3-26-2020

NewHope: A Mobile Implementation of a Post-Quantum Cryptographic Key Encapsulation Mechanism

Jessica A. Switzler

NewHope: A Mobile Implementation of a Post-Quantum Crytographic Key Encapsulation Mechanism

THESIS

Jessica A. Switzler

AFIT-ENG-MS-20-M-063

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT-ENG-MS-20-M-063

NewHope: A Mobile Implementation of a Post-Quantum Crytographic Key Encapsulation Mechanism

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

in Partial Fulfillment of the Requirements for the

Degree of Master of Science in Cyber Operations

Jessica A. Switzler, B.S.

March 26, 2020

AFIT-ENG-MS-20-M-063

NewHope: A Mobile Implementation of a Post-Quantum Crytographic Key Encapsulation Mechanism

THESIS

Jessica A. Switzler, B.S.

Committee Membership:

Laurence D. Merkle, Ph.D
Chair

Maj. Richard Dill, Ph.D
Member

Barry E. Mullins, Ph.D
Member

AFIT-ENG-MS-20-M-063

Abstract

NIST anticipates the appearance of large-scale quantum computers by 2036 [34], which will threaten widely used asymmetric algorithms, National Institute of Standards and Technology (NIST) launched a Post-Quantum Cryptography Standardization Project to find quantum-secure alternatives. NewHope post-quantum cryptography (PQC) key encapsulation mechanism (KEM) is the only Round 2 candidate to simultaneously achieve small key values through the use of a security problem with sufficient confidence its security, while mitigating any known vulnerabilities.

This research contributes to NIST project's overall goal by assessing the platform flexibility and resource requirements of NewHope KEMs on an Android mobile device. The resource requirements analyzed are transmission size as well as scheme runtime, central processing unit (CPU), memory, and energy usage. Results from each NewHope KEM instantiations are compared amongst each other, to a baseline application, and to results from previous work. NewHope PQC KEM was demonstrated to have sufficient flexibility for mobile implementation, competitive performance with other PQC KEMs, and to have competitive scheme runtime with current key exchange algorithms.

Table of Contents

Page

Page

Page

List of Figures

List of Tables

Table Page

Table Page

NewHope: A Mobile Implementation of a Post-Quantum Crytographic Key Encapsulation Mechanism

I. Introduction

1.1 Motivation

The National Institute of Standards and Technology (NIST) anticipates that large-scale quantum computers will become available by 2036 [34]. This raises an important issue within the cryptographic community: through the use of those quantum computers, the security of the most widely-used asymmetric cryptographic algorithms will be undermined. This is because the underlying primitives are based on mathematical problems that can be solved efficiently using quantum computing.

Since our current asymmetric cryptography will be insecure with the advent of large-scale quantum computers, it has become important to find cryptographic algorithms that are hardened against quantum attacks. In fact, switching to quantum-proof cryptosystems could take precedence over transitioning our current cryptosystems to higher bit-levels of security [34]. For this reason, post-quantum cryptography (PQC) has become a prevalent area of research that considers the development of algorithms that are thought to be able to withstand attacks from both classical and quantum computers [34]. The algorithms used in PQC are often referred to as quantum-proof, quantum-secure, or quantum-resistant.

Just like classical attacks, the quantum attack surface contains a variety of devices ranging from desktop computers to internet of things (IoT) devices, industrial control systems (ICS), mobile phones, and many more networked devices. Given the number

of devices that could potentially be targeted, it is important to consider and test how these quantum-secure algorithms will be implemented on all devices. However, a relatively small amount of research has been done on mobile phone implementations of quantum-resistant cryptosystems. This is despite the fact that mobile phones are considered to be the most common communication devices in history [4] and have uses that range from personal to confidential communication. In order to secure these communication avenues from the threat large-scale quantum computers present, this gap within current research needs to be bridged.

1.2 Problem Background

As stated above, the security of the most widely-used asymmetric cryptographic algorithms will be affected by the presence of large-scale quantum computers due to the mathematical problems on which these algorithms are based. For example, the security of the Rivest-Shamir-Adleman (RSA) algorithm is based on the apparent difficulty of the integer factoring problem [34]. Other examples include the Diffie-Hellman key exchange (DHKE) and elliptic-curve cryptography (ECC), both of which obtain their security from the difficulty of the discrete logarithm problem over various groups [34]. Through the use of Shor's algorithms [97], large-scale quantum computers will be able to solve each of these problems in polynomial time.

The risk that large-scale quantum computers present to asymmetric cryptography is a driving force behind the need to finding quantum-secure alternatives to current algorithms. Some of the most crucial communication protocols currently rely on public-key encryption (PKE), digital signatures, and key exchange, all of which rely in turn on asymmetric cryptographic algorithms [34].

The need to find PQC public-key algorithms was the impetus for NIST to start their Post-Quantum Cryptography Standardization project [34]. The goal behind

NIST's project is to standardize PQC asymmetric algorithms that will become publicly available before the emergence of large-scale quantum computers. These algorithms need to be designed and implemented to be interoperable with existing networks and communication protocols [34]. Many different variations of PQC asymmetric algorithms are being produced for this standardization effort, at the end of which NIST will standardize at least one PKE scheme, one key encapsulation mechanism (KEM), and one digital signature algorithm from among those submitted [34].

The submitted algorithms need to be assessed with respect to a number of requirements. First, they need to be verified to function properly and have a mathematically sound basis. Also, the cryptographic primitive for each algorithm should belong to a cryptographic family that has presumed hardness against quantum attacks and has undergone enough research to instill confidence in that presumed hardness. Next, the security of each submitted cryptosystem needs to be tested against classical attacks to expose any underlying vulnerability that might be exploited. Finally, each cryptosystem needs to be analyzed for its efficiency and interoperability to ensure that it will function given current communication protocols. NIST has addressed many of these requirements by requiring that each submission include a document outlining the cryptosystem and addressing the above specifications [83]. More detail about NIST's PQC Standardization Project is presented in Section 2.5.

Increasingly since the launch of the NIST PQC Standardization Project, research has been undertaken on the implementation of quantum-secure algorithms. The research efforts range from cryptanalysis of selected algorithms to specific device implementations. However, there is little research in the field of quantum-secure mobile devices. One exception is the work of Malina et al. [73], in which the authors implement two PQC KEM algorithms on an Android device: NewHope-Usenix [10] and MSR LN16 [68]. The authors implement the former in pure Java and the latter using

a native C library and a Java Native Interface (JNI). They compare the transmission size and runtime requirements of the two implementations. A more detailed description of this work is provided in Section 2.6.

The presumed security of both the NewHope-Usenix and MSR LN16 KEMs against quantum attacks is based on the presumed hardness of the ring learning with errors (RLWE) problem from the lattice-based cryptography family. This is also the case for the NewHope KEM, which is the only one of the three in Round 2 of NIST's Standardization Project.

The NewHope cryptosystem is built into four different KEM instantiations: two that are provably secure against chosen plaintext attack (CPA) and two that are provably secure against known chosen ciphertext attack (CCA). The four instantiations are referred to as NewHope512-CCA-KEM, NewHope1024-CCA-KEM, NewHope512-CPA-KEM, and NewHope1024-CPA-KEM, in which the 512 and 1024 parts of the names indicate choices for the dimension parameter. The NewHope specifications and documentation [8] justify that the 512 implementations are NIST security Level 1, whereas the 1024 implementations are Level 5. Section 2.10 provides a thorough description of NewHope implementations and Section 2.5 provides more information on NIST security levels.

1.3 Research Objectives

This research effort contributes to the current PQC research by assessing the feasibility, performance, and resource constraints of mobile implementations of a PQC KEM. The goal is to determine the extent to which current PQC KEMs can be used to secure mobile phones. Of primary interest is determining whether any resource constraints exist that might prohibit such use, including bandwidth, response time, CPU time, memory, or energy. This necessitates the design development, testing,

and analysis of one or more PQC KEMs. The specific research questions that this research effort addresses are:

1. How feasible is PQC KEM implementation on mobile devices?
2. How does NewHope[1] compare qualitatively to other NIST PQC Standardization Project Round 2 KEMs?
3. What performance tradeoffs and limitations are present among the NewHope instantiations?
4. How does the amount of Java code implemented affect the runtime of the native KEM scheme?

The question regarding performance tradeoffs and limitations is supported by several specific hypotheses:

1. How does the native scheme runtime, OnClick runtime, CPU usage, and transmission size differ between each NewHope instatiation?
2. How does average native scheme runtime and transmission size in bytes for NewHope compare to available work?
3. What resource constraints, if any, are presented when implementing a PQC KEM?

1.4 Approach

This section describes how the research objectives for this effort were accomplished. First, a KEM is selected from Round 2 of NIST Standardization Project. Next, the original codebase was downloaded, built, and tested for correctness. The

[1]The NewHope cryptosystem was selected for implementation based on comparisons among the PQC KEMs in Round 2 of the NIST PQC Standardization Project (see Section 2.7).

codebase was tested for correctness by comparing the output files to the known answer tests (KAT) that were provided by NewHope creators. The decision to implement the NewHope codebase as a native C library utilizing a JNI to call some NewHope functions from within the created Java code was a design choice that was made. This choice was made because the only available Java codebase was from NewHope-Usenix [7] and since then the link provided to that codebase no longer works. The next step was to modify the original NewHope codebase and to develop the Android application. After the codebase is complete, the application was tested using Android Profiler within Android Studio, and this allows for the evaluation of the usage of the central processing unit (CPU), memory and energy. Also, the application was evaluated using the transmission byte requirements as well as the runtime of the KEMs. With these results all four of the NewHope implementations will be compared to each other, and to those reported in the work by Malina et al. [73]. The comparison made between implementation involved an evaluation of native scheme runtime, Java code runtime, transmission byte requirements, and resource consumptions.

1.5 Assumptions and Limitations

This section outlines the assumptions and limitations present throughout this research effort.

Throughout this research effort it is assumed that whatever actions are taken for implementing one of the NewHope KEMs is also carried through to all the other NewHope implementations. Another assumption made is that when referencing post-quantum bit-level security, these security values are presumed and the calculations of bit-level security cannot be concrete.

Something that is not within the scope of this research effort is networking the produced Android application that implements the NewHope PQC KEMs. It is

assumed that this would not have an effect on the mobile implementation of NewHope because these PQC KEMs are designed to interoperate with existing communication protocols.

1.6 Document Overview

This section is designated to be a road-map to the contents within this document.

Chapter II provides the background information needed to understand this research effort, and Chapter II contains relevant literature reviews. Chapter III contains the methodology that this research effort follows along with the design choices that were made. Chapter IV contains the test results from the mobile implementations of NewHope, and an analysis of the test results along with comparisons to the work done by Malina et al. [73]. Finally, Chapter V provides the conclusions achieved from this research effort.

II. Background and Literature Review

2.1 Chapter Overview

This chapter outlines the preliminary research needed in order to understand post-quantum cryptography (PQC), NewHope cryptosystems, implementations of PQC algorithms, and C/Java Native Interface (JNI) Android implementations. Section 2.2 provides an overview of quantum computers and quantum computing. This is important background knowledge because the security of current asymmetric cryptographic algorithms is anticipated to be effected by utilizing quantum computing on a large-scale quantum computer. Also, this section provides a definition of the term large-scale quantum computer, which is commonly used throughout this research field. This is followed by Section 2.3, which provides two quantum algorithms and the effects these algorithms will have on current cryptography when implemented on a large-scale quantum computer. In Section 2.4, PQC is explained along with definitions of the families from which PQC algorithms obtain their cryptographic primitives. NIST has been a leading contributor to the study of PQC with the launch of their Post-Quantum Cryptography Standardization project, an overview of this project is provided in Section 2.5. Section 2.6 contains a literature review of available work focusing on PQC implementations on mobile phones and internet of things (IoT) devices. This provides important knowledge on how PQC algorithms can be implemented on these devices, and provides knowledge into the ways of which implementation can be improved. Ideally, this section would be dedicated solely to a literature review of PQC mobile phone implementations but due to the lack of available literature it was expanded to include IoT implementations as well. Section 2.7 provides an overview of all key encapsulation mechanism (KEM)s that have been progressed to round two of NIST's Post-Quantum Cryptography Standardization project. This provides an important

overlook into the reasoning behind selecting the NewHope PQC KEM for implementation. Section 2.8 provides an overview of lattice-based cryptography along with the lattice-based problems that are used as cryptographic primitives. Within this section the ring learning with errors (RLWE) problem is defined which is the problem that is the basis for NewHope. Section 2.9 provides a literature review on available research on the cryptanalysis of lattice-based cryptosystems. This provides valuable information into the possible vulnerabilities presented when implementing lattice-based cryptosystems. Section 2.10 provides background knowledge on NewHope as well as cryptosystem details. Finally, Section 2.11 provides information on JNIs because the NewHope mobile implementations is achieved by creating a NewHope codebase as a native C library and referencing the native library from Java utilizing a JNI.

2.2 Quantum Computers and Quantum Computing

When discussing post-quantum cryptography and how to achieve quantum-secure cryptography, it is vital to provide a foundational overview of quantum computers and quantum computing. Quantum computers have a negative impact on our cryptographic algorithms and have some unknown uses where quantum algorithms are used in order to speed up some current algorithms. This means that through quantum speedup some current algorithms can be made more efficient and built to feasibly break cryptographies. Quantum computers provide a machine that can be utilized to complete computations that would be infeasible to run within a reasonable amount of time on a classical computer. This section will be used to provide an overview of quantum computers, provide a definition of large-scale quantum computers and discuss the current state of quantum computers.

A quantum computer is defined as a machine that utilizes quantum mechanics to carry out computations [38]. A quantum computer uses a bit known as a quantum

bit (qubit). Qubits are two-level quantum systems [95]. Some examples of two-level quantum systems are the two states of a spin 1/2 particle, like the ground and excited states of an atom [38]. These two-level quantum systems hold properties which exhibit the power of a quantum computing.

Some of the most common quantum mechanics properties mentioned in quantum computing are superposition, entanglement, and decoherence times. Superposition refers to the ability of a qubit to exist in a combination of multiple states; it is not until the qubit is measured that it is seen in a single state [82]. Entanglement is the ability for the states of two qubits to become dependent meaning that if the state of one qubit is measured there is knowledge gained about the state of the entangled qubit [82]. Within quantum computing entanglement is a powerful property because it can allow for many parallel operations to take place [82]. Decoherence occurs when the quantum wave function (mathematical description of the quantum systems quantum state) collapses, which means that the quantum properties, like superposition or entanglement of the particle, would no longer be sustained within the quantum system [38].

Quantum computers are built with the basis being generating and managing qubits. Some quantum computers are built using superconducting circuits which is achieved by cooling the circuit to temperatures low enough to allow for superconductivity. Devoret et al. [37] describe superconducting qubits as a combination of two robust phenomena: superconductivity, a frictionless flow of electrical fluid (electric current where the flow cannot be resolved into individual electrons) through a metal at a temperature below the superconducting phase transition, and the Josephson effect, which endows the circuit with nonlinearity without introducing dissipation or dephasing. IBM, Google, and Rigetti Computing utilize superconducting circuits within their quantum computers [37]. Similiarly, ion traps can be utilized to build

quantum computers. According to DiVincenzo [38], the quantum computer holds the qubits in pairs of energy levels of ions held in a linear electromagnetic trap. Finding a well designed qubit that holds its coherence longer means that there is more time for gate operations to occur and more computations can be accomplished before the system decoheres [38].Since a basic understanding of a quantum computers has been achieved the discussion can transition to how quantum computing works.

Quantum computing is accomplished by utilizing one of many different qubit implementations where a quantum computer can encode information as 0's, 1's, or both at the same time. It is through the use of this superposition phenomena and other quantum mechanical phenomena, such as entanglement or tunneling, that allows a quantum computer to manipulate many combinations of states simultaneously [82].

However, these quantum mechanic phenomena also present some challenges for quantum computing. Due to decoherence, the collapsing of the wave function caused by time or interference, in order for a calculation to be reliable the calculations must be completed before the wave function decoheres [38]. This presents a time limit for computation to take place, and this time limit presents a possible detrimental constraint on quantum computers/quantum computing. Due to superposition, there is no way to measure the state of the system without decohering the wave function; this concludes that there is no method to measure qubits individually [82].

Despite these challenges, quantum computers possess a power that cannot be matched by classic computers. Neilsen and Chuang [82] stated that quantum computers offer an essential speedup advantage over classic computers. This speed up is so significant that many researchers believe that no amount of progression made in classical computation could overcome the power differential between a classic and quantum computer [82]. This leaves endless possibilities for quantum computing: from quantum speed-up of existing classic algorithms, which Grover's search algo-

rithm [52] is conjectured to be useful for this, to the development of new quantum algorithms that have the capacity to alter other aspects of cryptographies or solve problems quicker than classical computers.

Due to the potential of quantum computers not being fully developed, it is unknown to what extent or how much of an affect large-scale quantum computers will have on our current cryptosystems. Due to uncertainties in how well large-scale quantum computers will perform or advance, it makes it difficult to provide a bit-level security on many of the PQC algorithms being developed [34]. Due to this difficulty it can even become easy to underestimate or overestimate the security of quantum-secure cryptosystems.

Currently, there is a significant amount of research being done to obtain advancements for quantum computers. As of September 2019, IBM has a quantum computer which possesses a 53 qubit processor [80] and Google has the Bristlecone quantum computing processor which contains 72 qubits [61]. Also, D-Wave is anticipating that by mid-2020, it will have a functioning 5,000 qubit quantum annealing computer [93].

D-Wave quantum computers are quantum annealing computers, which utilize quantum annealing which is a metaheuristic for finding global minimum [60]. Typically when quantum computers are mentioned throughout literature they are referencing a quantum computer that utilizes quantum gates to carry out computations, which is not a quantum annealing computer. Annealing is much more limited in uses than gate-based quantum computers. Specifically, quantum annealing computers will never be utilized to run Shor's algorithms [97] and is successful at solving optimization problems [99]. The quantum computers by IBM and Google are quantum computers know as universal quantum computers [99] which are gate-based quantum computers.

Often in literature, references are made to large-scale quantum computers, but they are not formally defined within the literature. Through the examination of

references made to large-scale quantum computers, large-scale quantum computers provide significant improvements in the number of qubits implemented to current quantum computers. Providing more qubits allows for more calculations to be done before the system decoheres, meaning quantum computations that are carried out are more extensive. However, none of the literature is specific on the range of qubits that might be needed to create a large-scale computer. IBM research released an info-graphic on the three different types of quantum computing [60] and it states that for a universal quantum computer it is anticipated by researchers that it will contain more than $100,000$ physical qubits. In the work done by Harrow and Montanaro [55], they stated that $4,000$ qubits would be required to factor a 2048-bit number. With that in mind, a large-scale quantum computer is a quantum computer that contains enough qubits to carry out useful quantum computations and is anticipated to contain $4,000$ qubits, up to over $100,000$. However, like Dyakonov [41], there are still researchers that believe large-scale quantum computers will never overcome the physical limitations of decoherence times.

