

TLS protocol has several subroutines, the most important of which are *Handshake* and *Record* protocols. Each message is split into records, so every TLS packet starts with a record header (even the handshake one). TLS record packet structure can be seen in Figure 1.

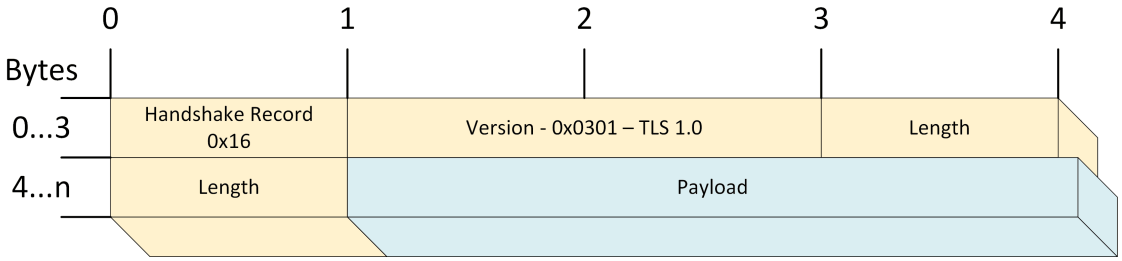

Figure 1: TLS Record Packet

A handshake protocol handles authentication of both communicating parties, negotiation of cryptographic parameters, and establishes a shared key.

Several communication messages are defined in a handshake, each with its specific name and structure. We explain both 1.2 and 1.3 handshakes in detail, as we need to understand both to understand our solution for a post-quantum handshake. We focus only on relevant fields for key establishment. Also, we explain TLS record protocol and authenticated encryption concept.

1.2.1 Authenticated Encryption with Associated Data

Many of the cryptographic protocols used in recent years (including TLS) use authenticated encryption. Authenticated Encryption with Associated Data (AEAD) [24] is cryptographic scheme where a symmetric cipher is used together with a cryptographic hash function to provide authenticity and confidentiality of transmitted data. There are two different AEAD for TLS 1.3 connections; AES-GCM and ChaCha20-Poly1305.

As described in [25] AEADs has two basic operations, "seal" and "open". The "seal" operation require the following:

- The plaintext (message).
- A secret key.
- An initialization vector (IV). It must be unique between invocations of the "seal" operation with the same key; otherwise, the secrecy of the cipher is completely compromised.
- Optionally, we can add other non-private data. The data will be authenticated but not encrypted. (AD in AEAD).

The TLS 1.3 removed all insecure options and ciphers and uses only ChaCha20 or AES in GCM mode to produce a ciphertext of equal length. Based on the key and IV, it hashes the ciphertext, additional data, and lengths. For hashing, Poly1305 or GHASH is used. Hash is encrypted to create the final MAC, and added to ciphertext. The "open" operation is the same but reversed.

1.2.2 Sub-protocols

1.2.2.1 Record TLS record protocol helps secure the application data using the keys and parameters established in the handshake. As described in [26], TLS Record is responsible not only for securing application data but also for verifying its integrity and origin. It manages the following:

- Dividing outgoing messages into manageable blocks and reassembling received messages.
- In obsolete versions, TLS 1.2 and less, TLS record supported compression of these outgoing blocks and decompression of received blocks. This feature was optional for TLS 1.2, and in TLS 1.3 is not present.
- In TLS 1.2, there were separate methods for:
  -applying MAC to messages before sending, and verifying received messages
  -encrypting transmitted blocks and decrypting received blocks
  In TLS 1.3, all ciphers are considered as AEAD. AEAD functions turn plaintext into authenticated ciphertext and back again. Each encrypted record consists of a plaintext header followed by an encrypted body with optional padding.

When the protocol finishes, the outgoing encrypted message is sent to the Transmission Control Protocol (TCP) layer for transport.

1.2.2.2 TLS 1.2 handshake In the first step, the client sends a list of supported ciphers (*cipher suites*) and other details in the message called *ClientHello*. The packet starts with record header information - type, version, and length (Figure 1). The Record message payload consists of the client protocol version, 32 bytes of random client data, an optional session ID to resume the previous session, a list of *cipher suites*, and a list of compression methods. It finishes with an extension's length and a list of extensions. ClientHello message details are shown in Figure 2.

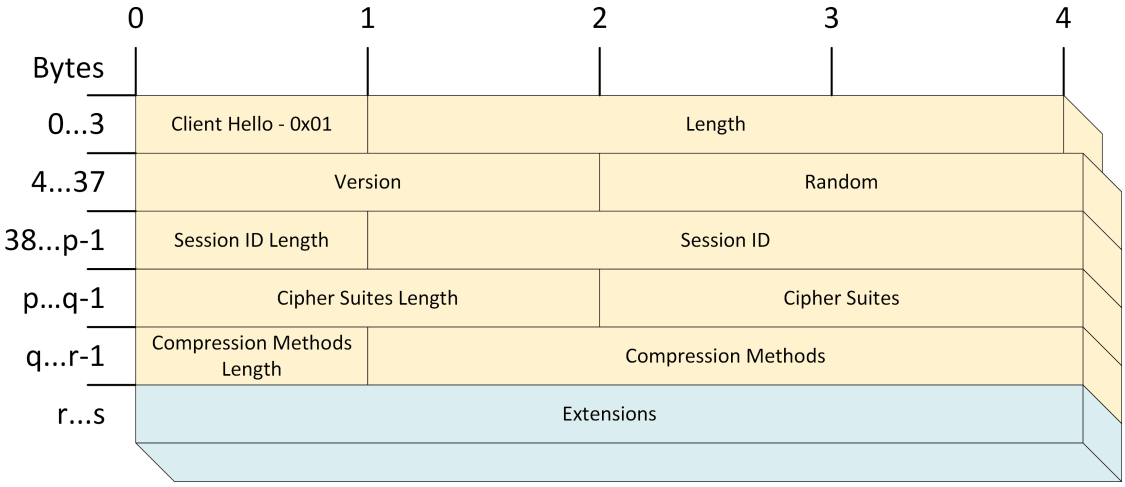

Figure 2: TLS ClientHello Packet

The *protocol type* for a record is 16. *Version* number 03 01 (TLS 1.0) is used in the record header of all TLS messages to preserve compatibility with older devices. The *Message type* for the ClientHello is 01, TLS protocol version is 03 03, (representing TLS 1.2). TLS 1.0 is represented by 3.1, TLS 1.1 is 3.2, etc.. The *Session ID* is used to resume the session with the client from the previous connection. Thirty-two bytes of *random client data* are used later in the handshake for a key derivation. A list of *cipher suites* is ordered by client preference. In TLS 1.2, there are several options for the public key system to be used in the communication. One option is to use RSA [21] system. In TLS RSA, the client chooses a shared secret, then uses RSA to encrypt it with a server public key and send it via a network. The other option is to use Diffie-Hellman [27] key agreement. In this scheme, each party publishes a binary sequence, then combines the received one with their private sequence, and both parties have the same secret (Pre-Master Secret). *Extensions Length* is the length of all other information that the client wants to send (Server Name, Supported Groups, Signature Algorithms, etc.).

The server processes the message, selects the correct version of the TLS protocol, com-

pression and encryption methods, and sends a ServerHello response as shown in Figure 3. It again consists of record header information (figure 1), handshake header - message type and length of the ServerHello, server protocol version, random server data, an optional session ID to resume a session, and a chosen session *cipher suite* a compression method and extensions length and a list of extensions.

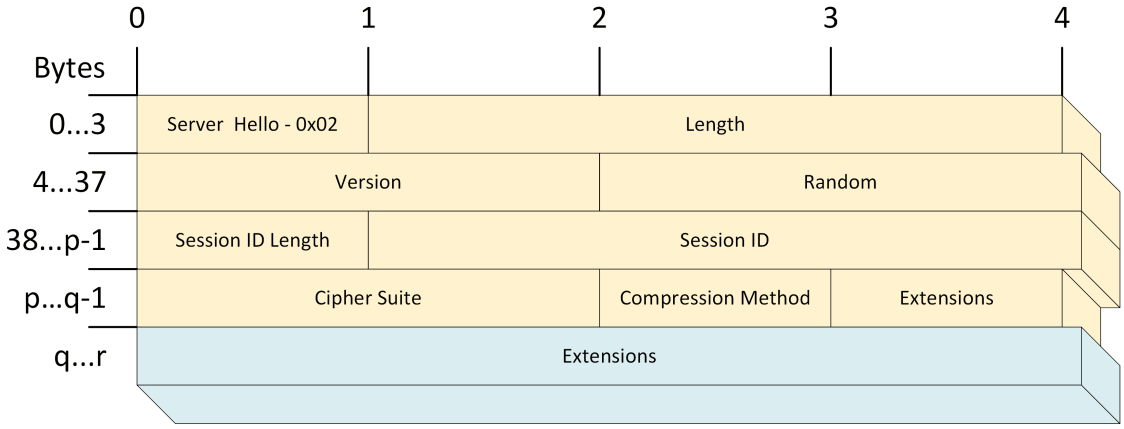

Figure 3: TLS ServerHello Packet

Message type for a ServerHello is 02, TLS protocol version is 03 03, (representing TLS 1.2). This number can also be found in *Server Version*. *The session ID* is again used to resume the session with the client from the previous connection. Thirty-two bytes of *Server Random Data* are used later in the handshake for key derivation. The server selects the *cipher suite* and sends it to the client. After sharing authentication information, the server announces the end of the Hello process with a *Server Hello Done*.

In the next phase, the client sends a ClientKeyExchange message that is either empty or can contain a Pre-Master Secret or a public key, depending on the selected cipher. The client and the server create a Master Secret from randomly chosen numbers and the Pre-Master Secret. All other keys are computed from Master Secret.

The final phase contains information that further communication will be encrypted. The client sends the ChangeCipherSpec and Finished message, which is encrypted and includes a MAC of a Finished message and a hash of all handshake messages. The server tries to decrypt this message and verify the client's MAC and hash. If the verification is not successful, the connection will be terminated. If the verification and decryption are successful, the server sends the ChangeCipherSpec and Finished messages. The client will decrypt and verify the message.

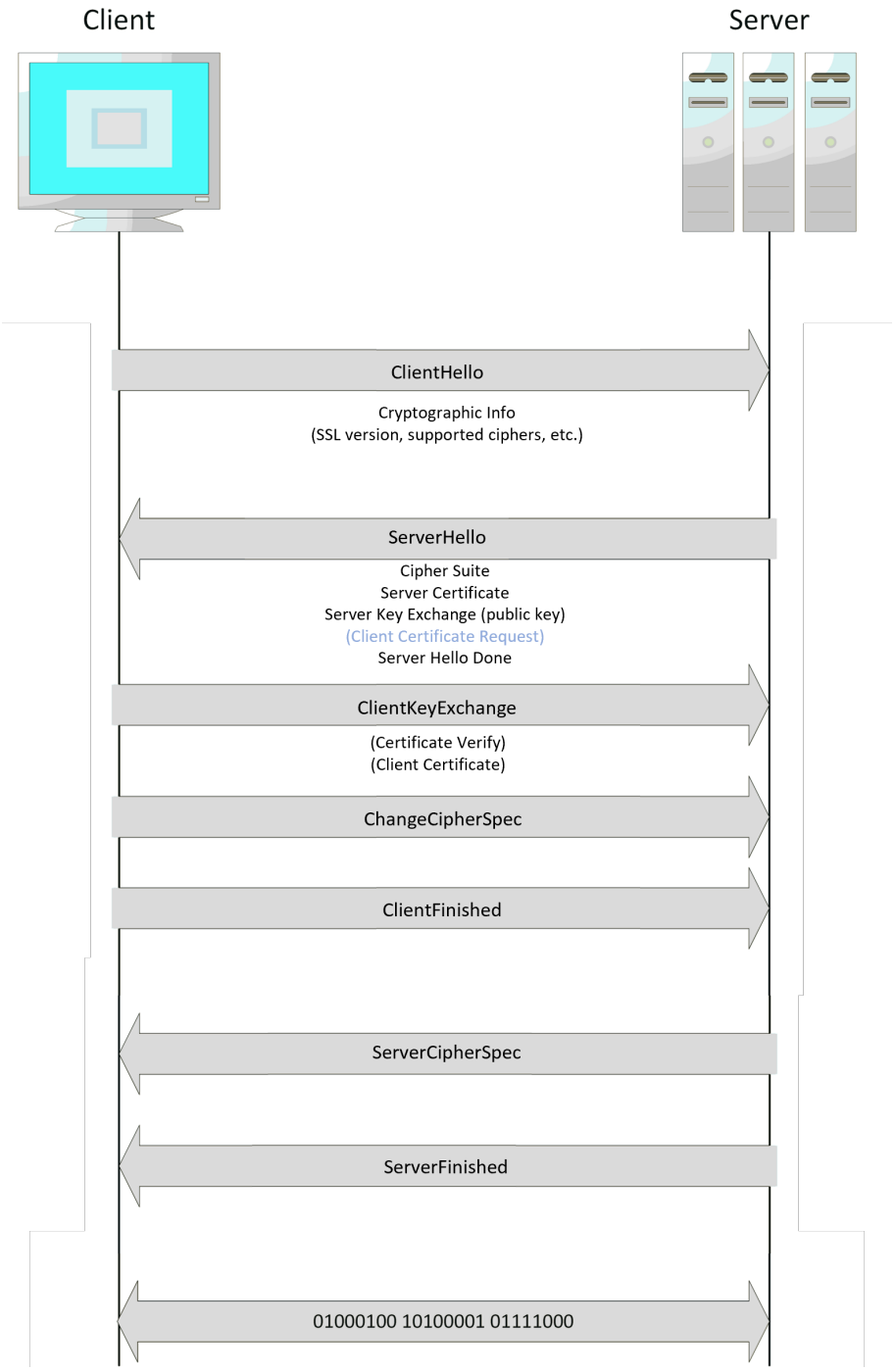

Figure 4: TLS Handshake Diagram

1.2.2.3 TLS 1.3 handshake Even if an attacker cannot gain access to encrypted communication, he can record TLS handshake, as seen in Figure 4, and later gain access to private keys, exploiting currently not known vulnerability. If the protocol is using the same (long-term) private key for all sessions, an attacker is able to access confidential data. Perfect Forward Secrecy is the ability of the cryptosystem to prevent that. TLS 1.2 and previous protocols use RSA [21] without Perfect Forward Secrecy. In TLS 1.3, RSA is removed, and ephemeral Diffie-Hellman [27] is the only key exchange mechanism present. That means that long term private key is not used for key exchange, but only for authentication. Diffie-Hellman secrets are used only once, for one key exchange, and then regenerated for the next session.

Because not all DH parameters are secure, TLS 1.3 restricted the parameters only to secure ones. These changes resulted in a faster and simpler protocol, which is easier to understand. TLS 1.3 abandoned the old concept of *cipher suites*. The table of *cipher suites* became too large because, throughout the years, each new entry in the table (e.g. new cipher) had to be combined with all the others (e.g. hash functions). Now TLS 1.3 uses three separate negotiations:

- AEAD + Hash (TLS 1.3 uses HMAC-based key derivation[28])
- Key Exchange
- Signature Algorithm

Figure 5 explains how new negotiations work in contrast to old *cipher suites* tables. Old negotiation shows a set of combinations of cryptographic primitives in form of *cipher suites* with underscore _ between each group (ciphers, key exchanges, and signature algorithms). New negotiation has three sets for each group, and because they are negotiated separately, we need much smaller sets.

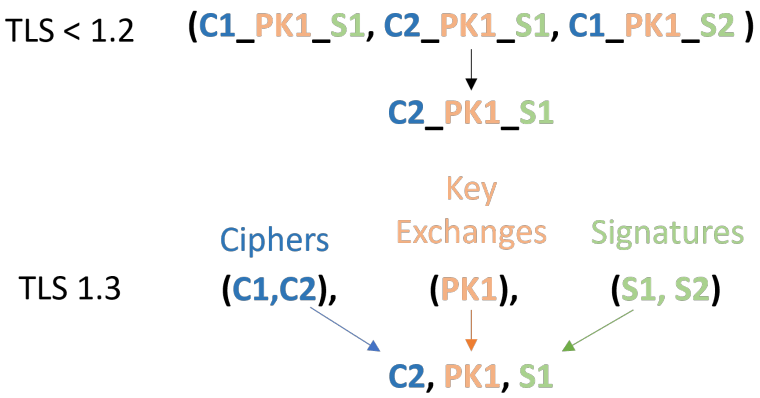

Figure 5: Cryptographic negotiations

All this leads us to an exciting result. As explained in section 1.2.2.2, two round trips are needed to complete the TLS 1.2 handshake. TLS 1.3 does not require two round trips, but only one. This cuts the encryption latency in half, resulting in a performance boost. This can be seen in Figure 6.

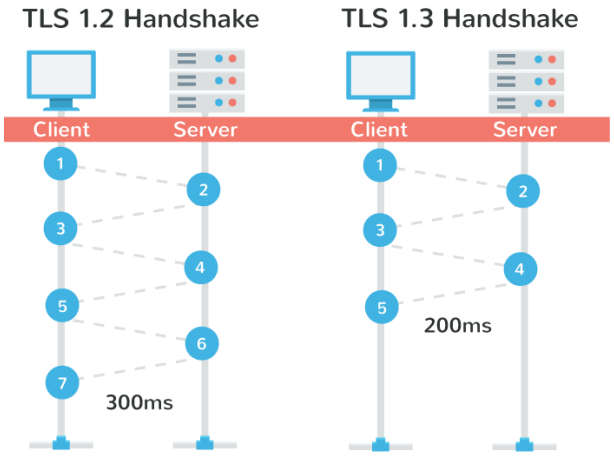

Figure 6: TLS 1.3 handshake time reduction [29]

Now we explain the reason for the latency cuts in more detail. Because of a limited set of choices for negotiation, the client sent a DH key in the first message. The server can learn the shared secret and send encrypted data one round-trip earlier. It is called 1-RTT mode. If the server does not support one of the keys from the client, the server will send the HelloRetryRequest to let the client know which groups it supports. This situation should be rare because the list of choices is short.

ClientHello The new ClientHello has the same structure as shown in Figure 2, with

minor differences. TLS protocol version is still 03 03 (representing TLS 1.2) because still used middle-boxes have been programmed not to allow protocol versions that they do not recognize. This field is hard-coded, and the actual negotiation is in the *Supported Versions* extension. The session ID is no longer needed because TLS 1.3 uses a different mechanism for session resumes. The *cipher suites* field is used for symmetric cryptography only and has only two ciphers, AES (128, 256) [15] and ChaCha20 [30] in AEAD form. *Compression Methods* are no longer allowed in TLS 1.3, so ClientHello sends only null value here. For agreement, public key algorithm negotiation *Supported Groups* extension is used. The client sends an ordered list of supported elliptic curve (EC) cryptography. *Key Share* extension is used to send the client's public key. Supported Versions extension indicates protocol version 03 04 (TLS 1.3).

ServerHello The server responds with its *ServerHello*, with the same structure as shown in Figure 3. TLS protocol version is again 03 03 (TLS 1.2), *Supported Versions* extension then shows 03 04, assigned for TLS 1.3. *Session ID* and *Compression Method* are not used. The server chooses a specific cipher and a hash and includes it in the *cipher suite* field. The server calculates the keypair for key exchange and attaches its public key in the *Key Share* extension. Because the *Diffie – Hellman* key exchange is used, the server already has the necessary secrets (client and server public keys) and calculates all secrets that are needed. After the delivery of *ServerHello*, the client has everything that is required for computation and also starts the calculation. On both sites, five keys and two initialization vectors are calculated.

- *handshake secret* - used later for calculating application keys (*Pre-Master Secret*).
- *client and server handshake traffic secret* - used in the end phase of a handshake (message *Finished*) for verification.
- *client and server handshake key* - symmetric key used for rest of the handshake.
- *client and server handshake IV* - Initialisation Vector (IV) for the rest of the handshake.

The key exchange mechanism is now finished. Even though this is not the main focus of our research, we can briefly mention other mechanisms of TLS 1.3 handshake. Everything is now encrypted using the selected cipher.

The server sends one or more certificates containing the identity, public key, and third-party signature. The server will also send proof that its certificate was linked with

the public key sent for the handshake. Both the server and the client exchange handshake termination messages (*Finished*). Both parties then compute the keys used for further communication based on the *handshake secret*.

- *client and server application key* - symmetric keys for application data.
- *client and server application IV* - Initialisation Vector for application data.

TLS 1.3 "remembers". When a new connection is established with a server that the client has not seen before, the handshake will be done as mentioned above. The so-called 0-RTT mode lets the client send encrypted data in the first message to the server, with no additional latency compared to unencrypted HTTP.

Of course, this comes with a trade-off. If an adversary captures 0-RTT data from the client to the server, he can replay it, and the server may accept it. If the client wants to avoid it, he should use a request that doesn't change the server state. For example, a browser can protect HTTPS servers against replay attacks by only sending GET requests in 0-RTT. If state-changing requests are sent in a 0-RTT packet, TLS 1.3 includes the elapsed time value in the session ticket. If the time is not valid, the server rejects it. We will not discuss this further in our work.

Following the trend of simplicity, TLS 1.3 helps administrators and developers not to misconfigure the protocol, and obsolete and insecure features of TLS 1.2 were removed. More specifically, SHA-1, RC4, DES, 3DES, AES-CBC, MD5, Arbitrary Diffie-Hellman groups, and others are vulnerable to specific attacks.

1.2.3 Comparison of SSL and TLS

TLS is relatively new, and some people may only be familiar with its predecessor, the ssl protocol. For those who are familiar with SSL, but not with TLS protocol, we mention the main differences between these two protocols as mentioned in [31].

Alert Protocol was changed, 11 more message types with error descriptions have been added, and one message, *NoCertificate*, has been removed. If the client does not have a certificate to use, it can return an empty certificate message. There are 23 alert messages in TLS 1.3.

The new protocol implements a standardized Hash-based message authentication code (HMAC) [32] that was already used in many other implementations. HMAC can operate with any hash function, not just MD5 or SHA 1, as is explicitly stated by the SSL protocol. SSL specifically supports RSA, DH, and Fortezza/DMS *cipher suites*. Fortezza is an information security system for PC Card-based security token, that was developed by U.S. Government and was used for the Defense Message System. TLS has stopped supporting Fortezza/DMS.

Before, SSL keys were generated from a combination of hash output, selected *cipher suite*, and parameter information (RSA, DH, or Fortezza/DMS output). TLS uses the HMAC standard and its Pseudorandom Function (PRF) output to generate TLS keys. It starts with the Pre-Master Secret to create the Master Secret. From Master Secret, all other secret keys are generated.

In SSL, the CertificateVerify message requires a complex procedure. With TLS, the verified information is wholly contained in the handshake messages previously exchanged during the session. SSL creates a Finished message in the same way as it generates key material. Also, in TLS, the PRF output of the HMAC algorithm is used with the Master Secret and either a *client finished* or a *server finished* designation to create the Finished message.

1.3 Algorithms used in IoT security

There are many protocols used to secure IoT communication. But as we go deeper and look at specific cryptographic algorithms they use, we can see that only a limited number of ciphers are used in these protocols.

Authors of [33] mentioned the most important protocols used in IoT. For each layer, we show the protocol and used relevant cryptographic algorithms:

- **Physical layer** - As we see in [34], most of the protocols of the physical layer (DASH7, LoRa) use **AES128** [15] for providing confidentiality of the data.
- **Data Link layer** - the security is provided by IEEE 802.15.4 [35], which specifies several cryptographic options, but all are based on **AES** [15].
- **Network Layer** - IPsec protocol is a requirement for IPv6 - allows **Diffie-Hellman**, **ECDH** [36], **RSA** [21], **AES** [15]. Another protocol of the network layer, 6LoWPAN protocol, only relies on the security of the transport layer [37].
- **Transport Layer** - in the transport layer, we can mainly use two types of protocols, TCP or UDP.
  - For TCP, security is provided by TLS, which in version 1.3 allows **AES** [15] and ephemeral **Diffie-Hellman** [27].
  - UDP is secured by DTLS or QUIC. These protocols allow using ephemeral **Diffie-Hellman** [27] for key exchange, and **AES** [15] for data confidentiality.
- **Application Layer** - CoAP protocol proposes to use DTLS to provide security, and AMQP protocol uses TLS. Therefore the same algorithms can be used as in the transport layer.

These algorithms are present in all protocols mentioned above: AES [15], RSA [21] or Diffie-Hellman [27] (or ECDH [36]). We can see that the situation in IoT is similar to the situation in the internet protocol suite. We need to use a larger keyspace for AES [15], and for public-key cryptography, we need to consider new algorithms.

1.4 Post-quantum movement

As we discussed in section 1.1, quantum technologies are threatening public-key cryptography, which calls for a new standard. Several public-key algorithms rely on other problems (some even NP-complete) that could meet the requirements for the new standard.

1.4.1 Post-quantum algorithms principles

Post-quantum algorithms are based on different problems than traditional public key systems. They are based on an assumption that in practice it is impossible to solve problems they are based on, without the key (even using a quantum computer). They don't rely on the integer factorization problem nor the Diffie-Hellman problem, but instead, they rely on the following principles:

1.4.1.1 Lattice-based Lattice is a discrete subgroup of a finite-dimensional Euclidean vector space. Lattice-based cryptography uses lattices for hiding information. The most commonly used problems are the shortest vector problem, closest vector problem, learning with errors, and variations of these problems.

1.4.1.2 Code-based It uses the hardness of the decoding problem. First, the message is encoded as a code-word of a special linear code, then errors are arbitrarily added to this message. The error-correcting codes are used to efficiently decode the message. The error-correcting code (a private key of the system) is masked, and a seemingly random code (public key) is used for encryption.

1.4.1.3 Multi-variate Multi-variate cryptography relies on the problem of solving a system of multivariate quadratic polynomial equations and the isomorphism of polynomials. It is used mostly for digital signatures.

1.4.1.4 Hash-based The usage of hash-based cryptography is currently limited to digital signatures. A one-time signature scheme is used to sign the message, and the Merkle tree structure is used to combine many one-time signature keys into a single data structure. There is a drawback, the number of signatures that can be signed using the corresponding set of private keys is limited.

