EXPLORING POST-QUANTUM CRYPTOGRAPHIC SCHEMES FOR TLS IN 5G NB-IOT: FEASIBILITY AND RECOMMENDATIONS

by

Kadir Sabanci

A Thesis Submitted to the Faculty of the Graduate School,
Marquette University,
in Partial Fulfillment of the Requirements for
the Degree of Master of Science

Milwaukee, Wisconsin

May 2023

ABSTRACT
EXPLORING POST-QUANTUM CRYPTOGRAPHIC SCHEMES FOR TLS IN 5G NB-IOT: FEASIBILITY AND RECOMMENDATIONS

Kadir Sabanci

Marquette University, 2023

Narrowband Internet of Things (NB-IoT) is a wireless communication technology that enables a wide range of applications, from smart cities to industrial automation. As a part of the 5G extension, NB-IoT promises to connect billions of devices with low-power and low-cost requirements. However, with the advent of quantum computers, the incoming NB-IoT era is already under threat due to conventional cryptographic algorithms that might be adapted to secure devices in NB-IoT being susceptible to be broken soon. In this context, we investigate the feasibility of using post-quantum key exchange and signature algorithms for securing NB-IoT applications. We develop a realistic ns-3 environment to represent the characteristics of NB-IoT networks and analyze the usage of post-quantum algorithms to secure communication. In this context, we investigate the feasibility of using post-quantum key exchange and signature algorithms for securing NB-IoT applications.

ACKNOWLEDGEMENTS

Kadir Sabanci

I would like to express my deepest gratitude to my thesis advisor Dr. Mumin Cebe, for his support, guidance, and encouragement throughout preparing this thesis. His insightful feedback and constructive criticism have been invaluable in shaping this work. I would also like to extend my sincere appreciation to the thesis committee members Dr. Thomas Kaczmarek and Dr. Debbie Perouli for putting their valuable time reviewing this thesis. Additionally, I would like to thank my family and friends for their unconditional support, constant encouragement, and understanding.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

Abbreviation	Meaning
IoT	Internet of Things
IIOT	Industrial IoT
MTC	Machine-type Communications
eNB	evolved NodeB
LPWAN	Low-power wide-area network
LTE	Long Term Evolution
3GPP	3rd Generation Partnership Project
PRB	Physical Resource Block
NB-IoT	Narrow Band IoT
PSM	Power Saving Mode
e-DRX	extended Discontinuous Reception
MCL	Maximum Coupling Loss
e-MTC	enhanced Machine Type Communication
LoRa	Long Range
EDT	Early Data Transmission
TLS	Transport Layer Security
16QAM	16 Quadrature Amplitude Modulation
IETF	Internet Engineering Task Force
TCP	Transmission Control Protocol
DNS	Domain Name System
HTTP	Hypertext Transfer Protocol
HTTPS	HTTP Secure
SMTPS	Simple Mail Transfer Protocol Secure
MAC	Message Authentication Code
CA	Certificate Authority
ECC	Elliptic Curve Cryptography

ECDHE	Elliptic Curve Diffie-Hellman Ephemeral
DHE	Diffie-Hellman Ephemeral
PQC	Post-Quantum Cryptography
NIST	National Institute of Standards and Technology
KEM	Key Exchange Mechanism
SIS	Short Integer Solution
LWE	Learning With Errors
DSA	Digital Signature Algorithms
ECDSA	Elliptic Curve Digital Signature Algorithm
ns-3	Network Simulator 3
PGW	Packet Gateway
SGW	Signaling Gateway
EPC	Evolved Packet Core
RTT	Round Trip Times
OQS	Open Quantum Safe
PKI	Public Key Infrastructure
PFS	Perfect Forward Secrecy

CHAPTER 1: INTRODUCTION

1.1 Motivation

The Internet of Things (IoT) technology holds great potential for connecting billions of intelligent devices situated in diverse locations, from urban centers to rural areas and even space [1]. Consequently, a range of IoT applications are emerging in transportation, healthcare, automation, and smart city industries, collectively known as Industrial IoT (IIoT) applications [2]. These applications are expected to rely on 5G/6G technologies that are specifically designed to support communication among IoT devices. For instance, Narrow Band - IoT (NB-IoT) technology in 5G provides extended coverage with enhanced signal penetration and longer battery life [3]. However, the expansion in coverage also means the connection of more IoT devices, which can put pressure on IIoT applications due to increased communication traffic. Furthermore, the increase in traffic also raises concerns about security management, as security protocols can create additional overhead on traffic [4].

Efforts are underway to manage the security keys and certificates required for millions of IoT devices, as reported in several research studies, including [5] [6] [7] [8]. Alongside these efforts, researchers are also exploring the complete overhaul of the security infrastructure, with communication protocols designed to be quantum resistant. However, these schemes come at the cost of increased communication and computation overhead. Recent developments in quantum computing have accelerated the expectation that the first generation of industrial quantum computers will soon be available, raising concerns about the security implications for IIoT systems [9]. On one hand, with the

increased data traffic there will be little bandwidth available for security management operations like authentication and key exchange, which are necessary for secure communications. On the other hand, migrating to post-quantum security protocols may add additional overhead, potentially causing delays that could compromise the application requirements.

1.2 Objectives

Post-quantum cryptography (PQC) and NB-IoT protocols come from two different venues of the technology landscape, while closely affecting each other from the performance point of view. In the literature surveys made, we found out that, this relation is not discussed thoroughly. Especially, regarding which specific post-quantum algorithms can be used with NB-IoT without experiencing performance degradation.

The main objective of this thesis is to conduct a real-world performance analysis of TLS handshake with post-quantum key exchange and signature algorithms on NB-IoT networks. Our analysis has two fundamental parts. First, we aim to analyze the post-quantum key exchange overhead. This overhead is originated from increased “key_share” data sizes of the new post-quantum key exchange protocol Crystals-Kyber. Second, we investigate post-quantum signature overhead. Post-quantum signature algorithms come with significantly bigger sizes compared to algorithms in use today such as RSA and ECDSA.

Through the analysis conducted, we want to shed light on whether these newly introduced post-quantum cryptographic algorithms can align with emerging IoT technology, particularly NB-IoT. We believe that this work can serve as a benchmark

guiding industry practitioners who want to transition to post-quantum algorithms and helping them identify which algorithms would be best for their interest.

1.3 Thesis Organization

In Chapter 2 we provide background information about topics related to the thesis, such as IoT technology, NB-IoT and working principles, TLS Protocol version 1.3, and finally Post-Quantum Cryptography.

Chapter 3 presents a literature review of studies focusing on the performance benchmarks of PQC algorithms, particularly those discussing implementation and network-based performance.

In Chapter 4 we present our approach to measuring the performance characteristics of PQC algorithms on an NB-IoT network. This chapter includes detailed information about the steps followed for software installation, key generation, simulation software, and simulation applications employed for benchmarks.

In Chapter 5 we evaluate the findings from Chapter 4. We discuss the results as well as the contributing factors.

Chapter 6 is dedicated to conclusions and remarks around Post-Quantum Cryptography algorithm usage with NB-IoT technology and possible future studies related to the thesis' context.

CHAPTER 2: BACKGROUND

2.1 Internet of Things (IoT)

Internet of Things (IoT) technology provides connectivity to everyday devices with the prospect of integrating them into the digital world. Household appliances, parking meters, healthcare devices, and industrial sensors are some examples of countless possible IoT application areas [10]. According to industry reports, the number of IoT devices worldwide is expected to exceed 43 billion at the end of 2023 [11]. With this, the simultaneous connectivity of a sheer number of terminals poses a technical challenge to the telecom industry and academia.

Low Power Wide Area Networks (LPWAN) are a class of connectivity methods, targeting IoT terminals that generally have different connectivity requirements than computers and smartphones. These requirements are characterized by high number of terminals, small payloads, and low power consumption [12]. On the other hand, connectivity technologies must be able to address different traffic patterns, such as periodic and burst traffic, as well as low and high mobility of the terminals depending on the use case [13]. In the last decade, the industry has come up with different technologies to answer the connectivity demands of different devices categories, these LPWAN technologies can be deployed on unlicensed spectrum, such as SigFox and LoRa, or licensed spectrum with standards such as e-MTC and NB-IoT [14].

2.1 NB-IoT

NB-IoT is an LPWAN technology designed to efficiently utilize the LTE spectrum to connect a high number of IoT terminals simultaneously. It was initially standardized in 2016 with 3GPP Release 13 document as an extension of the LTE, but with some

important changes, such as reduced bandwidth, enhanced coverage, and improved power-saving capabilities [15].

Moreover, NB-IoT comes with configurable specification elements such as terminal density, number of subframe repetitions, and power-saving timers. A mobile operator can choose proper settings according to its customer profile and terrain requirements to provide the best coverage and battery lifetime.

Another important aspect of NB-IoT design is support for simpler and cheaper half-duplex terminals, which is a crucial step towards realizing the billions of connected devices target [16]. In addition, NB-IoT applications are expected to tolerate latencies up to 10 seconds, as the devices using the same PRB might create bottlenecks at the random-access procedure phase [17].

NB-IoT has undergone further improvements with the subsequent 3GPP releases. Release 15 introduced Early Data Transmission (EDT) which decreases control signaling overhead for single message transmissions and therefore better latency, power consumption, and spectrum usage; this version also introduced Time Division Duplex (TDD) and coverage improvements [18]. Release 16 presented a key change, enabling NB-IoT to communicate with 5G core, which in essence is a 4G radio protocol [19]. Finally, Release 17 added 16-QAM modulation support to provide higher data rates [20].

With all the mentioned improvements, NB-IoT stands as a promising wireless access technology for the 5G era, which can help connect massive numbers of non-mission critical and delay-tolerant IoT devices. According to market research, the NB-IoT chipset market is expected to exceed $22 billion by 2030 [21], and NB-IoT as a connection technology is expected to consist of 43% of all LPWAN connectivity landscape by 2025

[22]. The rest of this section will cover technical specifications and working principles of NB-IoT technology.

2.1.1 NB-IoT Architecture

NB-IoT can be deployed using a single LTE Physical Resource Block (PRB) of 180 KHz for uplink and downlink channels. This PRB is also shared among 12 subcarriers, allocating 15KHz of bandwidth to each user equipment, compared to LTE's standard 20MHz bandwidth [15]. As NB-IoT uplink and downlink channels use different Transport Block sizes and different scheduling configurations, downlink bitrate is significantly lower than the uplink [23]. For existing LTE infrastructure, NB-IoT migration can be made with software updates, without new hardware or spectrum requirements. NB-IoT base stations (eNB) can serve both standard LTE users and NB-IoT terminals at the same time by assigning different PRBs to each device category [24].

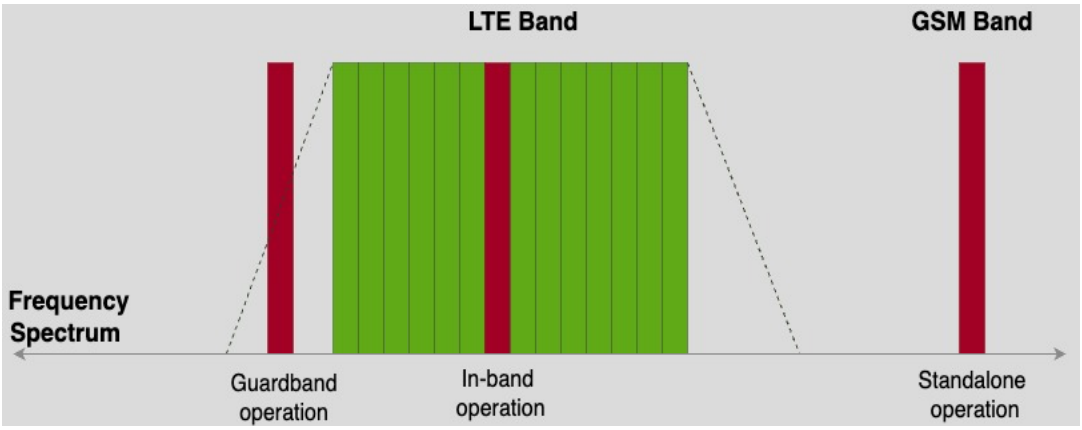

Figure 1 – NB-IoT Deployment Modes

NB-IoT standard supports different operation or deployment modes within existing LTE and GSM spectrum. This approach provides more flexibility to the service

providers as some operators have already abandoned 2G / 3G networks [25] and therefore can easily assign free of use bands to NB-IoT usage. These deployment modes are,

in-band mode: in a single resource block of the LTE band,

guard band mode: in a vacant guard spectrum between different LTE bands,

standalone mode: in a band allocated from the GSM spectrum [26].

2.1.2 Coverage Enhancement

NB-IoT offers three coverage enhancement modes: CE0, CE1, and CE2. CE0 has the same Maximum Coupling Loss (MCL) as GSM, which is 144 dB, whereas CE1 and CE2 provide 154 dB and 164 dB MCL respectively [27]. Increased MCL is achieved through repetitions. Nb-IoT can use up to 2048 repetitions in the uplink and 128 in the downlink, which is left to the network operator's preference to satisfy the best coverage while keeping connected devices battery efficient [28]. In addition to these technical capabilities in the physical layer, NB-IoT also allows the integration of half-duplex NB-IoT terminals, which significantly helps to reduce the overall cost of IoT device deployments since these terminals are cheaper than full-duplex terminals. This advantage is particularly significant for IoT applications that involve infrequent, low-bandwidth communication, where the latency introduced by half-duplex communication may not be a major concern.

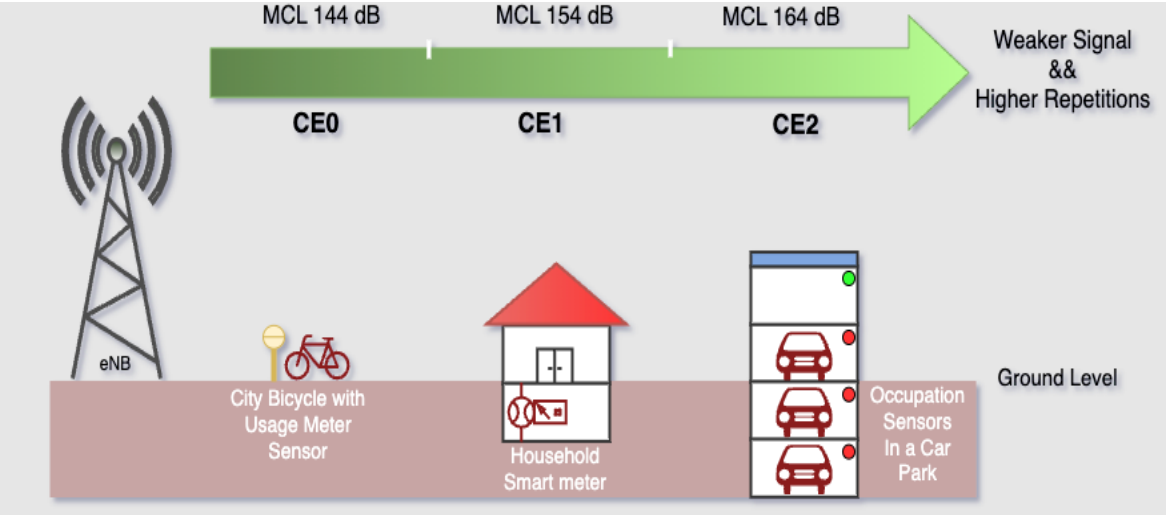

Figure 2 - NB-IoT Coverage Enhancement Modes

2.1.3 NB-IoT Power Saving Modes

NB-IoT also introduces advanced power management techniques through Power Saving Mode (PSM) and Extended Discontinuous Reception (eDRX), which can significantly improve the battery life of IoT devices. PSM, introduced in 3GPP version 12, enables IoT devices to turn off network functionality while remaining registered to the network, reducing power consumption without requiring re-attachment to the NB-IoT [29]. The eDRX mode, on the other hand, allows IoT devices to wake up periodically to check for incoming messages while remaining passive during uplink transmission, thereby conserving power even further. These power management techniques extend the capabilities of NB-IoT even more, considering the long-term operability of IoT devices in remote or hard-to-reach locations where battery replacement or recharging may be difficult or impossible.

2.2 Transport Layer Security (TLS)

TLS is an IETF cryptographic standard that provides security to Transmission Control Protocol (TCP) messages. Today, TLS is the de-facto protocol for securing

HTTP traffic and is used along with some other protocols like email and DNS [30]. TLS protocol stands between the Application and Transport Layers of the Internet Protocol Stack and provides encryption to the application data. When an application protocol is secured through TLS, it is renamed to imply added security, such as HTTP vs HTTPS, the last S letter indicating the added security. This naming pattern is also used with some other protocols like SMTP where the TLS-secured version is renamed as SMTPS [31].

TLS is designed for creating ad-hoc secure connections between the vast number of devices on the internet. To achieve this, TLS provides a couple of important security features:

Confidentiality: Confidentiality aims to hide the real communication data from anyone but the initiating parties. To achieve this, Application Layer data is encrypted using symmetrical keys. Encrypted data looks like random bits to outsiders and can only be recovered by using the symmetrical key [32].

Integrity: Integrity is about keeping messages secure against tampering. TLS uses Message Authentication Code (MAC) to ensure no changes are made during transmission. Sending party appends a MAC digest to the message, which is recalculated and checked against the appended MAC at the receiving party. If the MAC digests are not identical the message is discarded and receiving party asks for a TCP retransmission [33].

Authentication: In a TLS session, communicating parties need to make sure that, they are communicating with the intended entity. TLS conducts authentication using public key cryptography through X.509 certificates [34]. Certificates contain public keys along with some other meta information about the subject and issuer of the certificate. The

issuer which can be a Certificate Authority (CA) or Intermediate Certificate Authority (ICA), digitally signs the certificate data and adds this signature to the certificate. Digital signatures are asymmetric cryptographic primitives which use public/private key pairs. While the private key is used for signing data by the owner of the public/private key, the public key can be used by anyone to verify whether the signature is created by the key owner. Although TLS supports mutual authentication, certificate-based authentication is generally used by the servers, whereas clients use simpler methods like password authentication [35].

TLS is a protocol stack and is consisted of five different underlying protocols. These protocols are the Handshake Protocol, Record Protocol, Change Cipher Spec Protocol, Heartbeat Protocol, and Alert Protocol. Within the scope of this thesis, we will be reviewing the first two which are the most important and relevant ones.

2.2.1 TLS 1.3 Handshake Protocol

TLS is categorized as a hybrid protocol as it uses symmetric encryption together with asymmetric encryption. Asymmetric encryption is used to decide security parameters, authentication of communicating parties through digital certificates and session key establishment [36].