2.3 Effect of Popular Quantum Algorithms

This section is dedicated to discussing some of the most commonly known quantum computing algorithms, Shor's algorithms [97] and Grover's algorithm [52]. The selection of these algorithms is due to popularity, but more importantly the role these algorithms plays on the security of current cryptographic algorithms. According to NIST [34], the most crucial communication protocols rely on three core cryptographic functionalities: public-key encryption (PKE), digital signatures, and key exchange. The primary cryptosystems implementing these functionalities are Diffie-Hellman key exchange (DHKE), Rivest-Shamir-Adleman (RSA), and elliptic curve cryptosystems [34]. When utilized on a large-scale quantum computer, Shor [97]

proves that his algorithms render the most common public-key cryptosystems insecure. Shor's algorithms break the security of all public-key algorithms that are reliant on integer factorization and discrete logarithmic problems, which includes DHKE, RSA, and elliptic-curve cryptography (ECC) [34]. Another algorithm that will effect some cryptographic algorithms is Grover's algorithm [52] that is proven to offer a quadratic speed up of unstructured search problems. Although Grover's algorithm does not threaten to render any current cryptographic technologies obsolete, it does require the use of larger keys for some current cryptosystems, including common symmetric key algorithms like Advanced Encryption Standard (AES). In NIST's Report on Post-Quantum Cryptography [34] there is a table provided that indicates the effects large-scale quantum computers will have on current cryptographic algorithms, a recreation can be seen in Table 1. Some acronyms that are not previously mentioned but are needed to understand the table are Elliptic-Curve Diffie-Hellman (ECDH) and elliptic-curve digital signature algorithm (ECDSA) which are ECC used in asymmetric cryptosystems, as well as, digital signature algorithm (DSA).

Table 1: The effects large-scale quantum computers will have on common cryptographic algorithms based on National Institute of Standards and Technology (NIST) PQC Report [34]

Impact of Large-Scale Quantum Computers On Current Cryptography			
Cryptographic Algorithm	**Type**	**Purpose**	**Impact**
AES	Symmetric Key	Encryption	Larger key sizes needed
SHA-2, SHA-3	Hash Function	Hashing	Larger output needed
RSA	Public key	Signatures, key establishment	No longer secure
ECDSA, ECDH (ECC)	Public Key	Signatures, key exchange	No longer secure
DSA (Finite Field Cryptography)	Public Key	Signatures, key exchange	No longer secure

2.4 Post-Quantum Cryptography (PQC)

Since it has been proven that current cryptographic algorithms will be affected with the advent of large-scale quantum computers, post-quantum cryptography has become an important area of research. PQC refers to cryptographic algorithms that are secure today but will remain secure when large-scale quantum computers and practical quantum computing become a reality [84]. With the advancements being made in the field of quantum computers, this reality could soon be realized as long as the issue with decoherence times can be overcome. PQC creates quantum-resistant hashes, signatures, symmetric and public-key cryptosystems. PQC algorithms have gained most popularity for asymmetric cryptosystems, and that is due to the effect large-scale quantum computers will have on public-key cryptosystems [34].

There are many known categories from which PQC algorithms originate but due to the nature of the basis problems some of these categories are better suited for different uses. A description of some of the cryptographic families that PQC algorithms obtained their cryptographic primitives follows.

Lattice-based cryptography is based on lattice problems, and the produced key establishment algorithms are simple, efficient, and highly parallelizable [34]. However, even with known cryptanalysis techniques it can be difficult to provide an accurate security rating [34].

Code-based cryptography is based on error-correcting codes [34]. Although code-based cryptography is proven to run quickly, it is hindered by its large-key sizes [34]. Code-based cryptography has been proven to have more success with signature schemes than any of the other schemes [34].

Multivariate-polynomial cryptography schemes are based on the difficulty of solving systems of multivariate polynomials over finite fields [34]. Many of these schemes have been broken [34]. Just like code-based cryptography, multivariate-polynomial

cryptography has been more successful with signature schemes [34].

Hash-based signatures are algorithms that produce a digital signature by using an underlying hash function [34]. However, hash-based signatures have a downfall, these schemes require an internal count of the number of signatures that are produced because there is a limit for signatures these schemes can produce [34]. Although there is a limit, that limit can be raised to be effectively unlimited but that comes with the trade-off of larger signature sizes [34].

Another important area of research in PQC is the cryptanalysis of the cryptographic families. A number of these PQC algorithms have been found to be vulnerability to side-channel attack (SCA)s [34] and multi-key attacks; these vulnerabilities will need to be addressed by the cryptographic creators before implementation of these cryptosystems is considered. That is why PQC is a popular area of research and one that could benefit from contests and challenges [74]. Opening up algorithms for analysis and allowing the public to conduct cryptanalysis on these algorithms will allow for the vulnerable algorithms to be reconsidered and improved [74]. In Sections 2.5 an outline of NIST's Post-Quantum Cryptography Standardization effort is provided where the purpose is to find quantum-secure public-key algorithms.

2.5 NIST: PQC Standardization

An important contributor in the research field of PQC algorithms is NIST, and their PQC Standardization Project. This project plays an important role in choosing a PQC algorithm for this research effort. For those reasons, this section outlines the work by NIST and the algorithms within the PQC Standardization Project.

NIST is conducting a PQC standardization project, and have released a Report on Post-Quantum Cryptography [34] where it provides a detailed motivation behind this project. In an attempt to find suitable PQC algorithms submissions are accepted from

the public and narrow down the submissions in a competition-like fashion. The aim of this project is to find public-key algorithms that are capable of protecting sensitive U.S. government information into the foreseeable future, including into the quantum computer era [34]. Ultimately this project is to update the public-key standards to include at least one PKE scheme, one digital signature, and one key establishment algorithm [34]. In February 2016, NIST announced their PQC Standardization project at PQCrypto 2016 [5]. In NIST's presentation there is an outline for the projects Call for Submissions [79], and later released is a formal Call for Proposals [83]. The Call for Proposals outlines the requirements for the algorithms, how the algorithms are evaluated, and what the submitted packages must contain. The packages that were submitted were required to contain thorough documentation on the algorithms, including a complete written specification, detailed performance analysis, known answer tests (KAT) values, a thorough description of the expected security strength, analysis of the algorithm with respect to known attacks, and a statement of advantages and limitations [83].

These packages are required to include two implementations, a reference implementation and an optimized implementation [83]. The goal of the reference implementation is to promote the understanding of how the algorithm works, which means that the code should be well commented with references to the functions in the documentation [83]. Since clarity is emphasized in the reference implementation it is more important that the code focus on readability rather than focusing on efficiency [83]. The optimized implementation is intended to demonstrate the performance of the cryptosystem and should focus on optimization of the code [83]. Both implementations must be written in C.

Once the proper packages were submitted they are then evaluated by NIST and the public is encouraged to conduct their own evaluations because public evaluations

are taken into consideration for which algorithms move onto the next round [83]. NIST will evaluate the packages on the following criteria: security, cost, algorithm characteristics, and implementation characteristics [83].

The evaluation of security is completed by evaluating the submitted proofs of relevant models and the security they add to protocols in which the algorithms will be implemented [83]. The candidate PKE and KEM algorithms will be evaluated for their security against chosen ciphertext attack (CCA) and chosen plaintext attack (CPA) [83]. The candidate digital signature schemes will be evaluated on their unforgeability and security against adaptive chosen message attack (CMA)s [83]. Also, evaluations are made as to the estimated security strengths of the candidate algorithms and comparing the candidates in accordance with NIST security rankings [83]. Lastly, security will be evaluated base on the candidates ability to provide resistance to misuse, SCA, and multi-key attacks [83].

The evaluation of cost is completed by evaluating the size of the produced keys, ciphertexts, and signatures [83]. Also, the candidates will be evaluated by computational efficiency (speed of algorithms) of key generations and public and private key operations [83]. Another aspect that is taken into consideration when evaluating the cost is the probability of decryption failure [83]. NIST analyzed the performance of candidate algorithms on the referrenced platform but encourages public to conduct performance evaluations on addition platforms which is addressed by this research.

The evaluation of algorithm and implementation characteristic is completed by the flexibility the candidates possess because algorithms that have flexibility have the capability to be implemented on a range of devices [83]. NIST prefer simple designs and will evaluate any factors that might hinder adoption of the candidate algorithms [83].

The deadline for submissions happened at the end of November in 2017 [5]. During

the submission period 82 packages were submitted and evaluated. After the initial submissions were evaluated, NIST selected 69 candidates to move on to Round 1 in late December 2017 [5]. The public and NIST conducted their evaluation of the Round 1 candidate algorithms, and then NIST decided on 26 candidates to move to Round 2 [5]. Of the 26 Round 2 candidates, 17 were PKE or key-establishment schemes and 9 were digital signature schemes. An accurate timeline of NIST PQC Standardization project has now been provided, it is important to take a look at the upcoming timeline for the project.

As of January 2019, NIST is currently in Round 2 of the project, and have a rough timeline for the rest of the project. NIST has allotted 2020 and 2021 to either begin Round 3 of candidate selection or to select which algorithms to standardize [5]. NIST anticipates that a draft of the new standards will be made available between 2022 and 2024 [5]. If the reader wishes to remain up-to-date with the project timeline they are suggested to check-out NIST's website because NIST will provide more information as their timelines are formalized [5].

2.6 Implementations of PQC for IoT and Mobile Devices

Since the security of current cryptography is going to be affected by the development of large-scale quantum computers, it is important to evaluate the implementation of quantum-resistant algorithms on resource constrained devices. A resource constrained device is limited on power, memory, or CPU. This is becoming increasingly more important because IoT devices and mobile phones are collectively growing to become the largest number of devices connected to a network. According to Gartner Research, the number of Internet-connected things expected to be in use by 2020 is 20 billion and outnumbering humans 4-to-1 [59]. This section focuses on outlining the current implementations of PQC algorithms on IoT and mobile devices. Ideally

this section would be dedicated solely to examining current work on mobile implementations of PQC. However, due to the limited research in the field this section also includes IoT devices which are typically more resource constrained than mobile phones.

Ghosh et al. [50] contributed an implementation of the eXtended Merkle Signature(XMSS) scheme for extremely resource constrained IoT motes, small smart sensors. They constructed a hardware-software implementation that allowed for hash-based signatures to be produced with a small footprint (in terms of the amount of hardware and code) and low latency for sign and verify operations [50]. Their implementation provided 128-bit presumed security against quantum attacks [50], while their main motivation was providing security against attacks on secure boot/update and attestation. Ghosh et al. [50] provided documentation shows comparisons from their implementation and existing solutions, demonstrating that they were able to implement a quantum-secure public-key solutions for IoT devices [50]. Their implementation performed better than NIST p-256 and NIST p-256 HW-SW in both latency and resources, and ECC on 2^{131} in latency but did not outperform ECC on 2^{131} for resources required [50]. Their implementation did require the addition of hardware to these IoT motes, and as they stated these devices are in rather remote locations and stay in the field for years or even decades [50], which would make switching to this algorithm not only time consuming, but difficult to ensure that all IoT motes got the required upgrades.

Strahl and Johansson [102] conducted research on implementing post-quantum based Transport Layer Security (TLS) on a device and ran PQC algorithms on an IoT device to determine if the device was able to support PQC algorithms [102]. The authors were unable to get the PolarSSL library onto the intended IoT device due to memory constraints, so the authors switched to using a Raspberry Pi 2 Model

B [102]. The Raspberry Pi was utilizing the post-quantum secure TLS, a modified PolarSSL, and the performance of LATICEE a post-quantum key exchange was compared to the performance of a standard implementation for key exchange [102]. This paper does goes into great detail as to why the author chose LATICEE based on the RLWE problem [102]. In this implementation the LATICEE scheme was compared to DHKE. Overall, the results from his research was that not only did LATICEE consume a significantly larger amount of random access memory (RAM) but it also took a significantly larger amount of time [102]. Ultimately the author comes to the conclusion that it is not, at this time, necessary to switch to post-quantum algorithms considering that the post-quantum key exchange required more time and RAM usage [102]. However, this work was completed in 2016 and with the development of new PQC algorithms that are more competitive with DHKE the opinions of the authors could differ now.

Basu et al. [19] performed a series of test on NIST Round two PQC algorithms to provide hardware-based comparisons of the algorithms. This was accomplished by utilizing a high-level synthesis (HLS)-based hardware design methodology to map high-level C specifications into field-programmable gate array (FPGA) and application-specific integrated circuit (ASIC) implementations [102]. One of the important contributions from this research was that in low-power IoT devices a PQC algorithm should be selected with low-area and compact designs. The authors suggested for security level 1, HRSS and SPHINCS+ for FPGAs [102]. Basu et al. [19] also suggested FrodoKEM and SPHINCS+ ASICS for IoT devices due to their small decapsulation modules that consume less power [102]. Through the testing the authors found that algorithms could be optimized for low-area by rewriting the code [102]. Basu et al. [19] suggestion was to run HLS and utilize the flow representation to change aspects of the code, making the code not only run faster but also consume less memory [102].

Malina et al. [73] implemented several PQC key exchanges on a variety of devices. The first device was an Android phone with Qualcomm Snapdragon 801, and the next was a single-board device with a 32-bit CPU running ARMv71. On the Android devices they tested NewHope-Usenix and MSR LN16. Both PQC key exchange algorithms were based on lattice-based cryptography, but NewHope-Usenix had a 206-bit post-quantum security level compared to the 128-bit post-quantum security level of MSR LN16 [73]. In their results, MSR LN16 accomplished a key exchange over 100 times faster than New Hope, and at a fraction of the time it took the ECDH algorithms [73]. On the single board they ran the following key exchanges/post-quantum bit level security: NewHope-Usenix/206, NTRU/128, BCNS/78, FrodoKEM/130, McBits/120 and SIDH/128 [73]. They are all lattice-based key exchange algorithms with the exception of McBits which is code-based and SIDH which is isogeny-based [73]. The results from these experiments were that NTRU and NewHope-Usenix outperformed all of the other PQC KEMs. This was within 0.5ms of the ECDH implementation [73]. Their overall conclusion was that these PQC schemes could be efficient and comparable with classic schemes.

In the work done by Routray et al. [91] and Cheng et al. [35], the authors suggested alternatives to securing communication from quantum attacks. Respectively, they suggested quantum communication and physically unclonable functions (PUF). Cheng et al. [35] suggested using PUF that fingerprinted a unique way a device sends signals; this approach is targeted for use by IoT devices, and could thwart a quantum attack.

The work done by Strahl and Johansson [102] showed that implementation of cryptographic libraries on constrained devices is ineffective and another approach should be considered. However, the conclusion that Strahl and Johansson came to was in 2016. This means that more effective algorithms may have been developed.

Additionally, their conclusions were derived from the use of a modified PolarSSL that consumes a large amount of memory. This could be solved by looking at alternatives. While the research by Malina et al. [73] showed that an algorithm that performs well on one constrained device might not be the solution for all devices. In general, algorithms that might perform well on mobile phones could not perform comparably on IoT devices and it is difficult to compare PQC algorithms of different post-quantum bit-level security. Lastly, implementing code on an Android device utilizing C/JNI appears to more effective than a pure Java implementation. The research done by Basu et al. [19] and Ghosh et al. [50] showed that with optimization PQC algorithms can be well suited for constrained devices. However, it would be better to find optimization implementations that do not require hardware modification for the simplicity of updating all devices to PQC algorithms.

A huge realization is that there is a lack of research being done on mobile phone implementation of these PQC algorithms. This is important because mobile phones are considered to be the most common communication devices in history [4]. Furthermore, mobile phones are utilized for a multitude of different of things ranging from sending SMS messages to utilizing mobile banking applications and for each of these usages security, needs to be prioritized. It is especially important when it comes to the security of communications of mobile devices that are used for classified government purposes. It is for that reason ensuring these devices are quantum-secure before large-scale quantum computers make their emergence is an impactful area of research.

2.7 NIST Round 2 PQC KEM's

As stated in Section 2.5, NIST has reached the second round of the PQC Standardization Project. Round two contains 26 PQC algorithms, in which 15 contain

a PQC KEM scheme. Table 2 shows all of the PQC KEMs that have progressed to round two of NIST Standardization Project along with the underlying problems that give security to their schemes. Some of the security problems that have not previously been mentioned are Module Learning With Errors (MLWE), Ring Learning With Rounding (RLWR), Module Learning With Rounding (MLWR), MLWE, and Ideal Rank Syndrome Decoding (IRSD).

Table 2: Each KEM in Round Two of NIST Standardization Project

NIST Round 2 Submitted PQC KEM			
Algorithm name	**Primitive**	**Security Level**	**CCA or CPA**
CRYSTALS-KYBER [14]	MLWE	2, 3, 4	CCA
FrodoKEM [9]	LWE	1, 3	Both
LAC [69]	RLWE	1, 3, 5	CCA
NewHope [8]	RLWE	1, 5	Both
NTRUPrime [24]	RLWE	2, 3, 4	CCA
Round5 [15]	RLWR	1, 3, 5	Both
Saber [106]	MLWR	1, 3, 5	Both
Three Bears [54]	MLWE	2, 4, 5	Both
Classical McEliece [22]	code-based	5	CCA
NTS-KEM [6]	code-based	1, 3, 5	CCA
BIKE [13]	code-based	1, 3, 5	CPA
LEDAcrypt [18]	code-based	1, 3, 5	both
ROLLO [76]	code-based	1, 3, 5	both
RQC [75]	IRSD	1, 3, 5	CCA
Sike [32]	isogeny-based	1, 2, 3, 5	CCA

Through the use of NIST's Status Report of First Round Submission [5] and each KEMs documentation, a PQC KEM will be selected for mobile implementation for this research effort.

These paragraph describes the PQC KEM algorithm selection for this research effort. It is possible to eliminate LEDAcrypt [18] and NTS-KEM [6] from the possibility of selection because the packages were not submitted with any KAT. Without KAT, it is impossible to verify the mobile implementation for correctness. Next, it is

possible to eliminate Sike [32] and ROLLO [76] because they are built using security problems that have not had their security studied as much as the other submitted KEMs [5]. Also, Sike [32] performs an order of magnitude slower than all other submitted PQC KEMs [5]. RQC [75] takes a conservative approach where there is no assumption made in the presumed security of the IRSD problem [5]. This conservative approach has a direct effect on the decryption speed and ciphertext size [5]. For these reasons, RQC [75] will not be considered for mobile implementation. Another submission that can be eliminated is BIKE [13] because it only implements a CPA-secure KEM. Since there is no CCA-secure implementation, Aragon et al. [13] make the assumption that this PQC KEM is only intended to be implemented when there is a strict prohibition on key reuse in place [5]. Since there is no CCA implementation, there is no added security against a CCA in the case of key reuse. The next KEM that is eliminated is Classic McEliece, because it has large public-key values over one million bytes [5]. The Classic McEliece system would benefit from adding additional parameters sets that could reduce the size of the public-key values because otherwise it limits the platforms that can realistically implement it. Three Bears [54] is not selected for implementation because the provided documentation heavily relies on the use of unpublished work for understanding. Since it is difficult to fully understand how the Three Bears cryptosystem is built and implemented, it is not a good choice for selection. Next, Saber [106] was not selected because NIST debated on whether or not this cryptosystem is stretching the hardness of the MLWR problem [106]. Round5 [15] was not chosen for selection because according to the Status Report on the First Round of NIST Post-Quantum Cryptography Standardization Process [5] there are security concerns with how the Round5 algorithms calculate decryption and decapsulation failure. Also, LAC [69] was not selected because there has been public comments demonstrating a vulnerability to a CCA within its security level 5 param-

eter and also there is no SCA resistant implementation [34]. Lastly, FrodoKEM [9] was not selected because it takes a conservative approach that increases the key sizes in comparison to other lattice-based KEMs which makes other KEMs a more suited fit when trying to limit the resources used.