1.4.2 NIST

In response to the successes in quantum computing, NIST at PQCrypto 2016 [38] started the standardization process of the post-quantum public-key algorithm and announced the call for proposals. This new standard should contain the best candidates for key exchange mechanisms as well as signatures. The first round collected overall 64 candidates [7], which are summarized in Table 1.

	Signatures	Encryption	Total
Lattice-based	5	21	26
Code-based	2	17	19
Multivariate	7	2	9
Hash-based	3		3
Other	2	5	7
Total	19	45	64

Table 1: NIST post-quantum first-round candidates distribution

Several candidates had to be removed because successful attacks against them were found. In January 2019 second-round candidates were announced. The distribution of all 26 candidates can be found in Table 2.

	Signatures	Encryption	Total
Lattice-based	3	9	12
Code-based		7	7
Multivariate	4		4
Hash-based	1		1
Other	1	1	2
Total	9	17	26

Table 2: NIST post-quantum second-round candidates distribution

In 2020, seven finalists were announced, together with 8 alternate candidates. The first group of finalists (Table 3) will be considered for a new standard. However, candidates

from the second group (Table 4), may still be part of the new standard. Also, there were some minor changes required for the finalists, due to the attacks found in [39] and [40].

	Signatures	Encryption	Total
Lattice-based	2	3	5
Code-based		1	1
Multivariate	1		1
Total	3	4	7

Table 3: NIST post-quantum third-round candidates distribution

NIST also discussed intellectual property concerns, as some of the finalist and alternate candidates are developed by private companies and the legal claims should be cleared [41]. These aspects will also be taken into consideration, however, NIST has signed a waiver of rights from submitting groups.

	Signatures	Encryption	Total
Lattice-based		2	2
Code-based		2	2
Multivariate	1		1
Hash-based	1		1
Other	1	1	2
Total	3	5	8

Table 4: NIST post-quantum alternate third-round candidates distribution

There were several requirements for NIST candidates [41]. Along with the paper submission, each candidate should include reference implementation and optimized implementation. Optimized implementation targeted Intel x64 processor. The implementations also needed to be royalty-free. Thanks to these requirements, we were also able to integrate the candidates into our work.

We explain two of the most crucial requirement concepts for our work. As our work focuses on post-quantum secure key exchange, we will consider only encryption schemes. Signatures would also need attention, but that is beyond the scope of this work.

1.4.3 Key Encapsulation Mechanism

Key Encapsulation Mechanism (KEM) is a valuable tool to secure symmetric keys for transmission with a public key algorithm. Instead of using traditional cryptographic operations, key generation, encryption, and decryption, we have three slightly different operations: keypair, encapsulation, and decapsulation. We can define them in the following way:

- **(*pk, sk*) <- Keypair()** - This function does not take any argument as an input, other than implicit ones (more specifically, security parameter or sizes and random bits). It uses an underlying public-key-algorithm key generation to generate a public key *pk* and a corresponding secret key *sk*.
- **(*c, K*) <- Encaps(*pk*)** - Besides implicit security parameters and random bits, input *pk* represents the public key generated previously in *keypair* operation. *Encaps* generates a shared secret *K*. Public key *pk* and underlying public-key-encryption algorithm produce ciphertext *c* from shared secret *K* and output both *c* and *K*.
- **(*K*)<-Decaps(*c, sk*)** - Based on the input secret key *sk*, the algorithm decrypts the ciphertext input *c* to get the shared secret output *K*.

An trivial demonstration of creating a key encapsulation mechanism from standard public-key encryption is provided in Figure 7. We use notation $outputs < -function(inputs)$ to indicate inputs and outputs to function.

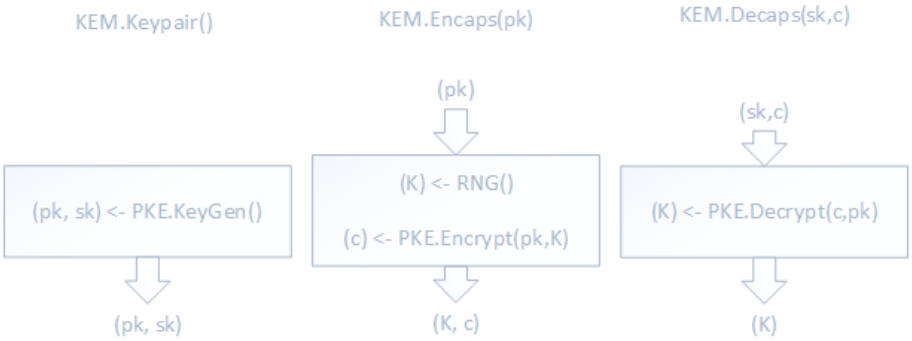

Figure 7: KEM conversion using Public Key Encryption

The usage of the key encapsulation mechanism is demonstrated in Figure 8. We use := to indicate key derivation. As we can see, both communicating sides (Alice and Bob) share the same key after one round trip. The first message contains a public key and thus

does not contain any confidential information. The second message contains ciphertext, an encrypted key using a public key cipher.

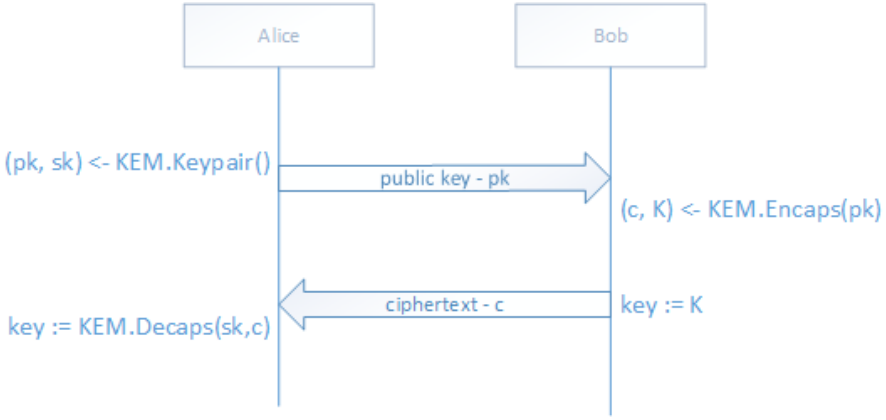

Figure 8: Key exchange using KEM

The requirements in[41] state that the submitting groups of the encryption schemes can choose between providing the implementation in public-key encryption form or KEM form. NIST also may apply standard conversion techniques to convert between them. As a result, all finalists of the third round have the implementation in KEM form.

1.4.4 Security Strength Categories

NIST call for proposals [41] stated several requirements. One such requirement was that the parameters of the individual cryptosystems should be chosen to allow a comparison of the candidates. NIST created five security categories that should be an etalon for the parameters of each candidate. Any attack that breaks the relevant security definition must require computational resources comparable to or greater than those required for the representative of the category:

- First category was defined by a key search on a block cipher with a 128-bit key (e.g. AES128 [15]).
- Second category was defined by a collision search on a 256-bit hash function (e.g. SHA256/ SHA3-256 [42]).
- Third category was again defined with a key search on a block cipher, but with a 192-bit key (e.g. AES192 [15]).
- Fourth category was defined by a collision search on a 384-bit hash function (e.g. SHA384/ SHA3-384 [42])
- Fifth category was once again defined with a key search on a block cipher, with a 256-bit key (e.g. AES 256 [15])

These categories can be used to create performance comparisons between submitted candidates so that the comparison would be made with parameters of comparable security levels. This also helps to understand the security/performance trade-off in each candidate. As the chain is only as strong as the weakest link, this may also help with the decisions of how strong symmetric cryptography components of the system should be. If we are aiming for post-quantum security in AES [15], we use a 256-bit key, so we need to use category five. This is also applicable for our design.

1.5 TLS Implementations

TLS is a cryptographic protocol. It is a standard that defines the principles and processes of secure communication between two parties, with precisely defined messages. These definitions have various implementations that put the processes defined in the standard into practice. For its undeniable security benefits, open-source implementations are prevalent. In this section, we compare some of the most notable implementations used on the servers and in the clients' browsers. Later on, we used some of them for our experiments.

1.5.1 OpenSSL

The OpenSSL project [43] started in 1998. It is a general-purpose cryptographic library written in C language. The OpenSSL toolkit is licensed under an Apache-style license, allowing developers to use it for commercial and non-commercial purposes subject to simple license conditions. OpenSSL is widely used, including SSL and TLS implementations up to TLS 1.3. This library is part of many software projects because it serves as a reliable collection of implementations of cryptographic algorithms also for purposes other than securing the transport layer. Because of the reliability of the open-source design, OpenSSL has become the basis for many other TLS implementations. OpenSSL is developed for Unix-like systems and Windows.

1.5.2 LibreSSL and BoringSSL

Both of the libraries are forks of OpenSSL. LibreSSL [44] by OpenBSD was forked as a response to the Heartbleed security vulnerability. Google forked BoringSSL to meet their needs. BoringSSL [45] is used in Chrome and Android. It is also a basis of Google's Tink cryptographic API.

1.5.3 NSS

Network Security Services (NSS) [46] is a set of libraries in the C language. NSS also includes other security features, providing complete TLS implementation with required cryptographic tools. It is developed by Mozilla, and following the company's policy, it is open source. NSS is not OpenSSL based. It uses libraries developed by Netscape when they invented the SSL protocol. NSS is used in Mozilla Firefox.

1.5.4 s2n

The s2n [47] is another open-source implementation that is a much clearer version of the OpenSSL. TLS part is less than 10% long in terms of lines of the code than original

TLS in OpenSSL, thus it is much easier to review and comprehend the mechanisms in the library. The developer can choose between OpenSSL, BoringSSL, or LibreSSL libraries for cryptographic primitives. It is developed by Amazon Web Services in the C99 language. s2n is used in Amazon S3 services.

1.5.5 Mbed TLS and wolfSSL

Both are open-source libraries developed for limited devices. They are small, readable, portable TLS implementation for embedded devices under 64 KB of RAM. Mbed TLS [48] is developed by the community under ARM, and wolfSSL[49] is developed by the wolfSSL company.

1.6 Operational Security

When designing a security system, we need to consider more than analytical proofs provided in submission papers of KEMs, or security provided by the strictly defined structure of messages in protocols. Even if the design and components are theoretically analyzed, it may not be sufficient, as we have seen in the past. Heartbleed OpenSSL vulnerability [50] or OpenSSL's timing attacks on the underpinning ciphers [51] are a great example that even projects as big as OpenSSL can be vulnerable to unexpected behavior or bugs.

Operational security can be perceived at various levels. The security threats can be then structured as defined in [52].

- **High level** - The protocol implementation is deviating from the design due to logical bugs.
- **Medium level** - The protocol implementation seems to follow its design, but still, the attacks reach their target (attacking execution runtime instead of the protocol's implementation).
- **Low level** - Threats are originating from programming bugs (exploiting arithmetic overflows, invalid pointer references, etc.).
- **Hardware level** - If the manufacturer of the hardware is not reliable or if side-channel attacks were found.

We can avoid high-level threats by strictly following the design (verification). Also, we need to implement secure coding practices to avoid low-level threats. We will discuss medium-level and hardware-level threats countermeasures in the following sections.

1.7 Side-channel attacks

In order to plaintext without using a key, cryptanalysis often examines the hardware or software properties of a particular implementation of a cryptographic algorithm. Side-channel attacks are **hardware-level** attacks. They are practical attacks that take advantage physical characteristics of the device during its operation. Those physical characteristics include power consumption, electromagnetic radiation, execution time, light emissions, and acoustic or heat emanations. Attackers can exploit hardware-level vulnerabilities if different inputs to the cryptographic system produce different outputs in measuring those physical characteristics.

Mainly, we can recognize two types of attacks [53]:

- Active attacks - they require the adversary to act directly on the device. He is able to change the messages, the behavior of the algorithm or device to modify or get more information.
- Passive attacks - they come from an adversary who is only "listening" to the communication. In passive attacks, the adversary is not able to send or alter messages, nor directly alter the device.

When using cryptographic devices, we can adopt another point of view for attacks from [54]:

- Invasive attacks - they require an attacker to have direct access to the device. The attacker alters the device physically. He may open the device and expose internal parts.
- Non-invasive attacks - on the other side, non-invasive do not require any device preparations before the attack. With this type of attack, the device is not damaged.

1.8 Trusted environment

To provide the mitigation against **medium-level** threads, a trusted environment can be used. By trusted environment, we mean the generalization of Trusted Execution Environment in the sense of the definition from [55]:

Definition 1.1 (TEE) *Trusted Execution Environment (TEE) is a tamper-resistant processing environment that runs on a separation kernel. It guarantees the authenticity of the executed code, the integrity of the runtime states (e.g. CPU registers, memory, and sensitive I/O), and the confidentiality of its code, data and runtime states stored on persistent memory. In addition, it shall be able to provide a remote attestation that proves its trustworthiness for third parties. The content of TEE is not static; it can be securely updated. The TEE resists all software attacks and the physical attacks performed on the main memory of the system. Attacks performed by exploiting backdoor security flaws are not possible.*

This definition requires a separation kernel on the system. In [56], National Security Agency (NSA) defines a separation kernel as follows:

Definition 1.2 (separation kernel) *A separation kernel is a hardware and/or firmware and/or software mechanisms whose primary function is to establish, isolate and separate multiple partitions and control information flow between the subjects and exported resources allocated to those partitions.*

In that light, we can view TEE as an abstraction of hardware security modules.

1.8.1 Hardware security modules

The idea of a Hardware security module (HSM) comes from the concept of a Trusted Platform Module (TPM) [57]. TPM is a microcontroller able to store keys, passwords, or certificates securely. It supports several cryptographic algorithms, including RSA, SHA-1, HMAC, and also ECC, (TPM 2.0). It is possible to generate and store the keys but not to compute any other cryptographic operations. That is where the HSM comes in. HSM is a special device that generates and stores the keys and performs encryption and decryption, signing, verification, and other cryptographic functions. Several manufacturers produce different hardware devices. The modules can be developed in Java, C, and other languages. Because it is not as strictly defined as TPM, HSMs are certified with standards such as Common Criteria or FIPS140.

When we use HSM, we usually trust it completely. In cryptography, we use the term Root of Trust (RoT) for a source that can always be trusted within the whole cryptographic system's scope.

Typical uses of HSM include payment cards, cryptocurrency wallets, and TLS connections to accelerate cryptographic operations [58]. The benefits of using HSM in TLS not only include speedups but mainly the separation of the critical code segments execution into the trusted environment (RoT).

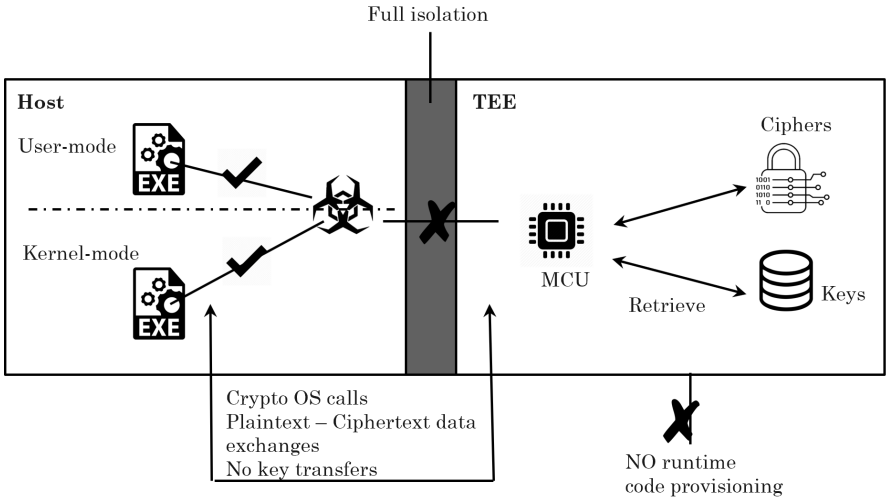

Figure 9: Isolation of critical functions into HSM

In Figure 9, we illustrated the situation where malware infected the host computer. If HSM is used for all cryptographic functions and key storage, the malware cannot see the keys or interfere with critical operations.

1.9 Similar efforts and related projects

The topic of designing a post-quantum-secure key exchange is very complex. The focus of this research is large, even with the help of the structure provided by the TLS protocol or other mechanisms described above. However, there is a lot of research in this field, and we mention several important research works and articles focusing on problems that are close to our own research. All this research is very recent and in continuous development, which sometimes made our work easier and sometimes more difficult. We present state of the art research in post-quantum algorithms library, post-quantum TLS prototyping, and post-quantum algorithms suitable for limited devices. In our work, we make use of some of the presented discoveries and implementations - first to provide an overview of the state-of-the-art and later to rely on them when creating our own implementation.

1.9.1 PQClean

PQClean is a collection of *clean* implementation of post-quantum schemes participating in the NIST post-quantum competition. The repository includes standalone implementation of all finalists of the third round and alternate candidates. The previous version also included candidates from the second round. Each scheme has its own original license, specified in separate files. The rest of the code, as well as tests, are open-source. Clean implementations must be buildable under Linux, macOS, and Windows must follow KEM API, are implemented in standard C99, and meet several other requirements.

1.9.2 Open Quantum Safe

Open Quantum Safe (OQS) Project [59] started in 2019. It is an open-source project that focuses on post-quantum cryptography implementations and their applications. It is strongly connected to the NIST PQC standardization project, so they work with new standard candidates.

liboqs is an open-source library, the first child of the OQS project. They collected post-quantum candidates in the form of a C library, following these principles:

- **Open source**. The library itself was released under the MIT License. Even so, it incorporates some external components which use a different license. It is fully open-source.
- **Multi-platform**. The library supports x86 and ARM architectures, most of the compilers, and all main operating systems (Linux, macOS, and Windows)

- **Common API**. The library incorporated the NIST requirement to provide a common API for all submissions.
- **Testing and benchmarking**. The library includes the tests and performance comparisons.
- **Application integrations**. the library cooperates with its applications, including TLS, SSH, X.509, etc.
- **Language wrappers**. The library provides wrappers for other programming languages.

The library contains all third-round candidates from the NIST standardization effort, finalists, and alternate algorithms. Some implementations are reused from the PQClean project.

OQS-OpenSSL is another exciting product of the OQS project. It implements post-quantum and hybrid key exchange and post-quantum public-key authentication in TLS 1.3. More precisely, it is the integration of the liboqs into OpenSSL.

Because the current architecture of OpenSSL does not allow to change key exchange mechanism easily, the architecture is changing in OpenSSL 3.0. OQS also brings a provider for a new concept of OpenSSL to enable post-quantum key exchange. This is still a work in progress.

OpenSSL has the concept of engines to allow new ciphers. Similarly to our previous research, [60], where we created a post-quantum OpenSSL engine in 2017, implementing McEliece code-based cipher, OQS project created OQS-Engine that integrates KEMs from libqos.

1.9.3 OpenSSLNTRU

Similar research to OQS-OpenSSL is the OpenSSLNTRU [61]. The authors proposed changes to the openssl library to allow post-quantum key exchange and created an OpenSSL engine that integrated NTRU (engNTRU). They managed to get a faster handshake than any other software included in OpenSSL.

1.9.4 Post-quantum algorithms prototyping in TLS

Similarly to this work and the work of OQS project, Douglas Stebila (from OQS project) et al. [62] introduced post-quantum (hybrid) key exchange in TLS. Along with the experiments in OpenSSL (TLS 1.3 and TLS 1.2), they used s2n implementation from AWS.

They integrated two NIST candidates, BIKE-L1 and SIKEp503, from the first round into s2n in TLS version 1.2.

They experimented not only with TLS but also with SSH protocol. OpenSSL and OpenSSH use liboqs library from OQS project to implement post-quantum KEMs.

1.9.5 Post-Quantum TLS 1.3 on Embedded Systems

Very recently, another similar research [63] was published. The authors focused on post-quantum TLS 1.3 for embedded systems. They integrated and benchmarked Kyber, Saber, Dilithium, and Falcon into the wolfSSL TLS 1.3 with a focus on ARM Cortex-M4 NUCLEO-F439ZI. They found that in some cases, the post-quantum handshakes were faster compared to ECDHE. They also included memory consumption of integrated candidates.

1.9.6 Post-Quantum TLS Without Handshake Signatures

Recently, Schwabe et al. in [64] did similar research to this work, but with a focus on authentication. They had designed and implemented a post-quantum TLS alternative to TLS 1.3 in Rustls, a TLS library written in Rust. They use KEMs instead of signatures for server authentication. As post-quantum public keys and signatures tend to be large, they decided to use only pre-distributed KEM keypair. They managed to reduce server CPU cycles by almost 90% compared to TLS 1.3. In their model, the server generated a KEM keypair once, serving for authentication. The keypair for a handshake is generated by the client.

1.9.7 pqm4

This is a very useful project for all who want to integrate post-quantum cryptography onto the ARM Cortex-M4 family of microcontrollers. The pqm4 is a library that provides the implementation of post-quantum NIST candidates for Cortex-M4 and specific optimizations, if available. Along with Cortex-M4 specific implementation, there is also PQClean one, reference implementation submitted to NIST, and optimized implementation in plain C for x86. In the publication [65], performance comparisons of KEMs and signature schemes tested on the STM32 Cortex-M4 chip are also provided. This GitHub repository contains python tests and benchmarks. Similarly to PQClean, each candidate has its license, and the rest of the code is open source.

2 Designing post-quantum-handshake key exchange

We propose a new solution to enhance the security of TLS communication, to match the state-of-the-art research. Taking into consideration the concepts we explained in the previous chapter; we can divide the requirements of a new design into several categories:

- Post-Quantum - TLS-like communication protocol should use only quantum-computer-resistant cryptography. That includes the key sizes for symmetric cryptography and the choice of public-key algorithms.
- Limited devices friendly - New protocol should be designed with limited devices in mind. The choice of algorithms and implementation of the algorithms should consider the limitations of these devices. Also, the protocol's design should have an option for devices with limited resources.
- Operational security - Along with the theoretical security of algorithms and protocols, we have to consider countermeasures for practical attacks. That includes side-channel attack resistance in the choice of implementations and separation of critical computation to the trusted environment.
- Effectiveness - The mechanisms used in our research require more resources, introducing the delay compared to the security mechanism used today. Irregardless of that, the choice of algorithms and their implementations should be aiming for a reasonable delay.

This work introduces the concept of TLS-like protocol, which is quantum-resistant and uses the physical separation of critical data and computations using HSM. Naturally, we introduce the delay due to communication with HSM, usage of post-quantum cryptography algorithms, and limitations resulting from the limits of the resources of HSM. It is the price we pay for enhancing security, resulting in limited use of such design. There are still some applications that may require such trade-offs, such as large finance operations or sensitive data operations. Our design also considers scenarios with limited clients.

In the following sections, we present the phases, concepts, and building blocks of our design.

2.1 Post-quantum algorithms replacing RSA/DH

When building a robust quantum-resistant mechanism for key exchange, an essential aspect is the choice of a public key algorithm. The selection of the algorithms is greatly facilitated by the NIST standardization process, which we described in section 1.4.2. We are interested in the finalists, i.e. the candidates that made it to the third round of the process. In [66], NIST reported the chosen candidates for the third round of the standardization process. The selected finalists are Classic McEliece [67], CRYSTALS-KYBER [68], NTRU [69], and SABER [70]. We decided to use those NIST-selected finalists in our implementations and experiments in our work.

- Classic McEliece falls into the code-based candidates category. Its strong point is that the McEliece cryptosystem has been available since 1979. Since then, it has not been broken; moreover, several modern improvements (for efficiency and CCA security) have been implemented. The downside of this candidate is the size of the public keys. On the other hand, the ciphertext is the smallest of all candidates. That may suit some specific applications, so NIST has chosen it as a finalist.

- CRYSTALS-KYBER relies on Module Learning With Errors problem (lattice-based). It uses Number theoretic transform (NTT) (Fourier transform in a finite field) for polynomial multiplication (more on NTT in section 3.6). The evaluated performance results are a good compromise for all aspects (sizes, speed). According to NIST, this puts CRYSTALS-KYBER in the top one position, but there is still work to do.

- NTRU is another lattice-based candidate. It is a widely analyzed scheme and has been standardized in several cases, e.g. IEEE 1363.1-2008 or ANSI X9.98-2010 (R2017). NIST wants NTRU to be a part of the standard to provide more diversity instead of a single point of failure (it is not based on the same problem as other lattice-based candidates). Because of that and its long history, it proceeded to the third round as a finalist.

- SABER is another Module Learning With Errors candidate; more precisely, its Module Learning With Rounding variation. SABER has good performance results, which helped it become one of the finalists in the third round.

2.2 Decoding problem

Code-based cryptography relies on the hardness of decoding a general linear code. An error-correcting code (able to correct t errors) is selected as a secret (private key) for which an efficient decoding algorithm is known. The generator matrix G of error-correcting code is hidden by two randomly selected invertible matrices S and P to produce a general linear code. When the error is introduced to the code-word in general linear code, it is hard to decode it unless G is known.

2.3 NTRU

NTRU has its own lattice-based problem. The simple NTRU problem relies on the hardness of factoring multiplication of polynomials in a truncated ring $R = Z[x]/X^{N-1}$. Polynomials have integer coefficients and the greatest possible degree $n - 1$. Also, integer q (power of 2) is chosen as a parameter.

Two polynomials f and g are picked from R, with coefficients in $-1, 0, 1$ most of them are zero. Public information h is again polynomial from R, such that

$$h \cdot f = 3g \mod q$$

Private information is f and f_3, where $f \cdot f_3 = 1 \mod 3$

The message m is hidden with random polynomial r from R in following way:

$$c = r \cdot h + m \mod q$$

With unknown f, it is hard to compute

$$a = f \cdot c = f \cdot (r \cdot h + m) = r \cdot 3g + f \cdot m \mod q$$

We can then obtain the message as $m = a \cdot f_3 \mod 3$, with private f and f_3.

This can be rewritten to a version of the lattice Shortest vector problem. More on this can be found in [71].

2.4 LWE, LWR, module-LWE, and module-LWR problem

In this chapter, we present the concepts of several lattice-based candidates problems. The definitions are adapted from [72].