TLS 1.3 was standardized by IETF in 2018 and is the most recent version of the protocol. TLS handshake protocol is redesigned to enhance security by starting data encryption earlier and decreasing latency by reducing the number of round-trip-times (RTT) of TLS 1.2 from 3-RTT to 1.5-RTT, with this version. This newly introduced key exchange mechanism is one of the most notable changes in TLS 1.3. The message flows in Figure 3 represent a server-only authentication handshake in Public Key Infrastructure

(PKI) settings with RSA or ECC signatures. TLS 1.3 supports Elliptic Curve Diffie-Hellman Ephemeral (ECDHE) and conventional Diffie-Hellman Ephemeral (DHE) key exchange schemes [37], however, in this work we use only ECDHE.

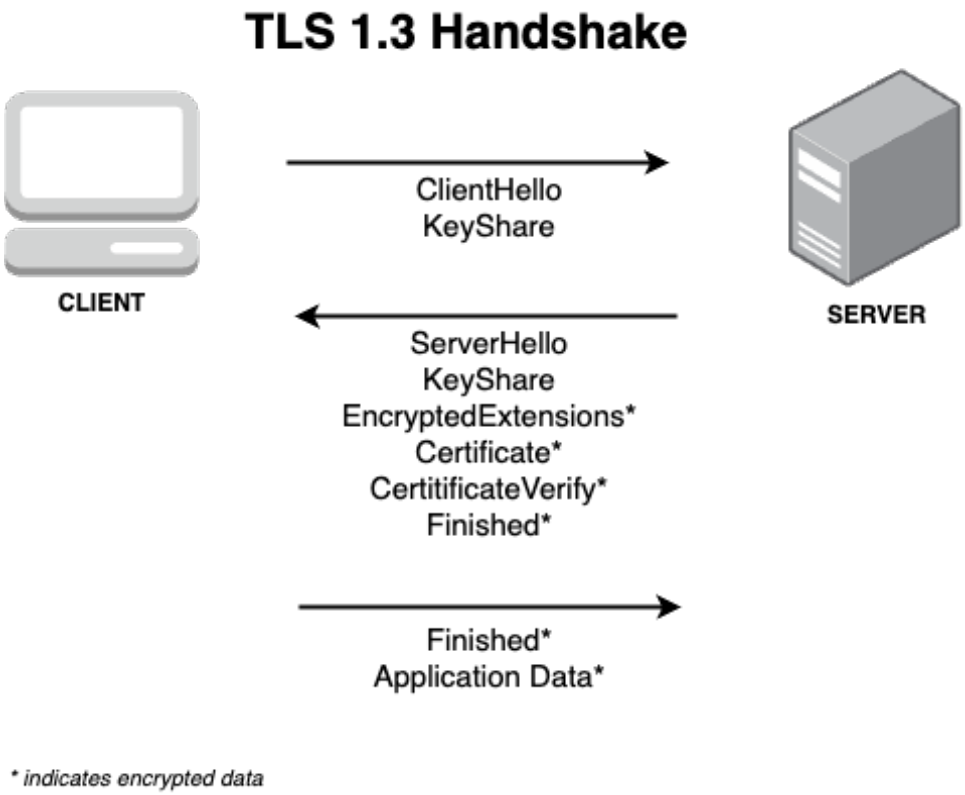

Figure 3 - TLS 1.3 1.5-RTT Handshake

TLS handshake protocol implements two important security functions: key exchange and authentication. The client initiates the process by sending a "ClientHello" message, which includes the cryptographic parameters and a nonce. The client also sends its ECDHE "KeyShare". The server responds with a "ServerHello" message, containing its selected cryptographic parameters and the server nonce. The server also sends its own ECDHE "KeyShare" and related "Extensions". For authentication, the server includes its "Certificate" (certificate chain) and a "CertificateVerify" message which contains a

signature generated by hashing previous handshake messages and signing them with the server’s long-term private key. The server's "Finished" message is a Message Authentication Code (MAC) which is also computed over the entire handshake, to ensure the integrity of the exchanged messages.

In some cases, the client might also need to authenticate itself using a digital certificate which is called mutual authentication. The client sends its "Certificate" (certificate chain) and "CertificateVerify" (traffic signature) messages together with the “Finished” message and its “ApplicationData”. The client also completes its handshake with a "Finished" message which similarly contains a MAC of the entire handshake, providing both integrity and key confirmation. If client authentication is not required, such as in daily web traffic, the "Certificate" and "CertificateVerify" parts of the handshake message are sent empty.

2.2.2 Elliptic Curve Diffie-Hellman (ECDHE) Key Exchange

Elliptic Curve Diffie-Hellman (ECDHE) and conventional Diffie-Hellman Ephemeral (DHE) protocols rely on the same mathematically hard problem, the discrete logarithm, while employing different arithmetic methods for key calculation.

Elliptic Curve Cryptography (ECC), is categorized as a generalized discrete logarithm problem, therefore can be applied to Diffie-Hellman key exchange. Typically, mathematical operations on an elliptic curve are point operations and results are another point on the curve, see Figure 4. For cryptographic usage, on the other hand, calculations are made over finite prime fields (Galois Field) which means modulo operation is applied to calculation [38].

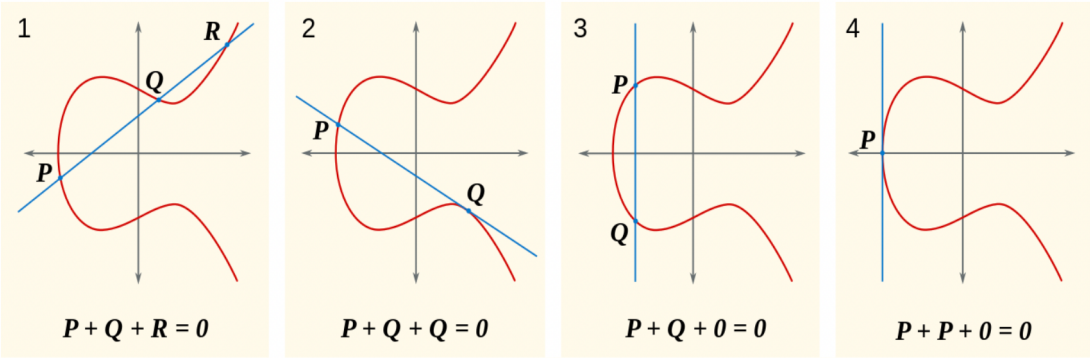

Figure 4 – Elliptic Curve Point Operations *[39]*

ECDHE needs some initial domain parameters for cryptographic calculations, such as:

a) A large prime number p for modulo operation,

b) an elliptic curve with a well understood security properties such as prime256v1 [40] which is denoted by,

$$y^2 = x^3 + a \,.\, x + b \bmod p$$

c) and a primitive element or starting point on the elliptic curve, $P = (x_p, y_p)$

Given these domain parameters, both A and B start with choosing private keys a and b which are large integers. Public keys will be calculated by point multiplications of P with itself private key times on the chosen elliptic curve. In the next step, both parties share their public keys which will be used to generate a shared secret, which is also a point on the curve [41].

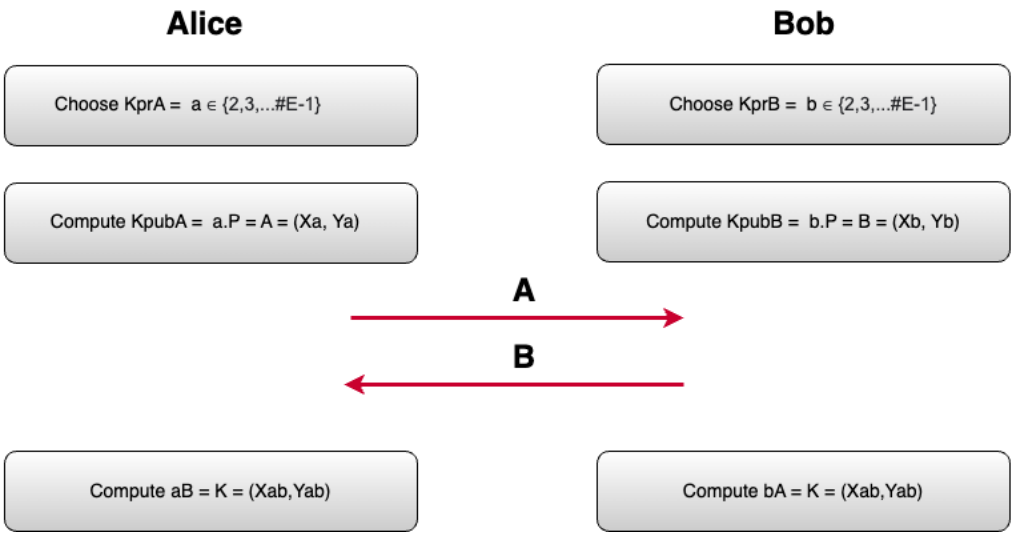

Figure 5 - ECDHE Key Exchange Protocol

2.2.3 TLS Record Protocol

When the TLS handshake is completed with agreed-upon encryption and MAC keys, both parties send “ChangeCipherSpec” messages to indicate starting symmetrically encrypted transmission to each other. In this phase, data coming from the Application Layer is secured and handed over to the TCP protocol for network transmission.

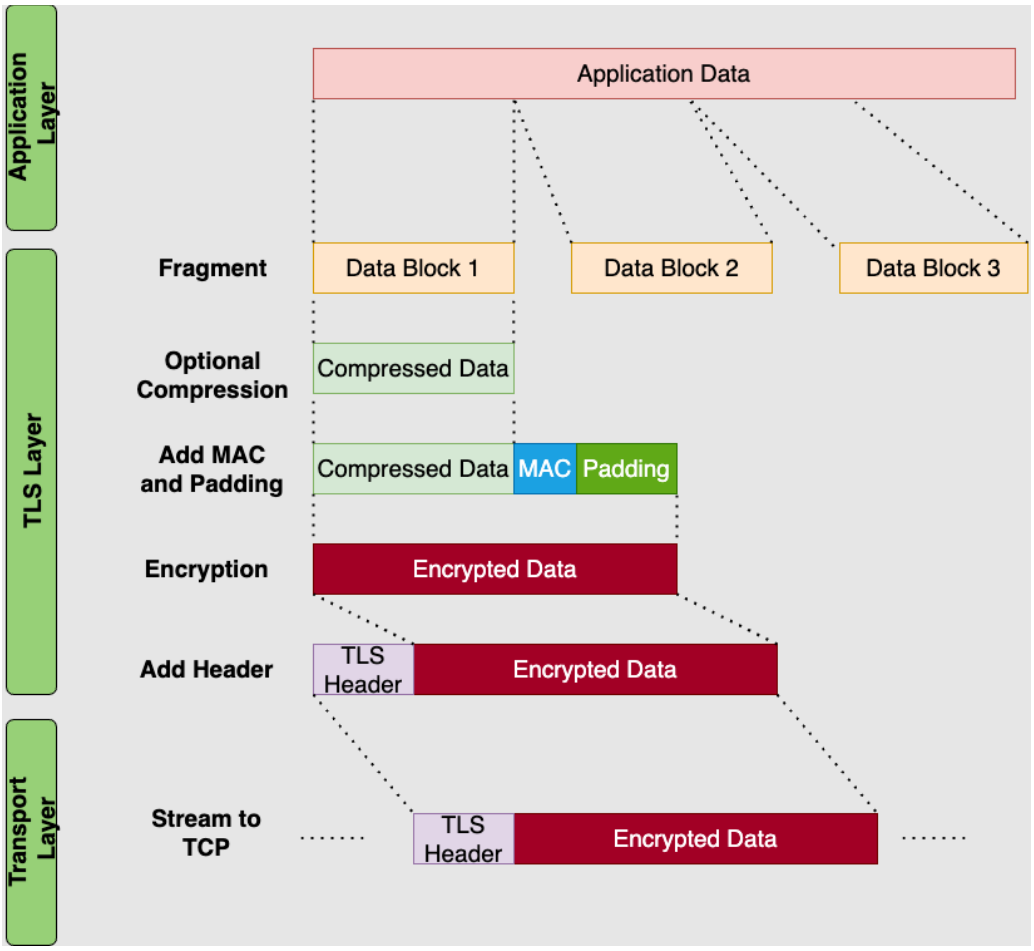

Figure 6 - TLS Record Protocol

TLS record protocol splits application data into data blocks and optionally applies compression. After these steps, MAC and Padding data are concatenated into data blocks. MAC field is used for assuring data integrity while padding is required to avoid some weaknesses of block ciphers [42]. Finally, TLS Header is added to the encrypted data, which is then streamed through the TCP protocol.

2.3 Post-Quantum Cryptography

In recent years, the encryption landscape is going through some drastic changes because of considerable development efforts put on quantum computers. Quantum computers are theoretically shown to be capable of solving a set of mathematically hard

problems that lay the foundations of modern cryptography. In 1997, Peter W. Shor published his paper which proposes a quantum algorithm to solve prime factorization and discrete logarithm problems in polynomial time [43], which ultimately means that today's public key cryptography namely RSA or ECC will be broken once the quantum computing is available at scale.

NIST Post-Quantum Cryptography (PQC) standardization process started in 2017 for selecting quantum-resistant public-key cryptographic algorithms to replace the current vulnerable algorithms, such as signature and key exchange or KEM which we use interchangeably in this thesis. Initial submissions to the NIST's process came from these five different families of encryption [44]:

2.3.1 Code-Based

This family is based on error-correcting codes and their usability in encryption. When a random linear code is used for encryption, there is no efficient decoder, whereas if a good code is used, there exists a decoder that can be used for decryption [45]. Classic McEliece (KEM), BIKE (KEM), and HQC (KEM) algorithms from this family are listed in the Round-3 qualifiers.

2.3.2 Isogeny-Based

This family is based on isogenies between two elliptic curves, which can be represented by polynomials where the addition operation in one curve would yield the same result when computed with the corresponding images on the second curve [46]. SIKE (KEM) is listed in Round-3 as an alternate candidate but was acknowledged as being insecure by the submitters, and there are no candidates left for this category in the standardization process [47].

2.3.3 Hash-Based

The idea is to employ hash functions to generate one-time or many-time signatures such as Lamport or Merkle signatures which are believed to be safe against quantum attacks [48]. Sphincs+ is a Round-3 alternate candidate which uses many few-time signature (FTS) keys from which a random key pair is chosen to sign a message [49]. Public and private key sizes for Sphincs+ remain small but the signature size is relatively higher than most of the other Round-3 qualifiers [50].

2.3.4 Multivariate

These are the schemes based on multivariate polynomials over finite fields. Decryption is based on the hidden structure of the polynomial so that inverting the polynomial is possible [51]. Round-3 finalist Rainbow (DSA) has small signatures but comes with large key pairs [50].

2.3.5 Lattice-Based

This cryptosystem family has lattice-based hard security proofs, and the idea is creating problems that are difficult to solve even for quantum computers [52]. Some of the hard problems are Learning-with-Errors (LWE), Learning-with-Rounding (LWR), and Short Integer Solution (SIS) [53].

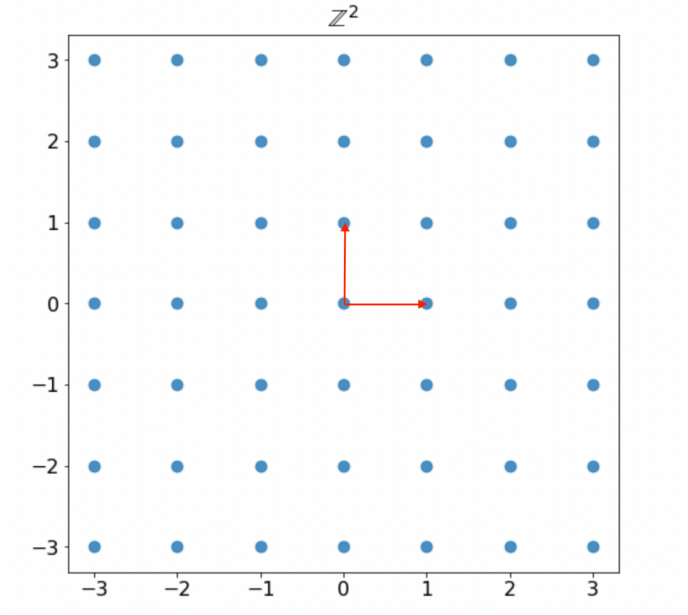

Figure 7 - A two-dimensional lattice generated using base matrix B = ([0,1], [1,0]) [54]

All lattice-based cryptography operations are defined on integers where:

- A basis is a set of vectors,

$$B = (b1, \ldots, bm)$$

- Lattice is defined by scalar and arithmetic operations calculated over basis vectors:

$$L(B) = B.Z^m = \sum_{i=1}^{m} Z \,.\, bi$$

Using mentioned properties, it is possible to draw an entire lattice which is a set of points, which Figure 7 represents an example [55].

2.3.5.1 Short Integer Solution (SIS)

Short Integer Solution (SIS) problem is an average case hard problem based on lattice cryptography [56]. SIS shows that on n-dimensional vectors modulo q (Z_q^n), it is hard to find a short integer $z \in Z^m$ such that:

$$(\dots A \dots)\,(z) = 0 \in Z_q^n$$

The solution is categorized as a lattice problem, as

$$A = (a_1, \dots, a_m) \in Z_q^n$$

defines a q-ary lattice.

2.3.5.2 Learning With Errors (LWE)

Learning With Errors (LWE) uses error terms added to inner dot products of matrices and is based on hardness of finding a vector $s \in Z_q^n$ from

$$A = (a_1, \dots, a_m) \quad \text{and} \quad b^t = s^t A + e^t$$

where e is an error vector [55].

2.3.5.3 Crystals-Kyber Key Exchange

Here we want to give details about how LWE based Crystals-Kyber (Kyber) key exchange mechanism works:

Given, **q** is a prime, **m** is a message, **s** is a random vector, **A** is a matrix of random polynomials and **e** is a random error vector, where all coefficients of the vectors are small integers. The definition is as follows [57] [58] :

Modulus: **q,** Private key: **s**, Public key: **t = (A.s + e) mod q** and **A**

a) Encryption:

- Choose a randomizer polynomial **r**, and random error polynomials $\mathbf{e_1}$ and $\mathbf{e_2}$.
- Encode message **m** as a binary polynomial to get $\mathbf{m_b}$.
- Scale $\mathbf{m_b}$ by multiplying with $\lceil \boldsymbol{q/2} \rceil$ to get $\mathbf{m_s}$.
- Encrypt $\mathbf{m_s}$ using public key **(A, t)** and calculate two ciphertext values **u** and **v** where:

$$u = A.r + e_1 \;\;, \; v = t.r + e_2 + m_s$$

b) Decryption

- Calculate $m_n = v - \mathrm{s}.\mathrm{u}$, which is equal to $m_n = m_s + e.r + e_2 - s.e_1$
- Descale $\mathbf{m_n}$ by dividing it by $\lceil q/2 \rceil$ to eliminate small error terms with small coefficients and get $\mathbf{m_b}$ again.
- Recover original message **m** from $\mathbf{m_b}$, by converting binary polynomial to a base 10 number.