The KEMs that were not selected for this research efforts mobile implementation selection and that narrows it down to CRYSTALS-KYBER [14], NTRUPrime [24], and NewHope [8]. Bernstein et al. [24] address the claims of some of these attacks within their documentation. However, NTRUPrime [24] has more added lattice structure than that of NewHope [8] which does not present any additional known attacks. If an attack is developed that takes advantage of the added structure of ideal lattices NTRUPrime will be at even more risk than NewHope. Ultimately the decision to implement NewHope was made, and this is based on the fact NIST acknowledged potential risks within the other two KEMs but did not acknowledge any potential risk within NewHope. It is also beneficial to select NewHope because available work presents data points for mobile implementations of a NewHope-Usenix and the selected NewHope scheme is a variant of NewHope-Usenix.

2.8 Lattice-Based Cryptography

Since latticed-based cryptography algorithms have hardness against quantum attacks, these algorithms have gained attention within the PQC community [64]. However, like all cryptographies, lattice-based cryptography has its downfalls such as inefficiency. These lattice-based cryptosystems can see a ciphertext expansion of a blowup factor about 25, whereas the classic cryptosystems produces ciphertext that is equivalent in size to the plaintext [78]. Lattice-based primitives are increasingly viewed as offering the most promising post-quantum alternative to DHKE or RSA [98].

Lattice-based cryptography has gained a lot of popularity within the post-quantum

cryptographic community and this is due to its conjectured strength against quantum attacks [74]. First, it is important to formally define lattice-based cryptography as the construction of cryptographic algorithms whose security is based on the conjectured intractability of computationally difficult lattice problems [77]; some of these problems are discussed in Section 2.8.2. In cryptographic applications these algorithms are typically required to be a one-way function that is computationally hard to invert [77]. These cryptographic algorithms are used to build cryptosystems for public-key cryptography, identity based encryption, fully homomorphic encryption, hashes, and signatures [77].

The security of lattice-based cryptographic algorithms tend to have an advantage on the security of classical algorithms. This is because classical algorithms are typically based on average-case problems, but lattice-based cryptography is based on worst-case problems [12]. The following is an oversimplified description of the difference between average case and worst-case problems but it will suffice for the purposes of this effort. Take the example of integer factorization, not all numbers are equally as hard to factor, in fact some numbers are easier to factor, like even numbers, but worst-case problems are difficult to solve [12]. Worst-case hardness implies that breaking these lattice-based cryptographic constructions would be as hard as solving several lattice problems, within polynomial factors, in the worst [20]. For example, the RLWE problem requires solving, the closest vector problem (CVP) problem as well as lattice-basis reduction. This is one reason why lattice-based cryptography is being strongly considered to be a quantum-secure replacement for classical asymmetric cryptographic algorithms. It defends against other mathematical breakthroughs that might efficiently solve integer factorization or compute discrete logarithms [30].

2.8.1 Lattices

In this section, a brief overview of the mathematical lattice structure is provided. For a more in depth understanding of the lattice structure the reader is directed to Mariano et al. [74], Nguyen et al. [81], Regev & Oded [88], and Micciancio [77].

Lattices are discrete subgroups of the n-dimensional Euclidean space $\mathbb{R}^n$, and presents a strong periodic structure [81] [88]. Specifically, $\mathbb{Z}^n \in \mathbb{R}^n$ represents an n -dimensional integer lattice in the n -dimensional Euclidean space. A set of linearly independent vectors $v_1, \ldots, v_m \in \mathbb{R}^n$ form the basis, $\mathbf{B}$, of the lattice $\mathcal{L}$ [74]. An integer latticed is formed by all integral linear combinations of the basis vectors, using integer scalars [74]. More precisely,

$$\mathcal{L}(\mathbf{B}) = \left\{ x \in \mathbb{R}^n : x = \sum_{i=1}^{m} u_i v_i, u \in \mathbb{Z}^m \right\}$$

where $m \leq n$ in the rank of lattices [74]. In the case that $m = n$, $\mathcal{L}(\mathbf{B})$ is said to be of full rank, and in the case that $n \geq 2$, then $\mathcal{L}(\mathbf{B})$ has an infinite number of possible bases [74]. An example of an integer lattice can be seen in Figure 1 and in this figure b_1, b_2 are basis vectors that form a lattice which contains b_3.

Non-integer lattices do exist, but integer lattices are usually used for lattice-based cryptography because the problems are still hard enough to be cryptographically secure and integer lattices are computationally easier to use [74]. Lattices have interesting applications in both cryptanalysis and cryptography, and the discussions for both follow.

2.8.2 Lattice Problems

This section provides a description to several hard lattice problems that provide the underlying security to lattice-based cryptosystems. Additionally, this section dis-

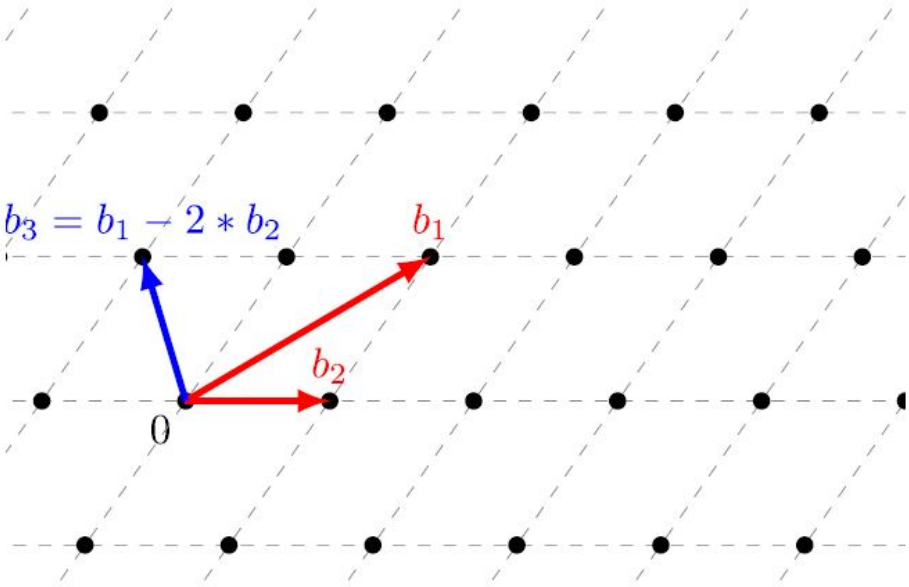

Figure 1: A lattice with b_1 and b_2 basis vectors, and b_3 is an element of the lattice. Figure by Mariano et al. [74]

cusses and describes these problems briefly to provide a better basis of understanding for the literature survey to follow. The lattice problems include lattice-basis reduction, the shortest vector problem (SVP), the CVP, the learning with errors (LWE) problem, and the RLWE problem. These problems were selected for their prevalence in cryptosystems, and since they are popularly used in cryptosystems, they are the most relevant problems for lattice-based cryptanalysis. Out of scope are hard lattice problems such as the shortest integer problem (SIS), decision shortest vector problem (DSVP), unique shortest vector problem (USVP), etc. that can be found in the work done by Laarhoven et al. [66].

2.8.2.1 Lattice-basis Reduction

Although it is not always mentioned throughout literature as a hard lattice-based problem, lattice-basis reduction is frequently used in the cryptanalysis of lattice-based cryptography and other cryptosystems. Some examples of lattice-basis reduction being utilized in the cryptanalysis of other cryptosystems can be seen in the work

done by Wang et al. [107] and Bi et al. [27]. Given a basis, $\mathbf{B}$, one can form the lattice, $\mathcal{L}(\mathbf{B})$. Lattice-basis reduction is the process of transforming $\mathbf{B}$ into a new basis, $\mathbf{B}'$, such that $\mathcal{L}(\mathbf{B}) = \mathcal{L}(\mathbf{B}')$ [74]. The basis $\mathbf{B}'$ should have vectors that are shorter and more orthogonal than those of $\mathbf{B}$ [74]. An example of a lattice-basis reduction can be seen in Figure 2.

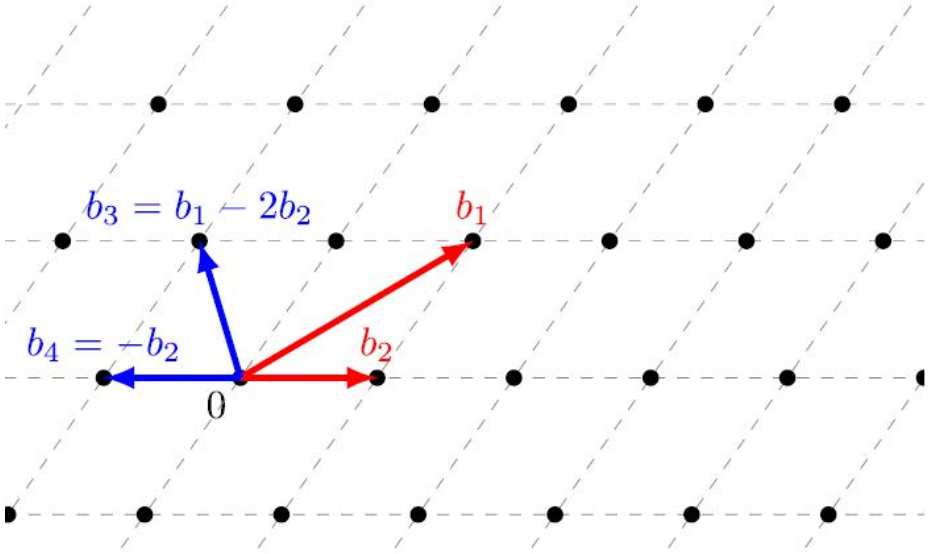

Figure 2: A lattice with b_1 and b_2 basis vectors, after lattice-basis reductions b_3 and b_4 are new basis vectors. Figure by Mariano et al. [74]

There are many algorithms that are used to reduce the bases of lattices and some of them will be discussed in Section 2.9.

2.8.2.2 Shortest Vector Problem (SVP)

The SVP is commonly used in lattice-based cryptography; it involves finding the shortest non-zero vector in the lattice which is measured by a norm [74]. More specifically, given a basis, $\mathbf{B}$, and its corresponding lattice, $\mathcal{L}(\mathbf{B})$, then the SVP is defined by finding a non-zero vector, $p \in \mathcal{L}(\mathbf{B})$, such that

$$\|p\| = min\{\|x\| : x \in \mathcal{L}(\mathbf{B}), \|x\| \neq 0\} \text{ and } \|p\| = \lambda(\mathcal{L}(\mathbf{B}))$$

[74]. An example for the SVP is provided in Figure 3.

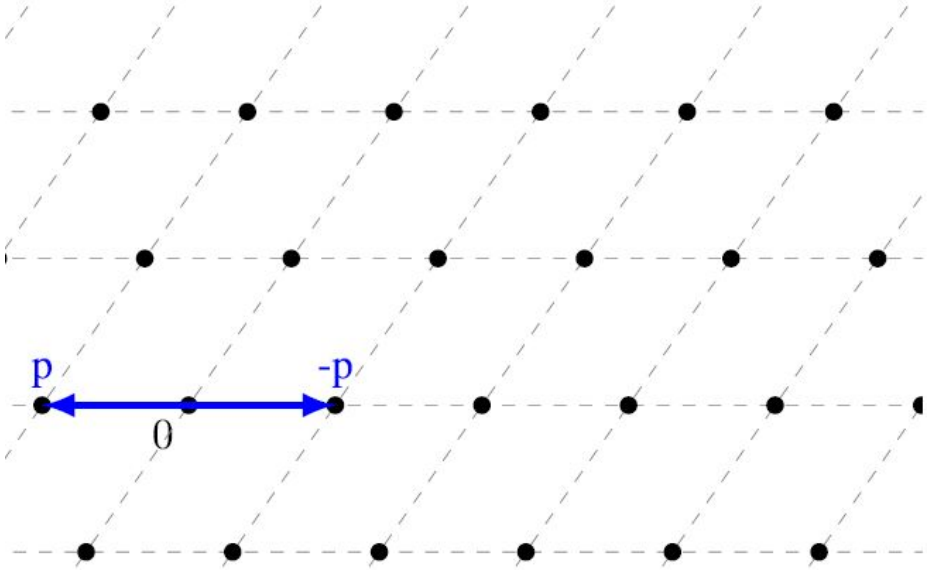

Figure 3: A lattice where p and $-p$ are the solution to the SVP for some $\mathcal{L}(\mathbf{B})$. Figure by Mariano et al. [74]

However, this is a definition for the exact version of the SVP, and there is an approximate version that is sometimes referred to as the γ-SVP [77]. The γ-SVP is similar to the exact SVP except that the vector found is a factor bigger than the exact shortest vector [77]. More formally, for $\gamma \geq 1$ and a basis, $\mathbf{B}$, the γ-SVP is finding a non-zero vector $p \in \mathcal{L}(\mathbf{B})$ such that $\|p\| \leq \gamma * \lambda(\mathcal{L}(\mathbf{B}))$ [77]. When $\gamma = 1$ then the solution for the exact SVP is equal to that of the γ-SVP [77]. A part of the reason that the SVP is difficult to solve is because typically the lattice basis is comprised of long vectors that are significantly larger than the shortest vector. This is why lattice-basis reduction is utilized to help solve the SVP [77]. It is difficult to predict for which dimension of a lattice causes solving the SVP to be infeasible with the computing power available today [88]. This makes it difficult to decide which security parameters are needed for lattice-based cryptosystems.

2.8.2.3 Closest Vector Problem (CVP)

The CVP is an NP-Hard problem which was proven by Van Emde Boas [105]The CVP is described by trying to find a vector in the lattice that is closest to a vector v in vector space V [74]. To be more precise, give a basis, $\mathbf{B}$, for lattice, $\mathcal{L}(\mathbf{B})$, in the vector space, V. The CVP is to find a vector $l \in \mathcal{L}(\mathbf{B}) \in V$ that is closest to vector $v \in V$ [74]. That is, $\|l - v\| = min\{\, \|p - v\| : p \in \mathcal{L}(\mathbf{B})\}$ and $dist(v, \mathcal{L}(\mathbf{B})) = \|l - v\|$ [74]. An example of the CVP can be seen in Figure 4.

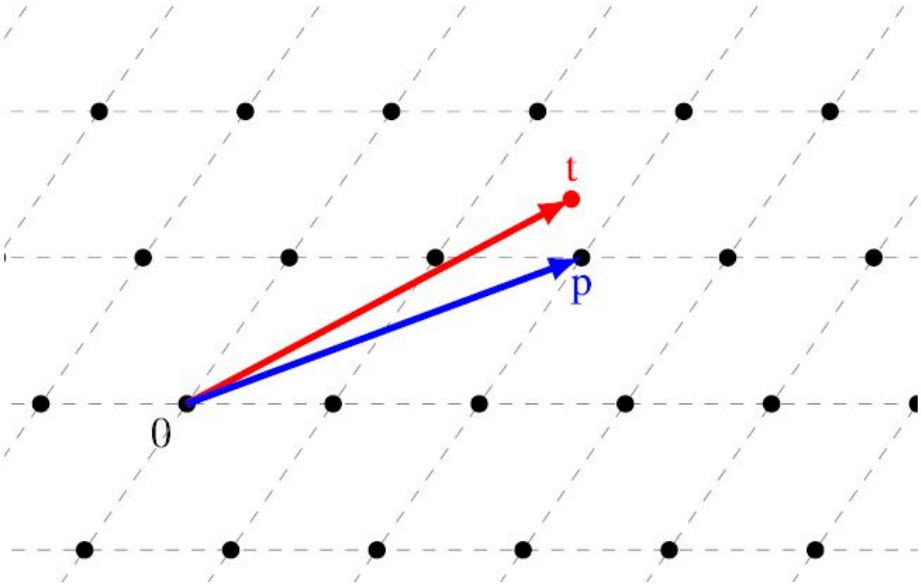

Figure 4: Vector t is the target vector, vector p is an element of the lattice, and the solution to the CVP. Figure by Mariano et al. [74]

In the CVP, the target vector does not have to be in the lattice. Additionally, just like in the SVP there is an exact version and an approximate version. The above definition is the exact CVP, and the approximate version, γ-CVP, is described by the distance from the lattice vector, l, and the target vector, v, being a factor of γ away from the exact distance [77]. To be more precise, for $\gamma \geq 1$, vector $l \in \mathcal{L}(\mathbf{B})$, target vector v in vector space V the goal is to find l such that $\|l - v\| \leq \gamma * dist(v, \mathcal{L}(\mathbf{B}))$ [77]. When $\gamma = 1$, the exact-CVP and γ-CVP are equal [77].

2.8.2.4 Learning With Errors (LWE) Problem

The LWE problem is a lattice-based hard problem introduced by Regev [89]. The parameters of this problem are the dimension, $n \geq 1$, an integer modulus, $q \geq 2$, and an error distribution, χ over $\mathbb{Z}_q$ [74]. Typically the LWE error distribution is either Gaussian or binomial centered around 0 [74]. The case when $q = 2$ is also known as the learning parity with noise (LPWN) problem. For the LWE problem, vectors $a_i \in \mathbb{Z}_q^n$ are generated uniformly at random and $b_i \in \mathbb{Z}_q^n$ is produced by $b_i = a_i * s + e_i \mod q$, where $e_i \in \mathbb{Z}_q$ are small error values chosen from χ and $s \in \mathbb{Z}_q^n$ is a fixed secret vector [74]. In the search variant of the LWE problem, the goal is to find a secret vector $s \in \mathbb{Z}_q^n$, and with the decision version the goal is to determine if the samples (a_i, b_i) are of the form described below [74].

Unlike SVP or CVP, it is not immediately apparent how the LWE problem is a lattice-based problem, but Mariano et al. [74] provide a detailed description of the relation. After collecting many samples, (a_i, b_i), the matrix A, is formed by using all of the collected a_i's as the rows for A [74]. Also, the vector b is all the b_i's that are collected, and can be seen by $b = As + e$, where e is the vector of all e_i's (error values) used [74]. In fact, b is a lattice vector, $b = As + e$, for the lattice created from the columns of A [74]. This means solving the difficult lattice problem CVP for target vector b would imply finding As and it is then trivial to find s [74]. It is important to note that in order to solve this problem enough transmission would need to be captured to rebuild the correct lattice and the hard lattice problem CVP would need to be solved.

Many cryptosystems are built with a security depending on the LWE problem. Schemes based on the LWE problem are known as standard lattice-based schemes. These scheme require matrix-vector multiplication, meaning these schemes require/use a large amount of memory and have a quadratic complexity [62]. There are many

different variants of the LWE problem, and one that is mentioned throughout the effort is the RLWE problem.