We can formulate the Learning With Errors (LWE) problem as a problem to distinguish between uniformly random samples from $\mathbb{U}(\mathbb{Z}_q^{l \times 1} \times \mathbb{Z}_q)$ and the same number of

samples (a, b) of the form

$$(a, b = a^T s + e) \in \mathbb{Z}_q^{l \times 1} \times \mathbb{Z}_q,$$

where $s \in \mathbb{Z}_q^{l \times 1}$ is a fixed secret vector, $a \leftarrow \mathbb{U}(\mathbb{Z}_q^{l \times 1})$ are freshly uniformly random vectors and $e \leftarrow \beta_u(\mathbb{Z}_q)$ are fresh and small error terms sampled from an appropriate error distribution.

The Learning With Rounding (LWR) problem is similar to LWE, but the errors come from rounding. It can be defined as distinguishing samples of the form:

$$(a, b = \lfloor \frac{p}{q}(a^T s) \rceil) \in \mathbb{Z}_q^{l \times 1} \times \mathbb{Z}_q,$$

from the same number of samples from $\mathbb{U}(\mathbb{Z}_q^{l \times 1} \times \mathbb{Z}_q)$ for a fixed secret vector $s \in \beta_u(\mathbb{Z}_q^{l \times 1})$, and a fresh uniform random vectors a.

In module LWE and LWR, the elements are now polynomials in $\mathbb{R}_q = \mathbb{Z}_q/(x^n + 1)$, not integers.

2.5 Basic post-quantum TLS Handshake concept

When designing a quantum-resistant TLS-like protocol, we start from the most recent TLS protocol versions. The first idea was to begin with TLS 1.3, as it is the newest and most secure version of the transport layer security. However, Diffie-Hellman is not secure against quantum computers, and we can no longer use it for key exchange in the Handshake process, communicating sides cannot "meet in the middle". We also need to modify the messages, or more specifically, the extensions. The simple idea is to use post-quantum KEM instead of Diffie-Hellman. The client sends an ephemeral public key, a server creates a keying material (PMS), encrypts it with the client's public key, and sends the ciphertext back to the client. Both sites can now use kdf to produce all keys for TLS (handshake, record, server, client). We refer to this situation as pqTLS protocol. The representative model in Figure 10 is simplified. It neglects the rest of the messages not important for key exchange, and authentication, as that is outside the scope of our research.

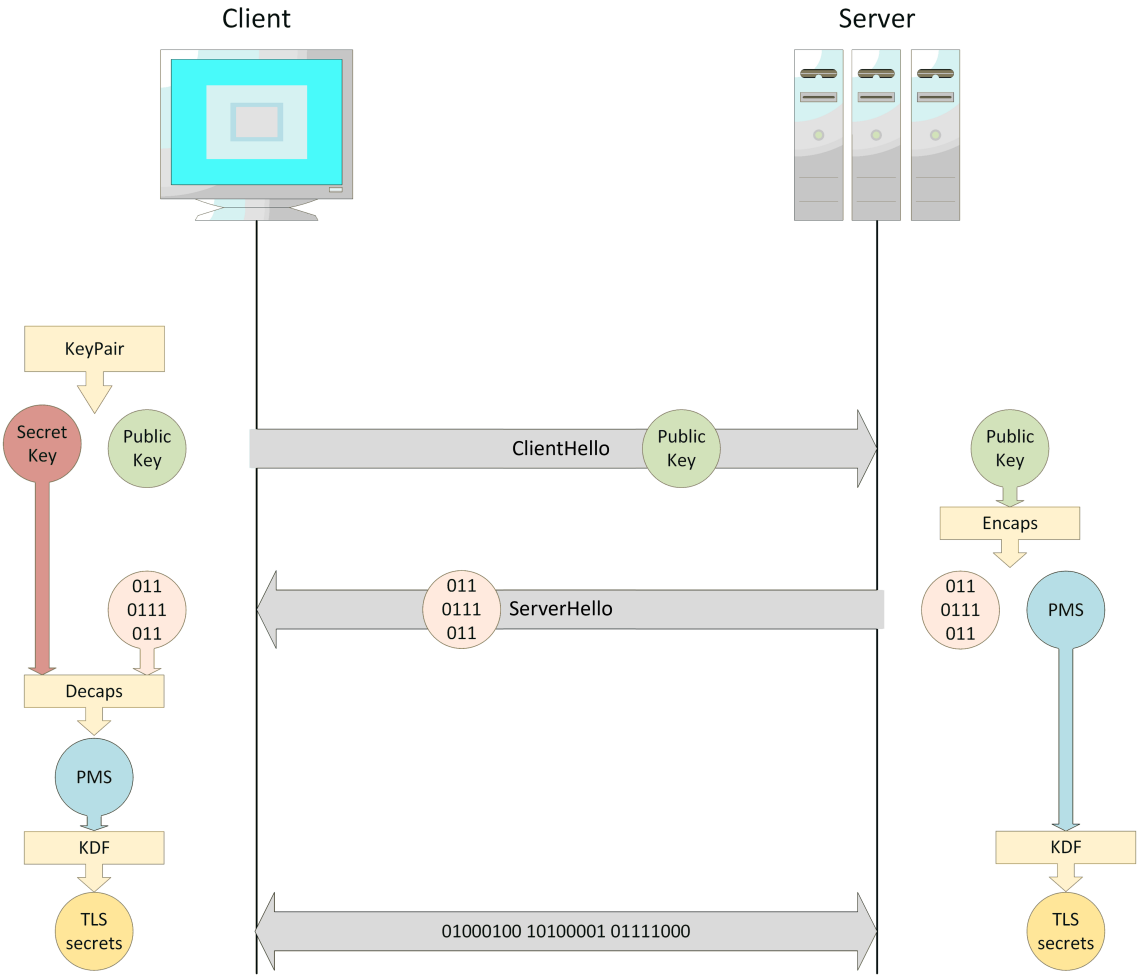

Figure 10: Basic pqTLS concept

As shown in Figure 10, such pqTLS may take only two roundtrips, the same as in TLS 1.3. In difference to TLS 1.2 with RSA or previous versions, the client sends the public key in the first message. The "difficult" part of the process, the key generation, is now on the client side. This may have several consequences:

- For lightweight devices, it may be impossible to do such computation. We need to add an option for sending a "not possible" message and change the roles of the client and the server in the protocol. This results in a similar protocol to TLS 1.2 with RSA key exchange.
- If the server doesn't support a specific cipher (for which the client generated a public key), the server needs to send a "supported public keys" message. The other option is that the client sends a list of "supported public keys" and its own public key in a ClientHello message. If the server doesn't support the client's public key, the server

knows which public key can be used and sends its own public key. The roles would be switched similar to the above.

- Because the "hard" part is not made on the server, the protocol is more resistant to Denial-of-service (DoS).

In the following subsections, we look into the specific messages in pqTLS -Handshake relating to the key exchange. For simplicity, some of the details of TLS are skipped here, but they would need to be considered in a real-world scenario (e.g. certificate, verify, and finished messages). For certificates, we can use post-quantum signature NIST candidates. For post-quantum KEM, we consider the NIST candidates mentioned in the sections above. More formal specification of pqTLS can be found in algorithm 1.

2.5.1 Client-side pre-computation

The client starts with calculating a keypair (private and public key) with default KEM. The client can generate more key pairs, one for each KEM that the client supports. However, this is not advised, as the message length may be too long.

2.5.2 ClientHello

The client sends the information structured in the ClientHello message, as shown in Figure 2. The changes to allow post-quantum key exchange are mainly in extensions. *Key Share* extension is used for sending a list of client public keys (of one or more different KEM keys generated in pre-computation). The server can use one in the key exchange. *Supported Groups* extension is no longer needed. Instead, the *Supported KEMs* extension is used for listing all KEMs that the client supports. *Supported Versions* extension indicates protocol version 03 05 (pqTLS). The rest of the ClientHello can stay as it is in TLS 1.3. The packet structure of ClientHello extensions required for post-quantum key exchange can be seen in Figure 11.

Algorithm 1 pqTLS

Client:

$TLS_c \Leftarrow$ TLS version (pqTLS)

for all supported_kems **do**

$(pk, sk) \Leftarrow KEM.KeyPair()$

KEMs+= (KEM, pk)

end while

$ClientHello \Leftarrow TLS_c, KEMs$

Client sends $ClientHello$ to Server

Server:

$TLS_s \Leftarrow$ TLS version (pq3TLS)

while KEMs[i] is not supported **do**

KEM=KEMs[i].KEM

pk=KEMs[i].pk

i++

end while

$(c, PMS) \Leftarrow KEM.Encaps(pk)$

$ServerHello \Leftarrow TLS_s, c, KEM$

Server sends $ServerHello$ to Client

Client:

$(PMS) \Leftarrow KEMs.Decaps(c, sk)$

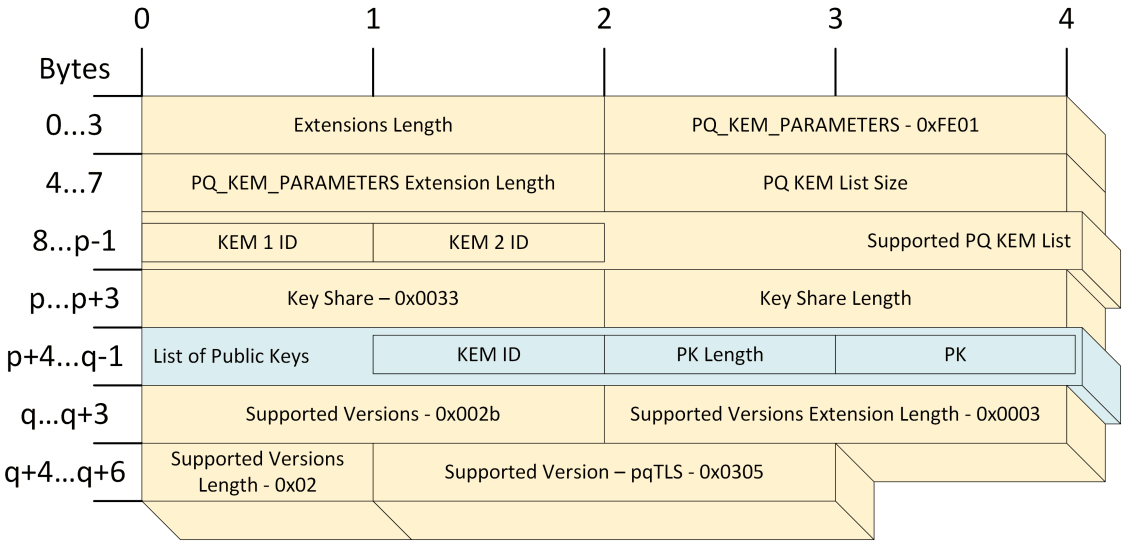

Figure 11: pqTLS ClientHello Extensions Packet

2.5.3 Server-side pre-computation

The server will use the public key from ClientHello, and the corresponding KEM *encaps* operation to create a cryptogram and a symmetric secret. The server now can compute other keys using symmetric secret as PMS. The Master Secret is produced using the *pbkdf2* key derivation function, with PMS as password, and *client random* concatenated with *server random* as a salt. OWASP recommended using 310000 iterations for PBKDF2-HMAC-SHA256 [73]. Server and Client key are generated similarly with Master Secret as a password and again client random concatenated with server random as a salt. *pbkdf2* is also used to produce implicit server Initialisation Vector (IV) with *server random* as a password, and *client random* as a salt, and vice-versa for client IV. Server key and server IV are used for encryption on the server side and decryption on the client side. Client key and client IV are used for encryption on the client side and decryption on the server side.

2.5.4 ServerHello

The server sends the information structured in a ServerHello (figure 3). It is similar to TLS 1.3, except for a few extensions. *Supported Versions* extension has protocol version 03 05 (pqTLS). *Key Share* extension is now used to transport encrypted symmetric secrets. After receiving ServerHello, the client can also compute all the keys and IVs using symmetric secret as PMS. The packet structure of ServerHello extensions required for post-quantum key exchange can be seen in Figure 12.

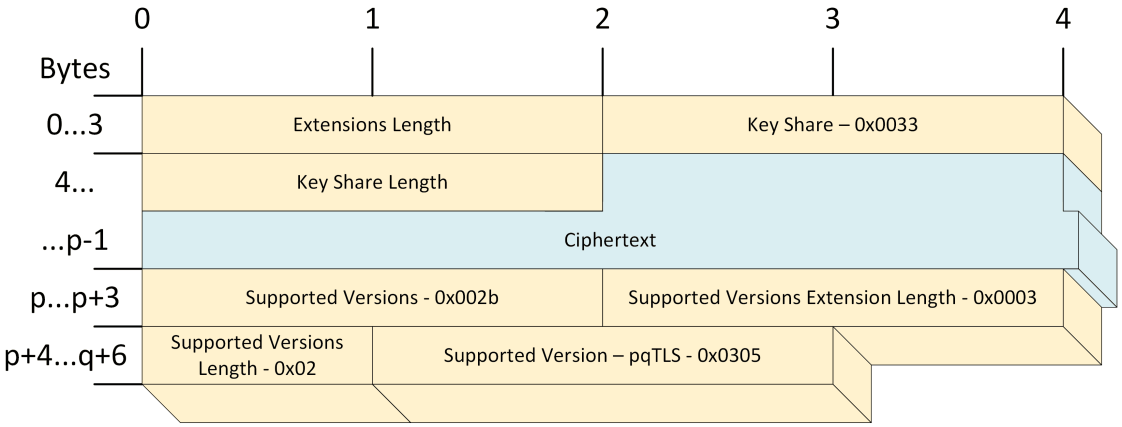

Figure 12: pqTLS ServerHello Extensions Packet

2.6 Post-quantum TLS for limited devices

As discussed earlier, some post-quantum KEMs may not be suitable for limited devices (e.g. Classic McEliece due to the size of public keys). Even if they were, the computational complexity of some operations may be so large, that computation may not be feasible or would be significantly slower on a limited device than on the server. This is why we present another option. If a limited client needs to use a KEM with a costly *keypair* operation, it can send the information about it in the first message. In practice, we can understand this scenario as the protocol with new messages, presented in Figure 13. The client asks for a connection with the first message. The server generates the key pair and sends it to the client. The client does only one KEM operation, *encaps*. The client saves generated Pre-Master Secret and sends the ciphertext to the server. The server now decapsulates PMS, and both sides share the same secret.

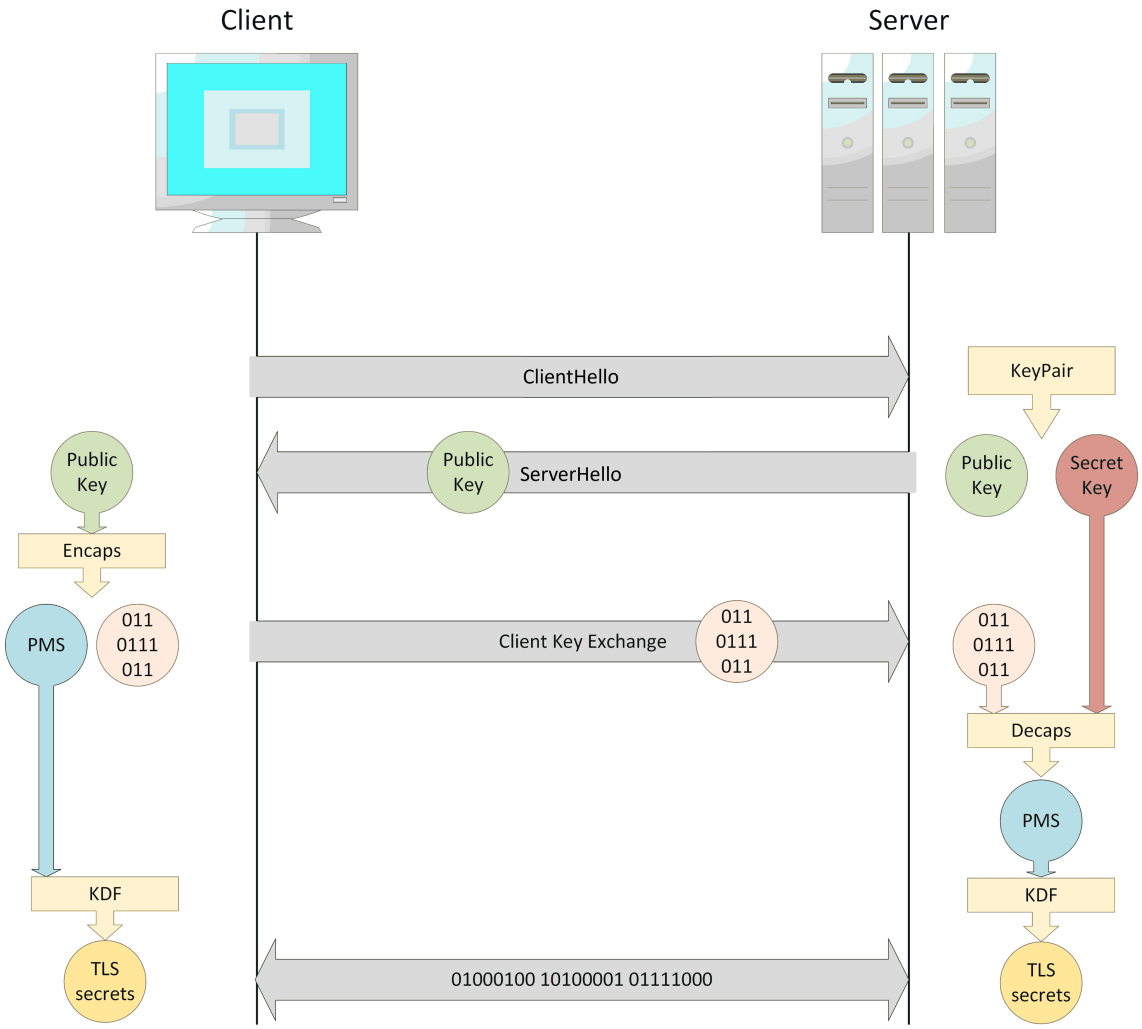

Figure 13: pqlimTLS concept

We explain each message in more detail below. We can see that we need one extra message, but it may be faster than waiting for a limited device to perform complex computations. We refer to this situation as pqlimTLS protocol. More formal specification of the protocol can be found in Algorithm 2.

Algorithm 2 pqlimTLS

Client:
$TLS_c \Leftarrow$ TLS version (pqlimTLS)
$KEMs \Leftarrow supported\_kems$
$ClientHello \Leftarrow TLS_c, KEMs$
Client sends $ClientHello$ to Server

Server:
$TLS_s \Leftarrow$ TLS version (pq3TLS)
while KEMs[i] is not supported **do**
  KEM=KEMs[i++]
end while
$(pk, sk) \Leftarrow KEM.KeyPair()$
$ServerHello \Leftarrow TLS_s, KEM$
Server sends $ServerHello$ to Client

Client:
$(c, PMS) \Leftarrow KEM.Encaps(pk)$
ClientKeyExchange $\Leftarrow c$
Client sends $ClientKeyExchange$ to Server

Server:
$(PMS) \Leftarrow KEMs.Decaps(c, sk)$

2.6.1 ClientHello

Here we refer to Figure 2 for the ClientHello message structure. We need to make a new ClientHello similar to the TLS 1.2 one. *Key Share* or *Supported Groups* extension is no longer used. *Supported KEMs* extension indicates which post-quantum KEMs will be used for key exchange. A list of all supported KEMs is sent. Supported Versions

extension indicates protocol version 03 06 (pqlimTLS). The packet structure of ClientHello extensions required for post-quantum key exchange can be seen in Figure 14.

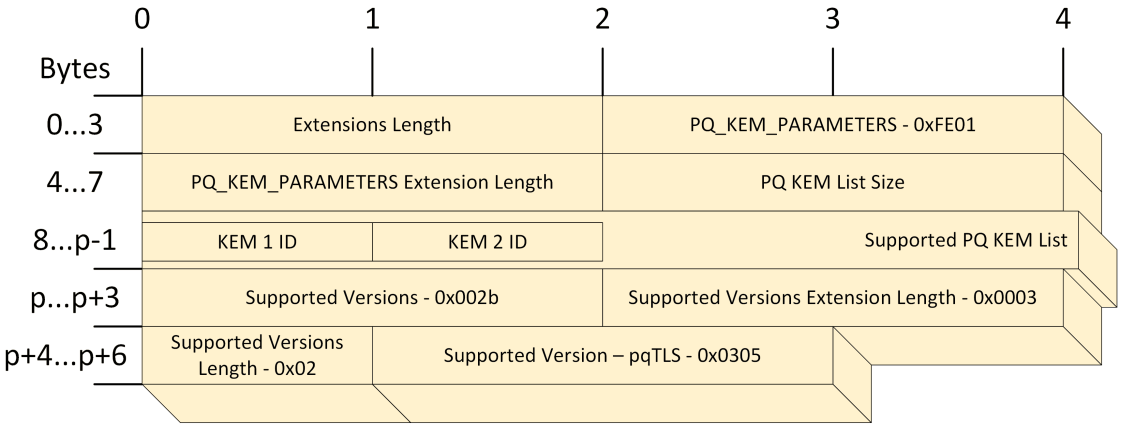

Figure 14: pqlimTLS ClientHello Extensions Packet

2.6.2 Server-side Pre-computation

After receiving ClientHello, the server chooses one of the client's supported KEMs and generates private and public keys using the KEM *keypair* operation.

2.6.3 ServerHello

In Figure 3, we can see the mandatory attributes of the ServerHello. Mostly, it is identical to TLS 1.3, but we need to change the extensions. Supported Versions extension indicates protocol version 03 06 (pqlimTLS). *Key Share* extension is used to transport the server public key for key exchange. The packet structure of ServerHello extensions required for post-quantum key exchange can be seen in Figure 15.

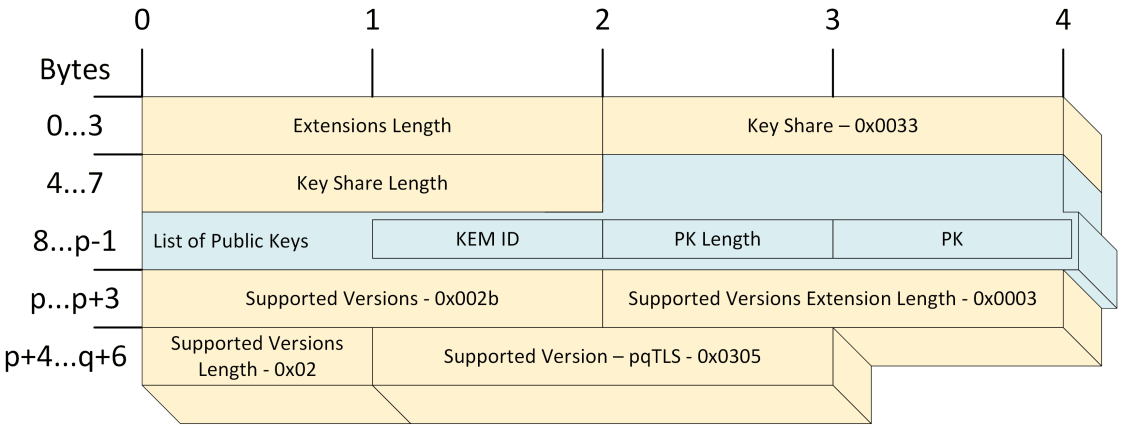

Figure 15: pqlimTLS ServerHello Extensions Packet

2.6.4 Client-side Pre-computation

After receiving ServerHello, with the use of the public key, a client generates a cryptogram and symmetric secret using encaps operation corresponding to KEM specified in ServerHello.

2.6.5 Client Key Exchange

Now we need one extra message in contrast to TLS 1.3. We can use TLS 1.2 Client Key Exchange message to transport encrypted symmetric secret (cryptogram). This message consists of a record and a handshake header as in other handshake messages, key length (in our case, length of the cryptogram), and key data (cryptogram). After receiving Client Key Exchange, Server uses decaps operation with a previously generated private key to decapsulate the cryptogram into the symmetric secret. The packet structure of our Client Key Exchange message can be seen in Figure 16.

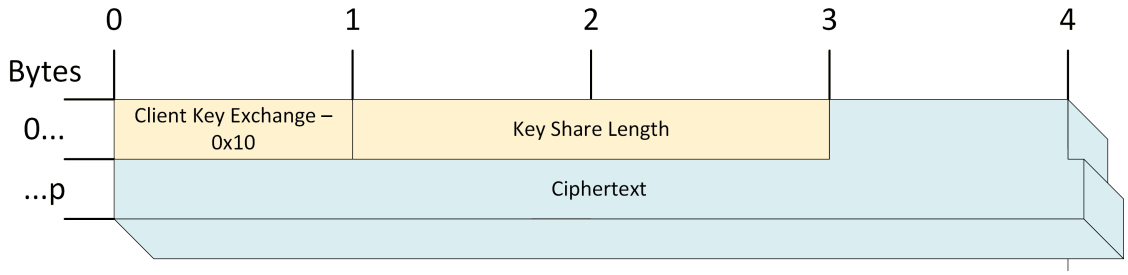

Figure 16: Client Key Exchange Packet

2.6.6 Differences between pqTLS and pqlimTLS

The main difference between pqTLS and pqlimTLS is that the client and server switched roles in the key exchange. In pqTLS, the client can start with the KeyPair operation before the first message, and ClientHello contains a public key. In pqlimTLS, the client just asks for connection, and the server starts with KeyPair operation. Similarly, Encaps and Decaps operations are switched between the client and server. As we can see from Figures 10 and 13, one extra message is required for pqlimTLS. We believe that even with one extra message, the benefit of doing complex operations on the server side, instead of a limited device is going to be more significant than the delay resulting from one extra handshake message.

2.7 Post-quantum authentication

TLS 1.3 uses signatures to provide authentication in a standard way. The server sends the certificate and signature in a ServerHello message. The client then verifies it and decides if it should be trusted. If the server sent a Certificate Request along with the ServerHello, then the client also sends the certificate, together with the Finished message. This model is widely analyzed [20] [74]. We can do the same in a post-quantum scenario, but we can use a post-quantum signature scheme instead of ECDSA [22] / EdDSA [23] or RSA [21] authentication (from the NIST standardization process).