Lattice-based PQC schemes are high performing in key generation, encapsulation, and decapsulation and lattice-based signatures have reasonable public key and signature sizes compared to other PQC schemes [59]. Crystals-Kyber (KEM), SABER (KEM), NTRU-Prime (KEM), Frodo (KEM), Falcon (DSA), and Crystals-Dilithium (DSA) are lattice-based schemes listed in NIST Round-3 either as a finalist or alternate candidates.

Signature Algorithm and Parameter	NIST Classical Security Level	Signature Size (bytes)	Public Key Size (bytes)
RSA-2048	<1	256	256
RSA-3072	1	384	384
ECDSA-prime256v1	1	64	64
SPHINCS-SHA256-128s-simple	1	7856	32
SPHINCS-SHA256-192s-simple	3	16224	48
SPHINCS-SHA256-256s-simple	5	29792	64
Falcon-512	1	690	897
Falcon-1024	5	1330	1793
Dilithium2	2	2420	1312
Dilithium3	3	3293	1952
Dilithium5	5	4595	2592

Table 1 - Signature and public key size comparison of traditional and post-quantum signature algorithms [50] [60]

On July 2022 NIST announced the first selected algorithms from Round-3, as Crystals-Kyber for Key Exchange Mechanism (KEM); Crystals-Dilithium, Falcon, and Sphincs+ for Digital Signature Algorithms (DSA). For Round-4 there are no DSA candidates left for consideration at the time of writing this thesis and one of the most important aspects of the selected DSA's is their large signature sizes compared to algorithms in use today, see Table 1 [61].

CHAPTER 3: LITERATURE REVIEW

Currently, there are limited number of studies that evaluate the performance characteristics of post-quantum algorithms. The literature on this topic can be divided into two categories: network performance related studies and implementation performance related studies.

3.1 Post-Quantum Network Performance Related Studies

The first notable study is conducted by [62]**,** which compares the performance of some NIST post-quantum signature candidates selected by authors, on a broadband connection between a cloud server and a client**.** Similarly, [63] evaluates post-quantum schemes on TLS by assessing the overhead of hybrid schemes that use post-quantum and conventional key exchange at the same time during TLS handshake. The purpose of the work is to measure the overhead of hybrid schemes while establishing a TLS connection on a regular network. Finally, in 2022 [64] conducts a benchmarking study of post-quantum schemes for QUIC protocol, which is a lightweight version of TLS protocol and evaluates the overhead of Dilithium-2 and Dilithium-3 certificate chains in a cloud environment.

3.2 Post-Quantum Implementation Performance Related Studies

In 2021, [65] modifies the TLS 1.2 to evaluate the effects of the post-quantum key exchange algorithm Kyber with the post-quantum signature scheme Sphincs+ on a resource-constrained device based on TPM, with a focus on measuring the computational overhead of the two post-quantum algorithms in TLS. Finally, [66] makes a comparative analysis of time and energy consumption characteristics of NIST PQC Round 2 candidates, with optimized C codes on a PC.

To the best of our knowledge, our study is the first comprehensive work that evaluates the effects of different post-quantum schemes in TLS on an NB-IoT network. In addition, we assess the overhead of post-quantum key exchange (KEM) and post-quantum signatures by explicitly comparing the former with conventional key exchange ECDHE and comparing the latter with conventional signature scheme ECDSA and RSA. By doing so, we shed light on performance differences of different post-quantum adaptation scenarios, such as separately evaluating the effect of post-quantum KEM and signatures to TLS. This provides valuable insights into the feasibility of adopting post-quantum schemes in various scenarios and highlights the areas where further considerations are needed according to the NB-IoT setup and the end-device density.

CHAPTER 4: APPROACH

NB-IoT is a relatively new technology that is increasingly gaining interest from the consumer market. One of the key promises of this technology is scalability for the vast number of IoT terminals through decreased spectrum usage. In this sense, we aim to test the network performance limits of NB-IoT against our main concern, the overhead brought about by the post-quantum era signature and key exchange algorithms.

To evaluate the performance of post-quantum algorithms on NB-IoT network, a network simulator is employed for cost and practicality reasons. Moreover, a simulator environment enables infrastructure-level debugging, giving the user freedom to conduct stress tests to see the real limits of the technology. To achieve these goals, a simulator that can reflect the real-world working conditions of an NB-IoT infrastructure is employed. The search for a reliable simulation environment led us to ns-3 Network Simulator [67] which is a well-established and reputable network simulation tool.

ns-3 simulator provides support for LTE networks through its LENA project. Current version is 5G-LENA which also supports 5G networks [68]. 5G-LENA code supports all network element types that a real IP-based LTE network requires, such as eNB, Packet Gateway (PGW), and Signaling Gateway (SGW) which is referred to as Evolved Packet Core (EPC). This realistic setup provides packet-level inspection capability of test scenarios, which we frequently used to augment the approach and update the simulation code and parameters.

Although NB-IoT is an extension of the LTE, it has some major differences compared to standard implementation in terms of Radio Resource Control (RRC) and Physical Layers. NB-IoT simulator code used in the experiments (LENA-NB) [69] is implemented

independently on top of the 5G-LENA project, and a paper regarding the implementation was published in 2022 at the WNS3 workshop [70].

The simulation application, which is basically a TCP client-server application, mimics the data exchange process of a TLS 1.3 handshake. The exchanged data traffic between the client and server application in the simulation is derived from the traffic generated from an *s_client* and *s_server* application of OpenSSL fork [71] of the Open Quantum Safe (OQS) project.

In the sequel, we give details regarding the ns-3 and OpenSSL program setup, signature and key generation steps, as well as data size extraction operations.

4.1 liboqs and OQS-OpenSSL

The OQS project is an initiative dedicated to open-source implementation of post-quantum algorithms. Starting from 2019, development efforts were conducted around the NIST Post-Quantum Cryptography standardization project and its candidate algorithms. Currently, the project libraries support all winner and candidate algorithms of Round 3.

The OQS development process is concentrated on two main development paths. The first one is *liboqs*, a C library implementation of post-quantum algorithms, the second one is implementation of industry standard protocols and applications such as OpenSSL, SSH, and X.509 [72]. In the following sections, we prefer to call the OQS fork of the OpenSSL as OQS-OpenSSL to avoid confusion with the standard OpenSSL program. For detailed setup instructions please refer to Appendix B-1 section.

4.2 Preparation of Post-Quantum Keys and Certificates

In a Public Key Infrastructure (PKI) setting, root certificate authorities (CA) act as a root of trust for other entities. Root CAs can self-sign their own certificates along with

Intermediate CA certificates and server certificates [73]. Intermediate CAs on the other hand, issue certificates for other Intermediate CAs or servers. When a client verifies a server certificate, it needs root and Intermediate CA certificates (public keys) to assure the authenticity of the signature chain, see Figure 8.

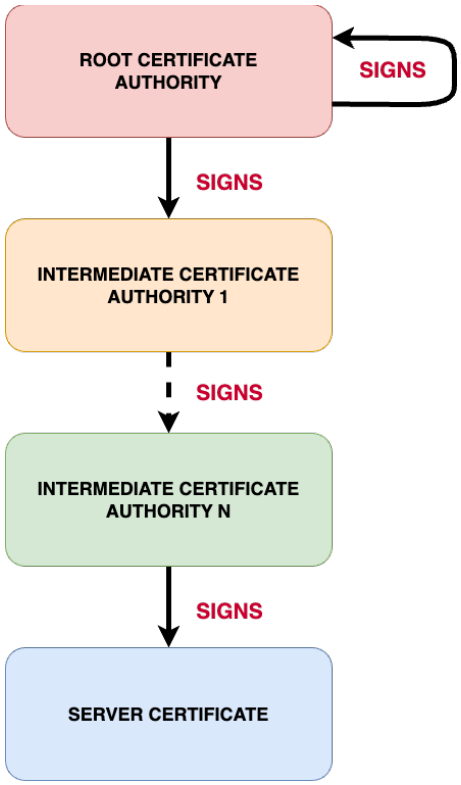

Figure 8 – Certificate Signing Chain

We used the OQS-OpenSSL program to generate X.509 certificate chains for NIST Round 3 selected signature algorithms: Falcon, Crystals-Dilithium, and Sphincs+ as well as RSA and ECDSA. Figure 9 shows a server certificate signed using the Falcon algorithm with a ring degree of 512, which we decode using Certlogik's CSR Decoder [74]. The error message ("Unable to load Public Key") in the decoded text indicates that the website still does not recognize Falcon certificates and therefore cannot show the

public key. Instead, it shows serial number 1.3.9999.3.1 in the output, which is the assigned object identifier for this algorithm [75]. A sample certificate chain creation process is provided in Appendix B-2 section.

```
    Certificate:
    Data:
        Version: 1 (0x0)
        Serial Number:
            13:77:40:ac:a4:ae:e9:2a:4d:1c:8b:ed:ce:c8:0d:de:13:2d:ff:fc
    Signature Algorithm: 1.3.9999.3.1
        Issuer:
            commonName                = oqstest CA
        Validity
            Not Before: Jan  9 20:28:35 2023 GMT
            Not After : Jan  9 20:28:35 2024 GMT
        Subject:
            commonName                = oqstest CA
        Subject Public Key Info:
            Public Key Algorithm: 1.3.9999.3.1
            Unable to load Public Key
139819800585888:error:0609E09C:digital envelope routines:PKEY_SET_TYPE:unsupported algorithm:p_lib.c:239:
139819800585888:error:0B07706F:x509 certificate routines:X509_PUBKEY_get:unsupported algorithm:x_pubkey.c:155:
    Signature Algorithm: 1.3.9999.3.1
         39:aa:6a:64:2c:4c:54:4c:5f:cd:40:12:e5:05:b5:70:e5:ff:
         3d:35:59:95:74:89:65:96:14:02:ea:26:d8:10:a1:17:71:c1:
         f8:bc:78:f4:17:bd:46:8f:43:b5:53:2a:b6:44:aa:3a:8e:a2:
         d5:0e:8d:33:a5:88:5f:bd:70:5b:77:7a:3f:cd:8f:39:4b:36:
         59:10:40:59:8f:46:e0:99:a1:38:a7:72:22:c0:b3:d6:76:18:
         82:68:56:d9:68:81:20:4a:74:a7:b9:17:32:31:d4:36:33:cb:
         51:1e:07:92:cd:d5:97:c5:69:07:cf:f3:1c:5a:a7:28:e3:21:
         1c:9e:1f:2e:a6:de:7b:0f:59:14:ba:b9:9d:d1:72:3e:90:2e:
         93:8d:9f:c5:9a:ce:4e:aa:44:8e:54:ba:6a:33:b8:dd:eb:91:
         f2:56:75:28:0b:42:32:cd:bf:7b:64:0d:be:a8:c0:f8:0d:2c:
         06:8b:b6:ba:18:e3:82:b3:fb:8f:07:de:e1:21:9e:ef:4e:73:
         6c:45:91:81:6b:e4:35:25:36:c5:be:af:43:a4:11:d6:49:b6:
         70:78:4f:9d:db:a9:6a:a3:b3:ec:26:20:f6:4d:6e:0b:35:5a:
         03:45:f6:db:89:ee:0b:d8:22:8c:55:23:82:3d:50:bd:34:5d:
         0c:26:70:04:65:cb:e0:b3:9d:44:03:c1:8a:6e:5c:a1:d0:60:
         85:d6:19:9f:ba:28:ec:c5:bb:f4:f8:92:8d:7b:04:d9:cb:38:
         bb:13:c8:95:cd:cd:9c:b1:e2:85:41:c4:93:b6:e8:a6:e1:d6:
         24:b7:63:66:85:43:c4:15:22:f2:d1:a4:70:ae:7b:19:52:88:
         ec:51:e7:c7:21:fa:4c:d2:99:04:5a:b0:d0:74:a3:55:5c:da:
         8b:e3:57:f5:d5:ff:b5:b3:46:fa:dc:65:09:14:d2:86:7f:0a:
         a6:8d:34:80:b6:9a:16:da:d1:b7:73:b5:0e:e4:97:21:08:b3:
         7c:e0:cb:6c:0e:0c:70:1a:de:54:f5:91:55:15:17:63:1b:77:
         c5:49:88:3f:c1:72:d8:27:0c:a3:1e:51:20:86:e5:68:53:f8:
         e6:2d:50:7b:67:b1:7b:b6:36:8f:b5:dc:a4:3d:bf:b5:56:60:
         ee:d6:58:24:4a:89:3b:9d:67:db:cc:b1:c2:f9:4c:ad:d8:eb:
         1e:bd:35:c9:8d:ba:8d:c7:20:0a:62:10:99:ce:8d:b1:83:df:
         97:1d:04:1d:8d:42:58:a2:f4:dd:45:3a:8c:9c:64:40:36:fd:
         cf:bf:f6:bf:c4:38:79:ec:09:4b:13:1b:2f:25:de:bb:31:1b:
         5e:ab:02:61:52:8c:55:b1:c7:dc:1a:44:42:4f:c3:73:59:bc:
         c3:13:1d:fc:66:1e:f9:13:29:8c:10:58:4c:b8:ce:e5:f8:52:
         cf:7a:d0:cb:b1:2d:38:f2:40:1c:2b:27:16:15:b5:d5:39:8f:
         56:9d:71:70:93:2c:e9:18:e0:f3:8d:df:c2:d9:d9:39:2f:49:
         07:9f:46:68:15:af:af:17:65:ae:c4:06:c7:7b:9c:b3:f5:35:
```

Figure 9 – A server certificate signed with Falcon-512

4.3 Obtaining TLS 1.3 Payloads After Post-Quantum Adaptation

After creating the necessary certificate chains, we run TLS 1.3 sessions between the OQS-OpenSSL's *s_server* and *s_client* programs. The objective of these runs is to determine the traffic overheads produced by various key exchange and signature algorithms during the TLS handshake and use them as input for our simulation. For all scenarios, we employ one root CA certificate, one Intermediate CA certificate, and one

server certificate. Intermediate CA certificates are directly used as the certificate chain files. The setup reflects the typical number of certificates that are chained for common client and server applications that use TLS. Figure 10 gives details about the algorithm specific information carried in the data fields of a TLS 1.3 handshake using Kyber key exchange and Falcon signature algorithms as an example.

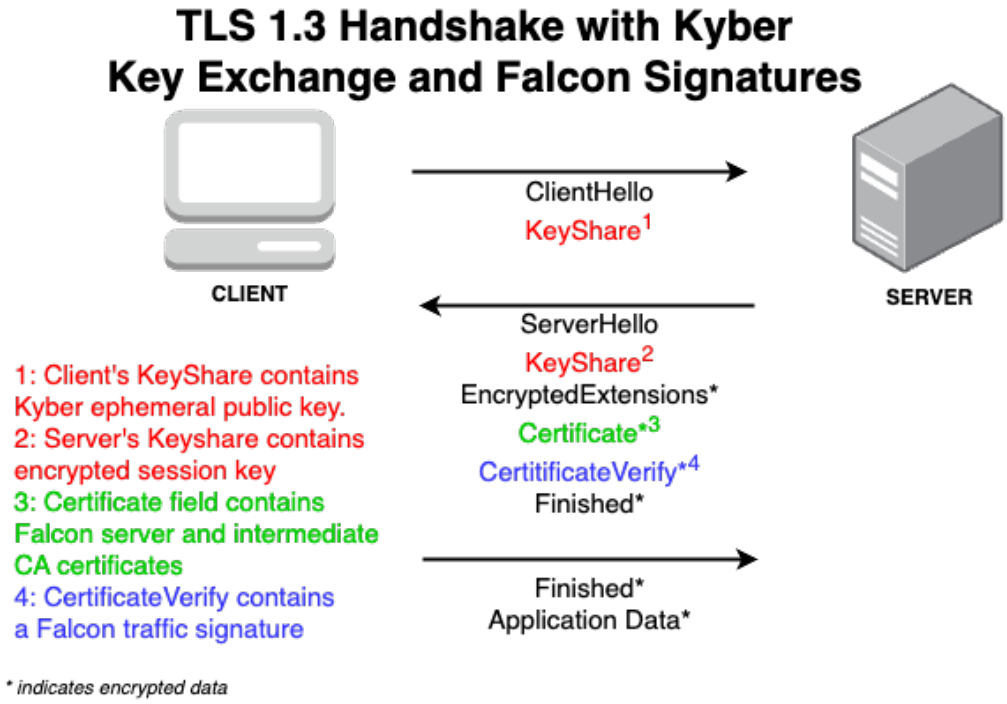

Figure 10 - TLS 1.3 Handshake with Kyber key exchange and Falcon signatures

To determine the exact payload size between the client and server, we capture the TLS handshake traffic using Wireshark Network Analyzer while *s_server* and *s_client* commands were running. We analyze every capture file to determine total data transmission on every round of this 2-RTT communication and determine the exact payload size of each message from the client to the server and from the server to the

client to accomplish a successful TLS handshake that uses post-quantum schemes. Figure 11 shows how this analysis works, by grouping the messages and the corresponding ACK messages. After analysis, we use payload sizes as input parameters for the applications that run on ns3. Detailed instructions and commands used are given in Appendix 2-C section.