2.8.2.5 Ring Learning with Errors (RLWE) Problem

When discussing RLWE cryptosystems, it is important to understand an oversimplified definition of a ring. In abstract algebra, a ring is an algebraic system consisting of a set, an identity element, two operations, and the inverse operation of the first operation [42]. A ring, $\mathcal{R}$, of integer polynomials modulo $f(x)$, is described by the following equation: $\mathcal{R} = \mathbb{Z}[x]/\langle f(x)\rangle$ [74].

The LWE problems has caused a rise to many different variants of the problem to be created because the inherent quadratic overhead when using the LWE problem [70]. The RLWE problem is an algebraic variant of the LWE problem that introduces the use of rings of integer polynomials and cyclotomic polynomials [70]. A more formal definition of a ring of integer polynomials is described as $f(x) = x^n + 1 \in \mathbb{Z}[x]$ and a ring of integer polynomials modulo $f(x)$ such that $R = \mathbb{Z}[x]/\langle f(x)\rangle$ [70]. The elements of R can be represented as polynomials less than n. Let q be a large prime. For the RLWE, there is a set of small random and unknown polynomials in R such that $e_i(x)$ is an element of that set [70]. Let $s = s(x) \in \mathbb{R}_q$ be uniformly random [70]. The samples taken are in the form of $(a_i(x), b_i(x))$ where $a_i(x)$ is a set of random but known polynomials and $b_i(x) = (a_i(x) * s(x)) + e_i(x)$ [70]. Similar to the LWE problem, there are two different versions- search and decision. The search version entails finding the unknown polynomial $s(x)$ given the sample $(a_i(x), b_i(x))$. The decision version entails finding out if $b_i(x)$ was constructed by $b_i(x) = (a_i(x) * s(x)) + e_i(x)$.

One of the main benefits of using RLWE cryptosystems over traditional LWE cryptosystems and other lattice-based techniques is its superiority with efficiency in both computation and key size [98]. The reason RLWE cryptosystems are efficient

is because the initial matrix is represented by a single row and all other rows of the matrix can be derived from cyclic shifts of the first row [70]. This reduces matrix-vector multiplication to polynomial multiplication and is memory and computation efficient [62]. RLWE cryptosystems also provide strong security against classical and quantum adversaries. In fact, research has be done to improve the efficiency of RLWE cyptosystems, for example Tan et al. [103] and Alkrim et al. [8], and the work they have done to improve RLWE cryptosystems through the use of a Number Theoretic Transform (NTT).

2.9 Cryptanalysis of Lattice-Based Cryptography

This section provides an overview of the previous literature that has been produced on the cryptanalysis of lattice-based cryptography. The section is broken up into subsections based on the types of problems that specific algorithms intends to solve. This is valuable to this research effort because it provides an in-depth look into how a cryptosystem built from lattice-based cryptography could have its security undermined.

2.9.1 Lattice-Basis Reduction

There are many algorithms that are used to accomplish lattice-basis reduction: LLL, Block Korkine-Zolotarev (BKZ), random sampling reduction (RSR), and PDR [74]. However, the most commonly represented across the literature are LLL and BKZ. Another interesting set of lattice-basis reduction algorithms are those that use quantum algorithms to speed up the reduction process, an example of one will be discussed in this section. There is a plethora of literature about lattice-basis reduction algorithms, techniques, optimizations, etc. Lattice-basis reduction is used throughout many attacks to aid in solving the hard lattice problems.

The lattice-basis reduction algorithm, Lenstra-Lenstra-Lovász, known as the LLL algorithm, has been applied to many areas of mathematics and computer science [110]. This algorithm generalizes the Lagrange-Gauss algorithm and exploits the Gram-Schmidt orthogonalization [46]. There are algorithms that were produced to try and optimize the LLL algorithm, like Backes et al. [17]; this produces a parallel variant of LLL that utilizes POSIX threads. This algorithm is based on Schnorr-Euchner algorithm, and it utilizes POSIX threads to make use of today's multi-core computer architectures [17].

Another lattice-basis reduction algorithm that is commonly seen is BKZ introduced by Schnorr & Euchner [94]. According to Schnorr et al. [64], although LLL is one of the most famous lattice reduction algorithms, the most promising according to them is a combination of LLL and BKZ. BKZ uses local blocks to achieve reduction, the size of these blocks are determined by an input parameter [64]. An improvement in reduction can be accomplished through the use of larger block sizes [64]. This algorithms starts by using LLL to reduce a given basis, $\mathbf{B}$, of a lattice, $\mathcal{L}(\mathbf{B})$ [104]. After $\mathcal{L}(\mathbf{B})$ has been LLL-reduced, the quality of the reduction is iteratively improved using the local blocks [104]. Unlike LLL, there is not much literature on the parallelization of BKZ. However, some literature does exist, Mariano et al. [74] provide a detailed survey of the BKZ and LLL algorithms and the corresponding optimizations.

In 2003, Ludwig [1] proposed a lattice-reduction method, quantum search reduction (QSR), that speeds up the original RSR algorithm by utilizing a quantum search. RSR is a good candidate for optimization through the use of implementing a quantum search algorithm because the running time of RSR is controlled by the exponential number of vectors that have to be sampled and discarded in order to find a sufficiently small vector [1]. The original algorithm that chooses the samples is random, and selects the samples with no inherent structure that would allow for a

speed up in the search process [1]. That is why Ludwig proposes the use of Grover's algorithm to speed up the search process for samples, and in return, speeding up the process of finding a sufficiently short vector [1]. Bernstein [65] states that Ludwig achieved an asymptotic speed up by replacing a random sampling from a big list with a quantum search algorithm. Zhoe et al. [110], state that the results from Ludwig's experiment imply that quantum computers will negatively impact the security of classical cryptosystems and lattice-based cryptosystems. Ludwig anticipates that the post-quantum cryptosystem, NTRU, would need to double its security parameters in order to remain secure [1]. To be more specific, it would need to be increased from 503 to 1277.

2.9.2 Shortest Vector Problem

There are two main families of SVP-solvers: enumeration and sieving. According to Kuo et al. [64], although lattice enumeration has an exponential time-complexity, it is the most efficient algorithm to solve SVP. According to Mariano et al. [74], the most practical enumeration algorithms are ENUM [44] and SE++ [49]. These algorithms could contain pruning techniques that minimize computation but these techniques would decrease the probability of success. There have been efforts made to try and optimize the ENUM and SE++ algorithm through the use of parallel implementations [74]. The sieve algorithms improve the runtime of the algorithms, but the improvement to the runtime comes at a cost of introducing randomization and an increase in memory consumed [74]. Lattice sieve algorithms are not viewed as effective due to their high exponential time and space complexity and the use of randomization [74].

2.9.3 Closest Vector Problem

Similiar to the SVP, the two families of CVP-solvers are enumeration and sieve algorithms. These algorithms face the same advantages and disadvantages as the SVP-solver algorithms. However, unlike SVP, CVP has not gained much attention in terms of practical assessments [74].

Cramer et al. [36] focus on recovering a short generator from any generator in a RLWE cryptosystem. The fastest way to find a short generator is by solving the CVP. Although it is not certain if the CVP of a long-unit lattice would recover a sufficiently short generator. Cramer et al. [36] show that given the generator of an ideal lattice, through the use of the naive "round-off" lattice-decoding algorithm, the secret short generator can be recovered [67] [16]. This work demonstrates that it is important for RLWE cryptosystems to have sufficiently large generators.

2.9.4 Learning with Errors Problem

In 2018, Wunderer et al. [108] show a successful hybrid attack on an instance of a binary-LWE cryptosystem. The hybrid attack is a hybrid lattice reduction and meet-in-the-middle attack, and it is currently known to be one of the best attacks on lattice-based cryptosystems like NTRUPrime [23], R-BinLWEEnc [31], BLISS [39], GLP [53], and NTRU [57]. However Wunderer et al. [108] restricted their studies to a hybrid attack on the LWE problem with a binary error distribution. They also state that the attacker must have access to a modern supercomputer [108]. Wunderer et al. [108] present a highly scalable and configurable parallel version of the hybrid attack on an instance of a binary-LWE cryptosystem. The multiple parallel instances employing distributed processes allows for the low probability of success to be amplified increasing the probability [108]. They also were able to drastically reduce the runtime of the guessing phase by implementing a shared memory parallelization tech-

nique [108]. However, there was a bottleneck presented within the shared memory parallelization technique because it was limited by the high runtimes of the serial BKZ [108]. The authors were able to utilize a parallel hybrid attack and their results show that the parallel hybrid attack was significantly faster than the serial implementation. However, this means that the cryptosystems mentioned above can be broken faster than previously estimated [108]. Due to the fact that the attacker has to have access to a supercomputer and that the probability of success is still low implies this attack does not present too much of a threat. However, LWE cryptosystems and cryptosystems that are variants of the LWE problem should still mitigate attacks of this nature to prevent from attacks of this nature.

Howe et al. [58] address the threat LWE signature schemes face from SCA and more specifically fault attacks, by proposing several countermeasures to use within the error samplers. The reason why the countermeasures are used within the error samplers is because the errors that are chosen to secure the secret-key operations have a statistically normal shape, which are usually Gaussian or binomial [74]. Specifically, lattice-based signatures typically require Gaussian sampling [74]. The need to ensure that these error samplers are operating correctly and are secure against SCA like fault attacks is because Bindel et al. [29] showed that error sampling modules can be targeted to obtain secret-key information. The results from the proposed countermeasures showed that it is possible to protect against error sampler attacks while having little to no effect on the efficiency of the module [58].

2.9.5 Ring Learning with Errors Problem

When it comes to RLWE cryptosystems, the literature differs on suggested parameter sizes for differing security levels, because for ideal lattices there are many parameters that directly affect the security level. Also, there are not many attacks

that exploit the added ring structure of RLWE cryptosystems, but rather exploits target the underlying LWE problem. In addition, if the attack does not exploit the ideal lattice structure of the RLWE cryptosystem, it is likely that the attack would also work against a LWE cryptosystem. This section describes some of the attacks against RLWE cryptosystems from existing literature.

Fluhrer [45] shows that the RLWE problem is vulnerable to attack if there is a reuse of keys. Fluhrer [45] shows that RLWE key agreement protocols can be broken if the private keyshare is reused. If one party is reusing the same private-keyshare then an attacker can send a series of messages to that party and analyze the response to obtain the public-keyshare [45]. After the attacker performs the key establishment protocol the attacker can guess the victims shared secret and generate the symmetric keys based on that guess [45]. After the attacker generates the keys, the attacker will send the keys to the victim. If the victim is able to decrypt the message based on those keys, then it is highly likely that the attacker guessed the shared key correctly [45]. One place where this can potentially become an issue is in the use of TLS 1.3 draft [90] that allows the server to declare a static keyshare. Currently, the TLS 1.3 draft uses DHKE or ECDH. However, if DHKE or ECDH were to be replaced with a RLWE-based key exchange it would no longer be secure [45]. Fluhrer [45] focuses on RLWE and it is believed that this attack would also be effective on a LWE key agreement protocol. RLWE/LWE cryptosystems are still believed to be secure as long as a fresh keyshare is generated for every key agreement protocol. Attacks targeting keyshare reuse have required for developed cryptosystems to contain CCA secure PQC KEMs and these cryptosystems are designed to conain security when key reuse is not strictly prohibited. If a PQC KEM claims to be CPA secure it is assumed that within the implementation there is a strict prohibition on key reuse.

In 2015, the work by Bos et al. [30] replaced the traditional number-theoretic key

exchange in the TLS protocol with one based on the RLWE problem. Based on the parameters selected for the RLWE problem, the authors discuss a possible classical and quantum attack. Based on their simulation results their implementation would remain secure against the classical attack [30]. It is valuable to discuss the classical attack because if improvements are made to any of the algorithms or techniques utilized, it could become an actual threat. The best classical attack is to solve the LWE problem with a bounded distance decoding (BDD) problem and then reduce the BDD to a USVP [30]. The next step is to implement the SVP oracle through sieving. Bos et al. [30] mention a quantum attack would implement Grover's search algorithm in order to offer a speed-up of the classical attack [30]. Both algorithms, the classical attack and the quantum attack, do not use the ideal lattice structure of the RLWE. The attacks treat the cryptosystem as if they are based solely on the LWE problem instead of targeting the added structure of the RLWE problem. This is not uncommon, to date, there is no known attack on RLWE that take advantage of the added structure of the RLWE cryptosystems but instead take advantage of the LWE basis.

In the documentation for the NTRUPrime: Round 2 (round two for NIST's Post-Quantum Cryptography Standardization Project) [25], the introduction begins with some warnings about previous attacks on cryptosystems. The NTRUPrime authors acknowledge that Gentry's original fully-homomorphic encryption (FHE) system [48] is known to be broken in polynomial time by a quantum algorithm [25]. The purpose the authors have for mentioning Gentry's FHE was to illustrate attacks on ideal lattices. The FHE system has standard cyclotomic choice of rings [25]. The weakness discovered was not due to the ideal lattice-based scheme but instead the choice of short generators for the ideal lattice-based schemes [25]. There is no known exploit to attack Ideal-SVP/BDD. NTRUPrime authors outline the known cryptanalytic at-

tacks on small lattice-based encryption schemes, including Streamlined NTRU Prime and NTRU LPRime.

Short generators paired with the structure of the principal ideal were the attack surface for Gentry's original FHE system. However, as discussed by Biasse et al. [28], there are some cryptosystems that rely on the hardness of finding a short generator of a principal ideal (short-PIP) of the cyclotomic ring $R = \mathbb{Z}[x]/(x^{2^n})$. Campbell et al. [33] conjectured that solving for the arbitrary generator of a principal ideal is feasible in quantum polynomial time. They used the concept of a lattice fingerprint to build their quantum attack [33]. The quantum attack was designed to find the small generator of the principal ideal scheme [33]. Ultimately, their suggestion was that cryptographic security should not be based on problems of finding hidden units in a number field or hidden generators of a principal ideal [33]. They explained that the mathematics behind cyclotomic number fields has cryptographic relevance for RLWE schemes. However, this attack is just a conjecture and is not currently tested or proven.

NTRU is a PQC PKE scheme where NTRU encrypt and NTRU sign use ring lattice structures [109]. The original NTRU public-key cryptosystem was proven to be vulnerable against a multiple transmission attack [109]. After the vulnerability surfaced, the designers proposed two countermeasure to the attacks [109]. In 2014, Xu et al. [109] proved that both countermeasures proposed do not secure the cryptosystem from the multiple transmission attack. The first countermeasure is accomplished by appending some redundant information to the plaintext before encryption [109]. However, the authors show that the countermeasure is ineffective and the plaintext can be recovered through a linearization attack technique [109]. They proved that the first countermeasure was vulnerable against broadcast attacks too [109]. The second countermeasure was accomplished by modifying the plaintext with random low-entropy

information and then encrypting it [109]. This countermeasure was proven insecure against the multiple transmission attack because for a case with specific parameters after performing lattice attack methods the plaintext can be obtained [109].

Multiple different research efforts show that RLWE cryptosystems are vulnerable to SCA. It is important to consider physical attacks that may be launched while the cryptographic schemes are being executed. The rest of this section is dedicated to the literature that shows lattice-based cryptography is vulnerable to SCA.

Park et al. [85] propose a simple power analysis (SPA) attack on an unprotected RLWE public-key scheme which was optimized by Roy et al. [92] by utilizing the NTT and an 8-bit implementation. They chose a SPA attack because it was simple and used lower traces than statistical analysis, such as differential power analysis [85]. Their implementation showed that through the use of a SPA attack they were able to recover the secret key using only $\log_2 q$ executions [85]. Also, they demonstrated the modular addition operation is vulnerable against an SPA attack when using the 8-bit implementation; thus it is advised to use the protected implementation when using RLWE public-key scheme on 8-bit processors [85]. For that reason a cryptosystems protection against SCA should be evaluated when analyzing RLWE cryptosystems.

In fact, Bindel et al. [29] show that lattice-based signature schemes are also vulnerable to side-channel attacks. The BLISS [39], ring-TESLA [3], and GLP [53] signature schemes were chosen to be analyzed against the attack due to their efficiency superiority over other signature schemes. These signature schemes were analyzed for vulnerability against randomization faults, skipping faults and zeroing faults [29]. All three schemes were found vulnerable again zeroing faults during the verification and signing algorithm. Also these three schemes were found vulnerable to skipping faults during the verification algorithm and the signature generation algorithm [29]. Moreover, the GLP scheme was vulnerable to some additional fault attacks [29]. An-

other important contribution of this paper was the analysis demonstrated that ideal lattice-based schemes, for example RLWE, was more frequently vulnerable to zeroing attack than standard lattice-based schemes LWE.

Lastly, Espitau et al. [43] presented two attacks that used SCA and a fault attack that allowed the attacker to cause a loop inside the signature generation algorithm to abort earlier than intended. The first attack targets BLISS [39], and the the authors show how it extends to GLP [53], PASSSign [56], and Ring-TESLA [3]. The authors inject a fault into the loop that generates a random "commitment value" and that value was used to generated a polynomial coefficient by coefficient [43]. This means that an early abort leads to an abnormally low degree polynomial, that implies the protocol is no longer zero knowledge [43]. This produced a faulty signature which was used to construct a point that was very close to a vector in an integer lattice of moderate dimension. The difference is a subset of the signing key [43]. Lattice reduction can then be used on the difference to recover the signing key [43]. The target of the next attack was the GPV-based hash-and-sign signature scheme of Ducas et al. [40]; against this signature scheme the early loop abort fault is used against the discrete Gaussian sampling used in signature generation [43]. The result of aborting the loop early is that when the signature generates, it is a linear combination of the last few rows of the secret lattice [43]. If enough faulty signatures are collected it is possible, through the use of lattice reduction, to rebuild the secret key [43].

2.9.6 Analysis of Cryptanalysis

This section provides an analysis of some aspects of the literature surveyed. This provides an idea on the accuracy of the cryptanalysis of lattice-based cryptographies. Also, common vulnerabilities for each problem can be exploited with a variety of attacks and this is also discussed in this section.

A powerful tool in cryptanalysis is the use of challenges, as pointed out by Mariano et al. [74]. By releasing challenges it allows for the community to try and expose any vulnerabilities cryptosystems might possess. Mariano et al. [74] provide information on the challenges available against lattice-based cryptographies.

It is important to analyze the usage of lattice-basis reduction throughout the cryptanalysis of lattice-basis cryptography. In many of the attacks, the solution can be found by reducing the problem down to a SVP, CVP, LWE problem, RLWE problem to a lattice-basis reduction. Lattice-basis reduction can be used to attack cryptosystems that are not built using lattice-based cryptography. Many of the lattice-basis reduction algorithms have been improved upon, whether it be parallelization or the use of a quantum search to accelerate the runtime. Some of the parallelization efforts being made on SVP or CVP-solvers exhibit a bottleneck when it comes to executing the lattice-basis reduction. This implies that when improvements are made on the runtime of lattice-basis reduction algorithms the parallelization of SVP and CVP-solvers could affect the parameter selection for lattice-based cryptography. This means that if lattice-basis reduction is tremendously improved upon, it could affect the security of lattice-based cryptography.