However, this is not the only way of providing authentication. The authentication can be achieved by successful decryption of the challenge. This type of authentication was presented in a protocol in work [75]. It uses DH key exchange for authentication. This is widely used nowadays, for example, in Signal [76], or Noise [77]. This idea and other improvements were concluded in the work [64], where post-quantum KEM was used for key exchange and authentication. The idea is described in the following paragraph.

The static long-term server's public key would be sent in the certificate with the ServerHello message. After receiving, a client would encapsulate challenge value using post-quantum encapsulate KEM operation with the server's public key. The challenge value is incorporated in key derivation to generate keys for client and server finished messages. The ciphertext is then sent to the server with the Client Finished message, also containing the HMAC of the client finished key. The server decapsulates it and produces the same set of finished keys. If HMAC verification fails, the server aborts the protocol run. The server sends HMAC of server finished key along with server finished message. The client also verifies it and aborts the process if verification fails.

The results [64] show that this can save CPU time. However, it also adds one more message to the TLS handshake. This idea can also be used for our system to add authentication. The interesting thing when incorporating this scheme into our concept for limited devices is this; one extra message needed for authentication can be integrated into our Client Key Exchange message. Also, the combination of this kind of authentication and our proposal for limited devices does not require difficult operations performed on the client side. The client uses KEM encapsulate message twice, once with a server public key for authentication and once with a public key for key exchange. The client still doesn't have to generate the keys or decapsulate. If a not computationally difficult KEM encapsulate operation is used, the handshake on a limited client might result in speed-ups.

3 TEE-based post-quantum TLS

We decided to use a Trusted Execution Environment (TEE) for critical computations and storage of the keys in our work. The benefits of using such an environment include:

- Full isolation - no runtime code provisioning, the malware has no access to the data or the program run.
- No key transfers - The secret key can be generated and stored in TEE and never leave the environment.
- TEE can be tested and evaluated separately, and then it can be used on various systems without losing trust.

We explained the concept of TEE in more detail in section 1.8.

When designing a system that includes TEE, there are few possibilities for what to use.

The first is to use TEE extensions to the CPU. There are several industry-level CPU-TEEs: Intel's SGX [78] and AMD's SVM [79] ARM's TrustZone [80]. However, since there is no absolute isolation, the attacks can be found [81] [82] [83] [84]. Also, special hardware with OS modification requirements or a TEE-enabling hypervisor is needed.

Another option is to use a Hardware security module (HSM). If the design of the systems supports HSM, the use of such a module can be "plug and play". In our work, we chose to use SEcube hardware [85] as our HSM. SEcube seems like a good match in terms of portable, inexpensive but powerful HSM.

3.1 SEcube

SEcube has an MCU, CC EAL5+ -accredited SmartCard, and an ultra-low-power FPGA on the same chip. FPGA and SmartCard are called through specific MCU instructions. The MCU is STM32F4 - ARM 32-bit Cortex-M4 CPU. Flash memory has a size of 2 MiB, and SRAM has 256 KiB. That should be enough for most of our needs. There is no SRAM cache, so associated side-channel attacks are not possible.

3.1.1 SEcube SDK

SEcube provides an open-source Software development kit (SDK) on their website [86]. We decided to reuse some parts of this code and design to build our Crypto OS. Our

Crypto OS only manages hardware and software resources of the MCU, providing API for cryptographical functions.

The state of the art of open-source code is designed as described in Figure 17. It consists of two projects, one for the host side and one for the MCU.

Hardware side - the code on the device side is divided into several main modules represented by .c and .h files:

- SEcube Core - handles the main device loop and basic commands such as echo or init. More specific commands such as login, key_edit, etc. are handled in Dispatcher Core. Communication Core handles USB packet-like communication.
- Communication Core - handles data of requests and responses. If a command is executed, its input or output data are written in the request and response blocks and then sent with SD Card Driver and USB driver communication to HAL/LL libraries.
- Dispatcher Code - on top of the login, key management, and challenge/response functions, it handles dataflow from/to Smart Card, FPGA, and Security Core.
- Security Core - this is where all cryptographic functions are handled. In SDK version 1.4.1 the only implemented crypto algorithms are AES256 [15], CRC16 [87], PBKDF2 [88] and SHA256 [42]. Again, the underlying features are provided by HAL/LL libraries.

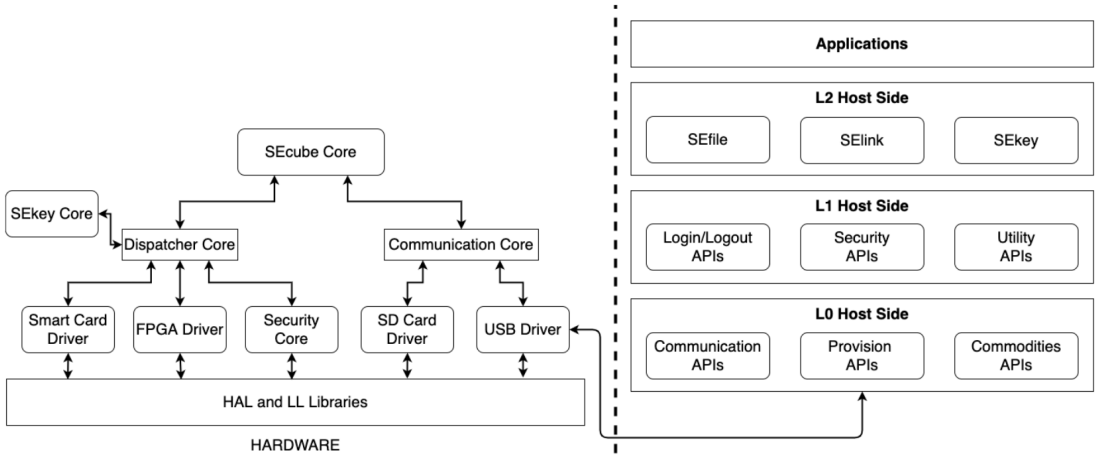

Figure 17: The architecture of the SEcube SDK [89]

Host Side - the code at the host side is divided into three libraries:

- L0 - this is the lowest library, handling TX and RX functions and opening and closing of the device (communication API), initialization of the device (provisioning API), and discovering SEcube devices (Commodities API)
- L1 - this is the most interesting library for us, as we are mainly changing and expanding this library on the host side. It handles login/logout key management and cryptographic algorithms.
- L2 - This library aims to bring an even higher level of abstraction, a key management system, data-at-rest protection, and data-in-motion protection. It is still in development. We are not using this library in our work.
- Newer documentations also mention the L3 library on the application level. As an example, they provided a Secure Database application [89]. It is not yet published at the time of writing this work.

The following chapters describe the changes and addition to this architecture to enable post-quantum security.

3.1.2 Side-channel attack resistance

In the work [90], the authors performed a side-channel analysis of SEcube. Namely, they studied the robustness of the device against Differential Power Analysis (DPA). DPA is a non-invasive passive attack that requires physical access to the device. The commercial USB token USEcube does not allow direct access to internal circuitry. That limits the possibility of attacks, making non-invasive attacks impossible. Nevertheless, the work describes the DPA attack on SEcube in comparison with the ST Microelectronics Nucleo board. They constructed a device to monitor USB power consumption to perform a non-invasive attack. DPA was performed on AES128 [15] ECB implemented in C. The attack focused on the whole AES, and AES S-box separately. The results were similar for both platforms. The attackers were able to reconstruct only around 4-5 bits of the secret key. Although this can speed up brute-force attacks on the key, it should not lead to severe damage.

3.2 Symmetric cryptography in a trusted environment

The first step in our HSM integration design is to provide a post-quantum level of security for symmetric cryptography (used e.g. in TLS record). In SEcube, SDK is AES implemented in the following modes; ECB, CBC, CTR, OFB, and CFB. As none of these modes is supported in TLS 1.3, we need to add GCM.

3.2.1 GCM

Galois/Counter Mode (GCM) is a newer child of counter mode that uses Galois field multiplication for authentication. We can describe the function of the mode using formalization from [91]. The field $GF(2^{128})$ is defined by the following polynomial:

$$x^{128} + x^7 + x^2 + x + 1$$

The tag provides the authentication, added to the end of the ciphertext. The tag is constructed using the GHASH function:

$$GHASH(H, A, C) = X_{m+n+1}$$

where $H = E_k(0^{128})$, which means a vector of 128 zero bits encrypted by an underlying block cipher, in our case AES256. A represents the plaintext data for authentication, and C is a ciphertext. m is the number of 128-bit blocks from A, n is the number of 128-bit blocks in C. X_i is then defined as follows:

$$X_i = \sum_{j=1}^{i} S_j \cdot H^{i-j+1} = \begin{cases} 0 & \text{for } i = 0 \\ (X_{i-1} \oplus S_i) \cdot H & \text{for } i = 1, ..., m+n+1 \end{cases}$$

S_i is defined as:

$$S_i = \begin{cases} A_i & \text{for } i = 1, ..., m-1 \\ A_m^* | 0^{128-v} & \text{for } i = m \\ C_{i-m} & \text{for } i = m+1, ..., m+n-1 \\ C_n^* | 0^{128-u} & \text{for } i = m+n \\ len(A) | len(C) & \text{for } i = m+n+1 \end{cases}$$

Where $M|N$ is concatenation of binary strings M and N. For a detailed scheme of the GCM operation, we refer to Figure 18. We can add GCM mode to SEcube SDK codebase and call it using the same mechanism as other AES modes. However, this brings us to the need for four inputs instead of three as in all other modes. We discuss this in section 4, which discusses implementation.

3.2.2 HSM symmetric cipher in TLS

For the scheme of usage of HSM in the TLS record, we refer to Figure 18. Red parts are encryption-related, and green are authentication-related data. Application data would be fragmented as in regular TLS 1.3. Data from each fragment are sent into HSM, divided into blocks of 128 bits. Each block is then xor-ed with an encrypted counter sequence. This produces ciphertext blocks. Each ciphertext block is then used to create an authentication block when it is XORed with the previous iteration of GHASH and inputted into the actual iteration of GHASH. The TAG also "consists" of additional data and the length of the plaintext, as can be seen in the scheme. The ciphertext is concatenated with TAG and sent back to the TLS record to be handled as usual.

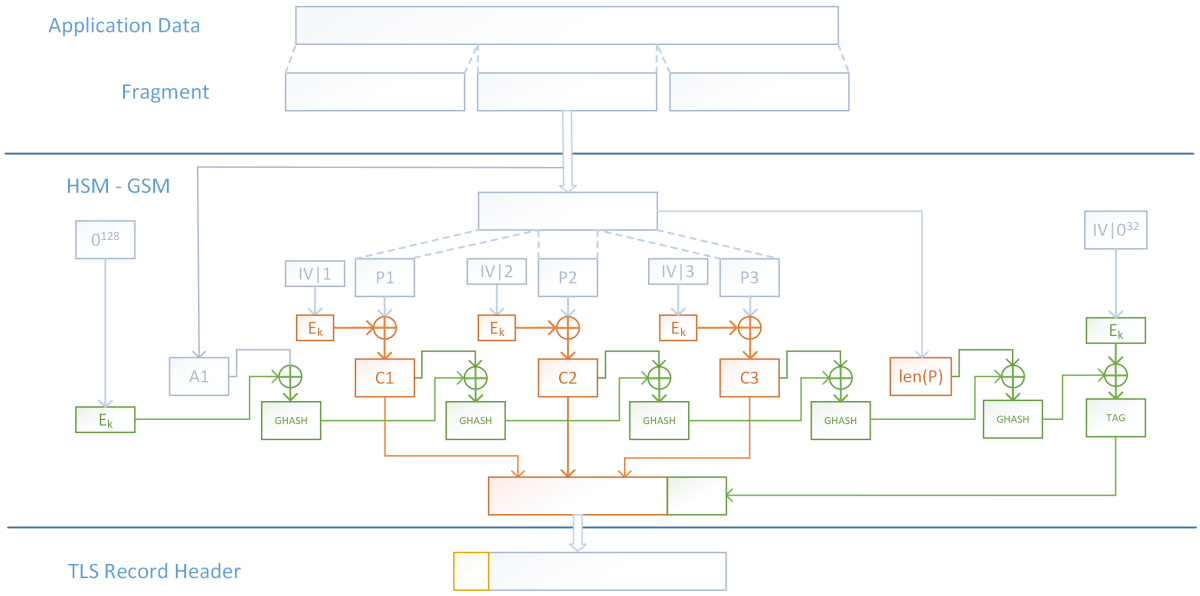

Figure 18: Integration of HSM GCM to TLS record

We can see that the block cipher is used to encrypt each counter iteration. Our design also assumes that keys are stored in the HSM and all AES key-related operations are performed in HSM. More of this is discussed in the following section 3.3.

3.3 Secret keys stored in the trusted environment

One of the benefits of using specialized hardware for security is separating unsecured and secured environments. If we want the keys to be protected from a potential attacker, they should be in a secure environment. In our system, we use the memory in the hardware device to store private keys. In the scope of HSM-aided TLS like protocol, we can identify three scenarios:

- The symmetric key for the record protocol cipher is written once into the HSM memory and never leaves. This mechanism is already supported in the SEcube SDK.
- The private key for the key exchange mechanism is stored in the memory of the secure hardware and never leaves. With the use of KEM API and keypair operation in the HSM, we can limit the output from keypair operation to provide only ciphertext. The secret key is stored in the device memory and not transported to the host. The only way to manipulate the secret key is to use it for *decaps* operation or destroy it. The secret key is never exposed; it is indexed with the key ID assigned to the key pair in the *keypair* KEM operation. This ID is the only information about the secret key on the host.
- The symmetric key is never stored in the host computer. KEM *encaps* or *decaps* operation outputs only key ID, so the keying material doesn't leave a trusted environment (HSM). KDF is performed in the HSM to produce all TLS keys and they are stored in the device memory. This way, no secret material or operation (except for the decrypted application data) is visible to the host.

3.4 Post-quantum public-key algorithms in TE

The introduction of HSM in TLS makes sense only if the key exchange mechanism (all KEM operations and key storage) is in the Hardware Security Module. We call such a module Post Quantum Cube (pq3). The module has a strictly defined API, following KEM scheme. We define a wrapper layer for KEM API:

- *keypair*: Output has one important modification. Instead of a public and a secret key, the HSM keypair operation returns the public key and only the ID of the secret key. We define secret key ID as input, to let the application have control over key IDs. Other inputs are KEM ID, key validation period, and HSM session information. Internally, the KEM *keypair* operation is called, then public and secret keys are stored in device memory under key ID with a specified validation period, and the public key is returned to the host.

- *encaps*: Input set consists of HSM session information, KEM ID, public key, validation period for the keys, master key ID, client key ID, server key ID, and client and server random nonces. *Encaps* operation outputs only ciphertext (encrypted PMS). Internally, the KEM *encaps* operation is called to encapsulate the PMS using a public key. Master Secret, Client, and Server keys are derived using the algorithm specified in section 2.5.3, and then they are all stored in the device memory with a specified key validation period. Encapsulated PMS is returned to the host.

- *decaps*: Inputs consist of HSM session information, KEM ID, secret key ID, ciphertext, validation period for the keys, master key ID, client key ID, server key ID, and client and server random nonces. HSM *Decaps* operation has no outputs. Internally, KEM *decaps* operation is called to decapsulate the PMS using stored secret key accessed with secret key ID. Master Secret, Client, and Server keys are derived using the algorithm specified in section 2.5.3, and then they are all stored in the device memory with specified key validation period.

3.5 Post-quantum cube TLS - pq3TLS

Finally, we describe the draft of Post Quantum TLS with the use of HSM. We consider HSM with implemented post-quantum cryptography described in previous sections (pq3). We call this protocol Post-Quantum Cube TLS (pq3TLS). The pq3 is used on the client side of the protocol; in this model, we trust the server side. We consider HSM as a device with limited computing power and memory. Thus, we start designing the protocol from our Post-Quantum TLS for limited devices described in section 2.6).

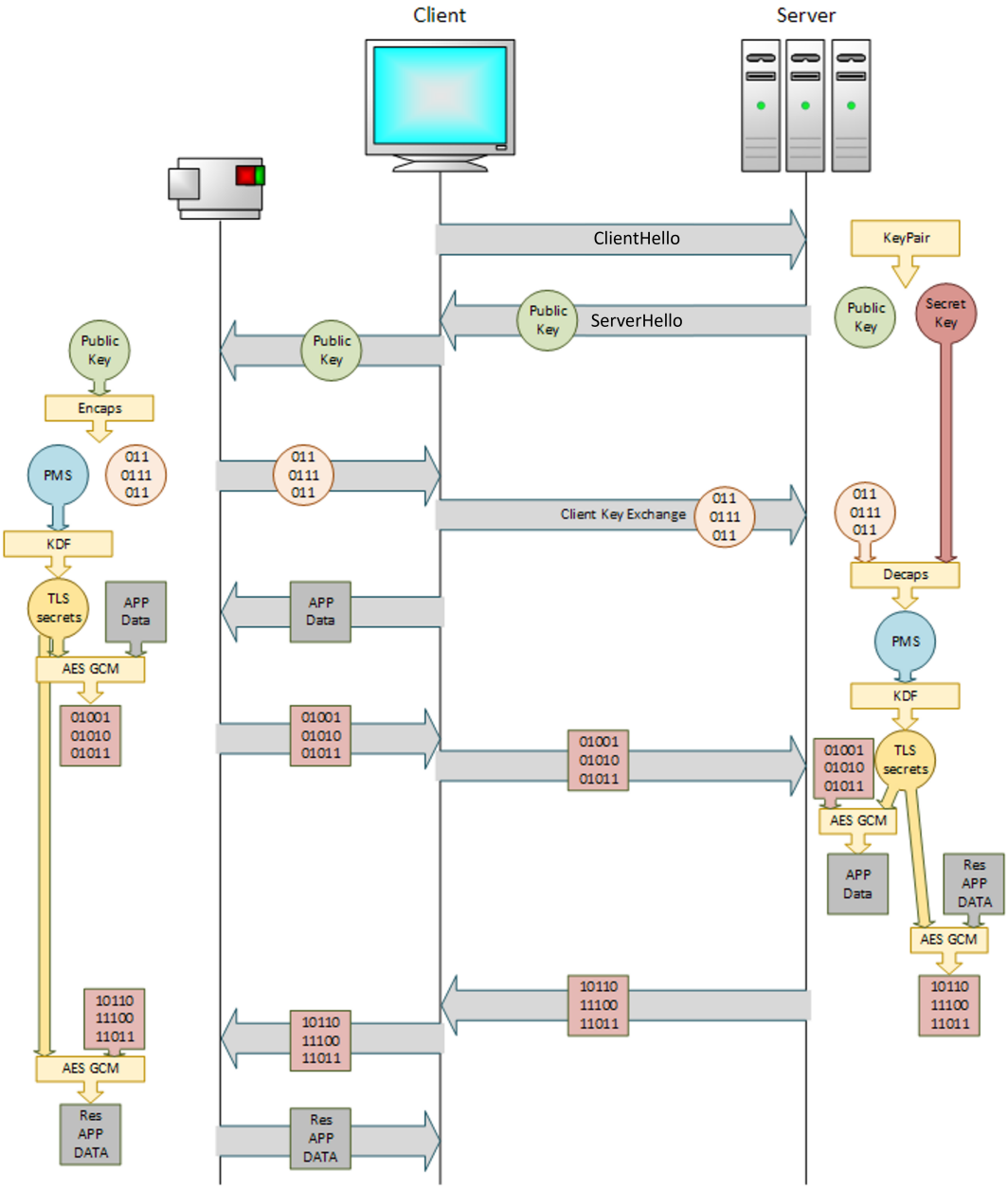

Figure 19: Post-Quantum Cube TLS

Again we neglect the authentication. As shown in Figure 19, the client does not store any secret key or perform any cryptographic operation. Before the ClientHello, the client plugs in the PQcube, logs in, and stores its session information.

We specified pq3TLS protocol in more formal way in Algorithm 3.

Handshake messages are specified as

ClientHello	C	$\longrightarrow$	S: $KEMs_C, TLS_C$
ServerHello	S	$\longrightarrow$	C: KEM, pk_S, TLS_S.
Host to HSM	C	$\longrightarrow$	H_C: pk_S.
HSM to Host	H_C	$\longrightarrow$	C: $E_{PK_S}(PMS_{CS})$.
ClientKeyExchange	C	$\longrightarrow$	S: $E_{PK_S}(PMS_{CS})$.

Where C is the Client, S is a server, and H_C is HSM plugged in to the client computer.

For ClientHello, we refer to Figure 2. Protocol version should be the same, only client-side implementation of KEMs should be changed to PQcube ones. The client sends a list of supported KEMs and TLS protocol version supported by the client. The server-side stays the same as in pqTLS. For the ServerHello message, we refer to Figure 3. The server sends TLS version and generated public key PK_S. The client-side pre-computation calls PQcube to generate the ciphertext $E_{PK_S}(PMS_{CS})$ and create master and client/server session keys. Key storage is in the device memory, as described in previous sections. Client Key Exchange message is the same as in Figure 16. The client sends ciphertext $E_{PK_S}(PMS_{CS})$ to server. The server decapsulates it and computes master and client/server session keys. Both sides has the same set of keys. Record protocol communication can be formalized in the following way:

$C \longrightarrow H_C$: $DATA$.
$H_C \longrightarrow C$: $E_{K_{CS}}(DATA)$.
$C \longrightarrow S$: $E_{K_{CS}}(DATA)$.

$S \longrightarrow C$: $E_{K_{SC}}(RESP)$.
$C \longrightarrow H_C$: $E_{K_{SC}}(RESP)$.
$H_C \longrightarrow C$: $RESP$.

Clients use HSM to encrypt application data that will be sent to the server. HSM encrypt it and send ciphertext back to the client. The client sends it to the Server. The server response is also sent to HSM to be decrypted.

Algorithm 3 pq3TLS

Client:

$TLS_c \Leftarrow$ TLS version (pq3TLS)

$KEMs \Leftarrow supported\_kems$

$ClientHello \Leftarrow TLS_c, KEMs$

Client sends $ClientHello$ to Server

Server:

$TLS_s \Leftarrow$ TLS version (pq3TLS)

while KEMs[i] is not supported **do**

  KEM=KEMs[i++]

end while

$(pk, sk) \Leftarrow KEM.KeyPair()$

$ServerHello \Leftarrow TLS_s, KEM$

Server sends $ServerHello$ to Client

Client sends pk to HSM

HSM:

$(c, PMS) \Leftarrow KEM.Encaps(pk)$

HSM sends c to Client

Client:

ClientKeyExchange $\Leftarrow c$

Client sends $ClientKeyExchange$ to Server

Server:

$(PMS) \Leftarrow KEMs.Decaps(c, sk)$

3.6 Side-channel attack resistance

The nature of our design provides some level of attack resilience. In addition to the side-channel attack resistance we describe in section 3.1.2, there are also other possibilities to prevent side-channel attacks for specific algorithms. These countermeasures provide a higher level of security at the cost of some extra time needed to perform the algorithms. To allow users to choose this higher level of security, we decided to integrate some additional countermeasures for algorithms that use number-theoretic transformation.

Some of the algorithms in the NIST standardization process use polynomial multiplication in the ring R_q. There are several options for implementing polynomial multiplication, such as schoolbook multiplication, Karatsuba algorithms, or number-theoretic transform (NTT) based multiplication. Lattice-based schemes such as Kyber (KEM), New Hope (KEM), or Dilithium (signature) use NTT-based one.

We briefly explain the idea of NTT, following [92]. The NTT is a bijective mapping from one polynomial to another in the same operating ring. Considering an (n-1) degree polynomial p in R_q, the polynomial p in the normal domain is mapped to its alternate representation $\hat{p}$ in the NTT domain through the NTT as follows:

$$\hat{p} = \sum_{i=0}^{n-1} p_i \cdot \omega^{i \cdot j} \tag{1}$$

where $j \in [0, n-1]$ and ω in the n^{th} root of unity in the operating ring Z_q. We can also denote ψ as $\psi^2 = \omega$. The powers of ω and ψ are called *twiddle constant*, and they are used in NTT computation. The inverse mapping from $\hat{p}$ to p is denoted INTT. The multiplication of polynomials x and y is then effectively done using NTT:

$$z = INTT(NTT(x) \bullet NTT(y)) \tag{2}$$

The NTT of a higher degree polynomial is then recursively broken down into smaller NTTs until we are left with atomic operations called *butterfly* operations, most common with size 2. The NTT/INTT of size n is typically computed in $log(n)$ stages with each stage consisting of $n/2$ butterfly operations. There are two types of butterfly operations. Cooley-Tukey (CT) operation is defined as

$$c = a + b \cdot w \tag{3}$$

$$d = a - b \cdot w \tag{4}$$

and Gentleman-Sande (GS) defines as

$$c = a + b \tag{5}$$

$$d = (a - b) \cdot w \tag{6}$$

with a and b as input coefficients, outputting c and d, and w as a *twiddle constant* (power of ω or ψ). More details on how NTT multiplication can be implemented can be found in [93].

We use the concept from [94] to introduce shuffling and masking countermeasures for NTT. Masking is done with random constant by introducing one additional multiplication:

$$c = (a + b \cdot w_x) \cdot w_y \tag{7}$$

$$d = (a - b \cdot w_x) \cdot w_y \tag{8}$$

where $w_x = \psi^x$ and $w_y = \psi^y$. Generally, n denotes the input length to the NTT and N the number of stages within the NTT. There are three possible options for masking countermeasures:

- Coarse Masking - uses a single mask for every stage, with same mask for inputs and same for output as:

  $$c' = (a' + b' \cdot w_x) \cdot w_y = c \cdot w_{(i+x)} \tag{9}$$

  $$d' = a \cdot w_{(i+x)} - b \cdot w_{(i+x+y)} \tag{10}$$

  where $a' = a \cdot w_i$ and $b' = b \cdot w_i$

- Fine masking - employs more masks in every stage. Adding another mask for input is done as:

  $$a'' = a' \cdot w_{(2ni+k)} = a \cdot w_k \tag{11}$$

  and butterfly operation is then:

  $$c' = (a' \cdot w_{(2n-i)} + b' \cdot w_{(2n-j)} \cdot w_x) \cdot w_y = c \cdot w_y \tag{12}$$

  $$d' = a' \cdot w_{(2n-i-l)} - b' \cdot w_x \cdot w_{(2n-j+l)} = d \cdot w_l \tag{13}$$

- Generic Masking - the number of masks is specified by choosing random number u, $1 < u < n$. Different butterfly operations (from previous maskings) are selected for specific stages.