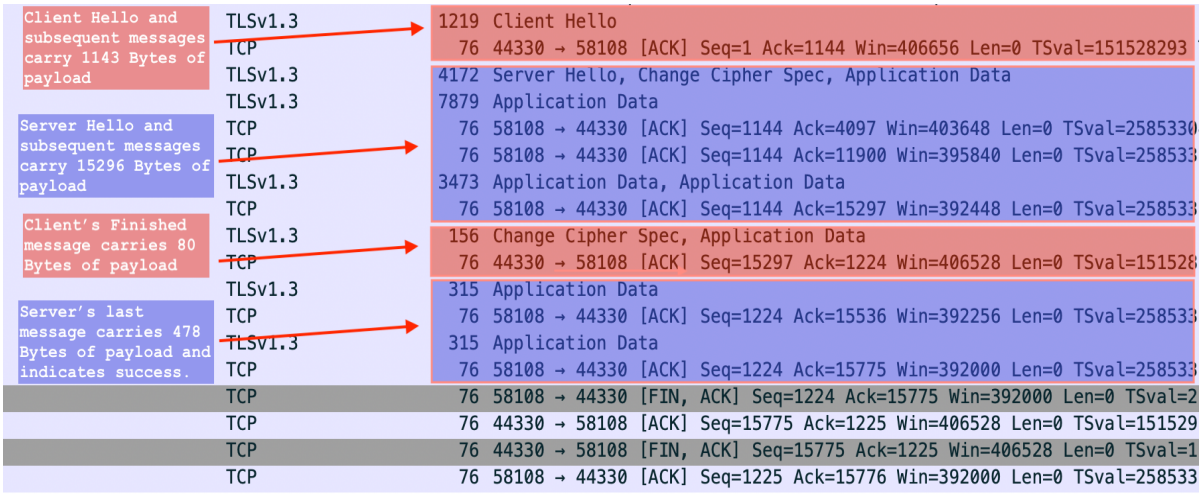

Figure 11 - Analysis of payload sizes for a TLS handshake session with Kyber key exchange and Dilithium signatures

4.4 Application to Mimic TLS 1.3

ns-3 applications are user programs running on ns-3 nodes that represent the end devices like IoT terminals or servers. These applications drive the simulations on the virtual infrastructure provided by the main ns-3 program [76].

For experiments, we create two TCP socket-based network applications written in C language which simulate TLS 1.3 handshake. Client application MyTcpEchoClient starts with sending a TCP session request and MyTcpEchoServer grants the TCP session. Next, the client sends the payload for the first round, simulating the "ClientHello" message,

while the server listens and waits until receiving the right amount of data. When the server receives the expected number of bytes, it can start sending its own data for the first round. This process continues until the client and server successfully send and receive their final payloads. Upon successful transmission rounds, the client sends a socket close request to the server, as shown in Figure 11. As a remark, we want to mention that our simulation uses a 2-RTT approach rather than 1.5 RTT of TLS 1.3. The reason for this is our client application's inability to determine whether the server successfully receives the final data before closing the connection. Therefore, we employ one more data stream from the server to the client, to indicate completion.

Payload sizes extracted in the previous section are provided to the applications as configuration parameters. Source codes for all the mentioned programs are given in the Appendix-A section.

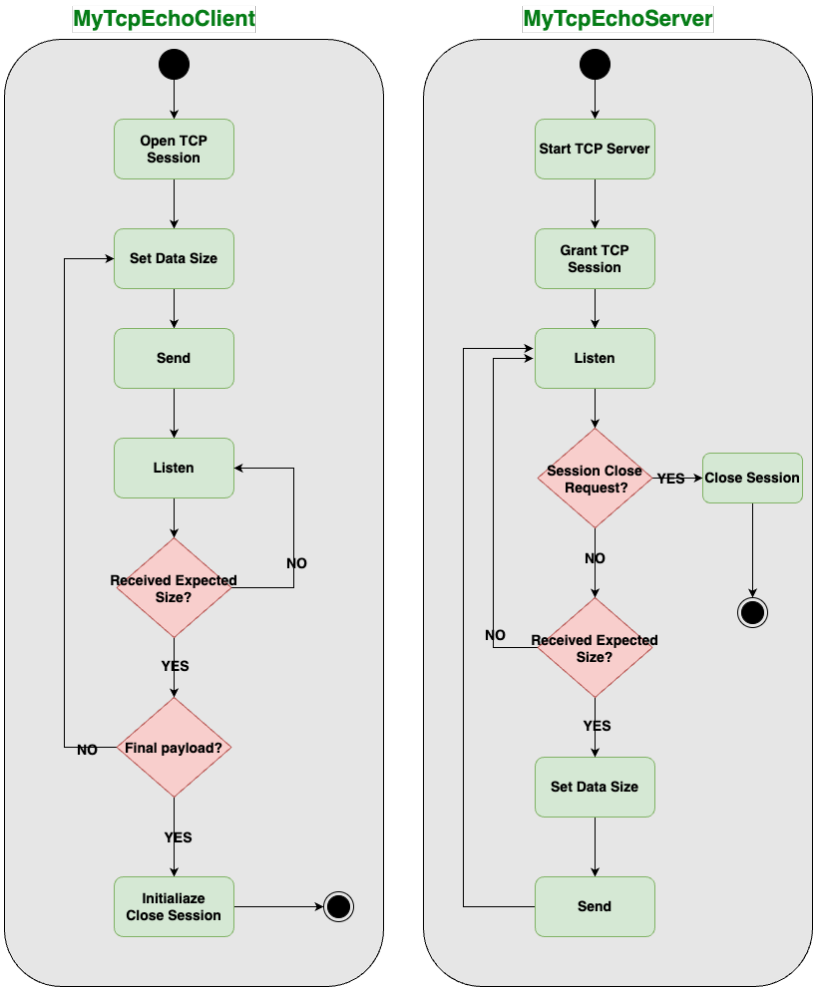

Figure 12 – UML State Diagrams for ns-3 applications MyTcpEchoClient and MyTcpEchoServer

4.5 Network Topology

For experiments, we create LTE-EPC networks with a varying number of IoT devices (5,10,15, and 20), randomly distributed within a circular area of a one-kilometer radius. An evolved NodeB (eNB) device is placed at the center of the circle, serving as a base station to connect the IoT devices in the coverage area. A remote host located on the internet hosts MyTcpEchoServer applications, whereas each IoT device runs one instance of MyTcpEchoClient application, which together simulate the TLS handshake.

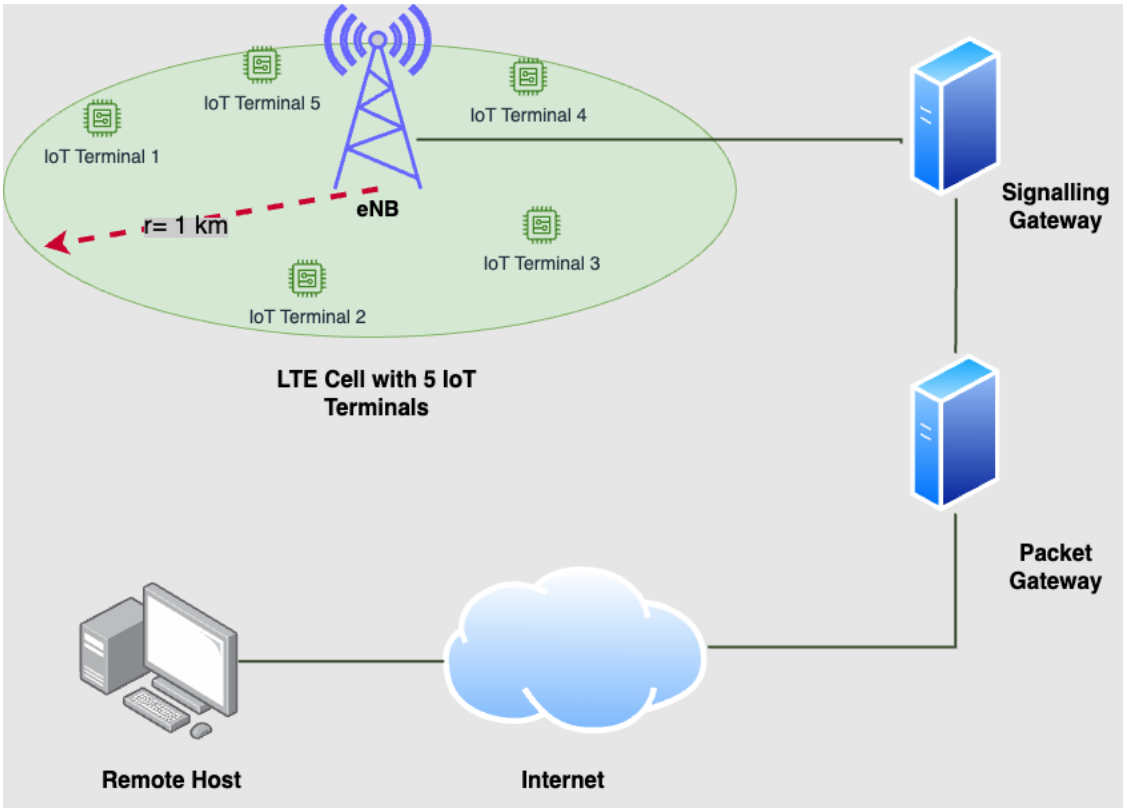

Figure 13 – Simulation Network Topology

This setup represents the ideal connectivity conditions (CE0 Level), to provide the best bitrates to terminals. We allowed only one eNB channel with a single resource block (RB) of 180KHz, which can accommodate a maximum of 12 IoT devices to connect at the same time. When the number of connected IoT devices exceeds 12, the remaining ones wait for their turn until some of the active devices are set to idle state (RRC_IDLE) by the network, allowing the waiting devices to attempt retransmission. We deliberately limit the base station to a single NB-IoT channel to observe the effects of channel saturation. However, in a real-world scenario, multiple PRBs might be available at the

same location, enabling each base station to serve multiple channels to support varying numbers of IoT devices (Figure 13).

4.6 Baseline and Performance Metrics

In this study, we aim to showcase the effects of post-quantum security schemes on the communication overhead of the TLS 1.3 handshake in an NB-IoT network, as compared to conventional security schemes. The digital signatures used in the TLS handshake are the main cause of traffic overhead, and they come in two different forms:

a. The signatures that are carried in certificates; during server-only authentication, the server typically transmits a minimum of two digital certificates in the "Certificate" data field of the handshake. These certificates include the server's own certificate (also known as the server certificate) as well as the certificate of the Intermediate CA.

b. The signature in the “CertificateVerify” data field which provides integrity of the handshake and authenticity of the server.

The digital signatures discussed here underscore the significance of choosing an appropriate signature algorithm, as it has a notable impact on TLS performance.

To evaluate the impact of post-quantum schemes on various stages of the handshake, we have developed the following scenarios:

1. As a benchmark, we utilize the ECDHE key exchange algorithm alongside ECDSA and RSA signature schemes, where certificates contain either RSA public key and RSA signature pairs or ECDSA public key and ECDSA signature pairs. Key sizes are selected as 2048 bits for RSA and 256 bits for ECC, as these are the most frequently used key sizes [77].

2. To assess the impact of post-quantum key exchange algorithms, we replace the ECDHE key exchange in TLS 1.3 with post-quantum key exchange Kyber-512 [78].

3. To evaluate the impact of post-quantum authentication overhead and the resulting increase in certificate sizes due to larger public key and signature sizes, we generated server, intermediate, and root CA certificates containing public keys and signatures from the Falcon, Dilithium, and Sphincs+ post-quantum schemes selected by NIST. Referring to OQS-OpenSSL's GitHub page we choose falcon-512 for Falcon, and sphincssha256128fsimple for Sphincs+ as algorithm variants, which have the smallest public key and signature sizes. With Crystals-Dilithium, we employed the dilithium3 parameter set, which is the recommended parameter set for achieving AES-128 bits of security by the creators of the scheme [79].

CHAPTER 5: EVALUATIONS

5.1 Overhead of Post-Quantum Key Exchange (KEM) on TLS 1.3 Handshake

In this section, we analyze the supplementary communication overhead of post-quantum key exchange compared to the traditional ECDHE key exchange method. To do so we substitute ECDHE with Kyber in two TLS cipher suites, namely TLS-ECDHE-RSA and TLS-ECDHE-ECDSA, where the RSA and ECDSA components represent the signature algorithms utilized. Through this, we can evaluate the performance of Kyber across different signature suites. In Figure 14, we present the average completion time of the TLS handshake under various numbers of IoT devices on a single NB-IoT channel.

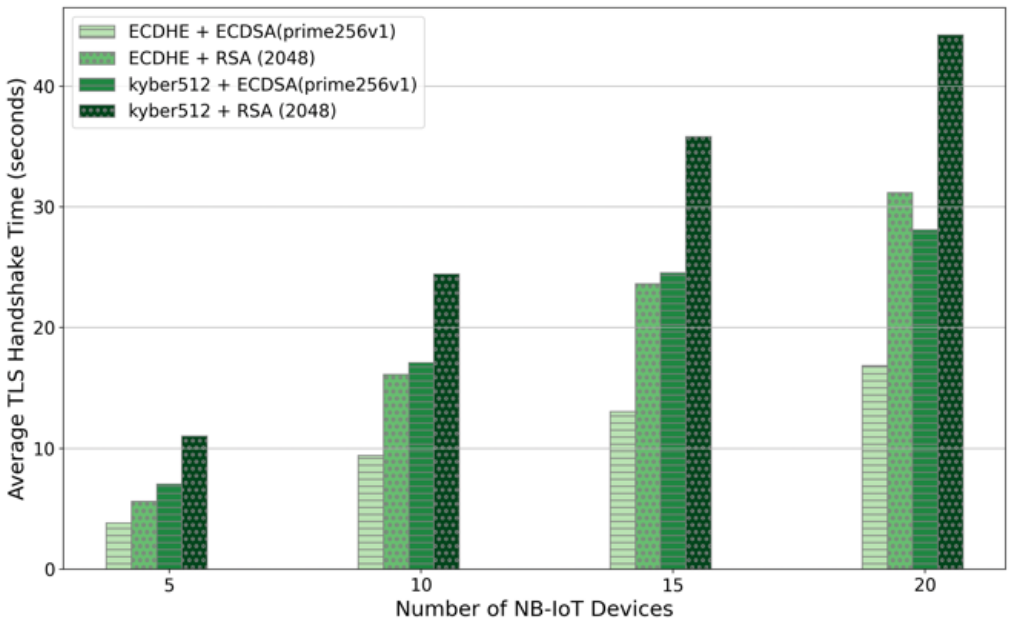

Figure 14 - Effect of key exchange algorithm on TLS handshake time

Based on the experiment results presented in Figure 14, it is evident that replacing ECDHE with the post-quantum algorithm Kyber has a noticeable negative impact on the

average time required to complete a TLS handshake. For instance, when comparing ECDHE-RSA with Kyber-RSA, the average TLS handshake time increased from 5.61 seconds to 11.02 seconds for just 5 IoT devices. This difference becomes even more significant when the number of devices increases to 20, where the average TLS handshake time almost reaches one minute.

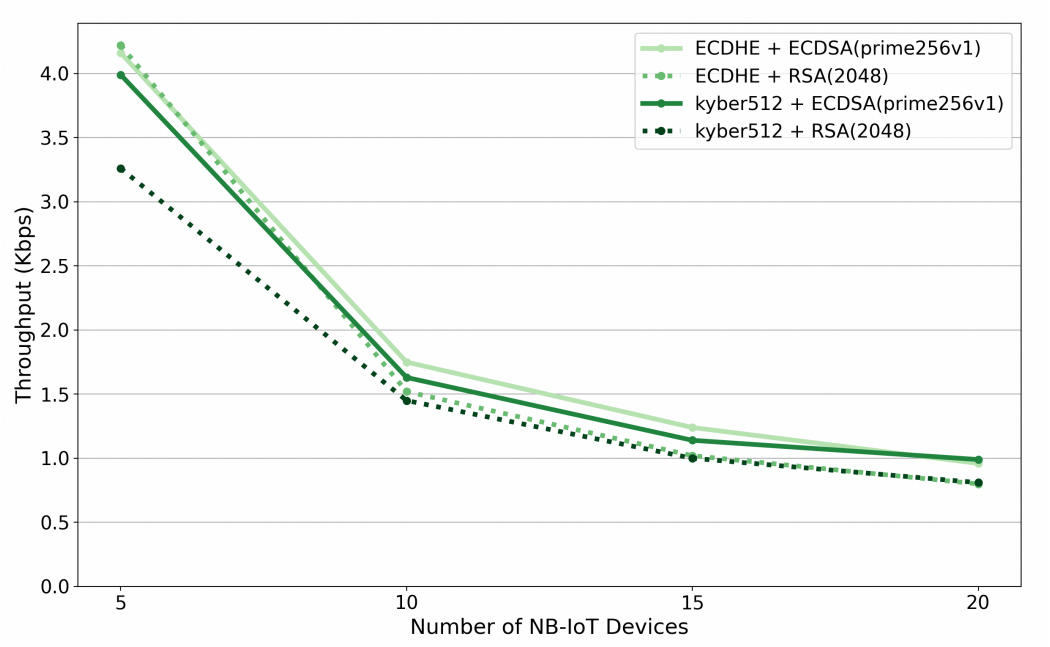

Figure 15 - Effect of key exchange algorithm on throughput

Similarly, when comparing ECDHE-ECDSA with Kyber-ECDSA, the average time increases from 3.85 seconds to 7.07 seconds for 5 devices. However, we note that the increase is less drastic when using ECDSA as the signature algorithm instead of RSA, and the TLS handshake takes less than 30 seconds on average for 20 devices. This is because the small overhead of ECDSA helps the NB-IoT network accommodate the overhead of post-quantum Kyber. In both cases, Kyber introduces a significant overhead, and this can be directly attributed to the difference in “key_share” size between Kyber

and ECDHE algorithms as shown in Table 2. Specifically, the “key_share” size of Kyber in the “ClientHello” message is 806 bytes, which is 768 bytes larger than the 38 bytes of ECDHE. The size increase is also observed with the “key_share” in the “ServerHello” message, which was 36 bytes for ECDHE and 772 bytes for Kyber. Overall, the substitution of key exchange algorithms in TLS handshake introduces an overhead of 1500 bytes, resulting in an additional 50% overhead compared to ECDHE + RSA and 73% compared to ECDHE + ECDSA, as shown in Table 2. These findings underscore the importance of selecting the appropriate key exchange algorithm to minimize network traffic size, particularly in the context of post-quantum algorithms and their larger key sizes.

KEM/Digital Signature	key_share Size in ClientHello (bytes)	key_share Size in ServerHello (bytes)	Total Traffic Exchanged During Handshake (bytes)
ECDHE + ECDSA (prime256v1)	38	36	2048
ECDHE + RSA (2048)	38	36	3026
kyber512 + ECDSA (prime256v1)	806	772	3560
kyber512 + RSA (2048)	806	772	4522

Table 2 – TLS handshake traffic size comparisons with ECDHE and Kyber key exchange algorithms

Experiment results indicate that NB-IoT provides similar throughput for Kyber compared to the ECDHE scheme. Figure 15 illustrates that the throughput for all

scenarios begins at approximately 4 kbps for 5 IoT devices and gradually decreases to 1 kbps for 20 devices in ECDSA scenarios and 0.8 kbps for RSA scenarios.