As with lattice-basis reduction, there has been focused research on using a quantum-search algorithm within SVP or CVP-solvers to optimize runtime. CVP and SVP-solvers have been used in attacks against the LWE and RLWE problems. The use of quantum search algorithms to solve hard lattice-problems, shows that quantum computers will not just affect current cryptosystems but they will also affect the security parameters chosen for post-quantum cryptosystems. This presents a problem, because throughout the literature there is a consensus on the difficulty of estimating security level against quantum attacks. Cryptanalysists are both overestimating and underestimating the security level of cryptosystems against quantum attacks [34].

Many attacks against RLWE cryptosystems show that these cryptosystems are attacked as if they are LWE cryptosystems because attempts to solve the RLWE and LWE problems are approached in the same way. The added structure of RLWE cryptosystems become favorable to LWE cryptosystems because the added ring structure allowed for improvements in the efficiency of the cryptosystems without affecting the security of the cryptosystems. Cryptanalysis of these systems becomes important to research to determine if the added structure of the RLWE cryptosystems adds an additional, unexposed, attack vector. However, only two attacks against the structure of the ideal lattices were seen. One, is if a short generator is used for the structure. Two, is a SCA where ideal-lattices are seen to be more vulnerable to zeroing attacks during the sign algorithm; this was not strictly proven to take advantage of the additional structure of the lattice.

2.10 NewHope

Since the PQC NewHope KEM cryptosystem was selected for this research, it is important to discuss the NewHope cryptosystem. This section references the NewHope Algorithm Specifications and Supporting Documentation document submitted to NIST by Alkim et al. [8]. NewHope proposed by Alkim et al. [8] is an encryption based KEM that is based on NewHope-Simple. NewHope-Simple is a variant of NewHope-Usenix with the main difference being that NewHope-Usenix is a reconcilliation based scheme and NewHope-Simple is an encryption based scheme. Within this section each of the following will be detailed: all four KEM types, the difference between CCA and CPA algorithms, descrption of algorithms and supplementary algorithms needed for functionality, parameter selection, the package submitted to NIST by the NewHope creators, how the submitted codebase works, and how the KAT works.

2.10.1 NewHope Cryptosystem

NewHope is a post-quantum cryptosystem that has presumed quantum security through the use of lattice-based cryptography and more specifically the RLWE problem. The NewHope KEM cryptosystem is packaged with a CPA-PKE scheme but this encryption scheme is not intended for use outside of the KEM. Utilizing the CPA-PKE scheme, NewHope is able to achieve two different styles of KEM's, CCA-KEM and CPA-KEM. Those different types are described in more detail in the next section. The NewHope authors claim to achieve a quantum-resistant KEM which is simple to implement, and due the use of the NTT, it is memory efficient. The authors do acknowledge some limitations: The use of the RLWE problems could present a problem if a vulnerability targeting the added structure of the ring is discovered. Lastly, the parameter sets are restricted due to the usage of the NTT. However, the advantages of using the NTT outweighs the downfall of parameter restrictions.

2.10.2 NewHope KEM Types

The NewHope cryptosystem contains four different kinds of KEM's at different NIST security levels. The NewHope cryptosystems are NewHope512-CCA-KEM, NewHope1024-CCA-KEM, NewHope512-CPA-KEM, and NewHope1024-CPA-KEM. Both NewHope512-CCA-KEM and NewHope512-CPA-KEM have a NIST security level of 1, a better description of what this specific security level means can be seen described in Section 2.5. NewHope512-CPA-KEM and NewHope1024-CPA-KEM have a NIST security level of 5. These KEMs are accompanied by NewHope PKE scheme, but unlike some of the other PKE schemes submitted to NIST this PKE scheme is not intended to be used on its own. The NewHope PKE scheme is only used to accomplish the key establishment.

Each cryptosystem is either CCA or CPA secure. The CPA cryptosystems are

built under the assumption that the public-key and private-key values are not reused over multiple exchanges. However, the CCA cryptosystems are not built under the same assumption. Since LWE key exchanges are known to be vulnerable under CCA when one party is using the same key values for multiple exchanges. In order to combat this the NewHope creators added extra security within their CCA implementations, to secure against these known CCA attacks. These added security measures causes the CCA implementations to run slower, which will be tested throughout this research as well. Since, the CCA implementations add extra security measures to secure against the reuse of key values this KEM is best utilized in a system where not many assumptions are being made. The CCA implementations also has the same security measures of the CPA implementations with the added security of private/public value reuse. Later, if a vulnerability is found against a CPA then it is safe to assume that the codebase for the NewHope-CPA implementations will have to be altered.

2.10.3 NewHope Cryptosystem Package

When the provided cryptosystems are built they produce an executable. When that executable is ran it produces a request and response file. Each of those correspond to a file provided within the KAT directory, that is used to verify the correctness of the implementation. This can be seen in more detail in Section 3.4. The importance of this is the ability to verify the implementation is done correctly. All of this is provided within the package submitted to NIST. However, that is not all that is included in the package. In fact, for each of the four KEM implementations there is a referenced implementation and an optimized implementation. The referenced implementation is commented thoroughly, that way the algorithm implementation can be identified throughout the code and can be traced back directly to the documentation. However, the optimized implementation is not written for direct comparison to the documented

algorithm, allowing it to be written in a more optimized fashion. They also included an additional implementation folder which includes an implementation for the use of vector extension. This implementation is not utilized for this research. Also included is a directory containing Python scripts. This directory contains a `PQsecurity.py` script that is utilized to analyze the cost of primal and dual attacks. This is then used to analyze the bit level security for NewHope. The other scripts in this directory are `failure-512k8.py` and `failure-1024k8.py`. These scripts are used to determine the decryption failure rate of the NewHope512 and NewHope1024 KEMs. Through the use of these scripts, the NewHope authors determined that the failure rate for NewHope512 is less than 2^{-213} and less than 2^{-216} for NewHope1024. This failure rate is important when analyzing it against an attack outlined in the paper written by Fluhrer [45]. Also the authors outlined an attack that exploits failures in the CPA NewHope KEM's, in which about 4000 decryption requests need to be generated. Alkrim et al. [8] state that the attack in Fluhrer's paper [45] requires much larger fail rates than defined by their protocol. The authors do mention that for CCA implementions, Grover's search algorithm could produce a failing ciphertext in time $2^{-216/2}$. However, that would require an adversary to determine offline whether a ciphertext triggers a failure, which is impossible. This leads the authors to conclude that decapsulation failure does not induce a weakness. The last directory included in the submitted package is the supporting documentation, which outlines the NewHope cryptosystem as well as addresses all requirements outlined by NIST.

2.10.4 NewHope Mitigating Attack Vectors

The NewHope authors addressed the attack detailed in Fluhrer's paper [45]. That was not the only attack addressed throughout the documentation. To protect against attacks involving disclosure of randomness the PKE coins (random bytes used for

entropy), secrets are computed by hashing coins instead of using the coins directly. Another common known attack utilizes vulnerable noise distributions. Not only are timing attacks against discrete Gaussian samplers hard to protect against; they are also difficult to implement and this can be seen in Bos et al [30] and Alkim et al [10]. Due to this, NewHope authors have chosen to use a centered binomial distribution. Another concern that the NewHope cryptosystems address is the presence of constant polynomials, which can be seen as the fixed system parameter a in Bos et al [30]. The authors address this by utilizing a pseudorandom generator (PRNG) for the generation of the system parameter for every public-key. A more in-depth description of how this parameter can be utilized in an attack can be seen in the NewHope documentation in an outline of the NTRU trapdoors seen in Hoffstein et al [57] and Stehlé et al [101]. In the NewHope documentation is a description of another countermeasure known as the "nothing-up-my-sleeve" process; however, even that countermeasure can be partially abused as shown by Bernstein et al [21]. The authors state that when not strictly required it seems preferable to avoid using this process. Another attack that is addressed is the all-for-the-price-of-one attack; this attack is important to address because even if the fixed system parameter is honestly generated the security of all of connections rely on a single instance of a lattice problem. However, this is difficult to do and would require an unforeseen cryptanalytic algorithm that would allow for a costly lattice reduction to occur. Once this occurs, the lattice basis, that was once good enough, can compromise all future communications using Babai's decoding algorithm [16]. This attack is similar to 512-bit "Logjam" data loss prevention (DLP) attack seen by Adrian et al [2] which was possible within a set time due to the use of fixed primes. The suggested mitigation is to avoid the use of fixed primes. All of the mentioned pitfalls becomes avoidable through the authors suggestion to generate a fresh a for each public-key. They suggest that if the computation of a fresh a for

each public-key is too expensive that it is possible to cache and use a^2 for a limited number of instances without weakening the security against all-for-the-price-of one attacks. However, the authors have mitigated the performance impact by sampling a uniformly directly in NTT format by transferring only a 256-bit seed for a. The authors also addressed the idea of which primitive to use to accomplish a random looking polynomial using a short seed. The authors acknowledge that in Galbraith's paper [47] it is shown that using a (non-programmable) random oracle model (ROM) is overkill and the use of a PRNG should suffice, but the authors state that the use of just a PRNG is not enough to argue security. In order to solidify the fact that it is not enough to argue security, the use the NTRU assumption to show a legitimate PRNG can still leave a scheme insecure. The authors decided to utilize a hash function, Keccak [26], that became standardized as SHA3 in FIPS-202 [100]. Keccak offers extendable-output functions (XOF) named SHAKE, that are utilized to avoid the cost of an external iteration of a regular hash function, which fits the small overhead needs of NewHope. More specificially, they have selected the use of SHAKE128 for the generation of a. They selected SHAKE128 versus selecting SHAKE256 with a small performance penalty because neither collisions or preimages lead to an attack against their proposed scheme. Many of these attacks are avoided by not relying a globally chosen public parameter a, this causes a slight efficiency disadvantage. However, by doing so the creators are able to defend against backdoor attacks, allow trusted generation of this value, and prevent attacks relying on a single instance of the lattice problem similar to [2].

2.10.5 NewHope Parameter Sets

It is important to discuss the parameter selection for NewHope512 and NewHope1024, shown in Figure **??**. This is important because the parameter sets cannot be chosen

freely and must be chosen specifically for the design of the scheme. The parameters in Table 3 fully define the parameters of NewHope and all other intermediary parameters can be calculated from those parameters.

The intermediary parameters are shown in Table **??**. As far as parameters not being able to be freely selected there are several reasons behind this. The dimension parameter, n, must be selected as an integer that is a power of two to support efficient NTT algorithms and to maintain the security properties of RLWE. By selecting an n that is an integer that is power of two, maintains the security properties of RLWE because choosing one that is not comes with several complications that can be seen in Lyubashevsky et al. [71] and Peikert et al. [87]. However, more specifically defining polynomial of the ring cannot have the form $X^n + 1$ anymore. If n is not chosen to be a power of two integer the scheme would be rendered insecure. That is not the only limitations placed on the dimension parameter, n must also be greater than or equal to 256 because of the encoding function that needs to encode a 256 bit message into a n-dimensional polynomial in NewHope-CPA-PKE. Another parameter that is limited for the support of efficient NTT algorithms is the modulus, q. The parameter q is an integer prime that is selected such that $q = 1 \mod 2n$. The noise parameter, k, is selected so that the probability of decryption errors is negligible. The authors go on to note that on a high-level the security of NewHope depends on (q, n, k) where

Table 3: NewHope parameters and derived high-level properties based on Alkim et al. [8]

Parameter Set	**NewHope512**	**NewHope1024**
Dimension (n)	512	1024
Modulus (q)	12289	12289
Noise Parameter (k)	8	8
NTT parameter (γ)	10968	7
Decryption error probability	2^{213}	2^{216}
Claimed post-quantum bit-level security	101	233
NIST security strength category	1	5

Table 4: NewHope's intermediary parameters. Based on Alkim et al. [8]

Intermediary Parameters	NewHope512	NewHope1024
$\gamma = \sqrt{w}$	10968	7
w	3	49
$w^{-1} \mod q$	8193	1254
$\gamma^{-1} \mod q$	3656	8778
$n^{-1} \mod q$	12265	12277

a larger n and a larger $\frac{k}{q}$ leads to a higher security level. The parameter γ does not directly affect the security of NewHope but it is needed for correctness and selected to be the smallest possible value. A more precise look of how the NTT is implemented effieciently is described in the NewHope documentation as:

> The NTT defined in $\mathcal{R}_q$ can be implemented very efficiently if n is a power of two and q is a prime for which it holds that $q = 1 \mod 2n$. This way a primitive n-th root of unity ω and its square root $\gamma = \sqrt{\omega} \mod q$ exist. By multiplying coefficient-wise by powers of γ before the NTT computation and after the reverse by powers of $\gamma^{-1} \mod q$, no zero padding is required and an n-point NTT can be used to transform a polynomial with n coefficients

The authors note that it is possible to select a different parameter value for q. However, there is no belief that it will improve performance or improve the performance/security trade-off. It is important to note that if a different value for q is chosen then the value for k will also have to be adapted. Also, in general for RLWE schemes q does not have to be chosen as a prime. However, since NewHope directly uses a negacyclic NTT parameters must be chosen such that q is prime and $q = 1 \mod 2n$. The authors note that a scheme without restriction to the q would look significantly different than NewHope from an implementers standpoint.

The authors also mention a parameter set that the refer to as, NewHopeLudicrous. This parameter set is unlikely to be needed, but is used when trying to achieve a higher security level while maintaining confidence in the RLWE assumption. This

parameter set increases the dimension n to 2048 and k to 8. Choosing this parameter set has the potential to double execution time and sizes of public-keys, ciphertexts, and secret keys. It is also possible to increase the security of NewHope-CPA-KEM in an empherical setting where decryption errors are less critical, to achieve this k could be slightly increase to a value of $k = 16$.

2.10.6 NewHope Key Encapsulation

Utilizing one of the selected NewHope KEM's a shared key can be created and distributed to the other party in the communication. This is done by utilizing NewHope's CCA-PKE inside of both the CCA-KEM and the CPA-KEM. This means that the basis of the key encapsulation is the same. To demonstrate how a key encapsulation process occurs Figure 5 was created. This figure is created using the NewHope CPA-KEM because it is the more simple of the two. Also, NewHope CCA-KEM utilizes the same functionality as NewHope CPA-KEM but with some added security measures to protect against attacks targeting key reuse vulnerabilities.

NewHope KEM's utilize SHAKE256 which is a strong hashing function that is specified in [96]. The following variables in this figure $\hat{a}$, s, $\hat{s}$, e, $\hat{e}$, $\hat{b}$, s', e', e'', $\hat{t}$, $\hat{u}$, v, and v' are all polynomials in a ring, $\mathcal{R}_q = \mathbb{Z}_q[x]/\langle f(x)\rangle$ such that $f(x) = x^n + 1 \in \mathbb{Z}[x]$. SHAKE256 takes two parameters, the first indicates the number of output bytes and the second indicates the input data byte array. The rest of the functions seen in the figure are all functions defined within the NewHope codebase. The GenA function creates a polynomails in $\mathcal{R}_q$ from a given seed. The Sample function is utilized to sample polynomials from a centered distribution, and its parameters are a seed and a nonce that indicates domain separation. The nonce is an important feature as it allows for multiple polynomials to be sampled with a single seed. The PolyBitRev function reverse the bits of polynomial because the NTT function assumes that the

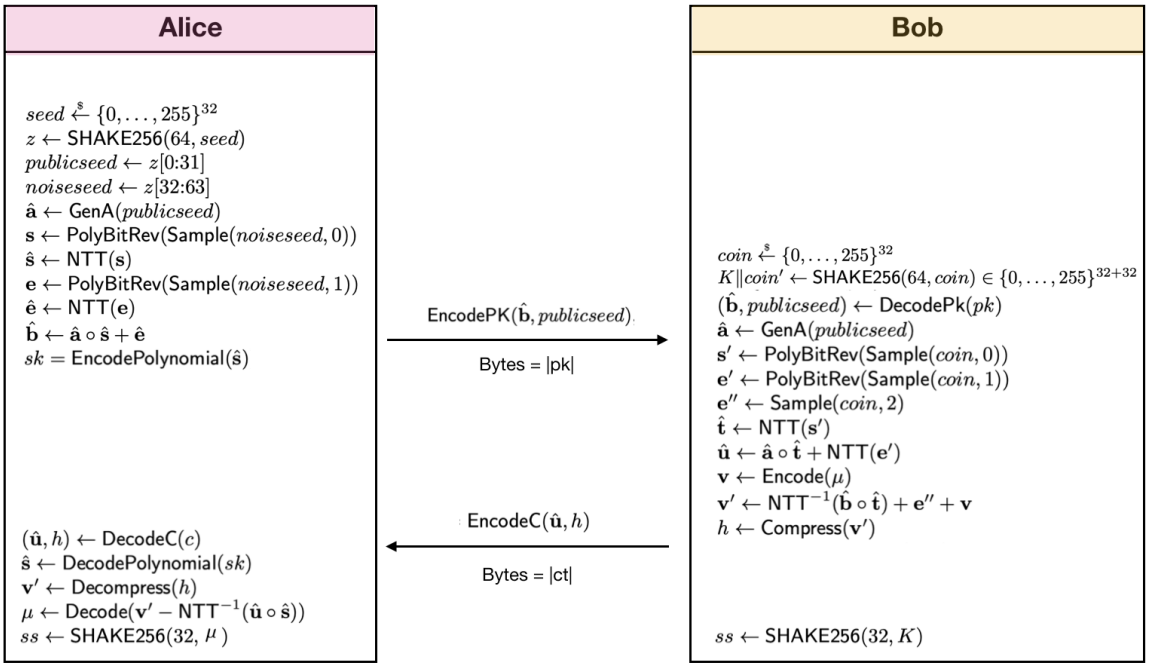

Figure 5: Implementation of a NewHope-CPA-KEM

passed parameter has its coefficients in bit-reversed order. The EncodePK function is used to encode into a byte array using a polynomial in the ring and a seed. The DecodePK function is used to decode the public-key back into the original byte array and polynomial. The Encode function encodes a byte array into an element of the polynomial ring, and Decode function decodes it back to a byte array. The Compress and Decompress functions perform coefficient-wise modulus switching between modulus q and modulus 8 by multiplying by the new modulus and then performing a rounding division by the old modulus. EncodeC is a function that encodes the ciphertext (ct) into a byte array using a polynomial and compressed byte array. The DecodeC function decodes the byte array into a polynomial and a byte array that needs to be decompressed. Finally, the DecodePolynomial converts a byte array into an element of $\mathcal{R}_q$

Within Figure 5 it is assumed that Alice is initializing communication with Bob. To begin Alice generates a seed, and this seed obtains its randomness from byte

arrays. It can be seen that the seed value is set utilizing a byte array with 32 values in the range of 0 to 255. Next, the SHAKE256 function is called to obtain 64 random bytes by utilizing the seed value as input. From those 64 bytes the first 32 are used as the publicseed and the second half are set as the noiseseed. The public seed is the utilized to generate a public polynomial, $\hat{a}$. After, a polynomial, $\hat{S}$, is created using the noiseseed and a nonce of zero; This value will also serve as Alices secret key. Then the polynomials bits are reversed and it passed in the NTT function. The same procedure is done but with a nonce of one. The produced polynomials are then utilized to create a polynomial in the for $\hat{b} = \hat{a} \circ \hat{s} + \hat{e}$, which is a common form used within the RLWE problems. This polynomial and the publicseed value are then passed into the EncodePK function to produces Alice's public-key value. Alice then sends her public-key value to Bob. Once received Bob can begin creating the shared secret value, ss.