Ravi in [94] also brings three options for shuffling countermeasures:

- Coarse (Full) Shuffling - All butterflies within any stage of the NTT can be computed independently. Coarse Shuffling randomly shuffles the order of execution of all (n/2) butterfly operations within any stage of the NTT.

- Coarse Group Shuffling - In that stage of the NTT, where every butterfly forms a unique group, we can do a complete n/2 length shuffle, which is considered enough in the scope of the third round NIST candidates' algorithms.
- Fine Bitwise Shuffling - Basic principle here is to shuffle the order of input loads and output stores for each butterfly. An order is generated randomly, using a randomly generated bit for masking, and then straightened using 16 iterative bitwise AND operations, in the case of 16-bit-long operands in Kyber.

4 Implementation details

This section describes the implementation details specific to the chosen implementation of TLS (s2n) and the selected instance of HSM (SEcube). We describe the findings and knowledge gained with the implementation of our design.

4.1 New public-key ciphers integration into s2n

Because TLS 1.3 and previous TLS versions explicitly specify what public key algorithm is to be used, only a few public-key ciphers are implemented in most TLS implementations. An open-source s2n implementation of TLS provides RSA [21], DH [27], or ECDH [36] implementations. On top of that, Crockett et al. implemented two post-quantum algorithms, Sike and Bike, in a hybrid scheme [62]. As we know, neither Sike nor Bike made it to the third round as finalists of the NIST standardization process.

The implementation of KEM schemes has a strictly defined form. It should contain kem.c file with following code (Figure 20):

```
int crypto_kem_keypair(unsigned char* pk, unsigned char* sk)
{ // key generation implementation }

int crypto_kem_enc(unsigned char *ct, unsigned char *ss, const unsigned char *pk)
{ // key encapsulation implementation }

int crypto_kem_dec(unsigned char *ss, const unsigned char *ct, const unsigned char *sk)
{ // key decapsulation implementation }
```

Figure 20: kem.c file structure

This NIST requirement, to have uniform interface helps with the compatibility of the KEM schemes. The switching from one scheme to another is straightforward, so it is possible to use several in one system. This aspect, combined with the collection of clean implementation of algorithms in PQClean helped us add all third-round candidates into s2n without significant difficulties. We used some of the structures already provided in s2n and added our own when necessary.

4.1.1 Classic McEliece

Classic McEliece comes in 10 different versions [95]. Because we are interested in strength equivalent to AES256 [15], we consider only category five versions, mceliece6960119, mceliece6688128, mceliece8192128, and their semi-systematic form versions. We inte-

grated mceliece6960119 implementation from PQClean. However, we decided not to use Classic McEliece implementation in our experiments for the following reasons:

The problem occurs if we want to have backward compatibility with previous and current TLS versions. The maximum size of a TLS record is 16 389 bytes. It consists of 1 byte for content type, 2 bytes for the protocol version, 2 bytes for the length field, and finally up to 2^{14} for the encrypted+compressed payload data. This would also include the public key for post-quantum KEM. Classic McEliece has the size of public keys from 1 044 992 bytes to 1 357 824 bytes. Here we have several options:

- Increase the maximum size of the TLS record.
- Split the McEliece public key into several record messages.
- Use only smaller (weaker) versions of Classic McEliece.

Changing the TLS record size or splitting the public key to several messages would affect the compatibility with other TLS versions or significantly slow down the communication (instead of one TLS record message, we would need around 80 messages). We also decided to avoid using weaker versions of Classic McEliece to have a consistent security level.

For this reason, we consider Classic McEliece KEM **not suitable** for TLS handshake in our system.

4.1.2 CRYSTALS-KYBER

CRYSTALS-KYBER [68] was specified and implemented in three versions, *Kyber512*, *Kyber768*, and *Kyber1024*, corresponding to categories 1, 3, and 5. We integrated the PQClean implementation of *Kyber1024*, as it is clean and easy to integrate.

4.1.3 NTRU

NTRU comes with 4 parameter sets: *ntruhps20485091*, *ntruhrss7013*, *ntruhps20486773* and *ntruhps4096821*. The security analysis [69] defined two evaluations: relative to non-local models of computation and relative to local models. In local models, signals propagate at finite speeds (e.g. speed of light). Non-local allows communication at an arbitrary distance. Version *ntruhps4096821* achieved category five relative to local models. We integrated it using PQClean implementation.

4.1.4 SABER

SABER [70] was specified with three-parameter sets, corresponding to categories 1 (lightSaber), 3 (saber), and 5 (fireSaber). We integrated fireSaber implementation from PQClean.

4.2 Implementation of pqTLS protocol

For building a post-quantum TLS-like protocol, we decided to use the s2n codebase in version 0.9.0 from 2019, which included modification from Crockett et. al. [62]. In the previous section, we described the integration of NIST post-quantum KEMs. This section explains the implementation of the post-quantum protocol for key exchange.

As the server and client have to wait for incoming messages during the handshake, the communication is not full-duplex. s2n uses a single *stuffer* for incoming and outgoing data. The so-called *stuffer* is a stream-like structure with a buffer (called *blob*) and read and write cursors.

The heart of TLS implementation is the handshake state machine located in tls/handshake_io.c. It is represented with a table of function pointers. These functions are handlers for each message, for server and client site. Each message contains a flag to indicate who is the writer and the receiver of the message. In the writer's handler function, the data are written to the handshake io stuffer, and in the receivers handler, the data are extracted from the stuffer. This stuffer is then used in the TLS Record part. Accordingly to the messages that we defined in section 2.5, we created a new state machine for pqTLS. The state machine can be seen in Figure 21.

```
static struct s2n_handshake_action pqtls_state_machine[] = {
    /* message_type_t        = {Record type, Message type, Writer, {Server handler, client handler} } */
    [CLIENT_HELLO]           = {TLS_HANDSHAKE, TLS_CLIENT_HELLO, 'C', {s2n_establish_session,
        s2n_client_hello_send}},
    [SERVER_HELLO]           = {TLS_HANDSHAKE, TLS_SERVER_HELLO, 'S', {s2n_server_hello_send,
        s2n_server_hello_recv}},
    [SERVER_FINISHED]        = {TLS_HANDSHAKE, TLS_FINISHED, 'S', {s2n_pqtls_server_finished_send,
        s2n_pqtls_server_finished_recv}},
    [CLIENT_FINISHED]        = {TLS_HANDSHAKE, TLS_FINISHED, 'C', {s2n_pqtls_client_finished_recv,
        s2n_pqtls_client_finished_send}},


    [APPLICATION_DATA]       = {TLS_APPLICATION_DATA, 0, 'B', {NULL, NULL}},
};
```

Figure 21: Minimal pqTLS state machine

The next step was to modify the rest of the files to recognize this new handshake in the mentioned file and the rest of the codebase. We created s2n_pqtls.c and s2n_pqtls.h with functions required for using this new protocol.

We needed to modify or create new handlers for each handshake message. For that, we prepare usage of KEM, similarly to the solution in [62], we added a new *key exchange*

instance and new *cipher suite*. We also made new *cipher preferences* that contain NIST third-round post-quantum candidates and a previously created *cipher suite* to allow *key exchange* operations for post-quantum algorithms. We added further codes for post-quantum algorithms:

- TLS_PQ_KEM_EXTENSION_ID_FIRESABER_R3 with id 32
- TLS_PQ_KEM_EXTENSION_ID_NTRU_R3 with id 33
- TLS_PQ_KEM_EXTENSION_ID_KYBER_R3 with id 34
- TLS_PQ_KEM_EXTENSION_ID_MCELIECE_R3 with id 35

4.2.1 Client Hello

As we explained in section 2.5.2, there were several challenges when creating the Client Hello message. We needed to change the s2n code not to require the ECC mechanism and not to use the *Supported Groups* extension. *Supported Versions* field was set to 03 05 to indicate pqTLS protocol. We used the standard *Key Share* extension wrapper for the public key, but we called the previously integrated *keypair* KEM operation to generate the public key. We used the KEM mechanism from post-quantum modification of s2n [62] to select the exchange algorithm. We used blu5 implementation of PBKDF2-HMAC-SHA256 to calculate PMS and other secrets and IVs for the rest of the record. We also adopted the *PQ_KEM_PARAMETERS* extension from [62] that suits well to serve as our *Supported KEMs* extension. *TLS_EXTENSION_PQ_KEM_PARAMETERS* has value 0xFE01 in s2n. After receiving Client Hello, the server sets corresponding *cipher preferences* and KEM from client preference (the list is ordered by preference). The server also saves the client public key into the *s2n_kem_keypair* structure.

4.2.2 Server Hello

Here we refer to section 2.5.4, where we explained the architecture of our new Server Hello. We implemented the pqTLS mechanism into the s2n, where we specify *Supported Versions* as 03 05 to indicate the pqTLS protocol. Using the *s2n_client_key_send* function in s2n, we called our KEM *encaps* operation to generate the PMS (symmetric secret) and fill the *Key Share* extension with it. The *s2n_client_key_send* function also calls a function to calculate keys from the generated PMS. After receiving Server Hello and parsing its extensions, s2n_client_key_recv is used on the client-side to call our KEM

decaps operation (using the previously generated private key) to decapsulate the PMS and calculate the rest of the keys.

4.3 Implementation of pqlimTLS protocol

In the implementation, we follow the design described in section 2.6. Similarly to pqTLS (section 4.2), we started with the handshake state machine. It can be seen in Figure 21. It contains one extra message, Client Key Exchange, as we designed in section 2.6.

```
static struct s2n_handshake_action pqlimtls_state_machine[] = {
       /* message_type_t        = {Record type, Message type, Writer, {Server handler, client handler} } */
       [CLIENT_HELLO]           = {TLS_HANDSHAKE, TLS_CLIENT_HELLO, 'C', {s2n_establish_session,
            s2n_client_hello_send}},
       [SERVER_HELLO]           = {TLS_HANDSHAKE, TLS_SERVER_HELLO, 'S', {s2n_server_hello_send,
            s2n_server_hello_recv}},
       [CLIENT_KEY]             = {TLS_HANDSHAKE, TLS_CLIENT_KEY, 'C', {s2n_pqlimtls_key_recv,
            s2n_pqlimtls_key_send}},
       [SERVER_FINISHED]        = {TLS_HANDSHAKE, TLS_FINISHED, 'S', {s2n_pqlimtls_server_finished_send,
            s2n_pqlimtls_server_finished_recv}},
       [CLIENT_FINISHED]        = {TLS_HANDSHAKE, TLS_FINISHED, 'C', {s2n_pqlimtls_client_finished_recv,
            s2n_pqlimtls_client_finished_send}},

       [APPLICATION_DATA]       = {TLS_APPLICATION_DATA, 0, 'B', {NULL, NULL}},
       };
```

Figure 22: Minimal pqlimTLS state machine

Analogically to pqTLS, we created and modified all the required files so s2n recognizes pqlimTLS protocol. We used the same KEMs implementations, with the same *cipher preferences* and KEMs.

4.3.1 Client Hello

When implementing the concept presented in 2.6.1 into s2n, the Client Hello is implemented similarly to the one in TLS 1.3. *Supported Versions* was set to 03 06 to indicate pqlimTLS protocol. Supported KEMs extension is again implemented with PQ_KEM_PARAMETERS s2n extension. It contains the list of client-supported KEMs. After receiving the Client Hello, like in the previous case, the server parses its extensions and sets corresponding *cipher preferences* and KEM from client preference.

4.3.2 Server Hello

Where generating the Server Hello, *s2n_extensions_server_pq_key_share_send* function is used. It uses the previously chosen implementation of KEM to generate a key pair. *PQ_KEM_PARAMETERS* extension includes public key length and data. When the

client receives the Server Hello, it parses the extensions and stores the public key in *s2n_connection.secure.s2n_kem_keys* structure.

4.3.3 Client Key Exchange

The client now encapsulates the PMS using the same *s2n_client_key_send* as in section 4.2.1, and calculates the rest of the secrets from the generated symmetric key. Client Key Exchange message contains an encapsulated symmetric key. The server similarly decapsulates the key and calculates the rest of the secrets using *s2n_client_key_recv* function. Now both parties share the same set of IVs and record secrets.

4.4 Modification of s2n code to enable external device

There are several things that need to be implemented to enable external HSM in s2n. We decided to use SEcube and its SDK in version 1.4.1 as a basis for our code. Version 1.4.1 is written in C on the host side, and s2n is also implemented in C. We used library *L0* with minimal modifications (i.e., TIMEOUT value) as it is in SDK code. The changes in *L1* follow our architecture (section 3.2 - section 3.5) and are described in the following corresponding sections.

In s2n, like in previous cases, we added new *key exchange*, *cipher suite*, and *cipher preferences* for post-quantum algorithms in an external device. The reason for avoiding McEliece was already explained.

- TLS_PQ_KEM_EXTENSION_ID_FIRESABER_PQ3 with id 64
- TLS_PQ_KEM_EXTENSION_ID_NTRU_PQ3 with id 65
- TLS_PQ_KEM_EXTENSION_ID_KYBER_PQ3 with id 66

In cipher preferences, we also set s2n_pq3tls_record_alg to handle Record operations.

A user needs to log in to perform a crypto operation on SEcube. We use the mechanism from SEcube SDK. After opening the device, a login session on the device is created. This session is on the host side, represented by the se3_session structure. It contains login information and is needed to perform crypto operations and logout. Because this session has to be stored in s2n, we created a structure named pq3_ctx. It contains all device and session information. We also added *present* flag to indicate if the HSM has been plugged in. We added this to s2n_connection structure to be present for the whole TLS communication process.

4.5 New symmetric cipher integration into s2n

File s2n_cipher.h contains the declaration of all supported ciphers. We added a new AEAD structure and a new cipher type (HSM-pq3 cipher). This new type has a dual implementation of AES GCM. One is on the host side (for server and for the client when PQcube is not plugged in), and the other is *wrapper* function calling HSM implementation of AES GCM. Session info is stored in s2n_connection (s2n_connection) to perform only one login and one logout for each TLS connection. We preserved a very similar architecture to the one that was already implemented in s2n for AEAD ciphers, but we added SEcube

session info and changed the TLS session key to the structure containing key ID and direction (encryption or decryption).

We tested our implementation using our created unit test (s2n_pq3_aead_aes_test). The test simulates client-server communication using Record protocol with our implementation in PQcube.

4.6 AES-GCM implementation in PQcube

We followed the recommendation of block cipher modes of operation from NIST, specifying the implementation of Galois/Counter Mode and GMAC [96] and the open-source implementation of GCM mode in the wpa_supplicant [97] tool. We created Cortex M4 implementation of GCM on top of Blu 5's implementation of AES.

One more modification of the SEcube SDK is needed. Besides IV, key (key ID if stored in HSM), and data to be encrypted, we need an extra input for additional data to provide encapsulation. We changed the crypto call accordingly, adding data0 field that is not used in all modes and cryptographic algorithms except AES GCM, which uses additional data for authentication only. The tag is then added after the last block of the ciphertext.

The change in implementation has to be done on both sides. On the host side, L1_crypto_update function has to be changed to send data0 from the host side to the device, and on the device side, crypto_update's request parameters are changed.

4.7 Post-quantum algorithms suitable for devices with limited resources

We have already described the implementation of TLS handshake using post-quantum KEMs. These KEMs are now moved to HSM, according to the architecture presented in section 2.1 and section 3.4.

4.7.1 PQClean implementation of algorithms

We started the integration of post-quantum algorithms with PQClean implementation [98]. PQClean is a collection of clean implementations of the post-quantum schemes from the NIST post-quantum project. The project aims to provide a standalone implementation of each version of each scheme suitable for the evaluation of implementation security and formal verification. The goal is to have the code that is clean and easy to understand. Along with the clean implementation, PQClean provides some target-specific implementation (avx2), but the code is not optimized for ARM Corex M4 in seCube. However, it is possible to run PQClean implementations in a 32-bit microcontroller unit.

4.7.2 Source of randomness

All PQClean KEM schemes need randomness to generate key pairs. PQClean provides the functions to generate the sequences of random bytes that depend on a specific operating system. SeCube includes a true random number generator that relies on 240 physical noise seeds [85]. This allows true random noise generation. We integrated this TRNG using HAL library in all post-quantum algorithms in our solution.

Another way of generating the randomness that we integrated is via libopencm3 [99] library. The library is an open-source firmware for ARM Cortex-M microcontrollers, including M4. Since it uses the same TRNG, there was no difference in performance between these two methods.

4.7.3 Hash function implementation

All third-round NIST candidates use fips202 standard of SHA-3. The implementation in PQClean is not optimized for Cortex-M4. Instead, in all schemes in our solution, we use optimized assembly implementations of SHA-3 from mupq back-end of pqm4 [65].

4.7.4 Classic McEliece

The sizes of public keys differ from 1 044 992 bytes to 1 357 824. This brings the limitation: the STM32F4 chip has 256 KB of RAM, which is too small for the classic McEliece public

key. We can run the code itself, but we cannot work with the public key. The authors of the paper [95] also provided FPGA-based Niederreiter Cryptosystem implementation [100] that may be implemented on seCube FPGA, but hardware implementation is outside the scope of our research.

4.7.5 CRYSTALS-KYBER

We integrated PQClean implementation of *Kyber1024* into the SEcube. To achieve better results in embedded devices, Botros, Kannwischer, and Schwabe in [101] implemented a memory-efficient and high-speed version of Kyber for Cortex-M4. We also integrated this category 5 *Kyber1024* lightweight version into the seCube.

4.7.6 NTRU

We integrated version *ntruhps4096821* using PQClean implementation. To optimize the performance of NTRU on Cortex-M4, we added the optimized implementation with faster polynomial multiplication by Kannwischer–Rijneveld–Schwabe[102] from pqm4[65].

4.7.7 SABER

Along with reference implementation, SABER comes with the implementation for Cortex-M4. Instead of optimizing speed or memory consumption, this implementation gives a trade-off between speed and memory with a low-performance penalty. We integrated this modification. Also, we integrated fireSaber implementation from PQClean.

4.8 Side-channel resistance

To implement side-channel resistant countermeasures discussed in section 3.6 into the SEcube chip, we decided to use the modified implementation from the project *Configurable SCA Countermeasures for the NTT Against Single Trace Attacks* [103]. The project contains several implementations of side-channel countermeasures for the Number Theoretic Transform (NTT) based on shuffling and masking. NTT is used for polynomial multiplication in the lattice-based Kyber KEM scheme. This implementation is targeted the ARM Cortex-M4 microcontroller, so it is possible to use it in SEcube.

4.9 PQcube system

The final step in our implementation was to enable PQcube in s2n as described in section 3.5. We created a simple *plug-and-play* solution, with automatic detection of the HSM module and automatic use of it on the client-side if it is present. There were a large number of small changes to s2n, and for the scope of the document, it would not be possible to list them all, so we mention only the most important ones.

4.9.1 Implementation of HSM KEM

The implementation follows the design specified in section 3.4. We created a structure called se3_kem_descriptor. It contains all operation handlers, sizes, and KEM info. Then, we created an array of se3_kem_descriptors in se3_security_core containing all implemented KEMs, described in section 4.7. Then, we created keypair, encaps and desaps functions corresponding to design in section 3.4, and connected them with dispatcher_call to handle HSM KEM commands. We parsed information from *the SEcube packet*, implemented functionality and constructed response. For key manipulation, we used already existing key management from SEcube SDK.

4.9.2 Key Derivation

After encapsulation or decapsulation operation, we use PBKDF2 to create keys. Even though NIST suggested using at least 1000 iterations [104], the performance of such a large amount of iterations for PBKDF2HmacSha256 is causing unacceptable delay on Cortex M4. KEM encaps operation outputs 32 bytes (256 bits) of entropy. As these 32 bytes of PMS are never exposed, we used only 100 iterations to create 48bits Master Secret and 100 to generate Client and Server Key. We used the implementation of PBKDF2HmacSha256 from SEcube SDK.

4.9.3 s2n PQcube handshake

The integration of PQcube into the pqlimTLS handshake required several major changes. pq3 cipher preferences have a dual KEM implementation set, one for the host side (for the server and the client without HSM) and the other for the device side (the client with HSM). We changed pqlimTLS implementation accordingly to call PQcube on the client-side if present. Also the public key, Master Secret and client and server keys are not stored on client-side, only their IDs (in *conn->secure.s2n_pq3_kem_keys.public_key*, *conn->secure.pq_client_key* and *conn->secure.pq_server_key*).

4.9.4 s2n PQcube record

We integrated the AES GCM PQ3 implementation as described in previous sections. To allow this new type of cipher, a record protocol (s2n_record_write and s2n_record_read) needed to be changed. Here we also ask whether the device is present. If not, we use our implementation of GCM mode on the host side. Else, we called the *wrapper* function of HSM GCM implementation.

5 Experiments

We designed a series of experiments to test and evaluate each specific component of our system and evaluate the whole system. Here we provide the results and benchmarks to show that the implementation of these constructs is possible, and we also show the practicality of our concepts. We designed the experiments to be simple and to stay as close as possible to the real-world usage of the system.

5.1 Measurement Methods

In our experiments, we used different ways of measuring desired features. The methodology of the measurements were adapted to the nature of the experiments. Some experiments require overall time measurements, some only the time when the CPU is not idle. There are several ways to measure the time of the program runtime and the efficiency of the algorithms. To be consistent with NIST post-quantum candidates publications, we decided to count CPU cycles combined with other time measurements, depending on the experiment.

5.1.1 Time measurements

We recognize two ways of measuring the time of the program run:

- **Process time**, sometimes called CPU time, is the amount of time for which the CPU was used for the specific task. It can be used to measure the efficiency of algorithms. It is not dependent on real-world variables such as CPU temperature and multitasking delays. The CPU time can be measured in clock ticks (CPU cycles) or seconds. It can be further divided into:
  - **User time** - is the time of the CPU used in user space;
  - **System time** - is the time of the CPU used in kernel space;
  - **Idle time** - is the time when the CPU was not busy.

  Another indicator of the program's efficiency may be CPU time as a percentage of the CPU's capacity. High CPU usage means the program is highly demanding of processing power.

- **Elapsed time** or wall-clock time is the overall time from the start of the program to the end. Although this type of time measurement contains delays from phenomena

unrelated to the program run, it is more suitable to show real-life use of the tested component. One other benefit of this type of measurement is that it can evaluate systems that run on several devices, where we are interested in overall time, including I/O operations, time spent in computation in other devices, etc.

We have several options how to measure under Linux operating system:

- GNU **time** - this command runs the specified program and returns general statistics about the program run. A *-v* option can be used to access Elapsed time, User and System time, and CPU capacity in percentage.
- The **gettimeofday()** function can be used to compute the time difference, and thus Elapsed time of the researched sub-procedure.
- The **clock()** function returns the CPU time elapsed since the start of the program. This can be used to determine the CPU time of researched sub-procedure.

CPU cycles can be measured with several methods, depending on the platform (STM32, Intel, AMD, etc.) or other specific properties.

5.1.2 CPU cycles

CPU cycles is a simple tool to receive cycle count [105]. It provides a script that creates cpucycles.h and cpucycles.o specifically for the platform used. It provides the implementation for 32 and 64-bit architecture and for many CPUs.

5.1.3 SysTick System Timer

ARM Cortex-M4 comes with a 24-bit system timer, SysTick, with four registers:

- SysTick Control and Status Register ($SYST\_CSR$) serves for timer configuration (enabling, disabling, exception on overflow, etc.). Register address is 0xE000E010.
- Reload Value Register ($SYST\_RVR$) sets the value that is reloaded when the timer reaches zero. Register address is 0xE000E014.
- Current Value Register ($SYST\_CVR$) holds the actual value of the timer. Register address is 0xE000E018.
- Current Value Register ($SYST\_CALIB$) - Some Arm devices also implement the Current Value Register for additional calibration. Register address is 0xE000E01C.

Timer counts from $SYST\_RVR$ value to zero. When it reaches zero, the timer can throw an exception and reload the countdown from $SYST\_RVR$ value again. The advantage of using SysTick is that the counter does not decrement if the processor is halted for debugging.

5.1.4 Data Watchpoint and Trace

Another method of counting CPU cycles is through Data Watchpoint and Trace (DWT) unit. It contains several different counters for clock cycles, sleep cycles, cycles per instruction (CPI), etc. DWT has various registers for measuring CPU cycles. We are using two of them:

- Control Register (DWT_CTRL) with address 0xE0001000
- Cycle Count Register (DWT_CYCCNT) with address 0xE0001004

The first step is to enable trace bit in Debug Exception and Monitor Control Register ($DEMCR$) on the address 0xE000EDFC and CPU cycle counter in DWT_CTRL. The DWT_CYCCNT is a 32-bit register that counts from zero up.

5.2 Measurements and results

Our goal was to test all aspects of our system. Our experiments evaluate post quantum KEMs in TLS setting running on common laptop as well as in HSM. Also we evaluated all proposed protocols. Table 5 summarise those experiments.

Experiment	Protocol	Tested algorithms	Reference
Post-quantum algorithms in TLS	-	Saber, Kyber, NTRU	secp256r1, secp384r1, x25519
Post-quantum TLS protocol	pqTLS	Saber, Kyber, NTRU	-
Post-quantum TLS for lightweight client	pqlimTLS	Saber, Kyber, NTRU	pqTLS
Benchmarking post-quantum security on SEcube	-	Saber, Kyber, NTRU	[65]
Benchmarking masked implementation of Kyber	-	3+3 protected options	Unprotected
SEcube post-quantum KEMs integration in s2n	-	Saber, Kyber, NTRU	PC
PQcube Client-Server Handshake	pq3TLS	Saber, Kyber, NTRU	PC
PQcube for symmetric crypto	TLS 1.3	AES-GCM	Alexa Top 100

Table 5: Summary of experiments

In following subsections we present each experiment, its setup and results. For better comprehension we also added diagrams.

5.2.1 Post-quantum algorithms in TLS

This experiment focused on the usability of post-quantum KEMs in TLS protocol. As we mentioned in section 1.9, similar experiments were conducted for several implementations. In this experiment, we tested the integration of the NIST round-3 candidates into s2n implementation of TLS as described in section 4.1.