Another observation is, although the kyber512+ECDSA scenario has a bigger total traffic size compared to ECDHE+RSA, they both yield 0.8 kbps of throughput. This result can be explained by the asymmetrical bitrate characteristic of NB-IoT networks, which does not favor large payloads in the downlink direction. On the other hand, the overhead caused by the bigger key_share of Kyber is split between uplink and downlink channels and yields better throughput results.

5.2 Overhead of Post-Quantum Signature Algorithms on TLS-Handshake

For our second benchmark, we examined the overhead generated by post-quantum signature algorithms in comparison to ECDSA, when post-quantum Kyber is utilized as the default key exchange algorithm. The use of post-quantum signature algorithms results in significant increases in both signature and public key sizes, which in turn leads to considerable latency spikes in our tests. Of the post-quantum signature schemes we tested, Falcon was the most lightweight, as indicated in Table 3. As a result, it performed the best, completing the TLS handshake for five devices in under 16 seconds on average, and in about 50 seconds for 20 devices. However, Dilithium and Sphincs+ took more than 40 and 70 seconds, respectively, even with only five devices in the channel, see Figure 16. This can be attributed to the total exchanged traffic sizes of these two algorithms, which are 16997 and 26830 bytes, respectively (Table-3).

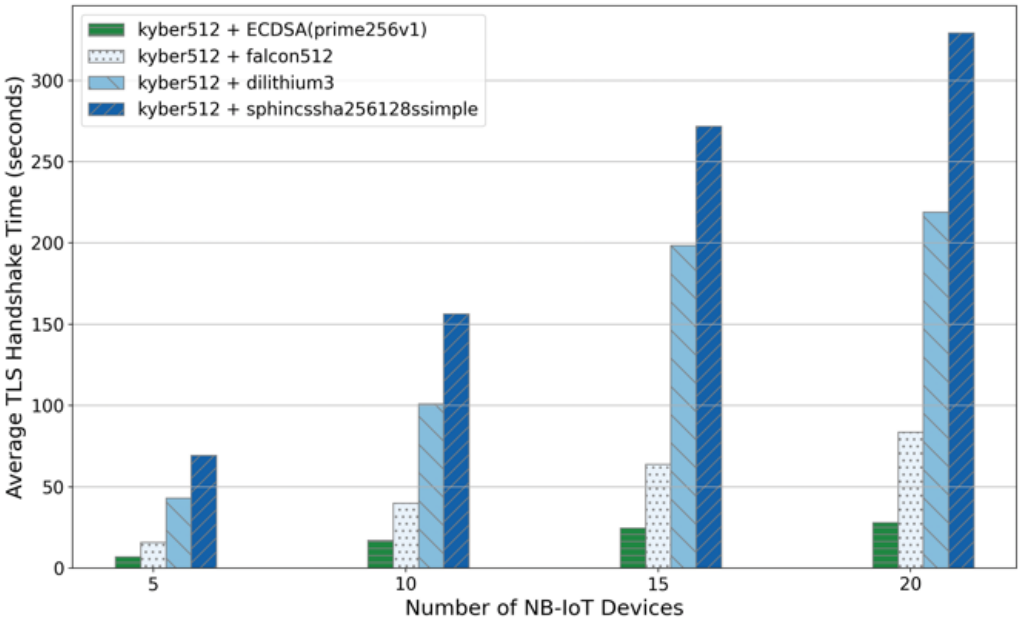

Figure 16 - Effect of Signature Algorithm on Average TLS Handshake Time

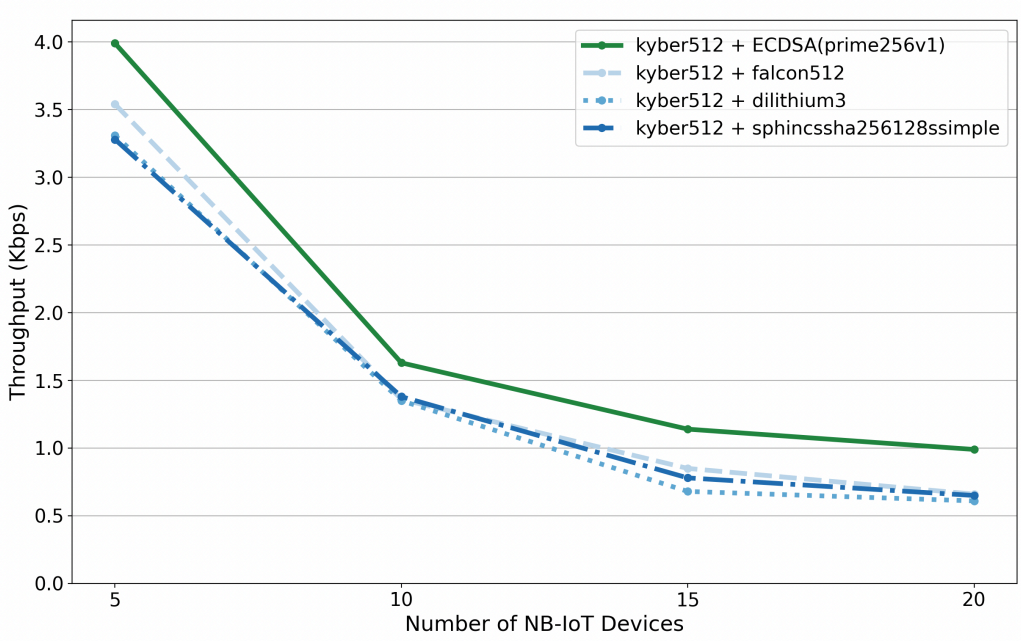

Figure 17 - Effect of Signature Algorithm on Throughput

In this set of experiments, measured throughput is about 3.5 kbps with 5 devices, which is slightly lower than traditional signature scheme scenarios (ECDSA/RSA), but

with the increased number of devices, results decrease even more. In comparison to ECDSA, we observed a 28% decrease in throughput for 10, 15, and 20 devices when post-quantum authentication schemes were used, resulting in a throughput of only 0.6 kbps, see Figure 17. These results suggest that the large traffic overhead introduced by new post-quantum signature schemes has an adverse impact on NB-IoT performance.

KEM/Digital Signature	ICA + Server Certificate Size (bytes)	Signature Size (bytes)	Total Traffic Exchanged During Handshake (bytes)
kyber512 + ECDSA (prime256v1)	592	64	3560
kyber512 + falcon512	3404	690	6965
kyber512 + dilithium3	10824	3293	16997
kyber512 + sphincsha256128fsimple	16070	7856	26830

Table 3 – Certificate, signature, and total traffic overheads of current vs post-quantum signature algorithms with Kyber key exchange

Replacing conventional signature schemes with post-quantum signature schemes during TLS handshake has a significant impact on performance compared to simply substituting the ECDHE with the post-quantum key exchange scheme Kyber. Falcon, while the best performing among the post-quantum signature schemes we tested, introduces nearly 5 times the overhead when compared to ECDHE/ECDSA and 2.5 times the overhead when compared to ECDHE/RSA. Despite this, it remains a feasible candidate for post-quantum signature migration in low-density NB-IoT networks, with a latency of 15 seconds for five devices. However, the results indicate that Dilithium and Sphincs+ may be impractical for most applications that run on IoT networks.

CHAPTER 6: CONCLUSION AND FUTURE WORK

6.1 Conclusion

In this study, we examine the feasibility of using post-quantum cryptographic schemes in TLS to ensure the security of NB-IoT applications. Drawing on the experimental results, we can conclude that NB-IoT applications can transition to post-quantum key exchange Kyber with an acceptable performance degradation when ECDSA is used. However, the use of post-quantum signature schemes would lead to impractical TLS handshake times and lower throughputs. If there is a need to adopt post-quantum signatures, the most viable choice would be Falcon with a ring degree of 512.

In light of these findings, we suggest migrating to Kyber as a viable option to prevent potential "store-now, decrypt later" type attacks that may arise from breaking ECDHE encryption with quantum computers in the future [45]. However, as there is currently no evident threat, it is advisable to continue using conventional signature schemes for TLS authentication until a quantum computer powerful enough to breach the security of ECDSA is developed.

For the post-quantum era, on the other hand, the industry needs to formulate alternative approaches and methods to handle the signature overheads of cryptographic operations. While this work specifically focuses on TLS handshake, the scope can be extended to include other protocols like VPN and SSH which also use signatures in their authentication phase. Here we list some methods to mitigate the effects of post-quantum signatures:

Long Term Symmetric Key Usage: A practical method to avoid signature overheads is to utilize symmetrical keys for subsequent sessions and conduct less

frequent handshakes. While this approach eliminates the Perfect Forward Secrecy (PFS) feature of TLS 1.3, it can bring more advantages compared to disadvantages when overall network performance is considered. This approach might be best compatible with IoT devices that communicate to a limited number of servers on the internet, which decreases the number of new authentication requirements.

Certificate Caching: Another option is caching the Intermediate CA certificates on the IoT device to avoid retransmission. With this, one of the two necessary signatures used for authentication in the handshake process is eliminated. However, this approach like the previous one requires a previous TLS session to cache the certificate, and certificates should be checked against revocation on every usage.

6.2 Future Work

In the future, we aim to extend our work by focusing on unvisited aspects of the NB-IoT technology in this work, and by running our experiments on other LPWAN technologies. Specifically, all the experiments conducted within the scope of this work feature the best NB-IoT coverage level, namely CE0. Although deploying IoT terminals in CE0 coverage helps with identifying the best throughput and handshake times, running the experiments with distant terminals would help identify adjustable NB-IoT network parameters such as repetition numbers for other coverage enhancement levels and therefore cell sizes.

Another intention is to test post-quantum key exchange and signature performances on other LPWAN technologies. LoRa, SigFox, and e-MTC feature different bandwidth allocations and need to be investigated with bigger traffic loads. What is more, NB-IoT

features stationary terminals, and the effect of mobility should also be investigated with further studies.

BIBLIOGRAPHY

[1] J. Kua, C. Arora, S. W. Loke, N. Fernando and C. Ranaweera, "Internet of Things in Space: Review of Opportunities and Challenges from Satellite-Aided Computing to Digitally-Enhanced Space Living," *CoRR,* 2021.

[2] E. Sisinni, A. Saifullah, S. Han, U. Jennehag and M. Gidlund, "Industrial Internet of Things: Challenges, Opportunities, and Directions," *IEEE Transactions on Industrial Informatics, vol. 14, no. 11,* pp. 4724-4734, 2018.

[3] G. Valecce, P. Petruzzi, S. Strazzella and L. A. Grieco, "NB-IoT for Smart Agriculture: Experiments from the Field," in *2020 7th International Conference on Control, Decision and Information Technologies (CoDIT)*, Prague,Czech Republic, 2020.

[4] S. Zhang, Y. Wang and W. Zhou, "Towards secure 5G networks: A Survey," *Computer Networks,* vol. 162, p. 106871, 2019.

[5] S. Narayanan, D. Tsolkas, N. Passas and L. Merakos, "NB-IoT: A candidate technology for massive IoT in the 5G era," in *2018 IEEE 23rd International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD)*, Barcelona, Spain, IEEE, 2018, pp. 1-6.

[6] M. De Ree, G. Mantas, A. Radwan, S. Mumtaz, J. Rodriguez and I. E. Otung, "Key Management for Beyond 5G Mobile Small Cells: A Survey," *IEEE Access ,* vol. 7, pp. 59200-59236, 2019.

[7] X. Huang, V. Tsiatsis, A. Palanigounder, L. Su and B. Yang, "5G Authentication and Key Management for Applications," *IEEE Communications Standards Magazine,* vol. 5, no. 2, pp. 142-148, 2021.

[8] T. Hewa, A. Bracken, M. Ylianttila and M. Liyanage, "Blockchain-based Automated Certificate Revocation for 5G IoT," in *ICC 2020-2020 IEEE International Conference on Communications (ICC)*, Dublin, Ireland, IEEE, 2020, pp. 1-7.

[9] J. Gambetta, "IBM’s roadmap for scaling quantum technology," September 2020. [Online]. Available: https://research.ibm.com/blog/ibm-quantum-roadmap.

[10] Oracle, "What is IoT?," [Online]. Available: https://www.oracle.com/internet-of-things/what-is-iot/.

[11] B. Marr, "The Top 4 Internet Of Things Trends In 2023," 22 November 2022. [Online]. Available: https://www.forbes.com/sites/bernardmarr/2022/11/07/the-top-4-internet-of-things-trends-in-2023/?sh=26bc7b9d2aea. [Accessed March 2023].

[12] K. Mekki, E. Bajic, F. Chaxel and F. Meyer, "A comparative study of LPWAN technologies for large-scale IoT deployment," *ICT Express,* pp. 1-7, March 2019, Volume 5.

[13] N. Ahmed, D. De and I. Hussain, "Internet of Things (IoT) for Smart Precision Agriculture and Farming in Rural Areas," *IEEE Internet of Things Journal,* vol. 5, no. 6, pp. 4890 - 4899, 2018.

[14] M. Iqbal, A. Y. M. A. and S. Farzana, "An Application Based Comparative Study of LPWAN Technologies for IoT Environment," in *2020 IEEE Region 10 Symposium (TENSYMP)*, Dhaka, Bangladesh, 2020.

[15] 3GPP, "3GPP TS 36.104 version 13.4.0 Release 13," ETSI, 08 2016. [Online]. Available: https://www.etsi.org/deliver/etsi_ts/136100_136199/136104/13.04.00_60/ts_136104v130400p.pdf.

[16] Rohde&Schwarz, "Narrowband Internet of Things Whitepaper," [Online]. Available: https://cdn.rohde-schwarz.com/pws/dl_downloads/dl_application/application_notes/1ma266/1MA266_0e_NB_IoT.pdf. [Accessed February 2023].

[17] A. P. Matz, J.-A. Fernandez-Prieto, J. Cañada-Bago and U. Birkel, "A Systematic Analysis of Narrowband IoT Quality of Service," *Sensors (Basel),* vol. 20, no. 6, p. 1636, 2020.

[18] P. Jörke, T. Gebauer, S. Böcker and C. Wietfeld, "Scaling Dense NB-IoT Networks to the Max: Performance Benefits of Early Data Transmission," in *2022 IEEE 95th Vehicular Technology Conference: (VTC2022-Spring)*, Helsinki, Finland, 2022.

[19] Mpirical, "Introducing NB-IoT (Narrow Band Internet of Things) YouTube Page," November 2022. [Online]. Available: https://www.youtube.com/watch?v=9WMbgvnpWf0. [Accessed 1 March 2023].

[20] G. Medina-Acosta, L. Zhang, J. Chen, K. Uesaka, Y. Wang, O. Lundqvist and J. Bergman, "3GPP Release-17 Physical Layer Enhancements for LTE-M and NB-IoT," *IEEE Communications Standards Magazine ,* vol. 6, no. 4, pp. 80 - 86, 2022.

[21] Allied Market Research, "Narrowband IoT (NB-IoT) Chipset Market," July 2021. [Online]. Available: https://www.alliedmarketresearch.com/narrowband-iot-nb-iot-chipset-market-A09846. [Accessed March 2023].

[22] Statista, "Share of LPWA connections worldwide in the first half of 2020 and 2025, by technology," June 2022. [Online]. Available: https://www.statista.com/statistics/1244778/lpwa-market-share-by-technology. [Accessed March 2023].

[23] TechPlayon, "NB IOT Peak Data Rate Calculation," May 2020. [Online]. Available: https://www.techplayon.com/nb-iot-peak-data-rate-calculation/.

[24] J. Ajayi, S. Weber, T. Braun, B. Stiller and E. Schiller, "Toward a Live BBU Container Migration in Wireless Networks," *IEEE Open Journal of the Communications Society ,* vol. 3, pp. 301 - 321, 2022.

[25] FCC, "Plan Ahead for Phase Out of 3G Cellular Networks and Service," October 2022. [Online]. Available: https://www.fcc.gov/consumers/guides/plan-ahead-phase-out-3g-cellular-networks-and-service. [Accessed March 2023].

[26] K. Kousias, G. Caso, O. Alay, L. De Nardis, M. Neri, A. Brunström and M. G. Di Benedetto, "Coverage and Deployment Analysis of Narrowband Internet of Things in the Wild," *CoRR,* vol. abs/2005.02341, 2020.

[27] 3GPP, "3GPP TR 45.050 version 13.1.0 - Release 13," 08 2016. [Online]. Available: https://www.etsi.org/deliver/etsi_tr/145000_145099/145050/13.01.00_60/tr_145050v130100p.pdf.

[28] S. Tabbane, "ITU Asia-Pacific Centre of Excellence Training On “Traffic engineering and advanced wireless network 1 planning”," October 2018. [Online]. Available: https://www.itu.int/en/ITU-D/Regional-Presence/AsiaPacific/SiteAssets/Pages/ITU-ASP-CoE-Training-on-/Session5_NB_IoT%20networks%20web.pdf. [Accessed February 2023].

[29] 3GPP, "3GPP TS 23.682 version 12.2.0 - Release 12," October 2014. [Online]. Available: https://www.etsi.org/deliver/etsi_ts/123600_123699/123682/12.02.00_60/ts_123682v120200p.pdf. [Accessed February 2023].

[30] Internet Society, " TLS for Applications," [Online]. Available: https://www.internetsociety.org/deploy360/tls-for-applications-2/. [Accessed February 2023].

[31] P. Hoffman, "SMTP Service Extension for Secure SMTP over Transport Layer Security," February 2002. [Online]. Available: https://www.ietf.org/rfc/rfc3207.txt. [Accessed March 2023].

[32] IBM, "About encryption keys," February 2022. [Online]. Available: https://www.ibm.com/docs/en/ts11xx-tape-drive?topic=STPRH6/com.ibm.storage.drives.doc/top_tscom_reuse_encryptoview_keys.htm. [Accessed March 2023].

[33] P. Gutmann, "Using Message Authentication Code (MAC) Encryption in the Cryptographic Message Syntax (CMS)," January 2012. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc6476. [Accessed March 2023].

[34] CLoudflare Blog, "What is mutual TLS (mTLS)?," [Online]. Available: https://www.cloudflare.com/learning/access-management/what-is-mutual-tls/. [Accessed March 2023].

[35] IBM, "Digital certificates for user authentication," April 2021. [Online]. Available: https://www.ibm.com/docs/en/i/7.3?topic=dcm-digital-certificates-user-authentication. [Accessed March 2023].