First Bob samples a byte array from a centered distribution and stores that value into a coin parameter. From there the SHAKE256 function is called, with the coin as a seed, to obtain 64 bytes. From the 64 bytes the first half is stored into a K value and the second half will be stored in $coin'$. Bob then decodes Alice's public-key using the DecodePK. Using the same public-key as Alice, Bob then generates the same $\hat{a}$ polynomial as Alice. From there he creates three polynomials, s', e', and e'', and then he reverses the bits to s' and e'. Passing the s' and e' polynomials through the NTT, he creates two new polynomials $\hat{t}$ and $\hat{m}$ which are the NTT($\hat{s}$) and NTT($\hat{e}$) respectively. Following, Bob is able to create a new polynomial using polynomial multiplication such that $\hat{u} = \hat{a} \circ \hat{t}\hat{m}$. The message, μ=K, that is intended to be sent is the encoded into a polynomial. After, Bob constructs a new polynomial which equals the NTT^{-1} function on Alices public-key multiplied by the $\hat{t}$ variable adding an error polynomial and the encoded message. This is done so that Alice can use

Bob's public-key, which he will send, and her secret key to obtain the shared secret key. Then, Bob compresses that produced polynomial into a byte array. Finally, he uses the EncodeC function to encode the ciphertext using the $\hat{u}$ polynomial and the compressed v' polynomial. He then sends the ciphertext to Alice so that she can begin obtaining the shared secret key.

To obtain the shared secret key Alice starts by decoding the ciphertext. She then decodes her secret key to obtain her original $\hat{s}$ polynomial. Then by decompressing the h byte array she can perform a series of reverse operations to obtain the message, which is the encoded K value. Then Alice and Bob can both perform the SHAKE256 function with an output of 32 bytes to obtain a shared secret key that they both have.

As you can see from above, this follows the format of the RLWE problems, and if Alice's $\hat{s}$ value is found then the messages could be intercepted and the shared secret could be calculated. As stated in Section 2.8.2.5, solving the RLWE problem would require that enough transmissions are captured to rebuild the lattice and then the NP-hard CVP would need to be solved.

2.11 Java Native Interface (JNI)

Since NIST required the submitted code to be in C, it was then a question of how to implement the C code on a mobile phone. The options were to rewrite the C codebase to be a purely Java codebase or use a JNI to implement the native C functions in an Android Package (APK). For this research effort, the decision was ultimately made to implement the C codebase through the use of a JNI, this is similiar to how Malina et al [73] implemented NewHope on their Android device. More reasoning behind the selection of using a JNI versus rewriting the existing codebase in Java can be seen in Section 3.9. Since a JNI was utilitzed, this section is dedicated to giving an overview of how JNI's work, how it is implemented in Android Studio, and what considerations

had to be made when using a JNI.

2.11.1 JNI Basics

This entire section is referencing the Oracle documentation on the Java Native Interface Specification [72]. A JNI is a native programming interface that allows Java code that runs inside a Java Virtual Machine (JVM) to interoperate with applications and libraries written in other programming languages, such as C, C++, or Assembly. Developers should consider the use of a JNI when they need to utilize the JNI to write native methods to handle situations where an application cannot be written entirely in Java. Within the documentation, Oracle outlines several reasons that would require the use of a JNI. They include but are not limited to: Java alone does not support the needs of the application, the standard Java class library does not support the platform-dependent features needed by the application, or you already have a library written in another language that the application needs to be accessible to Java code through the JNI [72]. This research effort finds the need for a JNI due to the fact that OpenSSL and the NewHope codebase implemented is written in C.

An important benefit of utilizing a JNI is that it does not add any impositions on the JVM vendors to change the functionality of their underlying Virtual Machine (VM). This means that JVM vendors can add support for the JNI without affecting other aspects of their VM. This also means that a programmer can write a version of a native application or library and expect it work on any JVM that supports JNI. This is a tremendous benefit to have because when writing a native application or library you are not confined to any specifications that would be JVM specific. Utilizing a JNI gives programmers a multitude of options that are available to utilize on native methods. Programmers can use native methods to create, inspect and update Java objects, call Java methods, catch and throw exceptions, load classes and obtain class

information, and perform runtime type checking.

The Oracle JNI documentation [72] has a section that addresses native method programmers. The documentation states that native programmers should program to the JNI, this will protect the programmers from unknowns that might affect the end user, such as the vendors VM the end user might be running. Programmers conforming to the JNI standards will give the native library the best chance to run on any given VM.

2.11.2 JNI Within Android Studio

This section describes how to create a native C library which can be utilized from within Java code within Android Studio. Android Studio utilizes a Native Development Kit (NDK), CMake, and Gradle to achieve a JNI. The Android NDK is a set of tools that allows for the use of C/C++ within Android and it provides platform libraries that can be used to manage native activities and access physical device components [11]. The NDK uses CMake to compile the C/C++ code within the application. CMake is an open-source set of tools that is used to build the C/C++ code by utilizing its compiler to generate native makefiles from independent configuration files to build the native library [63]. Gradle is an advance open-source build tool that manages dependencies and allows the user to define custom build logic [51].

The first step in creating a JNI environment is to build a native library. In order to create a native library, it is required to have an independent configuration file that indicates how the native library should be built. Within Android Studio the default file name is `CMakeLists.txt`. Within this text file the add_library function will be utilized to name the native library and indicate which source files will be utilized to build the native library. It is also possible to define paths that the native code

might need to access (i.e., dependencies required to compile the code). In this file, the native library can also be linked to other required libraries needed for compilation. For example, NewHope requires the OpenSSL crypto library and within this file the native library will be linked to the OpenSSL library. These libraries can either be explicitly defined or the find_library function can be utilized to locate the desired library.

After defining how the native library needs to be built, the application Gradle build settings need to modified. There are a number of different Gradle build scripts throughout the project but the one that will need to modified to specify how to build the native library is the application Gradle build settings. Within this Gradle file the external native build settings need to be added to contain the CMake path variable and the cmake version. The CMake path variable will be set to the path of the `CMakeLists.txt` file and the CMake version needs to be set to the appropriate CMake version. Also, within this file it is possible to set flag variables to be used for C/C++ compilation.

The last step is to utilize the JNI within the Android codebase. To access the native library from within the Java code the native library has to be loaded into Java code and the native functions that are used have to be declared within the Java code. After, the Java code is able to call native C/C++ functions. In order for the C/C++ functions to be able to modify the Java environment or return Java variables the C/C++ functions will have to be declared with the Java environment and Java object parameters. More in depth examples can be seen in Appendix A for the Android Java code and Appendix B.

2.12 Chapter Summary

In Section 2.2, a brief overview of quantum computers and quantum computing was provided. This section resulted in the definition of large-scale quantum computers and provided an insight in the quantum mechanic properties that could present an issue on the successful creation of large-scale quantum computers. In Section 2.3, a look into how Grover's and Shor's algorithms will affect current cryptographic methods. In Section 2.4, an overview of post-quantum cryptography was provided. This section also contains the motivation into PQC research efforts.

In Section 2.5, an overview and timeline of NIST PQC Standardization Project was provided. This section provided valuable insight into a project that is contributing the the development of quantum-secure cryptosystems.

Section 2.6 provided a literature review of IoT and mobile device implementations of quantum-resistant algorithms. Examining the available literature provided valuable insight into known complications and expectations for PQC implementations.

Section 2.7 provided a table of PQC KEMs that have progressed to the current round of NIST's PQC Standardization Project. This section will be utilized in the following chapter for the selection of a PQC KEM to implement on an Android device.

In Section 2.8, background knowledge was provided on lattice-based cryptography because NewHope's presumed quantum hardness is a derivative of lattice-based cryptography.

In Section 2.9, an in-depth literature review of the available literature on cryptanalysis of lattice-based cryptographies was provided. This is important to understanding the types of vulnerabilities NewHope might be vulnerable to. An important take away from this section, is that there is currently no known attack that takes advantage of the added structure a ring adds to a lattice, but instead all attacks on the RLWE problems took advantage of the underlying RLWE problem.

In Section 2.10, background knowledge for the NewHope cryptosystem is provided. This section provides necessary details into understanding how to accomplish an implementation of NewHope. An important product from this section is Figure 5 which gives visual representation as to how the underlying NewHope-CPA-KEMs are accomplished.

Lastly, in Section 2.11 knowledge into a JNI was provided along with how a JNI environment can be achieved within Android Studio. This is pertinent to this research effort because the design choice was made to implement NewHope as a native C library that is interacted with through the use of a JNI.

III. Methodology

3.1 Chapter Overview

This chapter describes the methodology used throughout this research effort.

Section 3.2 outlines the methodology procedure for this research effort. This section provides a step-by-step view of the methodology approach.

Section 3.3 provides the justification for choosing the NewHope key encapsulation mechanism (KEM) and the criteria that was used to select the algorithm. KEM selection was achieved by using National Institute of Standards and Technology (NIST) post-quantum cryptography (PQC) Standardization project, and the documentation of each Round 2 KEM candidate algorithm.

Section 3.4 outlines the steps taken to build and execute the NewHope codebase. The NewHope codebase was built to verify the correctness of the codebase and allow for reverse engineering of the codebase. Section 3.5 outlines the steps that were taken to reverse engineer the NewHope KEM scheme. Throughout this effort reverse engineering refers to the process of using a codebase to understand the KEM scheme and the transmission requirements. Section 3.6 provides details about transmission byte requirements, key, and ciphertext sizes for the NewHope KEMs. This data will be used when comparing NewHope implementations to the available work by Malina et al. [73].

Section 3.8 provides a description of the preparation required to complete the mobile implementation of the KEM. The preparatory steps include device selection, installing any software that is required, building Android specific OpenSSL libraries, and creating a Java Native Interface (JNI) environment to achieve NewHope KEM mobile implementation.Section 3.9 provides an overview of how a mobile NewHope C/JNI codebase was created and a description of how the implementation was achieved.

In addition, this section provides code functionality that was added to create a codebase that was more indicative of real world use than the original codebase.

Lastly, Section 3.10 provides details on Android Studios Android Profiler and the efficiency tests that were ran on the NewHope KEM mobile implementations. Also, this section provides a testing procedure and a description of what comparisons were used to analyze the data that was collected.

3.2 Methodology Procedure Outline

This section provides a step-by-step procedure on the methodology approach used for this research effort.

1. A PQC KEM was selected
2. The NewHope codebase was obtained and built for correctness
3. The codebase was reverse engineered to obtain a full understanding of transmission requirements for the NewHope KEM scheme
4. From the NewHope codebase the NewHope transmission byte requirements, key and ciphertext sizes was obtained
5. Using a modified Android Studio C/JNI example a JNI baseline was created to be used in comparison
6. Steps were taken to ensure a proper environment to complete a NewHope mobile implementations
7. NewHope codebases were created for each NewHope KEM scheme
8. The created NewHope codebases were implemented on a Google Pixel

9. Using Android Profiler within Android Studio the NewHope mobile implementations were tested for CPU, energy, and memory usage. More details about the testing procedure is outlined in Section 3.10.2

10. Comparisons to available work, the created JNI baseline, and comparisons between NewHope KEM schemes are made using the data that was collected from each NewHope mobile implementation

3.3 Choosing a KEM Algorithm

When choosing a quantum-secure KEM for mobile implementation several considerations were made, the algorithm had to work well on a mobile phone and had to have a presumed quantum-secure cryptographic primitive. After NIST posted their call for proposals for their PQC Standardization Project, and received several submissions. NIST narrowed their selections down using competition-like rounds and this was detailed in Section 2.5. NIST is currently in Round 2 of selections, that was the starting place for looking for PQC KEM algorithms to be considered for mobile phone implementation.

Based on current literature one of the most promising post-quantum public-key encryption (PKE) and KEM schemes are based on lattice-based cryptography, more specifically the ring learning with errors (RLWE) problem, which is described in detail in Section 2.8.2.5. This is due to RLWE cryptosystem having small keys and ciphertext values due to the added structure of the ideal lattice. As seen in Section 2.9, the only attacks that are known to take advantage of the added structure of an ideal lattice target the use of short generators in lattice construction. If RLWE cryptosystems use a sufficiently large generator they are not known to be vulnerable, and they should be considered over learning with errors (LWE) cryptosystems because RLWE cryptosystems have better time and computationally efficiency.

Utilizing NIST Status Report on Round 2 Submissions [5] and the documentation for each submitted candidate KEM, selection eliminations were made based on key sizes, known attacks, or security concerns. Section 2.7 provided more specifics on the selection of NewHope.

This led to the selection of NewHope KEM algorithms. The authors of NewHope claim that NewHope is fast in software and achieve simplicity in terms of implementation [8]. Also, they claim that implementations on ARM devices are fast; this makes the NewHope cryptosystem a good choice for mobile phone implementation with a targeted device of a Google Pixel with an ARM64 processor [8]. Some of the disadvantages that are acknowledged by NewHope creators are parameter restrictions due to the use of Number Theoretic Transform (NTT) and RLWE cryptosystems having more structure compared to LWE cryptosystems [8]. As stated in Section 2.9, there is no known attack that targets the added structure of RLWE cryptosystems. So this disadvantage has yet to be proven to be a hindrance. The NewHope KEM cryptosystem is described in more depth in Section 2.10.

3.4 Building the Original KEM Codebase

Each NewHope KEM codebase that was submitted to NIST was downloaded and built to ensure the correctness of the codebase. The NewHope code base was verified for correctness using the known answer tests (KAT) provided to NIST by the NewHope creators.

In order to properly build the NewHope codebase, OpenSSL was required. More specifically, the Crypto library from OpenSSL was needed to build each version of the NewHope KEM; this was identified in each NewHope `makefile`. Within the `makefile` the build flag -lcrypto was used and refers to the use the OpenSSL crypto library.

After properly building and installing OpenSSL, it was possible to build the each NewHope codebase. This was done by following the make commands outlined within the `makefile`, simply run `make` if this was the first time building the executable and run `make clean` if the executable was previously built. Each provided NewHope `makefile` was modified to enable the `make clean` command to remove the request file, response file, and the executable. Additionally, each NewHope `makefile` had to be modified to include the correct path for the C compiler.

The built executable produced two files, a request file and response file. The produced files are intended to be identical to the provided KAT. Within the request file there was 100 produced seeds. The response file contained a count to label the seed values to 100, the seed, and empty values for private key, public key, ciphertext, and shared key. The response file was produced by reading in the seed value from the request file and using the seed value to produce the corresponding private key, public key, ciphertext, and shared key values. To create the full response file this was done iteratively for all 100 seed values in the request file. The original functionality of the NewHope codebase was not indicative of real-world use because the key values were not produced directly after generation of a random seed. Also, the original codebase pulled seed values from a file which, for security purposes, is not a viable solution when creating the mobile implementation.

The request and response files created when running each executable was compared to the provided KAT. An example comparison can be seen in Figure **??** and it shows the count, seed, and shared key where each of these values originate from the response files. Comparison of these files was done by using the `cmp` unix command because the produced files, when built correctly, will be the same as the KAT files. The `cmp` command will output the first line that the files differ and if the files are identical, nothing will be output. if the files differ in any way and return the number

of the first line that the files differ. Figure **??** shows an example of this unix command on the produced NewHope512-CCA-KEM response and request files compared to the corresponding KAT files. For each NewHope KEM no differences was returned and the files produced were the same as those provided for the KAT.

Each NewHope KEM was built, executed, and tested for correctness. The executables produced files that were identical to the KAT; it was concluded that each NewHope codebase was correct and could be used as a basis to build each NewHope mobile implementation. The original codebase could not just be placed on the phone and used because it was not applicable for the intended purpose of creating a singular shared key. However, there are some source files and functionalities that could be utilized within the new codebase.

3.5 Reverse Engineering NewHope KEM

After building and verifying the correctness of the original codebase it was required to revers engineer the original codebase. Within this research effort reverse engineering refers to the analysis and dissection of a codebase to gain a complete understanding of how the algorithms and schemes within the codebase are implemented. The NewHope codebase was accompanied with documentation that outlines algorithms but does not provide codebase and transmission requirement specifics. Therefore, reverse engineering the codebase was required.

After building the NewHope codebase, it was required to parse through the source code to identify the purpose of each submitted source file as this was not outlined throughout documentation. This had to be done in order to determine which files could be utilized in the mobile implementation, and which files had to be recreated. The only file that was not usable for the mobile codebases was the driving source file, `PQC_genKAT_kem.c`. This was an indicator that, most of the functions provided

in the NewHope codebase could be reused. More specifically, the files that were able to be reused built the underlying RLWE structure of the KEM, PKE schemes, chosen plaintext attack (CPA) schemes, chosen ciphertext attack (CCA) schemes, hashing algorithms, and NTT. However, the driving file had to be rewritten to use the remaining functionality from the original codebase and additional functions to achieve a shared key.

Throughout the reverse engineering process it was realized the transmission requirements between parties and how the transmitted data is created. Section 2.10.6 explains the results from reverse engineering the NewHope codebase. Figure 5 was a created transmission diagram between two parties, Alice and Bob. This diagram was created utilizing algorithms defined within the NewHope documentation [8]. The first portion of the diagram shows Alice computing her public key and secret key, this was seen in the documentation as the CPA-PKE key generation algorithm [8]. Alice then sends her encoded public key to Bob, the next portion of the diagram is Bob's computation. Bobs computations are a combination of NewHopes CPA-KEM encapsulation and CPA-PKE encryption algorithms [8]. After sending the encoded ciphertext to Alice, she then begins her final computations which are a combination of NewHopes CPA-KEM decapsulation and CPA-PKE decryption algorithms [8].

Through the use of reverse engineering it was possible to identify source files that could be reused in a mobile implementation and which files would have to be rewritten for the new codebases. Also, reverse engineering the codebase was required to understand the transmission requirements between the two parties, Alice and Bob.

3.6 Analyzing Transmission Byte Requirement

Although the bytes that are transmitted to establish a shared key is constant, dependant on the selection of NewHope KEM variation, it is important to examine

the number of bytes that need to be transmitted because the set of constraints a device possesses. In Table 5, the KEM-specific values can be seen. These values were obtained from the `api.h` and `params.h` source files from within the original codebases, an example from the NewHope512-CCA-KEM codebase can be found in Appendix C and Appendix D respectively. The definition for these value sizes was found by first examining `api.h` to see what the values are being mapped to within `params.h`. Then within `params.h` a definition of each of these values was located and calculated. These values all differ depending on which NewHope KEM is selected for implementation and Table 5 indicates the byte sizes for each.