5.2.1.1 Experiment setup We implemented this experiment as a unit test in our modified s2n. We measured the time and CPU cycles of keypair generation, encapsulation, and decapsulation of a Pre-Master Secret. Every operation was performed ten times in a row on a Dual-Core Intel Core i5-3317U CPU/6GB RAM machine. Here we present average values. For measuring CPU cycles, MIT's cpucycles tool [105] is used. The time is the time consumed by the algorithm (User time + System time) without Idle time.

5.2.1.2 Results The results can be seen in Table 6 and in diagram in Figure 23. We included time in milliseconds, and CPU cycles count for each KEM operation for all three candidates. For reference, we also included three supported key exchanges in TLS 1.3. The *keypair* is the sum of key generation on the client-side and server-side, *encaps* is Pre-Master Secret computation for server, and *decaps* Pre-Master Secret computation for a client. The results show that selected candidates are comparable to the public cryptography used in TLS in recent versions of TLS, and that the post-quantum TLS key agreement is feasible.

KEM	keypair ms	keypair cycles	encaps ms	encaps cycles	decaps ms	decaps cycles
FireSaber	2.32	3985254	3.05	5758520	3.61	6473779
Kyber1024	1.11	1906907	1.45	2469797	1.90	3247865
ntruhps4096821	128.78	221008637	1.75	2987527	3.98	6791942
secp256r1	0.29	495620	0.85	1457620	1.51	2578596
secp384r1	6.28	10650280	16.44	28115380	24.78	42268140
x25519	0.25	422364	0.47	1022152	0.65	1318440

Table 6: Post-Quantum KEMs Benchmarks

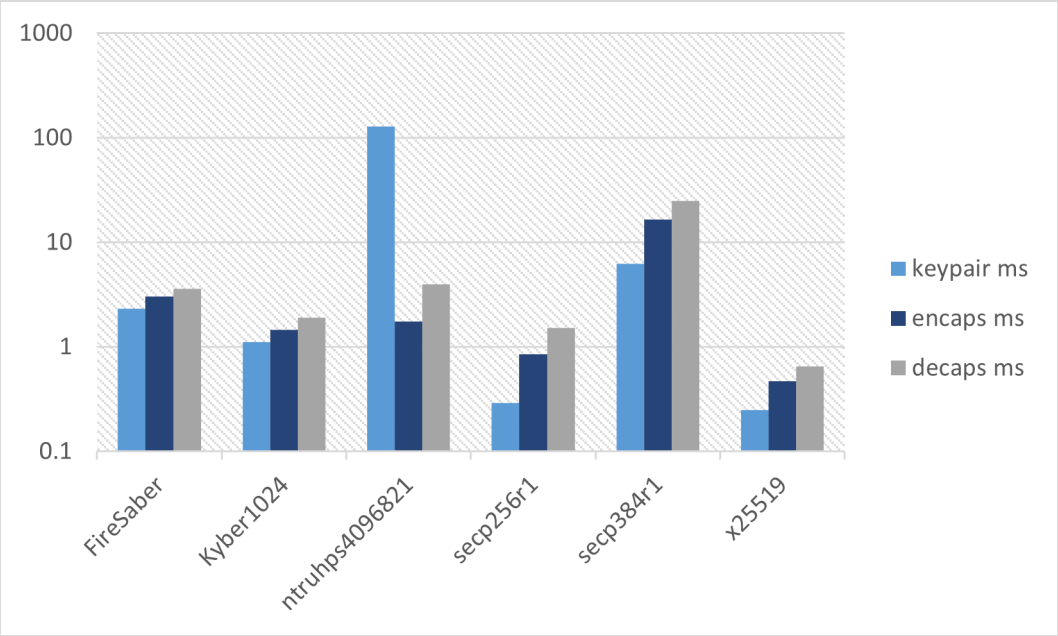

Figure 23: Post-Quantum KEMs

5.2.2 Post-quantum TLS protocol

The results from the previous experiment show us that it is possible and practical to use post-quantum key exchange in TLS like protocol. We designed (section 2.5) and implemented (section 4.2) such protocol using post-quantum KEMs. This experiment shows the real-world use of post-quantum key exchange in TLS handshake.

5.2.2.1 Experiment setup We decided that it would be best to test post-quantum TLS protocol close to a real-world scenario, so we created a client and server program communicating via a socket connection. Programs were implemented as unit tests, testing the whole handshake. Because the client is initiating the connection, we measured the performance of the client program, including connection to the server, negotiation of the handshake (with post-quantum key agreement), and the connection closing. We run the handshake negotiation five times for every post-quantum KEM for this experiment. We used the GNU time command for the performance measurements because we are interested in the time for the whole connection, not only the client CPU time. We used the same Dual-Core Intel Core i5-3317U CPU/6GB RAM machine.

5.2.2.2 Results Table 7 shows us the benchmarks of post-quantum TLS handshake. The values are averages of five measurements. Times are in seconds, User and System time shows us the complexity of the client-side, and Elapsed time is the overall time for handshake negotiation (client and server-side). CPU percentage shows how difficult the client-side was for the processor. As the previous experiment already suggested, the NTRU time is much greater than the rest. It is caused by the time complexity of keypair operation. Saber and Kyber perform better in this scenario because their key generation is faster. For better understanding we also provide diagram in Figure 24

Handshake KEM	User time (s)	System time (s)	CPU %	Eleapsed time
FireSaber	0.16	0.03	62.00	0.30
Kyber1024	0.02	0.00	70.10	0.05
ntruhps4096821	1.47	0.03	94.40	1.58

Table 7: Post-Quantum TLS Benchmarks

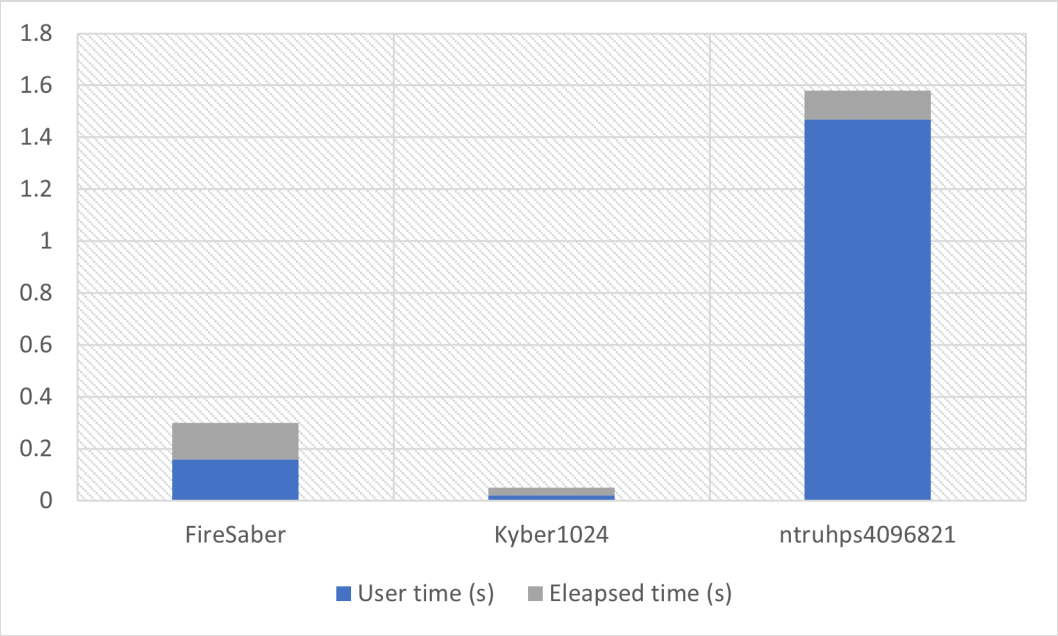

Figure 24: Post-Quantum TLS

5.2.3 Post-quantum TLS for lightweight client

In the previous experiment, we have shown that post-quantum cryptography can be used for TLS handshake key agreement. The results hinted that for NTRU, it may be difficult or not feasible to be run on a limited device. We presented the solution in sections 2.6 and 4.3, where we switched the roles of client and server in terms of KEM operations. This experiment focuses on finding whether this may benefit the TLS handshake in limited devices.

5.2.3.1 Experiment setup For this experiment, we used RaspberryPi zero (ARM 1 GHz CPU / 0.5GB RAM) for the client and Dual-Core Intel Core i5-3317U CPU/6GB RAM laptop for the server. We used a similar scenario to the previous experiment, but client and server programs now communicate using WiFi. We measured handshake negotiation from the client-side, again with the help of the GNU time -v command. Because previous experiments indicated that NTRU is the candidate that may not be able to be run on limited devices, we measured the handshake using NTRU. We used the GNU time command for measuring performance.

5.2.3.2 Results The results of the experiment can be seen in Table 8 and in diagram in Figure 25. Client time is the time spent by the client in a handshake. The Client CPU is the busyness of the client CPU in percents. The first two rows show averages of five measurements. The third row shows the percentage (pqlimTLS/pqTLS), and the last row shows the performance bonus gained by our modification. We can see that even though the RaspberryPi zero is more powerful than average IoT device, the overall time of handshake improved by almost four percent. This time benefit would be even more significant on a less powerful device. Furthermore, the client involvement in handshake was reduced by more than 90 percent in both CPU workload and amount of time spent in computation by the client. This result shows that our modification can save a lot of time for IoT clients using cryptography and even make some algorithms possible or practical that would not even be considered before the modification.

	Client time (s)	Client CPU %	Overall eleapsed time
pqTLS	1.50	94.40	1.58
pqlimTLS	0.13	8.20	1.52
%	9.09	8.69	96.08
performance bonus	90.90%	91.31%	3.92%

Table 8: Benchmarks of post-quantum TLS for limited devices

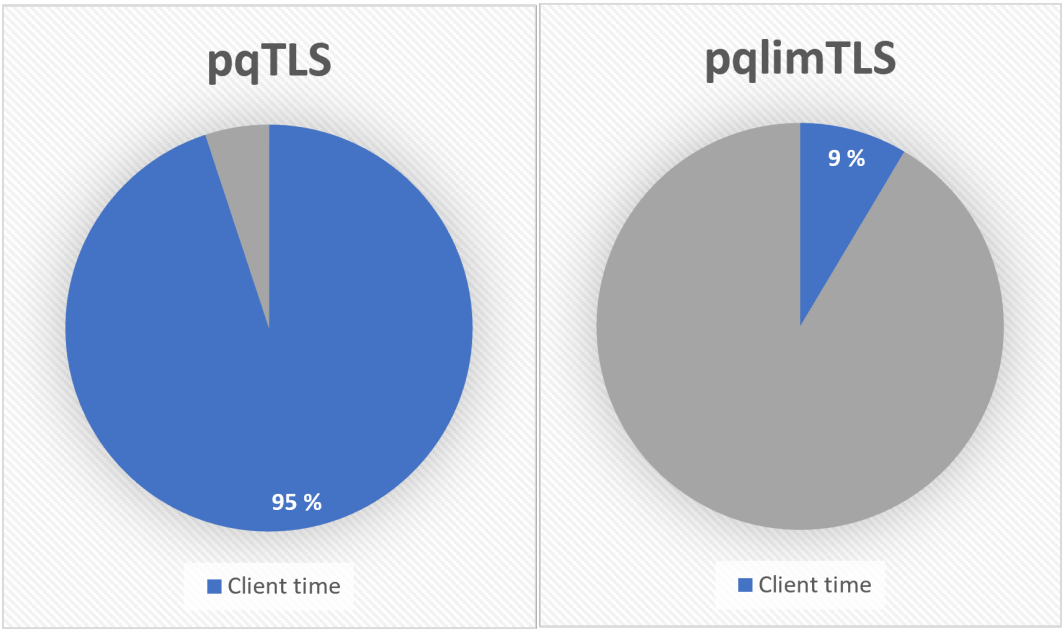

Figure 25: Post-Quantum TLS for limited devices

5.2.4 Benchmarking post-quantum security on SEcube

Our next experiment describes the possibility of using post-quantum algorithms in HSMs or micro-controllers in general. We describe the performance of post-quantum algorithms in terms of CPU cycles. We also offer the comparison of reference implementation and platform-optimized implementation.

5.2.4.1 Experiment setup As our Cortex-M4 target, we use SECube. We used the implementation described in section 4.1. The default frequency on Cortex-M4 is 168 MHz. To avoid wait-states when the processor fetches instructions, we use a 24 MHz clock for the speed tests.

First, we wanted to use the SysTick timer that is present on ARM-based microcontrollers. For most cases, it is an efficient and easy solution. Still, we found that it is not suitable for our need, as with some measurements (of ntruhps4096821), even the maximal capacity of the SysTick RELOAD register (24 bits) is too small. We decided to work with DWT, where we can use 32 bit, which is enough for all tests.

Implementation	Key generation	Encapsulation	Decapsulation
Our experiments (PQClean based)	4 390 006	5 323 618	5 004 678
Our experiments (Optimized for M4)	3 871 506	4 638 430	4 075 736
From paper [65] (PQClean based)	1 891 737	2 254 703	2 407 858
From paper [65] (Optimized for M4)	1 575 052	1 779 848	1 709 348

Table 9: Table of Kyber1024 speed in CPU ticks on Cortex-M4

Implementation	Key generation	Encapsulation	Decapsulation
Our experiments (PQClean based)	226 037 026	2 951 205	7 483 009
Our experiments (Optimized for M4)	221 874 517	1 179 052	1 578 356
From paper [65] (PQClean based)	289 736 570	7 046 106	19 262 764
From paper [65] (Optimized for M4)	211 758 452	1 205 662	1 066 879

Table 10: Table of ntruhps4096821 speed in CPU ticks on Cortex-M4

5.2.4.2 Results In Tables 9,10, and 11, we present the comparison of various implementations and measurements. Each test was performed ten times in a row for keypair

Implementation	Key generation	Encapsulation	Decapsulation
Our experiments (PQClean based)	6 528 985	8 098 992	8 504 554
Our experiments (Optimized for M4)	3 649 327	4 511 426	4 180 554
From paper [70] (Optimized for M4)	1 360 577	1 674 409	1 703 896
From paper [65] (Reference)	3 815 672	4 745 405	5 402 295
From paper [65] (Optimized for M4)	1 448 776	1 786 930	1 853 339

Table 11: Table of FireSaber speed in CPU ticks on Cortex-M4

generation, encapsulation, and decapsulation of the symmetric key. We compare the results of our measurements with those from available literature. We can see that although we did the experiments with very similar implementations, it was not possible to achieve the same results. We discussed this with the authors of the paper [65], and compared the process, but we did not find a significant difference. The difference in speed is probably the result of different platforms used (STM32F4DISCOVERY in [65] and SeCube). However, the results show that even with the challenges that post-quantum algorithms bring, it is feasible to use Cortex M4 based HSM in real-life scenarios.

5.2.5 Benchmarking masked implementation of Kyber

Besides the side-channel resistance presented in section 3.1.2, we described the integration of side-channel resistant implementation of Kyber in section 4.8. We use masked and shuffled NTT implementation. This experiment shows the performance of different versions of integrated countermeasures.

5.2.5.1 Experiment setup We were interested in total time for the whole procedure, including random generation, NTT operations, and kyber-related operations. Keypair generation, encapsulation, and decapsulation tests were performed ten times in a row for each masking/shuffling setting on the SeCube ARM Cortex-M4 chip.

5.2.5.2 Results In Table 12, we can see the difference between the implementation with no protection and the masking and shuffling countermeasures described in section 3.6. We can see that for masking options, the overhead is negligible, and thus we advise using generic multiplicative masking. If possible, even the use of fine multiplicative masking is advised. For shuffling options, overhead can go up to 74.63%. However, we think that if a higher level of security against side-channel attacks is required, even this overhead would be tolerable and that the USB interface would still make a backbone of this system.

Protection	**Keypair**	**Overh.**	**Encaps.**	**Overh.**	**Decaps.**	**Overh.**
	CPU cycles	%	CPU cycles	%	CPU cycles	%
No protection	5 069 586		5 768 645		5 358 027	
Coarse shuffling	8 267 776	63.09	9 372 600	62.47	9 356 986	74.63
Group coarse shuffling	7 853 633	54.92	8 890 793	54.12	8 838 696	64.96
Fine shuffling	5 981 787	17.99	6 799 084	17.86	6 506 489	21.43
Coarse multiplicative masking	5 206 074	2.69	5 924 128	2.70	5 532 311	3.25
Fine multiplicative masking	6 264 827	23.58	7 107 005	23.20	6 839 041	27.64
Generic multiplicative masking	5 443 916	7.38	6 190 240	7.31	5 826 209	8.74

Table 12: Performance of masked kyber in CPU cycles.

5.2.6 SEcube post-quantum KEMs integration in s2n

From previous experiments, we found that most of the candidates are suitable for implementation on Cortex M4, with promising results. In this experiment, we focused on the integration of SEcube post-quantum HSM (PQcube) into s2n as described in section 3.4 and section 4.7.

5.2.6.1 Experiment setup The experiment was performed on a Dual-Core Intel Core i5-3317U CPU/6GB RAM computer. We measured calls of s2n wrapper implementations of corresponding KEMs implementations in SEcube. We performed each measurement ten times in a row using clock().

5.2.6.2 Results Results can be seen in Table 13 and on diagram in Figure 26. Shown results are averages in milliseconds and CPU cycles for each KEM operation. For reference, we also added measurements from section 5.2.1, showing benchmarks of post-quantum implementations without HSM. We can see that although the HSM speed and USB communication introduce slowdown, it is not big enough to be considered a major obstacle to our project. It shows us that we can introduce another security countermeasure for the 10-100 millisecond trade-up.

KEM	keypair	keypair	encaps	encaps	decaps	decaps
	ms	CPU cycles	ms	CPU cycles	ms	CPU cycles
pq3-FireSaber	11.29	175 591 257	3.80	58 048 684	3.97	55 810 915
pq3-Kyber1024	12.24	195 639 355	3.17	56 847 923	2.86	51 105 791
pq3-NTRUhps4096821	219.80	3 531 194 675	1.47	22 990 808	1.61	25 375 951
FireSaber	2.32	3 985 254	3.05	5 758 520	3.61	6 473 779
Kyber1024	1.11	1 906 907	1.45	2 469 797	1.90	3 247 865
NTRUhps4096821	128.78	221 008 637	1.75	2 987 527	3.98	6 791 942

Table 13: Post-Quantum CUBE KEMs Benchmarks

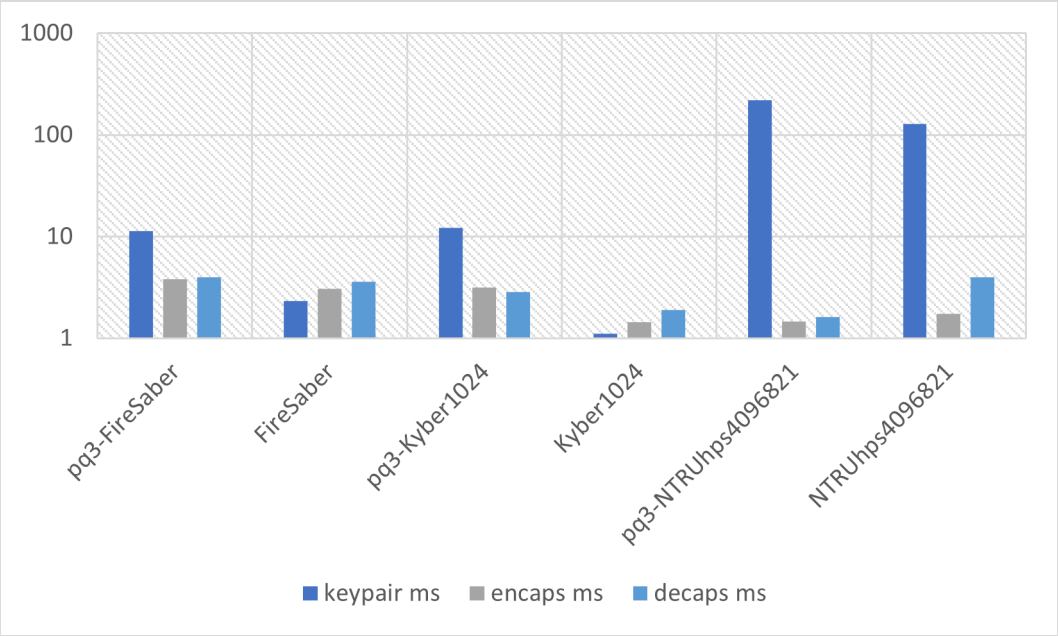

Figure 26: Post-Quantum CUBE KEMs

5.2.7 PQcube Client-Server Handshake

Finally, this experiment shows the usability of the PQcube system in a real-life scenario. This experiment sums up our efforts to create an HSM-driven post-quantum TLS handshake key exchange as designed in section 3.5 and implemented in section 4.9.

5.2.7.1 Experiment setup We created two separate test programs, one for the server-side and the other for the client-side. The server is the same as in section 5.2.2. Client uses our PQcube HSM. The experiment was performed on a Dual-Core Intel Core i5-3317U CPU/6GB RAM computer, entire server computations were performed there, and all client cryptography was performed on PQcube. Because the basic model for pq3TLS is pqlimTLS, the client-side is doing encapsulation and master key, client key, and server key derivation, all in PQcube. The result is that both sides share the same set of keys on the client-side stored in PQcube HSM. For reference, we also measured the client that is not using PQcube. All measurements were done with GNU time -v command.

5.2.7.2 Results Table 14 contains the average values made from five measurements in a row. User and System time in seconds, CPU busyness in percentage, elapsed time in seconds, and overhead introduced by HSM usage in seconds and percentage. We can see that usage of HSM is introducing delays, and although these delays may seem to be significant in relative percentage (817.86%), we need to consider that handshake is an only a small proportion of the connection, and this delay in absolute (0.4s) may not disturb user experience as much. We provide a diagram in Figure 27 for better understanding.

Handshake KEM	User time s	System time s	CPU %	Elapsed time s	Overhead s	Overhead %
pq3-ntruhps4096821	0.010	0.016	1.8	1.69	0.34	25.56
pq3-FireSaber	0.006	0.024	7.2	0.54	0.44	442.00
pq3-Kyber1024	0.006	0.034	8.2	0.51	0.46	817.86
ntruhps4096821	0.024	0.002	2.0	1.34		
FireSaber	0.032	0.000	36.8	0.10		
Kyber1024	0.014	0.002	39.8	0.06		

Table 14: Post-Quantum TLS Benchmarks

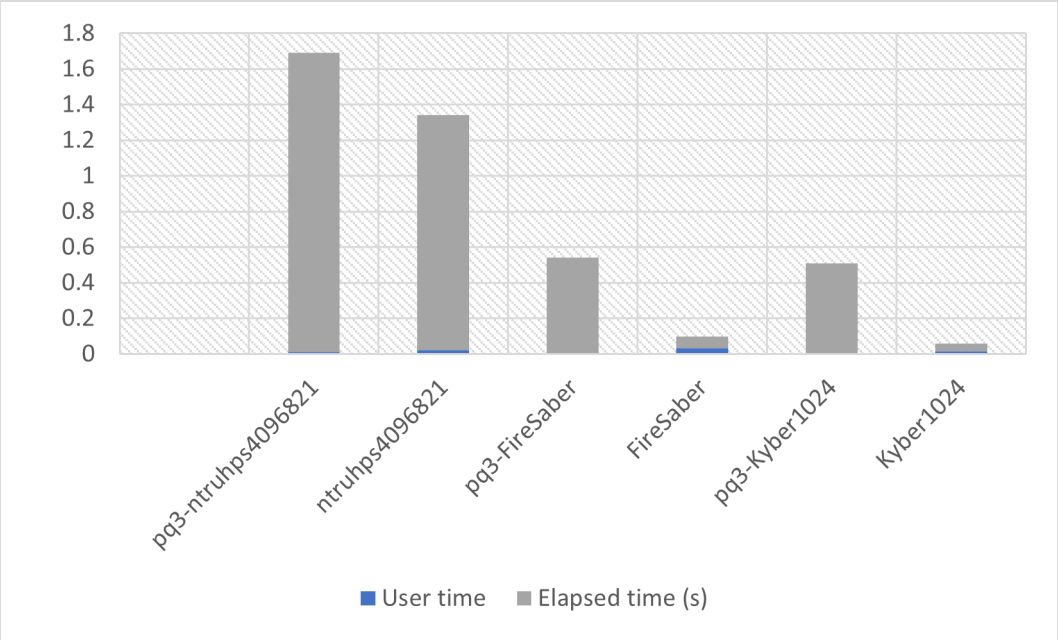

Figure 27: Post-Quantum TLS

5.2.8 PQcube for symmetric crypto

The last thing that we tested was the practicality of using HSM in TLS record. This experiment intended to find the cost of using HSM for a symmetric cipher. We decided to use a different setup with the NSS implementation of TLS. This experiment was also published as a part of secure cryptographic protocol execution based on runtime verification research [106].

5.2.8.1 Experiment setup Using OWASP® Zed Attack Proxy (ZAP) [107] and Mozilla Firefox, Alexa top 100 sites (as of 05/06/2019), were accessed, with all traffics collected in plaintext. Also, for each site, time taken to completely load the website was measured. Next, we isolated the Firefox browser's AES encryption by calling directly the AES implementation of its underpinning NSS library. We encrypted the collected traffic (all requests and responses needed to fully load the page, with all scripts etc.) in Galois Counter Mode (GCM), an authenticated encryption scheme supported by TLS 1.3 [108]. The encryption for each website was performed ten times in a row on a Dual-Core Intel Core i5-3317U CPU/6GB RAM machine. Next we did the AES GCM encryption again, but instead of using NSS, we called an AES-GCM implementation on SECube.