[36] N. Sullivan, "A Detailed Look at RFC 8446 (a.k.a. TLS 1.3)," 08 October 2018. [Online]. Available: https://blog.cloudflare.com/rfc-8446-aka-tls-1-3/. [Accessed February 2023].

[37] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3 RFC 8446," 2018 August. [Online]. Available: https://datatracker.ietf.org/doc/rfc8446/. [Accessed March 2023].

[38] D. Volya, "Elliptic Curve Cryptography," October 2020. [Online]. Available: https://volya.xyz/ecc/. [Accessed March 2023].

[39] Wikipedia, "Wikipedia Page for Elliptic curve," [Online]. Available: https://en.wikipedia.org/wiki/Elliptic_curve. [Accessed March 2023].

[40] S. Turner, D. Brown, K. Yiu, R. Housley and T. Polk, "Elliptic Curve Cryptography Subject Public Key Information," March 2009. [Online]. Available: https://www.ietf.org/rfc/rfc5480.txt. [Accessed March 2023].

[41] C. Paar and J. Pelzl, "Elliptic Curve Cryptosystems," in *Understanding Cryptography*, Bochum, Springer, 2010, p. 250.

[42] M. Dworkin and R. Housley, "Advanced Encryption Standard (AES) Key Wrap with Padding Algorithm RFC 5649," September 2009. [Online]. Available: https://datatracker.ietf.org/doc/rfc5649/. [Accessed March 2023].

[43] P. W. Shor, "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer," *Society for Industrial & Applied Mathematics Journal on Computing,* vol. 26, no. 5, pp. 1484-1509, 1997.

[44] NIST - Information Technology Laboratory, "Post-Quantum Cryptography - Round 1 Submissions," 03 January 2017. [Online]. Available: https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-1-submissions. [Accessed February 2023].

[45] N. Sendrier, "Code-based Cryptography," 2017. [Online]. Available: https://2017.pqcrypto.org/exec/slides/cbctuto-ecrypt.pdf. [Accessed February 2023].

[46] ISARA Corporation, "Isogeny-Based Cryptography Tutorial," August 2019. [Online]. Available: https://www.isara.com/downloads/crypto_tutorials/Intro-to-Iso-v3.pdf. [Accessed February 2023].

[47] SIKE Team, "SIKE and SIDH are insecure and should not be used," August 2022. [Online]. Available: https://csrc.nist.gov/csrc/media/Projects/post-quantum-cryptography/documents/round-4/submissions/sike-team-note-insecure.pdf. [Accessed February 2023].

[48] W. J. Buchanan, "Hashed-based signatures," Asecuritysite.com, 2023. [Online]. Available: https://asecuritysite.com/subjects/chapter106. [Accessed February 2023].

[49] C. D. M. E. S. F. S.-L. G. A. H. P. K. S. K. T. L. M. M. L. F. M. R. N. C. R. J. R. P. Daniel J. Bernstein, *SPHINCS+ - Submission to the NIST post-quantum project,* 2017.

[50] W. J. Buchanan, "PQC Digital Signature Speed Tests," Asecuritysite.com, 2023. [Online]. Available: https://asecuritysite.com/pqc/pqc_sig. [Accessed February 2023].

[51] J. Ding, "Multivariate Public Key Cryptography," [Online]. Available: http://www.fields.utoronto.ca/av/slides/06-07/number_theory/ding/download.pdf. [Accessed February 2023].

[52] J. Alwen, "What is Lattice-Based Cryptography & Why You Should Care," Wickr Blog on Medium, June 2018. [Online]. Available:

https://medium.com/cryptoblog/what-is-lattice-based-cryptography-why-should-you-care-dbf9957ab717. [Accessed February 2023].

[53] C. Peikert, "Post-Quantum Cryptography," Youtube, 3 March 2022. [Online]. Available: https://www.youtube.com/watch?v=dbP2cgTsrRo. [Accessed February 2023].

[54] T. Anderton, "Visualizing Lattices," April 2020. [Online]. Available: https://asymptoticlabs.com/blog/posts/visualizing-lattices.html. [Accessed March 2023].

[55] C. Peikert, "Presentation," 2013. [Online]. Available: Lattice-Based Cryptography: Short Integer Solution (SIS) and Learning With Errors (LWE). [Accessed March 2023].

[56] "Wikipedia Page on Short integer solution problem," [Online]. Available: https://en.wikipedia.org/wiki/Short_integer_solution_problem. [Accessed March 2023].

[57] D. Wong, Real-World Cryptography, Manning, 2021.

[58] R. Gonzalez, "Kyber - How does it work?," September 2021. [Online]. Available: https://cryptopedia.dev/posts/kyber/. [Accessed March 2023].

[59] B. Buchanan, "What's Next After Lattice Cryptography? Will it be BIKE, HQC, McEliece or SIKE?," Prof Bill Buchanan OBE Blog on Medium, October 2022. [Online]. Available: https://medium.com/asecuritysite-when-bob-met-alice/whats-next-after-lattice-cryptography-will-it-be-bike-hqc-mceliece-or-sike-7d543a8a6fb7. [Accessed February 2023].

[60] E. Barker, "NIST Special Publication 800-57 Part 1 - Revision 5," May 2020. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57pt1r5.pdf. [Accessed February 2023].

[61] NIST - Information Technology Laboratory, "Digital Signatures," [Online]. Available: https://csrc.nist.gov/Projects/digital-signatures. [Accessed February 2023].

[62] D. Sikeridis, P. Kampanakis and M. Devetsikiotis, "Post-Quantum Authentication in TLS 1.3: A Performance Study," *Cryptology ePrint Archive, Paper 2020/071,* 2020.

[63] C. Paquin, D. Stebila and G. Tamvada, "Benchmarking Post-quantum Cryptography in TLS," *PQCrypto 2020. Lecture Notes in Computer Science,* vol. 12100.

[64] P. Kampanakis and . T. Lepoin, "Do we need to change some things? Open questions posed by the upcoming post-quantum migration to existing standards and deployments," *Cryptology ePrint Archive, Paper 2023/266,* 2023.

[65] S. Paul, F. Schick and J. Seedorf, "TPM-Based Post-Quantum Cryptography: A Case Study on Quantum-Resistant and Mutually Authenticated TLS for IoT Environments," *Proceedings of the 16th International Conference on Availability, Reliability and Security (ARES 21),* pp. 1-10, 2021.

[66] C. A. Roma, C.-E. A. Tai and M. A. Hasan, "Energy Efficiency Analysis of Post-Quantum Cryptographic Algorithms," *IEEE Access,* vol. 9, 2021.

[67] "ns-3 Network Simulator," [Online]. Available: https://www.nsnam.org/. [Accessed March 2023].

[68] "5G-LENA simulator," [Online]. Available: https://5g-lena.cttc.es/. [Accessed March 2023].

[69] T. Gebauer and P. Jörke, "NS-3 LENA-NB," February 2023. [Online]. Available: https://github.com/tudo-cni/ns3-lena-nb.

[70] P. Jörke, T. Gebauer and C. Wietfeld, "From LENA to LENA-NB: Implementation and Performance Evaluation of NB-IoT and Early Data Transmission in ns-3," in *WNS3'22*, 2022.

[71] Open Quantum Safe Github, "OQS OpenSSL Fork," February 2023. [Online]. Available: https://github.com/open-quantum-safe/openssl. [Accessed February 2023].

[72] "OPEN QUANTUM SAFE," [Online]. Available: https://openquantumsafe.org/. [Accessed March 2023].

[73] "How Certificate Chains Work," [Online]. Available: https://knowledge.digicert.com/solution/SO16297.html. [Accessed March 2023].

[74] "CSR Decoder And Certificate Decoder," CertLogik, [Online]. Available: https://certlogik.com/decoder/. [Accessed March 2023].

[75] M. Ounsworth, J. Gray and M. Pala, "Composite Public and Private Keys For Use In Internet PKI," March 2023. [Online]. Available:

https://datatracker.ietf.org/doc/html/draft-ounsworth-pq-composite-keys. [Accessed March 2023].

[76] "ns-3 Tutorial," [Online]. Available: https://www.nsnam.org/docs/tutorial/ns-3-tutorial.pdf. [Accessed March 2023].

[77] D. Warburton and S. Vinberg, "The 2021 TLS Telemetry Report," F5 Labs, 2022. [Online]. Available: https://www.f5.com/content/dam/f5-labs-v2/article/pdfs/2021-TLS-Telemetry-Report.pdf. [Accessed February 2023].

[78] "Kyber," [Online]. Available: https://pq-crystals.org/kyber/. [Accessed February 2023].

[79] "Dilithium," [Online]. Available: https://pq-crystals.org/dilithium/. [Accessed February 2023].

[80] K. Sabanci, "Kadir Sabanci Github Repository," 2022. [Online]. Available: https://github.com/ksabanci-marquette/ns3_allinone/.

APPENDICES

A. TLS 1.3 SIMULATOR APPLICATION CODE [80]

a. MyTcpEchoClient

```
/* -*- Mode:C++; c-file-style:"gnu"; indent-tabs-mode:nil; -*- */ /*
 * Copyright 2007 University of Washington
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation;
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 */
#include "ns3/log.h"
#include "ns3/ipv4-address.h"
#include "ns3/nstime.h"
#include "ns3/inet-socket-address.h"
#include "ns3/socket.h"
#include "ns3/simulator.h"
#include "ns3/socket-factory.h"
#include "ns3/packet.h"
#include "ns3/uinteger.h"
#include "ns3/trace-source-accessor.h"
#include "my-tcp-echo-client.h"
#include "ns3/tag.h"
#include "TimestampTag.h"
#include  "string.h"
#include "ns3/config-store-module.h"
#include "ns3/core-module.h"
#include <thread>
#include <iostream>
#include <chrono>

namespace ns3 {

    NS_LOG_COMPONENT_DEFINE ("MyTcpEchoClientApplication");
    NS_OBJECT_ENSURE_REGISTERED (MyTcpEchoClient);

    TypeId
    MyTcpEchoClient::GetTypeId (void)
    {
        static TypeId tid = TypeId ("ns3::MyTcpEchoClient")
                .SetParent<Application> ()
                .AddConstructor<MyTcpEchoClient> ()
                .AddAttribute ("MaxPackets",
                               "The maximum number of packets the application will send",
                               UintegerValue (100),
                               MakeUintegerAccessor (&MyTcpEchoClient::m_count),
                               MakeUintegerChecker<uint32_t> ())
                .AddAttribute ("Interval",
                               "The time to wait between packets",
```

```
                                TimeValue (Seconds (1.0)),
                                MakeTimeAccessor (&MyTcpEchoClient::m_interval),
                                MakeTimeChecker ())
                .AddAttribute ("RemoteAddress",
                                "The destination Ipv4Address of the outbound packets",
                                AddressValue (),
                                MakeAddressAccessor (&MyTcpEchoClient::m_peerAddress),
                                MakeAddressChecker ())
                .AddAttribute ("RemotePort",
                                "The destination port of the outbound packets",
                                UintegerValue (0),
                                MakeUintegerAccessor (&MyTcpEchoClient::m_peerPort),
                                MakeUintegerChecker<uint16_t> ())
                .AddAttribute ("PacketSize", "Size of echo data in outbound packets",
                                UintegerValue (100),
                                MakeUintegerAccessor (&MyTcpEchoClient::SetDataSize,
                                                      &MyTcpEchoClient::GetDataSize),
                                MakeUintegerChecker<uint32_t> ())
                .AddAttribute ("TagCounter","Number of RTTs",
                                UintegerValue (0),
                                MakeUintegerAccessor (&MyTcpEchoClient::SetTagCounter),
                                MakeUintegerChecker<uint8_t> ())
                .AddAttribute ("ClientNo","ClientNo",
                                UintegerValue (0),
                                MakeUintegerAccessor (&MyTcpEchoClient::SetClientNo),
                                MakeUintegerChecker<uint8_t> ())
                .AddAttribute ("MSS","MSS",
                                UintegerValue (0),
                                MakeUintegerAccessor (&MyTcpEchoClient::SetMSS),
                                MakeUintegerChecker<uint32_t> ())
                .AddTraceSource ("Tx", "A new packet is created and is sent",
                                  MakeTraceSourceAccessor (&MyTcpEchoClient::m_txTrace),
                                  "ns3::Packet::TracedCallback")
                .AddTraceSource ("Rx", "A packet has been received",
                                  MakeTraceSourceAccessor (&MyTcpEchoClient::m_rxTrace),
                                  "ns3::Packet::TracedCallback")
                .AddTraceSource ("TxWithAddresses", "A new packet is created and is sent",
                                  MakeTraceSourceAccessor
(&MyTcpEchoClient::m_txTraceWithAddresses),
                                  "ns3::Packet::TwoAddressTracedCallback")
                .AddTraceSource ("RxWithAddresses", "A packet has been received",
                                  MakeTraceSourceAccessor
(&MyTcpEchoClient::m_rxTraceWithAddresses),
                                  "ns3::Packet::TwoAddressTracedCallback")
        ;
        return tid;
    }


    TypeId MyTcpEchoClient::GetInstanceTypeId() const {
        return MyTcpEchoClient::GetTypeId(); }


    MyTcpEchoClient::MyTcpEchoClient ()
    {
        NS_LOG_FUNCTION (this);
        m_sent = 0;
        m_socket = 0;
        m_sendEvent = EventId ();
        m_data = 0;
        m_dataSize = 1000;
    }

    MyTcpEchoClient::~MyTcpEchoClient()
    {
```

```
    NS_LOG_FUNCTION (this);
    m_socket = 0;

    delete [] m_data;
    m_data = 0;
    m_dataSize = 1000;
    m_tagCounter = 0;
}

void
MyTcpEchoClient::SetTagCounter (uint8_t tagCounter)
{
    NS_LOG_FUNCTION (this << "tag Counter "<< tagCounter);
    m_tagCounter = tagCounter;
}

void
MyTcpEchoClient::SetClientNo(uint8_t clientNo)
{
    NS_LOG_FUNCTION (this << "clientNo: "<< clientNo);
    m_clientNo = clientNo;
}

void
MyTcpEchoClient::SetIsComplete (){
    isComplete=true;
}

bool MyTcpEchoClient::GetIsComplete(){
    return isComplete;
}


void
MyTcpEchoClient::SetRemote (Address ip, uint16_t port)
{
    NS_LOG_FUNCTION (this);
    m_peerAddress = ip;
    m_peerPort = port;
}

void
MyTcpEchoClient::SetRemote (Address addr)
{
    NS_LOG_FUNCTION (this << addr);
    m_peerAddress = addr;
}

void
MyTcpEchoClient::DoDispose (void)
{
    NS_LOG_FUNCTION (this);
    Application::DoDispose ();
}



void
MyTcpEchoClient::StartApplication (void)
{
    startingIndex = 4;
    NS_LOG_FUNCTION (this);
    NS_LOG_FUNCTION ("client MSS" << m_maxSegmentSize);

    Config::SetDefault ("ns3::TcpSocket::SegmentSize", UintegerValue
```

```
(m_maxSegmentSize));
        Config::SetDefault ("ns3::TcpSocket::InitialCwnd", UintegerValue (1));


        if (m_socket == 0)
        {
            TypeId tid = TypeId::LookupByName ("ns3::TcpSocketFactory");
            m_socket = Socket::CreateSocket (GetNode (), tid);
            if (m_socket->Bind () == -1)
            {
                NS_FATAL_ERROR ("Failed to bind socket");
            }
            m_socket->Connect (InetSocketAddress (Ipv4Address::ConvertFrom(m_peerAddress),
m_peerPort));
        }

        m_socket->SetRecvCallback (MakeCallback (&MyTcpEchoClient::HandleRead, this));

        ScheduleTransmit (Seconds (0.));
    }

    void
    MyTcpEchoClient::StopApplication ()
    {
        NS_LOG_FUNCTION (this);

        if (m_socket != 0)
        {
            m_socket->Close ();
            m_socket->SetRecvCallback (MakeNullCallback<void, Ptr<Socket> > ());
            m_socket = 0;
        }

        Simulator::Cancel (m_sendEvent);
    }

    void
    MyTcpEchoClient::SetDataSize (uint32_t dataSize)
    {
        NS_LOG_FUNCTION (this << dataSize);

        //
        // If the client is setting the echo packet data size this way, we infer
        // that she doesn't care about the contents of the packet at all, so
        // neither will we.
        //
        delete [] m_data;
        m_data = 0;
        m_dataSize = 0;
        m_size = dataSize;
    }

    uint32_t
    MyTcpEchoClient::GetDataSize (void) const
    {
        NS_LOG_FUNCTION (this);
        return m_size;
    }

    void
    MyTcpEchoClient::SetFill (std::string fill)
    {
        NS_LOG_FUNCTION (this << fill);

        uint32_t dataSize = fill.size () + 1;
```

```
    if (dataSize != m_dataSize)
    {
        delete [] m_data;
        m_data = new uint8_t [dataSize];
        m_dataSize = dataSize;
    }

    memcpy (m_data, fill.c_str (), dataSize);

    //
    // Overwrite packet size attribute.
    //
    m_size = dataSize;
}

void
MyTcpEchoClient::SetFill (uint8_t fill, uint32_t dataSize)
{
    if (dataSize != m_dataSize)
    {
        delete [] m_data;
        m_data = new uint8_t [dataSize];
        m_dataSize = dataSize;
    }

    memset (m_data, fill, dataSize);

    //
    // Overwrite packet size attribute.
    //
    m_size = dataSize;
}

void
MyTcpEchoClient::SetFill (uint8_t *fill, uint32_t fillSize, uint32_t dataSize)
{
    if (dataSize != m_dataSize)
    {
        delete [] m_data;
        m_data = new uint8_t [dataSize];
        m_dataSize = dataSize;
    }

    if (fillSize >= dataSize)
    {
        memcpy (m_data, fill, dataSize);
        return;
    }

    //
    // Do all but the final fill.
    //
    uint32_t filled = 0;
    while (filled + fillSize < dataSize)
    {
        memcpy (&m_data[filled], fill, fillSize);
        filled += fillSize;
    }

    //
    // Last fill may be partial
    //
    memcpy (&m_data[filled], fill, dataSize - filled);
```

```
        //
        // Overwrite packet size attribute.
        //
        m_size = dataSize;
    }



    void MyTcpEchoClient::SetPacketSizes(uint32_t * sizes){
        m_PacketSizes = sizes;
    }

    void MyTcpEchoClient::SetMSS(uint32_t mss){
        NS_LOG_FUNCTION (this<< mss);
        m_maxSegmentSize = mss;
    }


    uint32_t * MyTcpEchoClient::GetPacketSizes(){

        return m_PacketSizes;
    }

    void
    MyTcpEchoClient::ScheduleTransmit (Time dt)
    {
        NS_LOG_FUNCTION (this << dt);
        m_sendEvent = Simulator::Schedule (dt, &MyTcpEchoClient::Send, this);
    }

    void
    MyTcpEchoClient::Send (void)
    {
        NS_LOG_FUNCTION_NOARGS ();

        NS_ASSERT (m_sendEvent.IsExpired ());

        Ptr<Packet> p;
        uint32_t *arr = GetPacketSizes();
        //uint32_t newSize = arr[4-(+startingIndex)];
        uint32_t newSize = arr[4-(startingIndex)];
        startingIndex = startingIndex -2;

        if (m_dataSize)
        {
            p = Create<Packet> (m_data, m_dataSize);
        }
        else
        {
            p = Create<Packet> (newSize);
        }
        Address localAddress;
        m_socket->GetSockName (localAddress);
        m_txTrace (p);

        if (Ipv4Address::IsMatchingType (m_peerAddress))
        {
            m_txTraceWithAddresses (p, localAddress, InetSocketAddress
(Ipv4Address::ConvertFrom (m_peerAddress), m_peerPort));
        }
        else if (Ipv6Address::IsMatchingType (m_peerAddress))
        {
            m_txTraceWithAddresses (p, localAddress, Inet6SocketAddress
(Ipv6Address::ConvertFrom (m_peerAddress), m_peerPort));
```

```
        }

        m_socket->Send (p);
        ++m_sent;

        if (Ipv4Address::IsMatchingType (m_peerAddress))
        {
            NS_LOG_INFO ("At time " << Simulator::Now ().As (Time::S) << " client " <<
+m_clientNo <<" sent " << p->GetSize() << " bytes to " <<
Ipv4Address::ConvertFrom (m_peerAddress) << " port " << m_peerPort);
        }
        else if (Ipv6Address::IsMatchingType (m_peerAddress))
        {
            NS_LOG_INFO ("At time " << Simulator::Now ().As (Time::S) << " client " <<
+m_clientNo <<" sent "<< p->GetSize() << " bytes to " <<
Ipv6Address::ConvertFrom (m_peerAddress) << " port " << m_peerPort);
        }
        else if (InetSocketAddress::IsMatchingType (m_peerAddress))
        {
            NS_LOG_INFO ("At time " << Simulator::Now ().As (Time::S) << " client " <<
+m_clientNo <<" sent " << p->GetSize() << " bytes to " <<
InetSocketAddress::ConvertFrom (m_peerAddress).GetIpv4 () << " port " <<
InetSocketAddress::ConvertFrom (m_peerAddress).GetPort ());
        }
        else if (Inet6SocketAddress::IsMatchingType (m_peerAddress))
        {
            NS_LOG_INFO ("At time " << Simulator::Now ().As (Time::S) << " client " <<
+m_clientNo <<" sent " << p->GetSize() << " bytes to " <<
Inet6SocketAddress::ConvertFrom (m_peerAddress).GetIpv6 () << " port " <<
Inet6SocketAddress::ConvertFrom (m_peerAddress).GetPort ());
        }

        if (m_sent < m_count)
        {
            ScheduleTransmit (m_interval);
        }
    }


    void
    MyTcpEchoClient::HandleRead (Ptr<Socket> socket)
    {
        NS_LOG_FUNCTION (this << socket);
        Ptr<Packet> packet;
        Address from;
        Address localAddress;
        TimestampTag timestampTag;

        NS_LOG_INFO("INSIDE MyTcpEchoClient::HandleRead , startingIndex IS : "<<
startingIndex << " for client " << +m_clientNo );

        while ((packet = socket->RecvFrom (from)))
        {
            if (InetSocketAddress::IsMatchingType (from))
            {
                NS_LOG_INFO ("At time " << Simulator::Now ().As (Time::S) << " client " <<
+m_clientNo <<" received " << packet->GetSize () << " bytes from " <<
InetSocketAddress::ConvertFrom (from).GetIpv4 () << " port " <<
InetSocketAddress::ConvertFrom (from).GetPort ());
            }
            else if (Inet6SocketAddress::IsMatchingType (from))
            {
                NS_LOG_INFO ("At time " << Simulator::Now ().As (Time::S) << " client " <<
+m_clientNo <<" received "  << packet->GetSize () << " bytes from " <<
Inet6SocketAddress::ConvertFrom (from).GetIpv6 () << " port " <<
```

```
Inet6SocketAddress::ConvertFrom (from).GetPort ());
            }
            socket->GetSockName (localAddress);
            m_rxTrace (packet);
            m_rxTraceWithAddresses (packet, from, localAddress);

            uint8_t ttl = 0;
            uint32_t packetSize = packet->GetSize ();
            uint32_t *arr = GetPacketSizes();

            //uint32_t expectedSize = arr[4-(+startingIndex +1)];
            uint32_t expectedSize = arr[4-(startingIndex +1)];
            totalPacketSize = totalPacketSize + packetSize;

            if(totalPacketSize >= expectedSize){
                //uint32_t newSize = arr[4-(+startingIndex)];
                uint32_t newSize = arr[4-(startingIndex)];

                if (newSize < packetSize){
//                    NS_LOG_INFO("newSize < expectedSize==> packet->GetSize(): "<<packet-
>GetSize() << " newSize: " << +newSize );
//                    NS_LOG_INFO("packet->RemoveAtEnd2 abs(packet->GetSize() - +newSize)
"<< abs(packet->GetSize() - +newSize));
                    packet->RemoveAtEnd2 ( abs(packetSize - +newSize) );



                }else if (newSize > packetSize){
//                    NS_LOG_INFO("newSize > expectedSize==> packet->GetSize(): "<<packet-
>GetSize() << " newSize: " << +newSize );
//                    NS_LOG_INFO(" packet->AddPaddingAtEnd abs(packet->GetSize() -
+newSize) "<< abs(packet->GetSize() - +newSize));
                    packet->AddPaddingAtEnd( abs(packetSize - +newSize) );

                }else{

                }

                startingIndex = startingIndex - 2;
                NS_LOG_INFO("startingIndex updated to: "<<startingIndex << " for client " <<
+m_clientNo );

                if (startingIndex < 0) {
//                    NS_LOG_INFO("ttl is 0 client " << +m_clientNo << " aborting packet
cycle! at" << Simulator::Now().GetNanoSeconds());
//                    NS_LOG_INFO("delay for client: "<< +m_clientNo << " "<<
Simulator::Now().GetNanoSeconds() - Seconds(2.0).GetNanoSeconds());
                    std::cout << "ttl is 0 client " << +m_clientNo << " aborting packet
cycle! at" << Simulator::Now().GetNanoSeconds() << std::endl;
                    std::cout << "delay for client: "<< +m_clientNo << " "<<
Simulator::Now().GetNanoSeconds() - Seconds(2.0).GetNanoSeconds() << std::endl;

                    m_socket -> Close();
                    SetIsComplete();
                    return;
                }else {
                    socket->SendTo(packet, 0, from);
                    totalPacketSize = 0;
                }

                if (InetSocketAddress::IsMatchingType(from)) {
                    NS_LOG_INFO("At time "
                                        << Simulator::Now().As(Time::S) << " client " <<
+m_clientNo <<" sent "
                                        << packet->GetSize() << " bytes from "
```

```
                                        << InetSocketAddress::ConvertFrom(from).GetIpv4() <<
" port "
                                        << InetSocketAddress::ConvertFrom(from).GetPort());
                } else if (Inet6SocketAddress::IsMatchingType(from)) {
                    NS_LOG_INFO("At time "
                                        << Simulator::Now().As(Time::S) <<  " client " <<
+m_clientNo <<" sent "
                                        << packet->GetSize() << " bytes from "
                                        << Inet6SocketAddress::ConvertFrom(from).GetIpv6()
<< " port "
                                        << Inet6SocketAddress::ConvertFrom(from).GetPort());
                }

            }



        }
    }

} // Namespace ns3
```