Table 5: Size of seeds, shared secret keys, public keys, secret keys, and cipher texts of NewHope instantiations in bytes

Size of Values Used in NewHope Instantiations in Bytes					
Parameter Set	**seed**	**|ss|**	**|pk|**	**|sk|**	**|ct|**
NewHope512-CPA-KEM	48	32	928	869	1088
NewHope1024-CPA-KEM	48	32	1824	1792	2176
NewHope512-CCA-KEM	48	32	928	1888	1120
NewHope1024-CCA-KEM	48	32	1824	3680	2208

First, these values can be utilized to examine the constraints of the target device and decide which NewHope KEM variation is most suitable. So if the target devices are extremely constrained on memory and are used in a ephemeral setting, public/private keys are freshly generated for key generation, then a CPA version of NewHope would be preferred because it has smaller values and does not require the added security against key reuse attacks. Another constraint that should be examined is the bandwidth on which these devices will be communicating. As seen in Section 2.10.6, the values that would initially need to be exchanged between Alice and Bob is Alice's public key value followed by Bob's response of the calculated ciphertext. So, when choosing which NewHope KEM to implement one should evaluate the intended bandwidth and the amount of bytes needed for key establishment. Those are just a

few examples of how these values can affect the choice of which KEM to implement, and becomes more important when implementing on a more constrained device than the modern day mobile phone. With that being said, these values did not affect this mobile phone implementation due to the increasing resources that are available in current mobile phones.

3.7 Creating a Baseline Application for Comparison

Due to the lack of available research on mobile phone implementations of PQC KEM, some additional data was desired to compare how the NewHope KEM mobile implentations run to other applications. This prompted the need for a C/JNI application to compare the NewHope test results to. In order to get an application that would be suitable for comparison to a NewHope mobile implementation, an application that utilizes a JNI was selected. To be more specific, an Android Studio JNI example code, `HelloJNI` found in Android Studio example repository, was downloaded and modified to provide an baseline JNI application. The example code was modified to contain a native function call within an OnClick event. The native C function obtains the application binary interface (ABI) of the device and creates a string using the ABI and a random number. The OnClick event then prints this string to the textview. Although this application does not require any heavy computations, it was still valuable to use in comparing consumed resources because it was used to conclude if the additional computations made throughout the native NewHope codebases caused an increase in resource consumption. This codebase was sampled using the same testing procedure outline in Section 3.10.2. The results from testing the JNI baseline are compared to the NewHope implementations by using a two-sample two-tailed T test.

3.8 Preparation for Mobile Implementation

After selecting NewHope KEM for implementation there were some preliminary preparatory steps that had to occur. Including downloading and installing software that was needed: Android Studio, Android Native Development Kit (NDK), OpenSSL, NewHope codebases, and NDK toolchains for x86 and ARM 64 architectures. These were required to achieve a mobile implementation of the NewHope PQC KEM schemes.

First, a mobile device had to be chosen for implementation and despite being selected because of availability; it was a good choice for implementation because, as of the end of 2018, it was the number three selling premium smartphone in the United States [86]. Specifications for the Google Pixel are as follows: OS: Android 9 (Pie), chipset: Qualcomm MSM8996 Snapdragon 821 (14 nm), CPU: Quad-core (2x2.15 GHz Kryo & 2x1.6 GHz Kryo), GPU: Adreno 530, and memory: 32 GB of memory with 4 GB of RAM.

Next, there was a design choice that had to made of which programming language to use to implement the NewHope KEM schemes. The full NewHope Round 2 submission package was downloaded from NIST PQC Standardization Project website, and more information on the NewHope package can be found in Section 2.10. The codebase for NewHope was written using C. Since this research effort was to put NewHope on an Android mobile device, the desired codebase would need to utilize Java. Alkrim et al. [7] mentioned an open source Java implementation of NewHope-Usenix. However, since publication the codebase mentioned by Alkrim et al. [7] was either removed or relocated. However, if the codebase was found, significant code changes would be required in order to achieve a NewHope implementation instead of a NewHope-Usenix implementation. Therefore, the choice was made to implement the NewHope KEMs using a native C library through the use of a JNI.

The JNI is accomplished by using Android Studio, NDK, Gradle, and CMake. Section 2.11.2 provides more specific information on the use of a JNI. The JNI allows for the use of some of the NewHope C code that was submitted to NIST as native functions in a native library. Within Android Studio the submitted code was placed in a `cpp` directory. When the project was built, the folder was compiled into a native library that Gradle would package into the final Android Package (APK). The written Java code could make function calls to functions contained in the native library through the use of JNI. The choice to use Android Studio is based on usability and the application testing tools available within Android Studio. NDK, Gradle, and CMake were chosen because this software is required to accomplish a JNI within Android Studio.

In order to build the codebase for a Google Pixel it was necessary to build OpenSSL for the specific Android architecture while designating an APK level. In order to build OpenSSL for Android devices additional NDK toolchains were needed. Two different toolchains needed to be created, one for x86 devices and one for ARM 64 devices. OpenSSL was needed for an x86 Android device for debugging purposes because there is no Android Studio x86 emulated device at the required API level. Additionally, OpenSSL was needed for an ARM 64 device because that is the targeted architure of the Google Pixel. To create the x86 and ARM 64 NDK toolchains the following commands were used, respectively: `./build/tools/makestandalonetoolchain.sh --toolchain=i686linux` and `./build/tools/makestandalonetoolchain.sh toolchain=arm-linuxandroideabi4.8`. Both of these commands produced directories that contained the 32 bit and 64 bit toolchains.

After building the correct toolchains, OpenSSL was built for the Android devices. To build OpenSSL the Android build instructions were followed, modifying paths and environment variables.

Building OpenSSL produced two libraries `libssl.a/so` and `libcrypto.a/.so`. The `libcrypto` library was needed for the NewHope codebases. This means that the library needed to be added to the project for the native code to access. This was accomplished by adding several lines to the `CMakeLists.txt` file; for a look into a well-commented `CMakeLists.txt` file see Appendix E. Section 2.11.2 provides descriptions of the uses of the `CMakeLists.txt`, CMake, and Gradle.

Within `CMakeLists.txt` the directories that the native files need additional dependencies from must be added; this is done by using an `include_directories` function and providing a path to the directories to be added to the project. Next, using an `add_library` function a shared native library utilizing the native C source files was created.

Next, in the `CMakeLists.txt`, the OpenSSL crypto library is added. This was accomplished using two functions, an `add_library` function and a `set_target_properties` function. The `add_library` function allows a definition for a static imported library and the `set_target_properties` function is used to declare the path to the library being imported.

After adding the OpenSSL Crypto library within `CMakeLists.txt`, it was necessary to find the log library, `log-lib`, included in the Android NDK to use logging capabilities throughout the mobile application. This was done using a `find_library` function. It was possible to add the OpenSSL library by using the `find_library` function. However, this could of presented issues because there were OpenSSL libraries in multiple locations. For example, OpenSSL libraries was installed within the system but there were Android-specific libraries located within the Android NDK home directory.

Next, it was time to link both the log library and the OpenSSL Crypto library to the native library in `CMakeLists.txt`. To accomplish this the `target_link_libraries`

function. The `target_link_libraries` function links the imported-crypto-lib and the found `log-lib` to the `native-lib` that was built from the created NewHope codebase. To see all the `CMakeLists.txt` source with the thorough comments see Appendix E.

After the `CMakesList.txt` was created, the `build.gradle` application settings was modified; Gradle builds the native library using the CMakesList.txt as an independent configuration file. This was accomplished by adding a line to the `build.gradle` applications settings that specifies the path for the `CMakeList.txt` file and indicates which version of CMake was used.

The last modification to the application settings, was to specify the architecture of the target device, this must match the architecture used to build the linked OpenSSL Crypto library. This was completed by adding an ABI filter within the `build.gradle` application settings. The full build.gradle application settings file can be seen in Appendix F.

3.9 Creating NewHope Mobile Implementations

In order to build an APK for an Android device, a JNI was used. As discussed above when the original codebase was run, the PQCgenKAT_kem.c acted as the main source file. Within this file, two files were created a request file and a response file. These files, if the implementation is done correctly, will match the provided KAT. However, this meant that in order to build an Android implementation this file had to be modified within the JNI implementation to run as wanted. The desired Android implementation will not just create two files that correspond to the KAT results; instead the desired implementation was meant to be as indicative to real-world application as possible. With that being said, the original implementation was not indicative to real world application for a number of different reasons. The desired

implementations will not be accomplished by reading produced seed values from a file in order to produce the the shared key and the implementation will not create 100 key values by producing 100 keys then the corresponding 100 shared key values.

To produce a mobile implementation of the NewHope KEM schemes, the layout of this mobile application contains three different buttons, and a singular textview. The mobile application layout is shown in Figure 6. The first button, `Create Keys`, is used to create the keys from a random seed. This button will modify the textview based on the OnClick functionality that is chosen. The `Clear Keys` button was used to clear the textview of all text; repressing the Create Keys button will clear the textview automatically and adjust the textview accordingly. The last button was a `Test For Correctness` button; this button is used to verify the correctness of the mobile implementation through the utilization of the KAT. Below is a description of how these functions are accomplished.

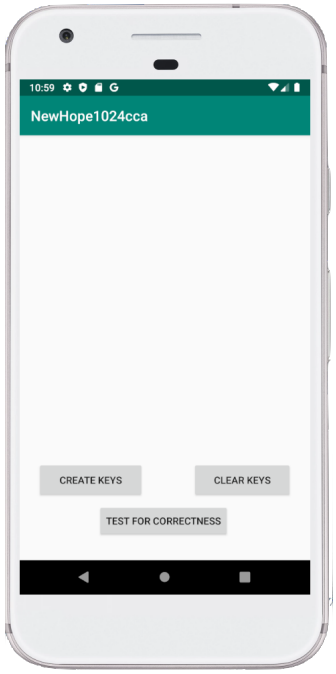

Figure 6: NewHope mobile application layout

The MainActivity.Java file contains hardcoded array values. These are the values

that are obtained from the KAT, and they were obtained by modifying the original main source file for each codebase of NewHope, this modified `PQCgenKAT_kem.c` file can be seen in Appendix G. The modifications made to this file was to produce console output that would be directed to a file such that the outputted values would mimic array initialization notation. The produced array values were verified to be identical to the values contained in the KAT. These hardcoded array values did take up memory that was not indicative of a real-world scenario; however, for the purpose of demonstrating correctness of this implementation it was necessary. Next within the file, there are several Java variables that were declared and set to match the values that resulted from the native C functions for use throughout the Java code. These variables will be discussed later in the section. Additionally within this file, were function calls made to load the native library and declare the used native functions. This had to be done in order for the Java code to properly interface with the native library and native functions. The code used to load the native library was

```
static{
    System.loadLibrary("native-lib");
}
```

and the code that was used to declare the native functions was

```
public native void RequestString();
public native void CorrectnessTest(String[]
    receivedSeedValues, String[] receivedPkValues,
    String[] receivedSkValues, String[]
    receivedCtValues, String[] receivedSsValues);
```

Also, within the MainActivity.Java file there are several different definitions for buttons along with their associated OnClick functions. The first OnClick event was for the Create Keys button; within this OnClick event the native function `RequestString`

is called and the textview was updated. This OnClick function was implemented in three different ways a verbose, succinct, and no output OnClick events. The verbose OnClick event displayed all the key and ciphertext values to the textview. The succinct OnClick function only displayed the calculated shared key value within the textview. Where as the no output OnClick event created the shared key without displaying anything in the textview.

The next OnClick event that was defined was for the `Clear Keys` button. In this event, the textview was set to an empty sting.

The last OnClick event that was declared was for the `Test For Correctness` button; in this OnClick event the native `CorrectnessTest` function was called, this function took several arrays as parameters. The purpose of this OnClick event was to verify the correctness of the mobile KEM implementation by comparing returned arrays to the hard-coded KAT arrays. If all of the values are equal, then the textview was set to indicate that the correctness check had passed. However, if the values were not equal a helper function was called that iterated through each array element and displayed each array index and which key or ciphertext values differed.

Next, the functionality of the `MainActivityDriver.c` source file will be explained. This source file contained two functions that were called from within the Java code, `RequestString` and `CorrectnessTest`. The functions had to be declared within the native library according the JNI standards meaning that The parameters must include the Java environment and an instance of the JObject class. Additional Java variables could be passed as parameters and that is the case for the `CorrectnessTest` function. For example, the native C `RequestString` function was declared using the following format:

```
JNIEXPORT void JNICALL
Java_com_example_newhope512cca_MainActivity_RequestString
```

```
    ( JNIEnv *env, jobject this){
     //NATIVE CODE GOES HERE
}
```

In this example code the function declaration is done such that `Java_com_example_newhope1024` refers to the Java package the native function interfaced with, `MainActivity` refers to the Java class the function is called from and `RequestString` is the name of the native function.

The `RequestString` function had to use helper functions. The first helper function converted a string to a hexadecimal string. The next, obtained a Java variables field id and set the Java variable to a value that was calculated from within the native C function. Within the `RequestString` function, the KEM was completed by creating a seed and then using a NewHope PKE with the seed as a parameter the users public-key and secret-key was created. After the public-key and secret-key values were created, these values were used to set corresponding Java variables. Within the C code, after the public key and secret key values were created, the public-key value was used to calculate the ciphertext and shared secret value. Again, these values were used to set the appropriate Java variables. After that was completed the secret-key was used to decrypt the message, decode the ciphertext, and calculate the shared secret key. Comparing this explanation to Figure 5, it is shown that they are equivalent but the actual code implementation happens at a higher level of abstraction due to the use of NewHope PKE and NewHope KEM functions.

The `CorrectnessTest` function follows the logic of the original NewHope code-base executable. This function created a 100 requests followed by the corresponding response values; there are a 100 response and requests because this function produced values to be compared to the KAT. The KAT contain 100 entries for both response and request. To accomplish this, 100 seed values were created and appended to its

corresponding Java array. Then, iteratively, the Java seed array was accessed to obtain the next seed in the Java array, then that seed was utilized to create the private-key, secret-key, ciphertext, and shared secret values. Those values were then appended to their corresponding Java arrays. These arrays are compared to the hard-coded values described earlier to verify correctness of each NewHope implementation. It is demonstrated that the `CorrectnessTest` function follows the same logic as the original NewHope executable, described in Section 3.4, but instead of using files this function utilizes interfaced Java arrays to achieve the same results.

After the codebase was complete the APK was compiled and built in Android Studio. Then the APK was placed on the Google Pixel and ran. After the APK was placed on the mobile phone, the correctness of all NewHope implementations was tested and verified using the `Test For Correctness` button.

3.10 Testing the KEM

After the NewHope implementation was placed onto the Google Pixel and tested for correctness, a series of efficiency and time-space complexity tests were performed to analyze each NewHope implementation. Through the use of Android Studio's Android Profiler, each NewHope KEM was analyzed for CPU, energy, and memory usage.

Analyzing the CPU usage indicated the applicability of this KEM for mobile implementation. If the KEM application consumed too much CPU it could cause other applications not to work properly or even cause too much latency in the delivery of a key for it to be reasonable to use. The data obtained from the CPU Profiler was used for analysis and could be used for code optimization.

Similarly, analyzing the energy usage was used to indicate whether the native NewHope library caused an increase in energy. Through analysis of energy usage it

was determined if the application would cause an effect on battery life.

The trace produced from the Memory Profiler indicated if the NewHope native library consumed too much memory to be implemented on a memory constrained mobile device. Memory constraint is device specific and each device would need to be analyzed. However, if the amount of memory consumption is low enough it would increase the number of platforms that NewHope could be feasibly implemented on.

3.10.1 Android Profiler

To use Android Profiler within Android Studio, the phone need to be connected to a laptop running Android Studio and was used to see the specified profiler running on the placed APK in real time. After selecting the specified profiler a trace could be recorded for the session and the profiler data collected from the application was saved. After an overall trace was recorded the usage of CPU, memory, and energy can be seen. In addition, two different CPU traces were recorded, the Java Methods trace and the C/C++ Native Functions trace. From each of those, the runtime of each function can be obtained. Within the Java Methods trace the runtime of OnClick functions and native functions were obtained by analyzing the function call chart. The Java Methods trace does not calculate the runtime of each function called within the native source, it only calculates the runtime of native C functions that were interfaced with the Java code. However, the C/C++ Native Function trace calculates the runtime of each function being utilized within the native source.

Using the Energy Profiler it was possible to evaluate the energy usage of the application and by analyzing the data it could allow for optimizations to the application, to either eliminate system calls or manage the usage of specific components which appear to be utilizing the most amount of energy. There is not any anticipated system calls that could affect the usage of the KEM. However, the Energy Profiler

allowed for an analysis of the amount of energy consumed on application startup and runtime. Also, this allowed for an analysis of how much energy the native call and OnClick events require. The energy analysis of the application is important because while the application is running, if it utilizes too much energy and causes the application to either run slowly or causes the device to run out of battery unusually fast then the feasibility of actually implementing this KEM onto a mobile device will be reconsidered.

The Memory Profiler allowed for analysis of how much memory the application consumed. Within the memory trace an analysis of the total amount of memory consumed on application startup and through runtime was completed. Within the memory trace it indicated if the memory being used came from the Java code, native code, the stack, and how much memory was being utilized by the graphics. However, that was one advantage to utilizing a KEM, once the key is obtained and in use there should not be a need to rerun the process to obtain another key until a new session is created. A new key will need to be generated at the start of each communication session, and should not be reused even if the two users had a session prior. This means that the resources consumed will only happen once per session.

3.10.2 Data Collection Procedure

Each NewHope mobile implementation had a profile trace recorded with 50 verbose OnClick events, 50 succinct OnClick events, and 50 no output OnClick events. The reason 50 samples was recorded was to collect enough data points to analyze statistical relevance. This was repeated for both the Java Methods Trace as well as the C/C++ Native Functions Trace. After all the data was collected it was compiled into tables and can be seen in Appendix H. From those results, the averages, minimum and maximums, and standard deviations was calculated using all of the data

points. Finally, the range of values on startup and throughout the entire runtime was sampled and analyzed. All of the test results and comparisons will be presented in Chapter IV.

The procedure that was used in testing the mobile implementations of NewHope KEMs was as follows:

1. Selected KEM and OnClick codebase to obtain samples from
2. Mobile application was placed onto the Google Pixel
3. Android Profiler starts trace capture on application startup
4. Began recording Java method CPU trace
5. Within the NewHope KEM application, the Create Keys button was pressed 50 times
6. Began recording C/C++ native CPU trace
7. Within the NewHope KEM application, the Create Keys button was pressed 50 times
8. Recorded runtimes of each OnClick and native function
9. Recorded runtimes of highest CPU percentage from within the OnClick event
10. Repeated for each NewHope KEM with every form of OnClick events

When recording the CPU traces the memory and energy traces were generated as well.

After collecting all of the samples and data points, they were used in comparison between each NewHope implemention, the JNI baseline application, and the available work by Malina et al. [73]. All collected data points were used when comparing

NewHope implementations amongst each other. The averages of runtimes were utilized in comparing NewHope implementations to the created JNI baseline, but more importantly the CPU usage will be compared. The runtimes of the native NewHope KEM scheme and transmission byte requirements are used in comparison to the work available by Malina et al. [73]. Additionally, the averages and standard deviations are utilized when completing t-tests to test for mean equivalence.