Sites	**Load Time**	**Data size**	**NSS**	**SECube**	**Overheads**	**Overheads**
	ms	*bytes*	*ms*	*ms*	*ms*	*%*
www.google.com	1158	1 367 595	6.942	913.599	906.657	78.76
www.youtube.com	1303	810 458	4.135	575.439	571.304	43.98
www.facebook.com	1045	1 511 775	7.717	944.329	936.612	90.29
www.baidu.com	6775	1 265 391	6.778	818.825	812.047	12.00
www.wikipedia.org	698	99 336	0.654	64.916	64.262	9.22
Top_100 average	5205	3 171 550	12.355	1735.728	1723.373	33.19

Table 15: SECube HSM overheads

5.2.8.2 Results Table 15 shows the overheads recorded for the encryption operation registered by SECube compared to Firefox's NSS library executing fully on the end-user machine. Results are shown both separately for the top five websites and the combined measurements for all hundred websites. In each case, the total page load time and the portion taken up by NSS encryption are shown. These values provide the context to analyze the increase in processing times once encryption is offloaded to SECube. While inevitably posing as a bottleneck due to the USB I/O involved, SECube's hardware spec-

ifications manage to keep overheads within a practically acceptable range. An average of 1723 ms may disturb the overall web browsing experience only a little. To keep the overhead as small as possible, we could use hardware acceleration for the encryption with SECube FPGA hardware implementation of AES, but that is not within the focus of this work. Overall, this experiment setup shows that using HSM for record protocol can be deployed at acceptable costs in terms of processing overheads and HSM costs.

5.3 Evaluation of results

The experiments described in the previous section evaluated proposed solution at several levels.

In the first experiment described in section 5.2.1, we have shown the difference in the performance of currently used public-key cryptography (ECC) and post-quantum candidates. We have demonstrated that post-quantum cryptography can match the performance of currently used key exchange algorithms.

We took the next step and evaluated the proposed algorithms in TLS handshake key exchange, which can be found in section 5.2.2. This required TLS design changes that we implemented in the s2n TLS library. We have shown that post-quantum cryptography in TLS handshake is feasible.

Next, we tested our hypothesis that it may be beneficial for lightweight devices to swap roles of server and client in handshake key exchange in section 5.2.3. The experimental results showed that this assumption was correct.

In section 5.2.4, we evaluated the integration of post-quantum algorithms in SEcube (Cortex-M4).

Because we also wanted to add side-channel analysis countermeasures, in 5.2.5 we evaluated the masked and randomized implementation of Kyber.

We were able to run post-quantum KEMs on the SEcube chip, which means we created the first hardware security module that provides a post-quantum level of security (PQcube). Experiments in section 5.2.6 have shown that we can use such HSM in the TLS library as a plug-and-play device, and section 5.2.7 has demonstrated successful integration of PQcube into handshake key exchange.

With this, we can conclude that our work was successful. To show the capabilities of complete isolation of the Root of Trust concept and make our research even more practical, we performed the experiment describing the capabilities of PQcube for symmetric cryptography, which we describe in 5.2.8. This experiment was also published and reviewed in [106].

We also want to point out the real-world usage of our design. For this, we use two separate programs, one for the client-side and the second for the server-side. The client-side has the possibility to plug in PQcube. Both sides run our post-quantum TLS. With PQcube plugged in, all critical operations are performed in HSM, as well as all keys are stored in HSM. After the handshake, application data of different sizes were transported from client to server and back. For this, we used a Dual-Core Intel Core

i5-3317U CPU/6GB RAM computer with Ubuntu 18.04.5 LTS.

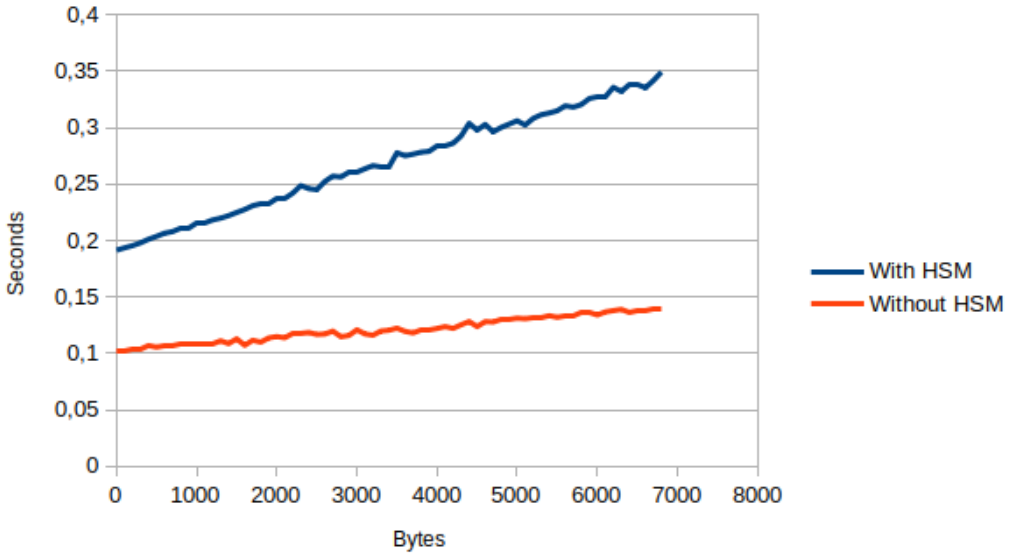

Figure 28: Performance of post-quantum handshake using Saber with and without HSM

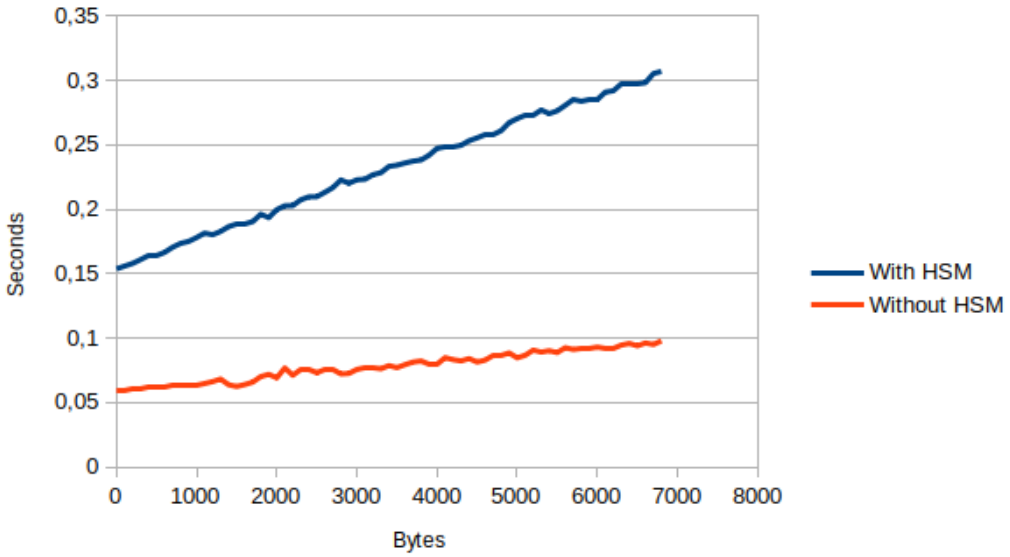

Figure 29: Performance of post-quantum handshake using Kyber with and without HSM

Figures 28 and 29 show that the use of HSM in post-quantum TLS communication introduces delays. The handshake part (with no authentification) needs roughly twice as much time for PQcube, and this delay also grows with the PQcube usage in TLS Record. This experiment aims to show the real-world use of our design, so other factors may interfere.

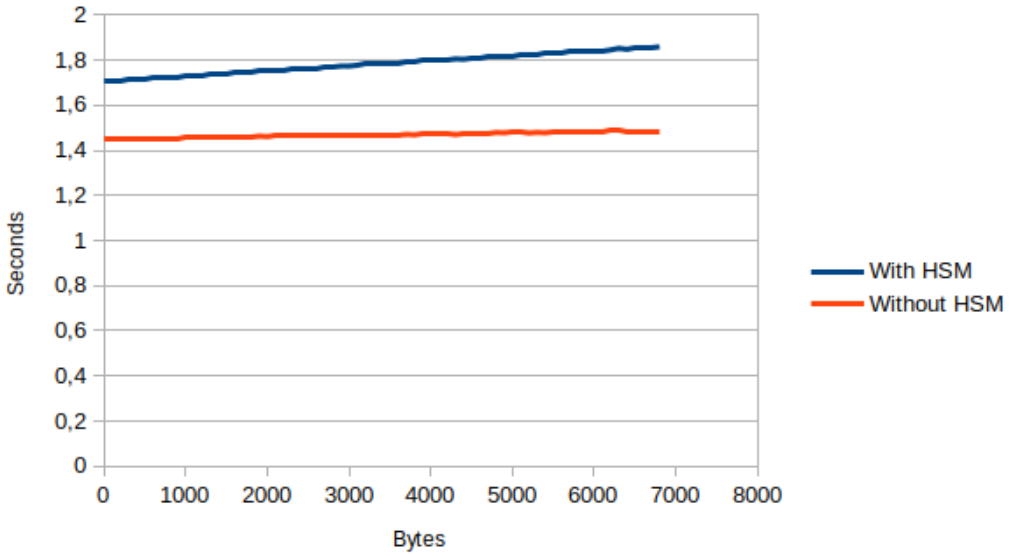

Figure 30: Performance of post-quantum handshake using NTRU with and without HSM

From Figure 30, we can see that for NTRU, the relative delay resulting from the use of PQcube usage is small. Actually, the delay mainly resulting from USB communication is more or less the same as in previous cases (Figures 28 and 29). The difference is in the keypair operation of NTRU, that is significantly slower, slowing the whole TLS communication. This is an example that shows the need for careful choice of post-quantum standard as well as the need to understand post-quantum cryptography to chose the best one for specific purpose;

In applications where keypair operation is not needed, e.g. authentication presented in section 2.7, NTRU would be as fast as other candidates. For key exchange in TLS, however, is not practical, and we advice to use other post quantum algorithms. All in all, our conclusion is that the best possible candidate for key exchange in post/quantum TLS handshake is Kyber. This statement is based on several reasons. As we presented in section 5.2.2, the performance of Kyber is comparable to currently used elliptic-curve cryptography. Kyber is suitable for IoT devices as can be seen in section 5.2.4. Moreover we can use randomisation and masking techniques, to provide higher level of security against side channel analysis (section 5.2.5).

6 Conclusion

In this work, we our goal was to examine and evaluate the possibilities of making TLS key exchange secure in the post-quantum world. We managed to meet the research objectives, which were set in the beginning of this research.

We have confirmed our hypotheses, and created the proof of concept of post-quantum TLS. We used SEcube and implemented Hardware Security Module for post-quantum public key algorithms and used it in our post quantum TLS. We found that the use of such setting is possible, and we consider delays caused still in acceptable range.

Our research in this field has opened even more questions. There are several possible directions of the future research following this work:

- Authentication of both server and client can be added to our design. As we mentioned in section 2.7, this can be done using post-quantum signatures or post-quantum KEM. The comparison of these two approaches under the same conditions would be interesting.

- More research in the protection of post-quantum algorithms from side-channel attacks is required. This is crucial for the integration of post-quantum algorithms into IoT devices.

- Some platforms (as SEcube) provide FPGA. It would be interesting to see hardware implementation of post-quantum algorithms PQcube system.

- Specification of several standards would need to be changed to allow post-quantum public-key cryptography. This includes recognizing post-quantum algorithms in Internet Assigned Numbers Authority (IANA), adapting protocols to allow key encapsulation mechanism API, etc.

The main contribution of the work can be perceived from several points of view: The first point of view is a contribution for Transport Layer Security (TLS) protocol research. Our work brought a post-quantum mechanisms for exchanging keys into the environment of one of the most used communication protocols today. From this point of view, the work pushes the boundaries of the TLS protocol to meet the requirements of security in the post-quantum world.

Another point of view is the view from the development of Hardware Security Modules. We designed and implemented an HSM module for post-quantum key exchange for

TLS Handshake purposes. The module offers not only asymmetric cryptographic algorithms for key exchange, but also key management and their use in symmetric cipher in a secure environment.

Last but not least, we can look at the contribution of the work from the perspective of IoT devices. We evaluated the possibilities of public key post-quantum algorithms for limited devices, we found out which algorithms can be used in such an environment and which cannot. We also proposed a way to modify the TLS protocol so that when using post-quantum asymmetric algorithms, the lightweight client does as little computation as possible.

We conclude that we were able to finish our research with positive results. We showed that the use of post-quantum cryptography in the TLS setting is possible and practical in real-world use. We were able to speed up the handshake for limited devices with the possibility of client and server swapping roles in the key exchange. We designed, implemented, and tested the first post-quantum hardware security module with auspicious results. There are still many steps that need to be taken to bring post-quantum cryptography into practical use, and this work is one of the steps to get closer to the goal.

Resumé

Motivácia

Motiváciou pre náš výskum je výrazný posun vo výskume kvantových počítačov za posledné roky [1], [2], [3], [4]. Kvantový počítač predstavuje riziko pre bezpečnosť komunikácie, z dôvodu, že zabezpečovacie mechanizmy sa spoliehajú na to, že nepoznáme efektívny sposob riešenia matematických problémov, použitých pri šifrovaní. Konkrétne riziko predstavujú dva algoritmy, navrhnuté pre kvantové počítače:

1. **Shorov alrogitmus**, ktorý v roku 1994 publikoval Peter Shor [9], je algoritmus pre kvantové počítače na faktorizáciu celých čisiel v polynomiálnom čase.

2. Lov Grover publikoval algoritmus na prehľadávanie databáz v roku 1996 [10]. Jeho zaujimavou vlastnosťou je, že **Groverov algoritmus** vie nájsť n-bitový kľúč so zložitosťou $\sqrt{2^n}$.

Pri komunikácií využívame dva typy šifier. Symetrické šifry sa používajú na zabezpečenie dôvernosti správy. Na to, aby sa dve komunikujúce strany dohodli na kľúči, sa využíva kryptografia s verejným kľúčom (asymetrická kryptografia).

Groverov algoritmus nedokáže symetrické šifry prelomiť úplne. Avšak kvadratické zrýchlenie útokov hrubou silou si vyžadujú, aby sme prehodnotili, čo považujeme za "bezpečné". Na zabezpečenie dôvernosti dát sa využíva Advanced Encryption Standard (AES) (pokročilý štandard šifrovania). Keď teda zohľadníme Groverov algoritmus, úroveň bezpečnosti *AES-128* je znížená na 64-bitov. To znamená, že nastavenie AES s dĺžkou kľúča 128 bitov alebo menej už nebude bezpečné, a šifra AES bude musieť mať dĺžku kľúča 192 alebo 256 bitov [16].

Najčastejšie využívané algoritmy verejných kľúčov na výmenu kľúčov alebo digitálne podpisy boli prelomené [5], [6]. Na základe problému faktorizácie celých čísel sa šifra RSA jasne stáva obeťou Shorovho algoritmu. Iné využívané šifry sú založené Diffie-Helman probléme, alebo jeho variantoch ECDH (založený na eliptických krivkách nad konečnými poliami). Podľa [12] môže byť Shorov algoritmus použitý aj na výpočet diskrétnych logaritmov a tým pádom aj na zlomenie Diffie-Helman problému. Proos a Zalka [14] dokázali, že prelomiť kryptografiu založenú na eliptických krivkách je jednoduchšie, ako prelomiť RSA.

Problematika práce

Cieľom nášho výskumu v oblasti bezpečnej post-kvantovej kryptografie je návrh protokolu, ktorý by vedel nahradiť súčasne využívaný protokol TLS, ale bol by odolný voči

útokom s použitím kvantového počítača. Taktiež treba vziať do úvahy rastúci podiel IoT zariadení. V práci skúmame, ktoré algoritmy sú pre takéto zariadenia vhodné, a ako ako upraviť samotný TLS Handshake protokol tak, aby to bolo pre tieto zariadenia výhodné. Podstatnou súčastou bezpečnosti systémov je aj operačná bezpečnosť. Preto sa treba zamerať aj na rôzne mechanizmy na zabezpečenie operačnej bezpečnosti, či už s využitím bezpečného exekučného prostredia (TEE), alebo metód na ochranu pred útokmi s využitím postranných kanálov. Ak využívame bezpečné exekučné prostredie pre kritické úkony v post-kvantovom TLS protokole, je treba zohľadniť aj správu kľúčov a dôsledky, ktoré z toho plynú.

Hypotézy a ciele

V dizertačnej práci skúmame a vyhodnocujeme možnosti zabezpečenia výmeny kľúčov v protokole TLS v post-kvantovom svete. Na základe tohto cieľa môžeme vymedziť nasledujúce hypotézy:

- Návrh a implementácia post-kvantového Handshake protokolu kompatibilného s TLS je možná.
- Modul HSM môže byť využitý pre post-kvantové algoritmy verejných kľúčov. Toto riešenie možno aplikovať v prostredí TLS.
- Oneskorenie spôsobené modulom HSM v post-kvantovom prostredí TLS neznemožnuje komunikáciu, ani ju nerobí nepraktickou.

V súvislosti s cieľmi práce môžeme identifikovať niekoľko úloh. Tieto čiastkové ciele nám poslúžia ako základná osnova výskumu.

- Preskúmame detaily protokolu TLS, jeho mechanizmy a štruktúru jeho správ. Identifikujeme súčasti TLS, ktoré môžeme použiť, a tie, ktoré budeme musieť nahradiť.
- Budeme hľadať nové mechanizmy výmeny kľúčov, ktoré by boli odolné proti kvantovým počítačom.
- Zozbierame vhodné implementácie TLS pre naše experimenty a pre praktickú časť výskumu si vyberieme najvhodnejší z nich.
- Navrhneme a implementujeme post-kvantový mechanizmus výmeny kľúčov v kontexte protokolu TLS.

- Pri návrhu nového protokolu podobného TLS vezmeme do úvahy rastúcu popularitu zariadení IoT. Odrazí sa to na voľbe mechanizmu výmeny kľúčov, ako aj na architektúre nového protokolu.
- Nájdeme dostupné metódy, ktorými zabezpečíme operačnú bezpečnosť, a danú metódu implementujeme.
- Naše riešenie otestujeme sadou experimentov, aby sme zistili, či je naše riešenie vhodné na použitie v praxi.

Obsah práce

Naša práca sa delí na niekoľko kapitol. Prvá kapitola predstavuje najnovší protokol TLS. Skúmame v nej detaily protokolu TLS, jeho mechanizmy a štruktúru jeho správ. Okrem toho prezentujeme post-kvantovú kryptografiu, operačnú bezpečnosť a súvisiaci výskum. Opisujeme aj implementácie protokolu TLS. Alternatívne mechanizmy výmeny kľúčov, ktoré sú odolné voči kvantovým počítačom, sme prevzali zo štandardizačného procesu NIST, a vysvetlili sme súvisiace koncepty.

Post-kvantová bezpečnosť v TLS si vyžaduje nový architektonický návrh. Naše návrhy sú predstavené v druhej kapitole, spolu s možnými kandidátmi na algoritmy výmeny kľúčov. Identifikujeme tu časti protokolu TLS, ktoré môžu byť využité, ako aj časti, ktoré treba nahradiť. Berieme pri tom do úvahy aj limitované zariadenia a rastúcu popularitu zariadení IoT. Predstavujeme SEcube a náš koncept post-kvantového HSM, ktorý poskytuje vyššiu úroveň operačnej bezpečnosti. Okrem toho ponúkame možnost využiť implementáciu chránenú proti útokom s využitím postranných kanálov (pre Kyber).

Zvolenú implementáciu TLS s2n sme upravili tak, aby zodpovedala našim potrebám a experimentom. V tretej kapitole spomíname niektoré z problémov, na ktoré sme narazili pri implementácii, aby sme bližšie priblížili artefakt nášho výskumu v zmysle metódy *design research*. Opisujeme aj jednotlivé stavebné diely a zdroje na implementáciu našich návrhov.

Kapitola štyri sa zaoberá testovaním a evaluáciou všetkých komponentov a krokov v post-kvantovej dohode o výmene kľúča v protokole TLS. Experimenty ukazujú úspešné post-kvantové dešifrovanie a úspešnú dohodu výmene kľúča a využitie HSM na strane klienta. Okrem toho poukazujú na úspešné využitie symetrickej kryptografie v protokole TLS Record po post-kvantovej výmene kľúčov.

Výsledky a prínos práce

Hypotézy, ktoré sme si stanovili na začiatku práce, sa potvrdili, a my sme vytvorili dôkaz koncepcie (*proof of concept*) post-kvantového TLS. Využili sme pri tom plat-

formu SEcube a implementovali sme modul Hardware Security Module (HSM) pre post-kvantové algoritmy výmeny kľúčov a aplikovali sme ich v post-kvantovom TLS. Zistili sme, že využitie takýchto nastavení je možné, a oneskorenie, ktoré vzniklo, považujeme za akceptovateľné. Môžeme teda skonštatovať, že výskum sa nám podarilo úspešne dokončiť. Dokázali sme, že využitie post-kvantovej kryptografie v prostredí TLS je možné a použitelné v praxi. Podarilo sa nám zrýchliť proces post-kvantového Handshake pre limitované zariadenia tak, že si klient a server vymenili role pri výmene kľúčov. Navrhli sme, implementovali a otestovali sme prvý post-kvantový modul HSM s mimoriadne sľubnými výsledkami. Na uvedenie post-kvantovej kryptografie do praxe je potrebnej ešte mnoho práce, a táto dizertačná práca je jedným z potrebných krokov, ktoré nás priblížia k cieľu.

Hlavný prínos práce je možné vnímať z niekoľkých uhlov pohľadu: Prvý uhol pohľadu je prínos pre výskum bezpečnosti protokolu na transportnej vrstve (TLS). Práca priniesla post-kvantový mechanizmus na výmenu kľúčov do prostredia jedného z najvyužívanejších komunikačných protokolov súčasnosti. Z tohoto pohľadu práca posúva hranice možností TLS protokolu tak, aby zodpovedali požiadavkám na bezpečnosť v post-kvantovom svete.

Ďalším uhlom pohľadu je pohľad zo strany vývoja bezpečnostných HSM modulov. Navrhli a implementovali sme HSM modul pre post-kvantovú výmenu kľúčov na účely TLS Handshake. Modul ponúka nie len asymetrické kryptografické algoritmy na výmenu kľúčov, ale aj správu kľúčov a ich používanie v symetrickej šifre v bezpečnom prostredí.

V neposlednom rade sa môžeme na prínos práce pozerať z pohladu IoT zariadení. Overili sme možnosti post-kvantových algoritmov verejného kľúča pre limitované zariadenia, zistili sme, ktoré algoritmy je možné v takomto prostredí využiť, a ktoré nie. Tiež sme navrhli spôsob, akým modifikovať TLS protokol tak, aby pri použití post-kvantových asymetrických algoritmov čo najmenej zaťažoval ľahkého klienta.

Náš výskum v tejto oblasti otvoril mnohé ďalšie otázky a možné smerovanie ďalšieho výskumu, nadväzujúceho na túto prácu, vieme zhrnúť do niekoľkých bodov:

- Do nášho návrhu môže byť pridaná autentikácia servera i klienta. Ako sme spomenuli v časti 2.7, tento cieľ možno dosiahnuť pomocou post-kvantových podpisov alebo post-kvantových KEM. Porovnanie týchto dvoch prístupov v rovnakých podmienkach môže priniesť zaujímavé výsledky.

- Je potrebný ďalší výskum v oblasti ochrany post-kvantových algoritmov pred útokmi z postranných kanálov. Je to nevyhnutné pre integráciu post-kvantových algoritmov do zariadení IoT.

- Niektoré platformy (ako SEcube) ponúkajú FGPA. Hardvérová implementácia post-

kvantových algoritmov do systému PQcube by bola určite podnetná.

- Aby bola možná post-kvantová výmena kľúčov, je potrebné zmeniť špecifikácie niektorých štandardov. Patrí sem napríklad rozpoznanie post-kvantových algoritmov v Internet Assigned Numbers Authority (IANA), adaptácia protokolov na umožnenie API mechanizmu zapuzdrenia kľúča (KEM), atď.

Bibliography

1. 40 years of quantum computing. *Nature Reviews Physics*. 2022, vol. 4, no. 1, pp. 1–1. Available from DOI: 10.1038/s42254-021-00410-6.
2. BALL, Philip. First quantum computer to pack 100 qubits enters crowded race. *Nature*. 2021, vol. 599, no. 7886, pp. 542–542. Available from DOI: 10.1038/d41586-021-03476-5.
3. MORZHIN, O. V. and PECHEN', A. N. Maximization of the Uhlmann–Jozsa Fidelity for an Open Two-Level Quantum System with Coherent and Incoherent Controls. *Physics of Particles and Nuclei*. 2020, vol. 51, no. 4, pp. 464–469. Available from DOI: 10.1134/s1063779620040516.
4. ARUTE, Frank et al. Quantum supremacy using a programmable superconducting processor. *Nature*. 2019, vol. 574, no. 7779, pp. 505–510. ISSN 1476-4687. Available from DOI: 10.1038/s41586-019-1666-5.
5. BUCHANAN, William and WOODWARD, Alan. Will quantum computers be the end of public key encryption? *Journal of Cyber Security Technology*. 2017, vol. 1, no. 1, pp. 1–22. Available from DOI: 10.1080/23742917.2016.1226650.
6. MAVROEIDIS, Vasileios, VISHI, Kamer, ZYCH, Mateusz D and JØSANG, Audun. The impact of quantum computing on present cryptography. *arXiv preprint arXiv: 1804.00200*. 2018.
7. ALAGIC, Gorjan et al. *Status Report on the First Round of the NIST Post-Quantum Cryptography Standardization Process*. NIST Interagency/Internal Report (NISTIR), National Institute of Standards and Technology, Gaithersburg, MD, 2019. Available from DOI: https://doi.org/10.6028/NIST.IR.8240.
8. CORPORATION, IBM. *TLS protocol overview* [online]. 2021 [visited on 2021-09-13]. Available from: https://www.ibm.com/docs/en/sdk-java-technology/7.1?topic=provider-tls-protocol-overview.
9. SHOR, P. W. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *SIAM Journal on Computing*. 1995, no. 5, p. 1484.
10. GROVER, L.K. A fast quantum mechanical algorithm for database search, Proceedings. *28th Annual ACM Symposium on the Theory of Computing*. 1996, p. 212.

11. ARUNACHALAM, Srinivasan and WOLF, Ronald de. *Optimizing the Number of Gates in Quantum Search*. arXiv, 2015. Available from DOI: 10.48550/ARXIV.1512.07550.