b. MyTcpEchoServer

```
/* -*- Mode:C++; c-file-style:"gnu"; indent-tabs-mode:nil; -*- */ /*
 * Copyright 2007 University of Washington
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation;
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 */

#include "ns3/log.h"
#include "ns3/ipv4-address.h"
#include "ns3/address-utils.h"
#include "ns3/nstime.h"
#include "ns3/inet-socket-address.h"
#include "ns3/socket.h"
#include "ns3/tcp-socket.h"
#include "ns3/simulator.h"
#include "ns3/socket-factory.h"
#include "ns3/packet.h"
#include "ns3/uinteger.h"
#include "my-tcp-echo-server.h"
#include "ns3/tag.h"
#include "string.h"
#include <thread>
#include <iostream>
#include <chrono>
```

```
#include "ns3/config-store-module.h"
#include "ns3/core-module.h"
#include <thread>
#include <iostream>
#include <chrono>


namespace ns3 {

    NS_LOG_COMPONENT_DEFINE ("MyTcpEchoServerApplication");
    NS_OBJECT_ENSURE_REGISTERED (MyTcpEchoServer);

    TypeId     MyTcpEchoServer::GetTypeId (void)
    {
        static TypeId tid = TypeId ("ns3::MyTcpEchoServer")
                .SetParent<Application> ()
                .AddConstructor<MyTcpEchoServer> ()
                .AddAttribute ("Local", "The Address on which to Bind the rx socket.",
                               AddressValue (),
                               MakeAddressAccessor (&MyTcpEchoServer::m_local),
                               MakeAddressChecker ())
                .AddAttribute ("Port", "Port on which we listen for incoming packets.",
                               UintegerValue (9),
                               MakeUintegerAccessor (&MyTcpEchoServer::m_port),
                               MakeUintegerChecker<uint16_t> ())
                .AddAttribute ("ServerNo","ServerNo",
                               UintegerValue (0),
                               MakeUintegerAccessor (&MyTcpEchoServer::SetServerNo),
                               MakeUintegerChecker<uint8_t> ())
                .AddAttribute ("MSS","MSS",
                               UintegerValue (0),
                               MakeUintegerAccessor (&MyTcpEchoServer::SetMSS),
                               MakeUintegerChecker<uint32_t> ())
                .AddTraceSource ("Rx", "A packet has been received",
                                 MakeTraceSourceAccessor (&MyTcpEchoServer::m_rxTrace),
                                 "ns3::Packet::TracedCallback")
                .AddTraceSource ("RxWithAddresses", "A packet has been received",
                                 MakeTraceSourceAccessor
(&MyTcpEchoServer::m_rxTraceWithAddresses),
                                 "ns3::Packet::TwoAddressTracedCallback")
        ;
        return tid;
    }

    MyTcpEchoServer::MyTcpEchoServer ()
    {
        NS_LOG_FUNCTION (this);
    }

    MyTcpEchoServer::~MyTcpEchoServer()
    {
        NS_LOG_FUNCTION (this);
        m_socket = 0;
    }

    void     MyTcpEchoServer::DoDispose (void)
    {
        NS_LOG_FUNCTION (this);
        Application::DoDispose ();
    }

    void MyTcpEchoServer::SetPacketSizes(uint32_t * sizes){
        m_PacketSizes = sizes;
    }
```

```
    uint32_t * MyTcpEchoServer::GetPacketSizes(){

        return m_PacketSizes;
    }

    void MyTcpEchoServer::SetMSS(uint32_t mss){
        NS_LOG_FUNCTION (this<< mss);
        m_maxSegmentSize = mss;
    }

    void     MyTcpEchoServer::SetServerNo(uint8_t serverNo)
    {
        NS_LOG_FUNCTION (this << "serverNo: "<< serverNo);
        m_serverNo = serverNo;
    }


    void     MyTcpEchoServer::StartApplication (void)
    {
        startingIndex = 3;
        NS_LOG_FUNCTION (this);
        NS_LOG_FUNCTION ("m_local: " << m_local);
        NS_LOG_FUNCTION ("m_local" << m_port);
        NS_LOG_FUNCTION ("SERVER MSS" << m_maxSegmentSize);

        Config::SetDefault ("ns3::TcpSocket::SegmentSize", UintegerValue
(m_maxSegmentSize));
        Config::SetDefault ("ns3::TcpSocket::InitialCwnd", UintegerValue (1));


        if (m_socket == 0)
        {
            TypeId tid = TypeId::LookupByName ("ns3::TcpSocketFactory");
            m_socket = Socket::CreateSocket (GetNode (), tid);
            InetSocketAddress local = InetSocketAddress (Ipv4Address::GetAny (), m_port);

//            NS_LOG_FUNCTION ("InetSocketAddress local" << local.GetIpv4());

            if (m_socket->Bind (local) == -1)
            {
                NS_FATAL_ERROR ("Failed to bind socket");
            }

            m_socket->Listen();

//            NS_LOG_INFO("Echo Server local address:  " << m_local << " port: " << m_port
<< " bind: " );
        }

        m_socket->SetRecvPktInfo (true);

        m_socket->SetRecvCallback (
                MakeCallback (&MyTcpEchoServer::HandleRead, this));

        m_socket->SetAcceptCallback (
                MakeNullCallback<bool, Ptr<Socket>, const Address &> (),
                MakeCallback (&MyTcpEchoServer::HandleAccept, this));

        m_socket->SetCloseCallbacks(
                MakeCallback(&MyTcpEchoServer::HandleClose, this),
                MakeCallback(&MyTcpEchoServer::HandleClose, this));
    }

    void MyTcpEchoServer::HandleClose(Ptr<Socket> s1)
```

```
    {
        NS_LOG_INFO(this);
    }

    void MyTcpEchoServer::HandleAccept (Ptr<Socket> s, const Address& from)
    {
        NS_LOG_FUNCTION (this << s << from);
        //NS_LOG_INFO("ACCEPT IN ECHO SERVER from " <<
InetSocketAddress::ConvertFrom(from).GetIpv4());
        s->SetRecvCallback (MakeCallback (&MyTcpEchoServer::HandleRead, this));

    }

    void
    MyTcpEchoServer::StopApplication ()
    {
        NS_LOG_FUNCTION (this);

        if (m_socket != 0)
        {
            m_socket->Close ();
            m_socket->SetRecvCallback (MakeNullCallback<void, Ptr<Socket> > ());
        }
    }


    void
    MyTcpEchoServer::HandleRead (Ptr<Socket> socket)
    {
        NS_LOG_FUNCTION(this << socket);

        Ptr<Packet> packet;
        Address from;
        Address localAddress;

        uint8_t ttl = 0;

        NS_LOG_INFO("--INSIDE MyTcpEchoServer::HandleRead , startingIndex IS : "<<
startingIndex << " for client " << +m_serverNo );

        while (packet = socket->RecvFrom (from))
        {
            uint8_t *msg;
            socket->GetSockName(localAddress);

            m_rxTrace(packet);
            m_rxTraceWithAddresses(packet, from, localAddress);
            uint32_t packetSize = packet->GetSize ();

            if (InetSocketAddress::IsMatchingType(from)) {
                NS_LOG_INFO("At time "
                                    << Simulator::Now().As(Time::S) << " server " <<
m_serverNo <<" received "
                                    << packet->GetSize() << " bytes from "
                                    << InetSocketAddress::ConvertFrom(from).GetIpv4() << "
port "
                                    << InetSocketAddress::ConvertFrom(from).GetPort());
            } else if (Inet6SocketAddress::IsMatchingType(from)) {
                NS_LOG_INFO("At time "
                                    << Simulator::Now().As(Time::S) << " server " <<
m_serverNo <<" received "
                                    << packet->GetSize() << " bytes from "
                                    << Inet6SocketAddress::ConvertFrom(from).GetIpv6() << "
port "
                                    << Inet6SocketAddress::ConvertFrom(from).GetPort());
```

```
            }

            uint32_t *arr = GetPacketSizes();
            uint32_t expectedSize = arr[4-(startingIndex +1)];

            totalPacketSize = totalPacketSize + packetSize;

            if(totalPacketSize >= expectedSize){

                uint32_t newSize = arr[4-(startingIndex)];

                if (newSize < packetSize){

                    packet->RemoveAtEnd2 ( abs(packetSize - newSize) );

                }else if (newSize > packetSize){

                    packet->AddPaddingAtEnd( abs(packetSize - newSize) );

                }else{

                }
                socket->SendTo(packet, 0, from);
                totalPacketSize = 0;
                startingIndex = startingIndex - 2;
                NS_LOG_INFO("startingIndex updated to: "<< startingIndex << " for server "
<< m_serverNo << " on port "<< InetSocketAddress::ConvertFrom(from).GetPort() );

                if (InetSocketAddress::IsMatchingType(from)) {
                    NS_LOG_INFO("At time "
                                        << Simulator::Now().As(Time::S) << " server " <<
m_serverNo <<" sent "
                                        << packet->GetSize() << " bytes from "
                                        << InetSocketAddress::ConvertFrom(from).GetIpv4() <<
" port "
                                        << InetSocketAddress::ConvertFrom(from).GetPort());
                } else if (Inet6SocketAddress::IsMatchingType(from)) {
                    NS_LOG_INFO("At time "
                                        << Simulator::Now().As(Time::S) << " server " <<
m_serverNo <<" sent "
                                        << packet->GetSize() << " bytes from "
                                        << Inet6SocketAddress::ConvertFrom(from).GetIpv6()
<< " port "
                                        << Inet6SocketAddress::ConvertFrom(from).GetPort());
                }

            }

        }
    }

} // Namespace ns3
```

c. ns-3 Runner Script (scenario_tcp.c)

```
#include "ns3/core-module.h"
#include "ns3/point-to-point-module.h"
#include "ns3/internet-module.h"
#include "ns3/applications-module.h"
#include "ns3/mobility-module.h"
#include "ns3/config-store-module.h"
#include "ns3/random-variable-stream.h"
#include "ns3/lte-module.h"
#include <ns3/winner-plus-propagation-loss-model.h>
#include <chrono>
#include <iomanip>
#include <stdlib.h>
#include <ctime>
#include <fstream>
#include <thread>
#include "ns3/flow-monitor-helper.h"
#include "ns3/my-tcp-echo-client.h"
#include "ns3/simulator.h"

//./waf --run "scratch/scenario_tcp --numberOfUEs=10 --mss=200 --verbose=false --
simulationTime=1000 --profile=EcRsa" >> scenario_tcp_10UE.log 2>&1 &

using namespace ns3;

uint32_t nClientApps = 0;
int numberOfUEs = 12;
ApplicationContainer clientApps;
ApplicationContainer serverApps;

NS_LOG_COMPONENT_DEFINE ("NbIoTExample");

void checkClients () {

    int numberOfCompleted=0;

    for (int i=0; i<nClientApps; i++) {
        Ptr <MyTcpEchoClient> client1 = DynamicCast<MyTcpEchoClient>(clientApps.Get(i));
        if(client1->GetIsComplete()){
            numberOfCompleted++;
        }
    }

    if (numberOfCompleted == numberOfUEs){
        NS_LOG_INFO ("numberOfCompleted: "<<numberOfCompleted<<" ENDING SIMULATION"<<
Simulator::Now().As(Time::S));
        Simulator::Stop ();
        Simulator::Destroy ();
        exit (0);
    }

    Simulator::Schedule (Seconds (1), &checkClients);
}


int main (int argc, char *argv[])
{

    Time::SetResolution (Time::NS);
    std::string simName = "NbIoTExample";
    bool verbose = false;
    double cellsize = 1000; // in meters
    int numberOfRemoteHosts = 1;
```

```
    int numberOfENBs = 1;//numberOfUEs/12;
    int simulationTime = 100;
    int mss = 600;
    std::string profile = "EcRsa";

    uint32_t packetSizesEcRsa[4] = {415,2037,80,494};
    uint32_t packetSizesEcEcdsa[4] = {415,1059,80,494};
    uint32_t packetSizesKyberRsa[4] = {1143,2821,80,478};
    uint32_t packetSizesKyberEcdsa[4] = {1143,1843,80,494};
    uint32_t packetSizesKyberFalcon512[4] = {1143,5248,80,494};
    uint32_t packetSizesKyberDilithium[4] = {1143,15296,80,478};
    uint32_t packetSizesKyberSphincs[4] = {1143,25113,80,494};
    uint32_t packetSizes[4] ;

    bool ciot = false;
    bool edt = false;

    CommandLine cmd (__FILE__);
    cmd.AddValue ("numberOfUEs", "numberOfUEs", numberOfUEs);
    cmd.AddValue ("mss", "mss", mss);
    cmd.AddValue ("verbose", "verbose", verbose);
    cmd.AddValue ("simulationTime", "simulationTime in seconds", simulationTime);
    cmd.AddValue ("profile", "profile", profile);

    cmd.Parse (argc, argv);

    if( profile == "EcRsa" || profile == "" ) {
        std::copy(std::begin(packetSizesEcRsa), std::end(packetSizesEcRsa),
std::begin(packetSizes));

    }else if (profile == "EcEc"){
        std::copy(std::begin(packetSizesEcEcdsa), std::end(packetSizesEcEcdsa),
std::begin(packetSizes));

    }else if (profile == "KyberRsa"){
        std::copy(std::begin(packetSizesKyberRsa), std::end(packetSizesKyberRsa),
std::begin(packetSizes));

    }else if (profile == "KyberEc"){
        std::copy(std::begin(packetSizesKyberEcdsa), std::end(packetSizesKyberEcdsa),
std::begin(packetSizes));

    }else if (profile == "Falcon"){
        std::copy(std::begin(packetSizesKyberFalcon512),
std::end(packetSizesKyberFalcon512), std::begin(packetSizes));

    }else if (profile == "Sphincs"){
        std::copy(std::begin(packetSizesKyberSphincs), std::end(packetSizesKyberSphincs),
std::begin(packetSizes));

    }else if (profile == "Dilithium"){
        std::copy(std::begin(packetSizesKyberDilithium),
std::end(packetSizesKyberDilithium), std::begin(packetSizes));

    }else {
        std::copy(std::begin(packetSizesEcRsa), std::end(packetSizesEcRsa),
std::begin(packetSizes));
    }

    uint32_t packetSize = 0;
    uint32_t maxPacketCount = 1;
    Time interPacketInterval = Seconds (1.);
    Time simTime = Seconds(simulationTime);

    LogComponentEnable ("NbIoTExample", LOG_LEVEL_INFO);
```

```
    LogComponentEnable ("MyTcpEchoClientApplication", LOG_LEVEL_INFO);
    LogComponentEnable ("MyTcpEchoServerApplication", LOG_LEVEL_INFO);
    LogComponentEnable ("MyTcpEchoClientApplication", LOG_LEVEL_FUNCTION);
    LogComponentEnable ("MyTcpEchoServerApplication", LOG_LEVEL_FUNCTION);

    if (verbose) {
        LogComponentEnable("LteUeRrc", LOG_LEVEL_INFO);
        LogComponentEnable("LteEnbRrc", LOG_LEVEL_INFO);
        LogComponentEnable("LteEnbMac", LOG_LEVEL_INFO);
        LogComponentEnable("LteUeMac", LOG_LEVEL_INFO);
        LogComponentEnable ("LteUePhy", LOG_LEVEL_INFO);
        LogComponentEnable("LteUeRrc", LOG_LEVEL_FUNCTION);
        LogComponentEnable("LteEnbRrc", LOG_LEVEL_FUNCTION);
        LogComponentEnable("LteEnbMac", LOG_LEVEL_FUNCTION);
        LogComponentEnable("LteUeMac", LOG_LEVEL_FUNCTION);
        LogComponentEnable ("LteUePhy", LOG_LEVEL_FUNCTION);
    }

    NS_LOG_INFO ("numberOfUEs " << numberOfUEs);
    NS_LOG_INFO ("simulationTime " << simulationTime);
    NS_LOG_INFO ("profile " << profile);
    NS_LOG_INFO ("mss " << mss);

    NS_LOG_INFO ("Create LTE.");
    Ptr<LteHelper> lteHelper = CreateObject<LteHelper> ();
    Ptr<PointToPointEpcHelper> epcHelper = CreateObject<PointToPointEpcHelper> ();
    lteHelper->SetEpcHelper (epcHelper);
    lteHelper->EnableRrcLogging ();
    lteHelper->SetEnbAntennaModelType ("ns3::IsotropicAntennaModel");
    lteHelper->SetUeAntennaModelType ("ns3::IsotropicAntennaModel");
    lteHelper->SetAttribute ("PathlossModel", StringValue
("ns3::WinnerPlusPropagationLossModel"));
    lteHelper->SetPathlossModelAttribute ("HeightBasestation", DoubleValue (50));
    lteHelper->SetPathlossModelAttribute ("Environment", EnumValue (UMaEnvironment));
    lteHelper->SetPathlossModelAttribute ("LineOfSight", BooleanValue (false));
    Config::SetDefault ("ns3::LteHelper::UseIdealRrc", BooleanValue (true));
    Config::SetDefault ("ns3::LteSpectrumPhy::CtrlErrorModelEnabled", BooleanValue (false));
    Config::SetDefault ("ns3::LteSpectrumPhy::DataErrorModelEnabled", BooleanValue (false));

    Ptr<Node> pgw = epcHelper->GetPgwNode ();
    // Create a single RemoteHost
    NodeContainer remoteHostContainer;
    remoteHostContainer.Create (numberOfRemoteHosts);
    Ptr<Node> remoteHost = remoteHostContainer.Get (0);
    InternetStackHelper internet;
    internet.Install (remoteHostContainer);

    NS_LOG_INFO ("Create Internet.");
    PointToPointHelper p2ph;
    p2ph.SetDeviceAttribute ("DataRate", DataRateValue (DataRate ("100Gb/s")));
    p2ph.SetDeviceAttribute ("Mtu", UintegerValue (1500));
    p2ph.SetChannelAttribute ("Delay", TimeValue (MilliSeconds (10)));
    p2ph.EnablePcapAll(profile +"_"+std::to_string(numberOfUEs) +"_mss"+std::to_string(mss)
+ "_scenario", true);
    NetDeviceContainer internetDevices = p2ph.Install (pgw, remoteHost);
    Ipv4AddressHelper ipv4h;
    ipv4h.SetBase ("1.0.0.0", "255.0.0.0");
    Ipv4InterfaceContainer internetIpIfaces = ipv4h.Assign (internetDevices);
    Ipv4Address remoteHostAddr = internetIpIfaces.GetAddress (1);
    Address serverAddress = Address(internetIpIfaces.GetAddress (1));
    Ipv4StaticRoutingHelper ipv4RoutingHelper;
    Ptr<Ipv4StaticRouting> remoteHostStaticRouting = ipv4RoutingHelper.GetStaticRouting
(remoteHost->GetObject<Ipv4> ());
    remoteHostStaticRouting->AddNetworkRouteTo (Ipv4Address ("7.0.0.0"), Ipv4Mask
("255.0.0.0"), 1);
```

```
    NS_LOG_INFO ("Set Position");
    NodeContainer enbNodes;
    enbNodes.Create (numberOfENBs);
    Ptr<ListPositionAllocator> positionAlloc = CreateObject<ListPositionAllocator> ();
    positionAlloc->Add (Vector (cellsize/2, cellsize/2, 25)); //last 25 parameter is
probably height of tower?

    MobilityHelper mobilityEnb;
    mobilityEnb.SetMobilityModel("ns3::ConstantPositionMobilityModel");
    mobilityEnb.SetPositionAllocator(positionAlloc);
    mobilityEnb.Install(enbNodes);

    NodeContainer ueNodes;
    ueNodes.Create (numberOfUEs);
    Ptr<ListPositionAllocator> positionAllocUe = CreateObject<ListPositionAllocator> ();


    ObjectFactory pos;
//    pos.SetTypeId ("ns3::UniformDiscPositionAllocator");
    pos.SetTypeId ("ns3::RandomDiscPositionAllocator");
    pos.Set ("X", StringValue (std::to_string(cellsize/2)));
    pos.Set ("Y", StringValue (std::to_string(cellsize/2)));
    pos.Set ("Z", DoubleValue (5)); //UEs at 5 meters
    pos.Set ("Rho", StringValue ("ns3::UniformRandomVariable[Min=0|Max=30]"));
    Ptr<PositionAllocator> m_position = pos.Create ()->GetObject<PositionAllocator> ();

    for (uint32_t i = 0; i < numberOfUEs; ++i){
        Vector position = m_position->GetNext ();
        positionAllocUe->Add (position);
        std::cout << "position: " << position.x << "," << position.y << "," << position.z <<
std::endl;
    }

    NS_LOG_INFO ("Set Mobility.");
    MobilityHelper mobilityUe;
    mobilityUe.SetMobilityModel ("ns3::ConstantPositionMobilityModel");
    mobilityUe.SetPositionAllocator(positionAllocUe);
    mobilityUe.Install (ueNodes);


    // Install LTE Devices to the nodes
    NetDeviceContainer enbLteDevs = lteHelper->InstallEnbDevice (enbNodes);
    NetDeviceContainer ueLteDevs = lteHelper->InstallUeDevice (ueNodes);

    // Install the IP stack on the UEs
    internet.Install (ueNodes);
    Ipv4InterfaceContainer ueIpIface;
    ueIpIface = epcHelper->AssignUeIpv4Address (NetDeviceContainer (ueLteDevs));

    // Assign IP address to UEs, and install applications
    for (uint32_t u = 0; u < ueNodes.GetN (); ++u){
        Ptr<Node> ueNode = ueNodes.Get (u);
        // Set the default gateway for the UE
        Ptr<Ipv4StaticRouting> ueStaticRouting = ipv4RoutingHelper.GetStaticRouting (ueNode-
>GetObject<Ipv4> ());
        ueStaticRouting->SetDefaultRoute (epcHelper->GetUeDefaultGatewayAddress (), 1);
    }

    // Install and start applications on UEs and remote host
    uint16_t ulPort = 2000;

    for (uint16_t i = 0; i < numberOfUEs; i++){
        NS_LOG_INFO ("Creating Applications. " << i);
```

```
        lteHelper->AttachSuspendedNb(ueLteDevs.Get(i), enbLteDevs.Get(i % numberOfENBs));
        //std::this_thread::sleep_for(std::chrono::milliseconds(100));
        Ptr<LteUeNetDevice> ueLteDevice = ueLteDevs.Get(i)->GetObject<LteUeNetDevice> ();

        Ptr<LteUeRrc> ueRrc = ueLteDevice->GetRrc();

        ++ulPort;
        MyTcpEchoServerHelper server (serverAddress, ulPort, mss);
        serverApps.Add(server.Install (remoteHost));
        server.SetPacketSizes(serverApps.Get (i),packetSizes);
        server.SetAttribute("ServerNo", UintegerValue(i));

        MyTcpEchoClientHelper ulClient (remoteHostAddr, ulPort);
        ulClient.SetAttribute ("Interval", TimeValue(interPacketInterval));//?
        ulClient.SetAttribute ("MaxPackets", UintegerValue (maxPacketCount));//?
        ulClient.SetAttribute ("PacketSize", UintegerValue(packetSize));//?
        ulClient.SetAttribute ("ClientNo", UintegerValue(i));//?
        ulClient.SetAttribute("MSS", UintegerValue(mss));

        ulClient.SetAttribute ("TagCounter", UintegerValue (sizeof(packetSizes) /
sizeof(int)));
        clientApps.Add (ulClient.Install (ueNodes.Get(i)));
        ulClient.SetPacketSizes(clientApps.Get (i),packetSizes);

    }
    nClientApps = clientApps.GetN ();
    Simulator::Schedule (Seconds (0), &checkClients);

    Ptr<FlowMonitor> flowMonitor;
    FlowMonitorHelper flowHelper;
    flowMonitor = flowHelper.InstallAll();

    NS_LOG_INFO ("Start sim with "<<nClientApps<< " client apps." );

    serverApps.Start (Seconds (1.0));
    serverApps.Stop (simTime);

    clientApps.Start (Seconds (2.0));
    clientApps.Stop (simTime);

    NS_LOG_INFO ("Run Simulation.");
    Simulator::Stop (simTime);
    Simulator::Run ();
    flowMonitor->SerializeToXmlFile(profile +"_"+std::to_string(numberOfUEs) +"_mss"+
std::to_string(mss) +"_flowMonitor.xml", true, true);

}
```

B. INSTALLATION and DIGITAL SIGNATURE GENERATION SCRIPTS

a. liboqs and OQS-OpenSSL Setup on MACOS

Install build / compile / debug tools for C language, standard OpenSSL binary using MACOS's HomeBrew package manager, and pytest library using Python3's package manager pip3:

```
brew install cmake ninja openssl@1.1 wget doxygen graphviz astyle valgrind
pip3 install pytest pytest-xdist pyyaml
```

Download source codes for liboqs and OQS-OpenSSL:

```
git clone -b main https://github.com/open-quantum-safe/liboqs.git
git clone -b OQS-OpenSSL_1_1_1-stable https://github.com/open-quantum-
safe/openssl.git
```

In this step we build liboqs. For this task we need standard OpenSSL binary to build liboqs, so we export its path as OPENSSL_ROOT_DIR environment variable. Another environment variable we need to set is DCMAKE_INSTALL_PREFIX which creates an install directory for liboqs header and library files inside the OQS-OpenSSL directory:

```
export OPENSSL_ROOT_DIR=/opt/homebrew/
cd liboqs
mkdir build && cd build
cmake -GNinja .. -DCMAKE_INSTALL_PREFIX=/Users/myuser/OQS-OpenSSL_1_1_1-
stable/oqs
ninja
```

Finally, configure and build OQS-OpenSSL with required parameters for ARM-64 instructions.

```
cd /Users/myuser/OQS-OpenSSL_1_1_1-stable/
export LD_LIBRARY_PATH=/opt/homebrew/Cellar/openssl@1.1/1.1.1m/lib/
./Configure no-shared darwin64-arm64-cc
make -j
```

b. Certificate Chain Creation for RSA-2048 on MACOS

Export OQS-OpenSSL program path, then create a root certificate and private key

```
cd /Users/myuser/OQS-OpenSSL_1_1_1-stable/

export DYLD_LIBRARY_PATH=/Users/myuser/OQS-OpenSSL_1_1_1-stable/oqs/lib/

apps/openssl req -x509 -new -newkey rsa:2048 -keyout rsa2048_root.key -out
rsa2048_root.crt -nodes -subj "/CN=oqstest root" -days 365 -config
apps/openssl.cnf
```

Create an Intermediate CA certificate signing request and private key

```
apps/openssl req -new -newkey rsa:2048 -keyout rsa2048_CA.key -out
rsa2048_CA.csr -nodes -subj "/CN=oqstest CA" -days 365 -config
apps/openssl.cnf
```

Sign Intermediate CA certificate signing request with the root certificate

```
apps/openssl x509 -req -in rsa2048_CA.csr -out rsa2048_CA.crt -CA
rsa2048_root.crt -CAkey rsa2048_root.key -CAcreateserial -days 365
```

Create server certificate signing request and private key

```
apps/openssl req -new -newkey rsa:2048 -nodes -keyout rsa2048_srv.key -out
rsa2048_srv.csr -nodes -subj "/CN=oqstest server" -config apps/openssl.cnf
```

Sign server certificate signing request with the Intermediate CA certificate

```
apps/openssl x509 -req -in rsa2048_srv.csr -out rsa2048_srv.crt -CA
rsa2048_CA.crt -CAkey rsa2048_CA.key -CAcreateserial -days 365
```

c. TLS 1.3 Handshake Test Cases

Terminal window 1

```
apps/openssl s_server -key rsa2048_srv.key -cert rsa2048_srv.crt -cert_chain
rsa2048_CA.crt -accept 44330 -www -tls1_3
```

Terminal window 2

```
apps/openssl s_client -groups kyber512 -CAfile rsa2048_root.crt -connect
localhost:44330
```

With the first command, a TLS 1.3 server instance starts on TCP port 44330. *s_server* program requires some information such as RSA private key, RSA server certificate and the certificate chain which makes verification of the server certificate possible on the client side. When *s_client* asks for a new session, server certificate is sent to the client along with the certificate chain. *s_client* on the other hand, needs to know the root certificate to verify the certificate chain which is provided with -*CAfile* parameter. The client also identifies the key exchange protocol that it prefers with -*groups* parameter, which would otherwise be ECDHE as default.

d. ns-3 and NB-LENA Setup on Ubuntu

1. Install git, C language make / build / install /debug tools, and python3

```
apt install g++ python3 cmake ninja-build git gdb valgrind
```

2. Download ns-3 3.32 and extract from zip

```
git clone https://gitlab.com/nsnam/ns-3-allinone.git
./download.py -n ns-3.32
tar xjf ns-allinone-3.30.tar.bz2
```

3. Configure and Install

```
./waf configure
./waf
```

4. Replace files under *"./ns-allinone-3.32/ns-3.32/src/LTE"* with NB-LENA source code for NB-IoT support.