12. EKERÅ, Martin and HÅSTAD, Johan. Quantum Algorithms for Computing Short Discrete Logarithms and Factoring RSA Integers. In: LANGE, Tanja and TAKAGI, Tsuyoshi (eds.). *Post-Quantum Cryptography*. Cham: Springer International Publishing, 2017, pp. 347–363. ISBN 978-3-319-59879-6.

13. MAURER, Ueli M. and WOLF, Stefan. The Relationship Between Breaking the Diffie–Hellman Protocol and Computing Discrete Logarithms. *SIAM Journal on Computing*. 1999, vol. 28, no. 5, pp. 1689–1721. Available from DOI: 10.1137/S0097539796302749.

14. PROOS, John and ZALKA, Christof. Shor's Discrete Logarithm Quantum Algorithmfor Elliptic Curves. 2003, vol. vol. 3, pp. 317–344.

15. HERON, Simon. Advanced encryption standard (AES). *Network Security*. 2009, vol. 2009, no. 12, pp. 8–12.

16. GRASSL, Markus, LANGENBERG, Brandon, ROETTELER, Martin and STEINWANDT, Rainer. Applying Grover's algorithm to AES: quantum resource estimates. In: *Post-Quantum Cryptography*. 2016, pp. 29–43.

17. HIDARY, Jack D. A Brief History of Quantum Computing. In: *Quantum Computing: An Applied Approach*. Cham: Springer International Publishing, 2019, pp. 11–16. ISBN 978-3-030-23922-0. Available from DOI: 10.1007/978-3-030-23922-0_2.

18. BRASSARD, Gilles, HØYER, Peter and TAPP, Alain. Quantum cryptanalysis of hash and claw-free functions. In: LUCCHESI, Cláudio L. and MOURA, Arnaldo V. (eds.). *LATIN'98: Theoretical Informatics*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 163–169. ISBN 978-3-540-69715-2.

19. NIST. *Guidelines for the Selection, Configuration, and Use of Transport Layer Security (TLS) Implementations: NIST SP 800-52 Rev. 2* [online]. 2019 [visited on 2021-09-13]. Available from: https://csrc.nist.gov/news/2019/nist-publishes-sp-800-52-revision-2.

20. DOWLING, Benjamin, FISCHLIN, Marc, GÜNTHER, Felix and STEBILA, Douglas. A cryptographic analysis of the TLS 1.3 handshake protocol. *Journal of Cryptology*. 2021, vol. 34, no. 4, pp. 1–69.

21. MORIARTY, Kathleen, KALISKI, Burt, JONSSON, Jakob and RUSCH, Andreas. *PKCS #1: RSA Cryptography Specifications Version 2.2* [RFC 8017]. RFC Editor, 2016. Request for Comments, no. 8017. Available from DOI: 10.17487/RFC8017.

22. PORNIN, Thomas. *Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)* [RFC 6979]. RFC Editor, 2013. Request for Comments. Available from DOI: 10.17487/RFC6979.

23. JOSEFSSON, Simon and LIUSVAARA, Ilari. *Edwards-Curve Digital Signature Algorithm (EdDSA)* [RFC 8032]. RFC Editor, 2017. Request for Comments, no. 8032. Available from DOI: 10.17487/RFC8032.

24. MCGREW, David. *An Interface and Algorithms for Authenticated Encryption* [RFC 5116]. RFC Editor, 2008. Request for Comments, no. 5116. Available from DOI: 10.17487/RFC5116.

25. BARNES, Richard and BHARGAVAN, Karthikeyan. *Hybrid Public Key Encryption*. Internet Engineering Task Force, [n.d.]. Available also from: https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-hpke-01. Work in Progress.

26. CENTER, Windows Dev. *TLS Record Protocol*. Microsoft - Online Source, 2017, [n.d.].

27. RESCORLA, Eric. *Diffie-Hellman Key Agreement Method* [RFC 2631]. RFC Editor, 1999. Request for Comments, no. 2631. Available from DOI: 10.17487/RFC2631.

28. KRAWCZYK, Dr. Hugo and ERONEN, Pasi. *HMAC-based Extract-and-Expand Key Derivation Function (HKDF)* [RFC 5869]. RFC Editor, 2010. Request for Comments, no. 5869. Available from DOI: 10.17487/RFC5869.

29. JACKSON, Brian. *An Overview of TLS 1.3 - Faster and More Secure*. Kinsta.com, Internet Source, december 2016, [n.d.].

30. NIR, Yoav and LANGLEY, Adam. *ChaCha20 and Poly1305 for IETF Protocols* [RFC 7539]. RFC Editor, 2015. Request for Comments, no. 7539. Available from DOI: 10.17487/RFC7539.

31. THOMAS, Stephen. *SSL and TLS Essentials: Securing the Web*. New York: John Wiley & Sons, Inc, 2000, [n.d.].

32. KRAWCZYK, Dr. Hugo, BELLARE, Mihir and CANETTI, Ran. *HMAC: Keyed-Hashing for Message Authentication* [RFC 2104]. RFC Editor, 1997. Request for Comments, no. 2104. Available from DOI: 10.17487/RFC2104.

33. JAYAPAL, Cynthia, SULTANA, Parveen, SAROJA, M. N. and SENTHIL, J. Security Protocols for IoT. In: 2019, pp. 1–28. ISBN 978-3-030-01565-7. Available from DOI: 10.1007/978-3-030-01566-4_1.

34. CHACKO, Smilty and JOB, Mr. Deepu. Security mechanisms and Vulnerabilities in LPWAN. *IOP Conference Series: Materials Science and Engineering*. 2018, vol. 396, p. 012027. Available from DOI: 10.1088/1757-899x/396/1/012027.

35. SASTRY, Naveen and WAGNER, David. Security Considerations for IEEE 802.15.4 Networks. 2004, vol. 2004. Available also from: https://people.eecs.berkeley.edu/~daw/papers/15.4-wise04.pdf.

36. LIUSVAARA, Ilari. *CFRG Elliptic Curve Diffie-Hellman (ECDH) and Signatures in JSON Object Signing and Encryption (JOSE)* [RFC 8037]. RFC Editor, 2017. Request for Comments, no. 8037. Available from DOI: 10.17487/RFC8037.

37. NARAYANAN, R., JAYASHREE, S., PHILIPS, N. D., SARANYA, A. M., PRATHIBA, S. B. and RAJA, G. TLS Cipher Suite: Secure Communication of 6LoWPAN Devices. In: *2019 11th International Conference on Advanced Computing (ICoAC)*. 2019, pp. 197–203. Available from DOI: 10.1109/ICoAC48765.2019.246840.

38. NIST. *The Future Is Now: Spreading the Word About Post-Quantum Cryptograph* [online]. 2016 [visited on 2021-09-30]. Available from: https://www.nist.gov/blogs/taking-measure/future-now-spreading-word-about-post-quantum-cryptography.

39. GRUBBS Paul; Maram Varun ;Paterson, Kenneth. *Anonymous, Robust Post-Quantum Public Key Encryption* [online]. 2021 [visited on 2021-09-30]. Available from: https://eprint.iacr.org/2021/708.

40. KARABULUT Emre; Aysu, Aydin. *Falcon Down: Breaking Falcon Post-Quantum Signature Scheme through Side-Channel Attacks* [online]. 2021 [visited on 2021-09-30]. Available from: https://eprint.iacr.org/2021/772.

41. NIST. *Submission Requirements and Evaluation Criteria for the Post-Quantum Cryptography Standardization Process* [online]. 2016 [visited on 2021-09-30]. Available from: https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf.

42. PUB, FIPS. Secure hash standard (shs). *Fips pub*. 2012, vol. 180, no. 4.

43. THE OPENSSL PROJECT. *OpenSSL: The Open Source toolkit for SSL/TLS*. 2003. www.openssl.org.

44. OPENBSD PROJECT. *LibreSSL*. 2021. www.libressl.org.

45. GOOGLE. *BoringSSL*. 2021. boringssl.googlesource.com/boringssl.

46. MOZILLA. *Network Security Services*. 2021. developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS.

47. AMAZON WEB SERVICES. *s2n*. 2021. https://github.com/aws/s2n-tls.

48. TRUSTED FIRMWARE. *Mbed TLS*. 2021. https://tls.mbed.org/.

49. WOLFSSL. *wolfSSL embedded TLS library*. 2021. https://www.wolfssl.com/.

50. POULIN, Chris. *What to do to protect against heartbleed openssl vulnerability* [online]. 2014 [visited on 2020-09-30]. Available from: https://www.yubico.com/.

51. BERNSTEIN, Daniel J. Cache-timing attacks on AES. 2005.

52. VELLA, Mark, COLOMBO, Christian, ABELA, Robert and ŠPAČEK, Peter. RV-TEE: secure cryptographic protocol execution based on runtime verification. *Journal of Computer Virology and Hacking Techniques*. 2021, vol. 17, no. 3, pp. 229–248. ISSN 2263-8733. Available from DOI: 10.1007/s11416-021-00391-1.

53. BHATTACHARYA, Anish. CYBER SECURITY ATTACK TYPES – ACTIVE AND PASSIVE ATTACKS. 2021. Available also from: www.encryptionconsulting.com/active-and-passive-attacks/.

54. BHUNIA, Swarup and TEHRANIPOOR, Mark. Chapter 10 - Physical Attacks and Countermeasures. In: BHUNIA, Swarup and TEHRANIPOOR, Mark (eds.). *Hardware Security*. Morgan Kaufmann, 2019, pp. 245–290. ISBN 978-0-12-812477-2. Available from DOI: https://doi.org/10.1016/B978-0-12-812477-2.00015-0.

55. SABT, Mohamed, ACHEMLAL, Mohammed and BOUABDALLAH, Abdelmadjid. Trusted Execution Environment: What It is, and What It is Not. In: *2015 IEEE Trustcom/BigDataSE/ISPA*. 2015, vol. 1, pp. 57–64. Available from DOI: 10.1109/Trustcom.2015.357.

56. LEVIN, Timothy, NGUYEN, Thuy, IRVINE, Cynthia and MCEVILLEY, Michael. Separation Kernel Protection Profile Revisited: Choices and Rationale. *Fourth Annual Layered Assurance Workshop (LAW 2010)*. 2010, vol. Electronic Archive.

57. TCG. *Tpm 1.2 main specification* [online]. 2011 [visited on 2021-09-30]. Available from: https://trustedcomputinggroup.org/resource/tpm-main-specification/.

58. AAS, Josh. *Preparing to Issue 200 Million Certificates in 24 Hours* [online]. 2021 [visited on 2021-10-30]. Available from: `https://letsencrypt.org/2021/02/10/200m-certs-24hrs.html`.

59. DOUGLAS STEBILA, Michele Mosca. *Open Quantum Safe project* [online]. 2017 [visited on 2021-10-30]. Available from: `https://openquantumsafe.org`.

60. ŠPAČEK, Peter. *DDP - Zverejnená diplomová práca Implementation of McEliece cryptosystem into TLS*. 2017. MA thesis. Slovak University of Technology.

61. BERNSTEIN, Daniel J., BRUMLEY, Billy Bob, CHEN, Ming-Shing and TUVERI, Nicola. OpenSSLNTRU: Faster post-quantum TLS key exchange. *IACR Cryptol. ePrint Arch.* 2021, vol. 2021, p. 826.

62. CROCKETT, Eric, PAQUIN, Christian and STEBILA, Douglas. Prototyping post-quantum and hybrid key exchange and authentication in TLS and SSH. *IACR Cryptol. ePrint Arch.* 2019, vol. 2019, p. 858.

63. GEORGE, Tasopoulos, LI, Jinhui, FOURNARIS, Apostolos P, ZHAO, Raymond K, SAKZAD, Amin and STEINFELD, Ron. Performance Evaluation of Post-Quantum TLS 1.3 on Embedded Systems. *Cryptology ePrint Archive*. 2021.

64. SCHWABE, Peter, STEBILA, Douglas and WIGGERS, Thom. *Post-quantum TLS without handshake signatures* [Cryptology ePrint Archive, Report 2020/534]. 2020. `https://ia.cr/2020/534`.

65. KANNWISCHER, Matthias J., RIJNEVELD, Joost, SCHWABE, Peter and STOFFELEN, Ko. *pqm4: Testing and Benchmarking NIST PQC on ARM Cortex-M4* [Cryptology ePrint Archive, Report 2019/844]. 2019. `https://ia.cr/2019/844`.

66. ALAGIC, Gorjan et al. Status report on the second round of the NIST post-quantum cryptography standardization process. *US Department of Commerce, NIST*. 2020.

67. CHOU, Tung et al. Classic McEliece: conservative code-based cryptography 10 October 2020. 2020.

68. AVANZI, Roberto et al. CRYSTALS-Kyber algorithm specifications and supporting documentation. [N.d.].

69. CHEN, Cong, DANBA, Oussama, HOFFSTEIN, Jeffrey, HÜLSING, Andreas, RIJNEVELD, Joost, SCHANCK, John M, SCHWABE, Peter, WHYTE, William and ZHANG, Zhenfei. NTRU, Algorithm Specifications And Supporting Documentation. 2019.

70. VERCAUTEREN, Ir Frederik. SABER: Mod-LWR based KEM (Round 3 Submission). [N.d.].

71. LANGE, Tanja. *Selected Areas in Cryptology* [https://hyperelliptic.org/tanja /teaching/pqcrypto21/]. [N.d.]. Accessed: 2021-10-30.

72. KARMAKAR, Angshuman, MERA, Jose Maria Bermudo, ROY, Sujoy Sinha and VERBAUWHEDE, Ingrid. Saber on ARM CCA-secure module lattice-based key encapsulation on ARM. *Cryptology ePrint Archive*. 2018.

73. *Password Storage Cheat Sheet* [OWASP Cheat Sheet Series.]. [N.d.]. Accessed: 2021-09-30.

74. CREMERS, Cas, HORVAT, Marko, SCOTT, Sam and MERWE, Thyla van der. Automated Analysis and Verification of TLS 1.3: 0-RTT, Resumption and Delayed Authentication. In: *2016 IEEE Symposium on Security and Privacy (SP)*. 2016, pp. 470–485. Available from DOI: 10.1109/SP.2016.35.

75. BELLARE, Mihir, CANETTI, Ran and KRAWCZYK, Hugo. A Modular Approach to the Design and Analysis of Authentication and Key Exchange Protocols (Extended Abstract). In: *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*. Dallas, Texas, USA: Association for Computing Machinery, 1998, pp. 419–428. STOC '98. ISBN 0897919629. Available from DOI: 10.1145/276698.276854.

76. PERRIN, Trevor and MARLINSPIKE, Moxie. The double ratchet algorithm. *GitHub wiki*. 2016.

77. PERRIN, Trevor. The Noise protocol framework. *PowerPoint Presentation*. 2018.

78. MCKEEN, Frank, ALEXANDROVICH, Ilya, ANATI, Ittai, CASPI, Dror, JOHNSON, Simon, LESLIE-HURD, Rebekah and ROZAS, Carlos. Intel® software guard extensions (intel® sgx) support for dynamic memory management inside an enclave. In: *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*. 2016, pp. 1–9.

79. JEE, Kangkook, PORTOKALIDIS, Georgios, KEMERLIS, Vasileios P, GHOSH, Soumyadeep, AUGUST, David I and KEROMYTIS, Angelos D. A General Approach for Efficiently Accelerating Software-based Dynamic Data Flow Tracking on Commodity Hardware. In: *NDSS*. 2012.

80. PINTO, Sandro and SANTOS, Nuno. Demystifying arm trustzone: A comprehensive survey. *ACM Computing Surveys (CSUR)*. 2019, vol. 51, no. 6, pp. 1–36.

81. WOJTCZUK, Rafal and RUTKOWSKA, Joanna. Attacking intel trusted execution technology. *Black Hat DC*. 2009, vol. 2009, pp. 1–6.

82. SEABORN, Mark and DULLIEN, Thomas. Exploiting the DRAM rowhammer bug to gain kernel privileges. *Black Hat*. 2015, vol. 15, p. 71.

83. SABT, Mohamed and TRAORÉ, Jacques. Breaking into the keystore: A practical forgery attack against Android keystore. In: *European Symposium on Research in Computer Security*. 2016, pp. 531–548.

84. KOCHER, Paul et al. Spectre attacks: Exploiting speculative execution. In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019, pp. 1–19.

85. VARRIALE, Antonio, DI NATALE, Giorgio, PRINETTO, Paolo, STEFFEN, Bernhard and MARGARIA, Tiziana. Secube (tm): an open security platform-general approach and strategies. In: *Proceedings of the International Conference on Security and Management (SAM)*. 2016, p. 131.

86. *Open Source SDK and Projects* [`http://secube.blu5group.com`]. [N.d.]. Accessed: 2021-09-30.

87. GELUSO, Joe. CRC16-CCITT. 2003. Available also from: `http://web.archive.org/web/20071229021252/http://www.joegeluso.com/software/articles/ccitt.htm`.

88. KALISKI, Burt. PKCS# 5: Password-based cryptography specification version 2.0. 2000.

89. *SEcube Open Source Platform - Introduction* [`https://raw.githubusercontent.com/SEcube-Project/SEcube-SDK/release-1.5.1/wiki/wiki_rel_011.pdf`]. [N.d.]. Accessed: 2021-09-30.

90. BOLLO, Matteo, CARELLI, Alberto, DI CARLO, Stefano and PRINETTO, Paolo. Side-channel analysis of SEcube™ platform. In: 2017, pp. 1–5. Available from DOI: `10.1109/EWDTS.2017.8110067`.

91. MCGREW, David and VIEGA, John. The Galois/counter mode of operation (GCM). *submission to NIST Modes of Operation Process*. 2004, vol. 20, pp. 0278–0070.

92. CARLET, Claude, HASAN, M Anwar and SARASWAT, Vishal. *Security, Privacy, and Applied Cryptography Engineering*. Springer, 2016.

93. PÖPPELMANN, Thomas, ODER, Tobias and GÜNEYSU, Tim. High-Performance Ideal Lattice-Based Cryptography on 8-Bit ATxmega Microcontrollers. In: LAUTER, Kristin and RODRÍGUEZ-HENRÍQUEZ, Francisco (eds.). *Progress in Cryptology – LATINCRYPT 2015*. Cham: Springer International Publishing, 2015, pp. 346–365. ISBN 978-3-319-22174-8.

94. RAVI, Prasanna, POUSSIER, Romain, BHASIN, Shivam and CHATTOPADHYAY, Anupam. On Configurable SCA Countermeasures Against Single Trace Attacks for the NTT. In: BATINA, Lejla, PICEK, Stjepan and MONDAL, Mainack (eds.). *Security, Privacy, and Applied Cryptography Engineering*. Cham: Springer International Publishing, 2020, pp. 123–146. ISBN 978-3-030-66626-2.

95. BERNSTEIN, Daniel J. et al. Classic McEliece:conservative code-based cryptography. 2017.

96. DWORKIN, Morris J. *Sp 800-38d. recommendation for block cipher modes of operation: Galois/counter mode (gcm) and gmac*. National Institute of Standards & Technology, 2007.

97. *wpa_supplicant* [https://wiki.archlinux.org/title/wpa$_s$*upplicant*]. [N.d.]. Accessed: 2021-09-30.

98. KANNWISCHER, MJ, RIJNEVELD, J, SCHWABE, P, STEBILA, D and WIGGERS, T. *The PQClean Project, August 2020*. [N.d.].

99. *LibOpenCM3* [https://libopencm3.org/]. [N.d.]. Accessed: 2021-09-30.

100. WANG, Wen, SZEFER, Jakub and NIEDERHAGEN, Ruben. FPGA-Based Niederreiter Cryptosystem Using Binary Goppa Codes. In: LANGE, Tanja and STEINWANDT, Rainer (eds.). *Post-Quantum Cryptography*. Cham: Springer International Publishing, 2018, pp. 77–98. ISBN 978-3-319-79063-3.

101. BOTROS, Leon, KANNWISCHER, Matthias J. and SCHWABE, Peter. *Memory-Efficient High-Speed Implementation of Kyber on Cortex-M4* [Cryptology ePrint Archive, Report 2019/489]. 2019. `https://eprint.iacr.org/2019/489`.

102. KANNWISCHER, Matthias J., RIJNEVELD, Joost and SCHWABE, Peter. *Applied Cryptography and Network Security – ACNS 2019*. Vol. 11464, Faster multiplication in $\mathbb{Z}_{2^m}[x]$ on Cortex-M4 to speed up NIST PQC candidates. Ed. by DENG, Robert H., GAUTHIER, Valérie, OCHOA, Martín and YUNG, Moti. Springer-Verlag Berlin Heidelberg, 2019. Lecture Notes in Computer Science. Available also from: `https://eprint.iacr.org/2018/1018`.

103. RAVI, Prasanna. *Configurable SCA Countermeasures for the NTT Against Single Trace Attacks* [online]. 2020 [visited on 2021-05-13]. Available from: https://github.com/PRASANNA-RAVI/Configurable_SCA_Countermeasures_for_NTT.

104. TURAN, Meltem Sönmez, BARKER, Elaine, BURR, William and CHEN, Lily. Recommendation for password-based key derivation. *NIST special publication*. 2010, vol. 800, p. 132.

105. ALBRECHT, Martin R. *cpucycles: counting CPU cycles*. 2015. Available also from: http://web.mit.edu/sage/export/libm4ri-0.0.20080521/testsuite/cpucycles-20060326/cpucycles.html.

106. VELLA, Mark, COLOMBO, Christian, ABELA, Robert and ŠPAČEK, Peter. RV-TEE: secure cryptographic protocol execution based on runtime verification. *Journal of Computer Virology and Hacking Techniques*. 2021, vol. 17. Available from DOI: 10.1007/s11416-021-00391-1.

107. BENNETTS, Simon. Owasp zed attack proxy. *AppSec USA*. 2013.

108. RESCORLA, Eric et al. RFC 8446: The Transport Layer Security (TLS) protocol version 1.3. *Internet Engineering Task Force (IETF)*. 2018.

Appendix

A GitHub Repository

Our systems consist of two code-bases, the first one consists of PQcube HSM, and the second represents post-quantum modification of s2n:

1. https://github.com/PeterSpacek/pq-cube-firmware
2. https://github.com/PeterSpacek/s2n

A.1 PQcube repository structure

/drivers

· Folder contains Common Microcontroller Software Interface and stm32f4xx Hardware Abstraction Level driver required for embedded development. We did not changed this folder.

/middlewares/st/stm32_usb_device_library

· Folder contains USB communication library for STM32 devices. We did not change this folder.

/secube

· Folder with source files and sample aplications

/secube-on-pc

· Folder contains SEcube features tests. We did not change this folder.

/secube-tests

· Folder contains L1 and L0 tests. We did not change this folder.

/secube-wrapper

· Folder contains SEcube python wrapper. We did not change this folder.

/src

· Folder contains source and header files containing the logic. We modified these files to allow post-quantum cryptography support

/sw4stm32/SEcubeDevBoard

· Folder contains generated files, project files and other build-related files. We did not change this folder.

ws/secubedevboard

· Project directory for STM32cubeIDE.

/Application/src/Device/pq-crypto

· Folder contains implementations of post-quantum algorithms.

/Debug

· Folder for debug binaries.

/Release

· Folder for release binaries.

A.2 PQ s2n repository structure

api

· Folder containing s2n public API. We modified it to support PQ3TLS and PQLIMTLS

bin

· Folder containing example files. We did not modify it.

cmake

· cmake configuration. We did not modify it.

codebuild

· AWS CodeBuild configuration. We did not modify it.

coverage

· Folder used for html and fuzz test files. We did not modify it.

crypto

· Source files for cryptography. We added and modified files to support pq3.

docs

· s2n documentation. We did not modify it.

error

· s2n error codes. We did not modify it.

lib

· Folder containing makefile describing library build. We added our files.

libcrypto-build

· Folder for locally built libcrypto: OpenSSL, LibreSSL or BoringSSL

pq-crypto

· Folder containing post-quantum cryptography implementations.

scram

· SCRAM algorithm. It is a legacy of amazon experiments. We did not modify it.

stuffer

· Memory management for s2n. We did not modify it.

tests

· Folder containing tests. We modified files in this folder, and added new unit tests.

tls

· TLS processes implementation. Most of our modifications and additions are here.

utils

· Essential services for s2n implementation. We did not modify it.

B Run Relevant Unit Tests

How to program pq3 under Windows (SEcube and ST-LINK/V2 required):

1. Clone pq-cube-firmware repository (Appendix A).
2. Start STM32CubeIDE and set its workspace to the "ws" directory
3. Plug in SEcube via USB
4. Connect ST-LINK/V2 debugger and programmer to computer and to SEcube
5. Build PQcube
6. Flash the binary

How to run relevant s2n tests in Ubuntu:

1. Clone our s2n GitHub repository (Appendix A).
2. # Go into s2n s2n folder.
   cd s2n
3. # Pick an "env" line from the codebuild/codebuild.config file and run it, in this case choose the openssl-1.1.1 with GCC 9 build
   S2N_LIBCRYPTO=openssl-1.1.1 BUILD_S2N=true TESTS=integration GCC_VERSION=9
4. source codebuild/bin/s2n_setup_env.sh
5. codebuild/bin/s2n_install_test_dependencies.sh
6. sudo -E prlimit –pid "$$" –memlock=unlimited:unlimited;
7. Run unit test, you are interested in:

UNIT_TESTS=s2n_pqlimtls_client_test make
UNIT_TESTS=s2n_pqlimtls_server_test make
UNIT_TESTS=s2n_pqlimtls_handshake_test make
UNIT_TESTS=s2n_self_talk_pqlimtls_test make
UNIT_TESTS=s2n_pq3_state_machine_handshake_test make

Chooses automatically between pq3tls and pqtls depending on se3 presence:
UNIT_TESTS=s2n_pq3tls_client_test make
UNIT_TESTS=s2n_pq3tls_server_test make

Requires pq3:
UNIT_TESTS=s2n_pq3_aead_aes_test make
UNIT_TESTS=s2n_self_talk_pq3tls_test make
UNIT_TESTS=s2n_pq3_benchmark_test make
Or you can run unit tests as cmake project in e.g. Clion where it is easy also to debug it.