3.11 Chapter Summary

Section 3.2 provided an overview of the methodology procedure that was used throughout this research effort. This procedure was presented in a step-by-step fashion.

Section 3.3, presented the reasoning behind selecting the NewHope PQC KEM. Also within this section, was a brief overview of the advantages and limititions of NewHope.

Section 3.4 was an overview of how to build and verify the original NewHope codebase for correctness. Also Section 3.4 described the functionality of the original codebase and provided insight into which portions of the NewHope codebase was able to reused when building mobile implementations of NewHope KEMs. Section 3.5 provided details on the process of reverse engineering the original NewHope codebase. Also, the need for reverse engineering was explained.

Section 3.6 detailed the analysis of transmission byte requirements. The transmission byte requirements were obtained by evaluating the NewHope codebase, and the results are used when comparing NewHope to the available work by Malina et al. [73].

Section 3.7 explained the process used to create a JNI baseline application that was used in comparing the NewHope KEM mobile implementations. Section 3.8 was

a description of any preparatory steps that were taken for mobile implementation to occur. These steps included outlining the software that was needed and the steps that were taken to set up a JNI environment.

Section 3.9 was a description of how mobile implementation was achieved. This section went into details about creating the NewHope codebase and into specifics about the functionality of the codebase. Additionally, this section provided an in-depth outline on how the mobile applications were tested for correctness utilizing the KAT

Section 3.10 was the details about how the NewHope KEMs are tested. The NewHope KEMs were tested for CPU, memory, and energy usage using Android Profiler in Android Studio. This section provided an procedure for how each NewHope mobile implementation was tested. Each NewHope implementation and OnClick combination was tested using this outlined procedure.

IV. Results and Analysis

4.1 Chapter Overview

This chapter is dedicated to outlining and analyzing the results found from testing the NewHope implementations. The results in this chapter were achieved utilizing Android Studios Android Profiler, which allows for an in-depth look at the applications CPU, memory, and energy usage.

In order to verify that each NewHope implementation was done correctly, the mobile implementations were tested for correctness. This is required to ensure the scheme that is being implemented worked as intended by the creators. This process is outlined withing Section 4.2.

Section 4.3 is dedicated to sharing the results for the created Java Native Interface (JNI) baseline. The results from the baseline are important in demonstrating the performance of an application implementing a post-quantum cryptography (PQC) key encapsulation mechanism (KEM) scheme compared to an application implementing a simple JNI.

Each NewHope implementation was tested using a three different OnClick event conditions: verbose, succinct, and no output. The verbose OnClick condition printed the keys and ciphertext values to the applications textview. The succinct OnClick condition printed a statement indicating that a shared key was created and appended the shared key to the textview. Finally, the no output OnClick condition did not display any values to the textview and created the shared key. The results from the verbose, succinct, and no output OnClick event conditions are described in Section 4.4, Section 4.5, and Section 4.6 respectively.

All the results from each NewHope implementation with each OnClick description are compared in Section 4.7. The comparisons made utilized the data sampled from

average scheme runtime and CPU usage. The data is displayed in two different box plots. The CPU box plots used the JNI baseline for comparison as well. The baseline was not utilize for comparison for runtime due to the incapability of obtaining individual runtime values from within Android Profiler.

Utilizing Android Profiler, the performance of succinct NewHope implementations and the JNI baseline were compared during application startup and during application runtime. The results were obtained by evaluating memory, CPU, and energy usage and are discussed in Section 4.8.

Section 4.9 outlines the statistical analysis that was performed on the data points gained. In this section the test results were examined and a series of hypotheses were formed and tested. These hypotheses were evaluated using T-tests and the results are provided.

Lastly, Section 4.10 the results obtained from this research effort is compared to the result found in the work done by Malina et al. [73]. Malina et al. [73] implemented PQC KEM's on an ARM board and on an Android device, more information about their research effort was provided in Section 2.6.

4.2 Verification of Correctness

After all the four NewHope KEM implementations were placed on the mobile device and the applications were tested by utilizing the correctness check button described in Section 3.9. Within the Java code there was hardcoded arrays that that were equivalent to the values of the known answer tests (KAT). The KEM scheme was ran 100 times, to produce values that could be stored into Java arrays. The hardcoded KAT Java array were used to verify that the Java arrays created from the native C KEM scheme produced the same values. Using these comparisons all the NewHope implementations were verified to be implemented correctly and returned

the correct KAT values.

4.3 Baseline JNI Application

In this section the results from the JNI baseline are provided. The baseline OnClick event was sampled 50 times. From the samples the average native function time, OnClick function time, and the highest CPU usage were recorded into Table 6.

Table 6: Results from the modified Android Studio sample code used to represent a baseline for a C/JNI application

Baseline Average Results	
Native Function Time(ms)	0.021266
OnClick Function Time(ms)	0.849467
Highest CPU Usage(%)	13.184

The table shows the native function and OnClick function times in milliseconds (ms) because the results from all the NewHope implementations are shown in ms; So for consistency these values were recorded ms versus microseconds. Table 6 will be referenced throughout the remainder of this chapter for comparison purposes.

All the other KEM samplings have the full tables with all 50 entries within Appendix H. However, the baseline implementation cannot be seen within the appendix due to Android Profiler not displaying the function times under 1 ms within the function call chart. Only the total function time during execution, the total amount of runtime for each function over the 50 samples, could be obtained and this was caused by the function runtimes being too small. This means it was possible to sample the 50 highest CPU percentages but it was only possible to obtain the total function time of all 50 samples for the native and OnClick functions.

4.4 Verbose OnClick Event Condition

This section provides the results from profiling all the NewHope implementations with an OnClick function that displays all the values used to obtain the shared key to the applications textview. This provided valuable information on how the application performed with an increase of Java code. This allows for an analysis on whether or not a more Java resource consuming application might have an effect on the native function runtime. For each of the following tables the application was profiled and 50 samples for native function time, OnClick function time, and highest CPU usage was recorded. The tables displaying all individual samples can be found in Appendix H. The tables displaying the averages, minimum, and maximum for each NewHope implementation can be found in Appendix I, and are referenced as follows: results from NewHope512-CCA-KEM are in Table 32, results from NewHope1024-CCA-KEM are in Table 33, results from NewHope512-CPA-KEM are in Table 34, and results from NewHope1024-CPA-KEM are in Table 35.

In Table 7, the averages for all values for each NewHope implementations are displayed.

Table 7: Comparison of create key average times of NewHope implementations with verbose OnClick function

Average Results From Create Key Test Runs with Verbose OnClick Function				
	512-CCA	**1024-CCA**	**512-CPA**	**1024-CPA**
Native Function Time(ms)	6.941	11.691	4.588	8.267
OnClick Function Time(ms)	89.034	155.644	70.447	119.616
Highest CPU Usage(%)	21.13	25.35	19.78	23.42

Within Table 7 it is shown that the NewHope-CCA implementations had longer

function times and higher CPU usage values than the NewHope-CPA implementations. This was expected to happen due to the extra security measures added against key reuse attacks within the chosen ciphertext attack (CCA) implementations. Along with that, the OnClick function times and CPU usage of the NewHope1024 implementations are higher than NewHope512 implementations, due to the size differential between the public key, secret key, ciphertext, and shared secret.

Comparing the results from Table 7 to the baseline averages in Table 6, it is shown that the average runtime of the native NewHope scheme was longer than the average baseline native function call. This was to be expected due to the amount of code being executed in the native functions. This can also be said for the OnClick functions; this OnClick function set and display five large strings within the textview. For those reasons, it was anticipated that average CPU usage of the native NewHope KEM schemes would be larger than that of the baseline. This was proven to be correct using a Two-Sample One-Tailed T test and the results are shown in Table 14

4.5 Succinct OnClick Event Condition

Within this section the results from profiling all four NewHope mobile implementations with a succinct OnClick function are displayed; the succinct OnClick function only displays a string indicating that the shared key has been created and prints the corresponding value within the textview. Per the data collection procedure in Section 3.10.2, 50 samples were obtained and the table displaying all of these values can be seen in Appendix H. Appendix I provides tables that show the averages of native function times, OnClick function times, and highest CPU usages for all implementations. The tables in the appendix can be referenced as follows: results from NewHope512-CCA-KEM are in Table 36, results from NewHope1024-CCA-KEM are in Table 37, results from NewHope512-CPA-KEM are in Table 38, and results from

NewHope1024-CPA-KEM are in Table 39.

In Table 8, the averages of all values are shown along with their corresponding NewHope implementations.

Table 8: Comparison of Create Key average times of NewHope implementations with a succinct OnClick function

Table Of All Average Results From Create Key Test Runs with Succinct OnClick				
	512-CCA	**1024-CCA**	**512-CPA**	**1024-CPA**
Native Function Time(ms)	6.658	11.900	5.158	7.816
OnClick Function Time(ms)	13.653	18.694	11.899	13.843
Highest CPU Usage(%)	13.500	14.054	13.514	13.396

In Table 8, it is shown that the OnClick function times and highest CPU usage within this table are smaller than those found in Table 7. This is due to the difference in the values that were set to the textview during the OnClick function. Comparison between these two tables revealed that the native function times appeared to remain the same and this shows that the amount of work being done within the Java code does not effect the performance of the native function call. This hypothesis was tested using a two-sample two-tailed T-test in Section 4.9.1 This is important knowledge when implementing a NewHope KEM within an application because no matter how labor intensive the application, it can be anticipated that key creation would not be affected. So long as the Java code does not cause the CPU usage to become too high such that the application starts to run slower.

Comparing the results from Table 8 to the baseline table, Table 6, it can be seen that function times for the baseline implementations faster. However, the CPU usages between the baseline and NewHope implementations are close in values. More specifically, NewHope512-CCA-KEM, NewHope512-CPA-KEM, and NewHope1024-

CPA-KEM are all less than 0.5% higher than the baseline and NewHope1024-CCA-KEM is less than 1% higher. These results coupled with the CPU trace of the C/C++ functions done by profiling the application indicates that the native functions do not have an impact on the average CPU usage. This was tested using a two-sample one-tailed t-test and can be seen in Section 4.9.2.

4.6 No Output OnClick Event Condition

This section provides the results from testing the NewHope implementations with a no output OnClick event. This testing procedure differed from the testing of NewHope implementations under verbose and succinct OnClick event conditions because this OnClick event was tested with the Google Pixel in airplane mode. All other aspects of the testing procedure was identical, including taking 50 samples. The samples that were collected were organized into different tables. The first set of tables shows the minimum, maximum, and average for the native scheme runtime, OnClick function runtime, and CPU usage. These tables are found in Appendix I; NewHope512-CCA is shown in Table 40, NewHope1024-CCA is shown in Table 41, NewHope512-CPA is shown in Table 42, and NewHope1024-CPA is shown in Table 43.

Table 9 shows the averages of native scheme runtime, OnClick function runtime, and CPU usage for all NewHope implementations.

In this table, it is shown that the average native scheme runtime is within at least 0.03 ms of the OnClick event runtime. This is due to the fact that the no output OnClick function did not set any text to the textview. Within the samples obtained there a maximum of one instance per NewHope implementation where the OnClick runtime was greater than the native scheme runtime.

Table 9: Comparison of Create Key average times of NewHope implementations with a no output OnClick function

Table Of All Average Results From Create Key Test Runs with No Output OnClick				
	512-CCA	**1024-CCA**	**512-CPA**	**1024-CPA**
Native Function Time(ms)	6.5146	12.2486	4.5224	7.7744
OnClick Function Time(ms)	6.5388	12.2486	4.5502	7.8398
Highest CPU Usage(%)	12.68	12.82	11.896	12.372

4.7 Comparing NewHope Implementations with Different OnClick Event Conditions

By using each sample set collect from each NewHope implementations with each OnClick event conditions and the baseline two figures displaying box plots from each was produced. These box plots include the average, dotted line within the boxs, and the standard deviation, the dotted line diamond centered around the average. Also, the box plots indicate which data points are considered to be outliers, solid points, and potential outliers, empty points.

Figure 7 includes the box plots that were produced utilizing the native scheme runtime of each NewHope implementations. Within this figure the box plots are grouped by NewHope implementation displaying the different OnClick event conditions. From this figure, a hypothesis was made, the average runtime of the native NewHope scheme is uneffected by the amount of Java code being executed in the mobile application. This is shown by the indicated averages for each NewHope implementation, and how they are approximately the same. In Section 4.9.1 provides the results from testing this hypothesis.

Figure 8 includes the box plots that were produced using the CPU usage samples obtained from each NewHope implementations with each OnClick event condition

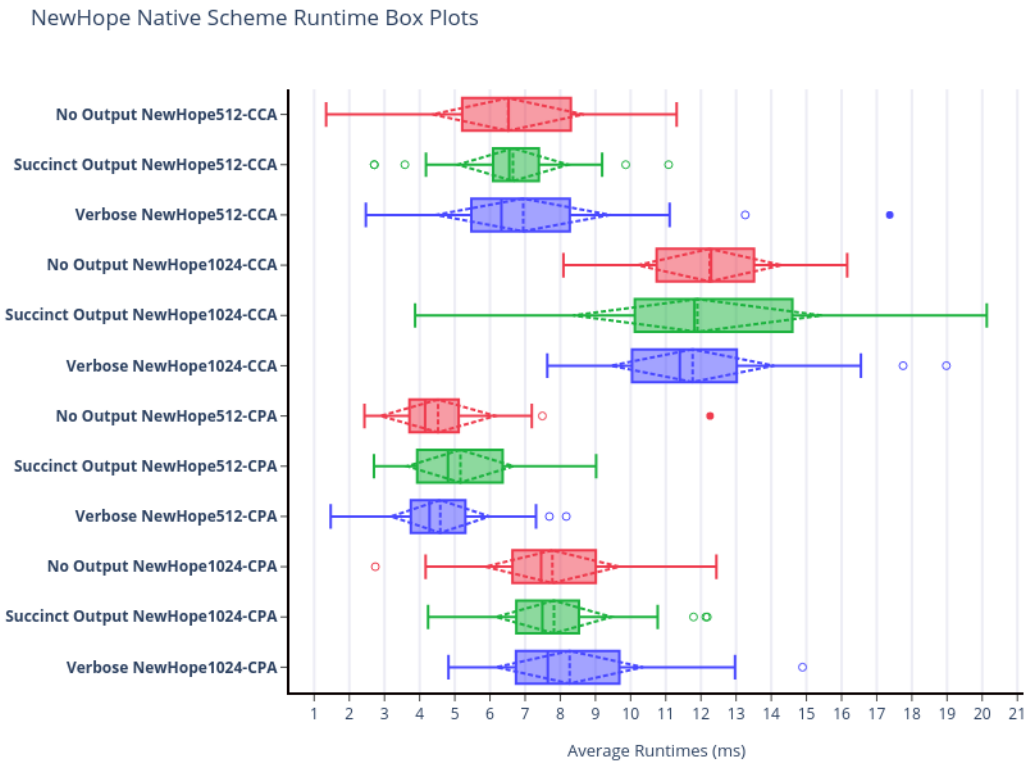

Figure 7: Box plots displaying native scheme runtimes for each NewHope implementations with different OnClick event conditions

and the baseline JNI application. Through analysis of this figure a few hypotheses can be drawn. The baseline and the NewHope implementations with succinct and no output OnClick event conditions are always lowerer than the verbose OnClick implementations. This indicated that the amount of Java code executed in the NewHope implementations effected the CPU usage of the application. The next hypothesis made was that the native NewHope schemes have no effect on the CPU usage of an application. This hypothesis was made by examining the CPU usage between the baseline and the NewHope implementations with succinct and no output OnClick event conditions. These hypotheses are addressed in Section 4.9.3 and Section 4.9.2.

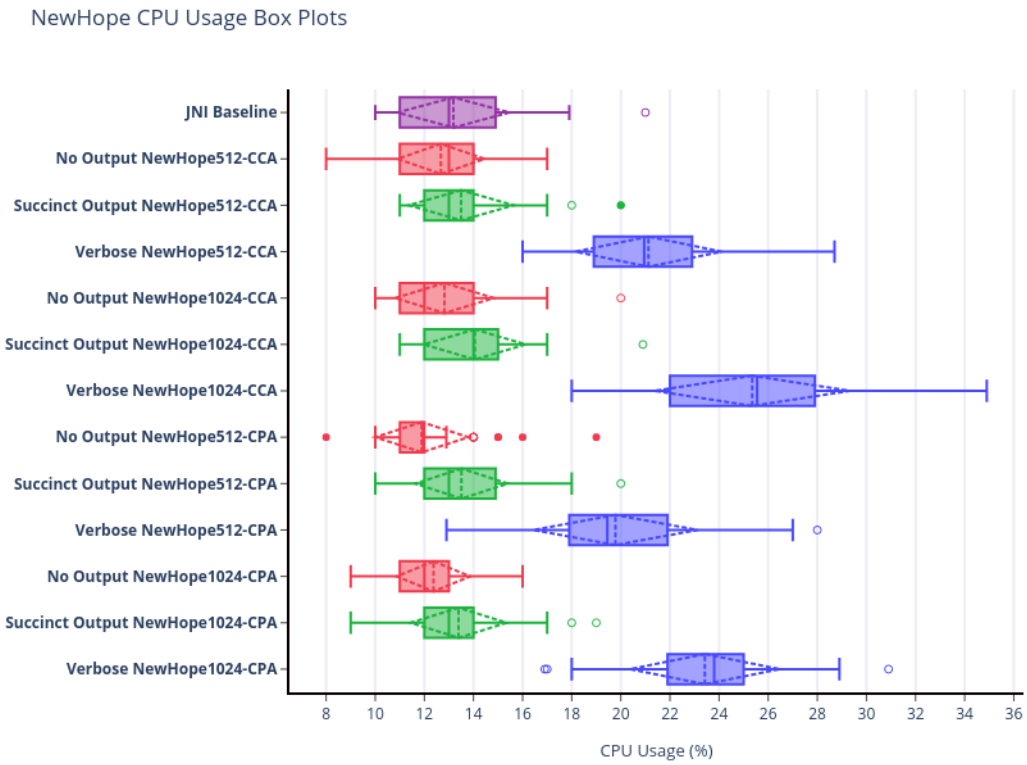

Figure 8: Box plots comparing CPU usage between each NewHope implementations with different OnClick event conditions and JNI baseline application

4.8 Performance Comparison of NewHope Configurations and Baseline Application

So far in this chapter each NewHope implementation had a sample of 50 `Create Key` functions evaluated and the results were show. This section is dedicated to analyzing the largest amount of resources consumed on application start up, as well as the range of resources used throughout the runtime of the application. The NewHope implementations with a succinct OnClick function were used for these comparisons. Table 10 displays the results as well as the baseline results for comparison.

Analyzing the resource usage upon application startup demonstrates that the NewHope implementations use less than 5% more CPU and uses the same amount of

Table 10: Comparison of performance results from NewHope implementations and modified Android Studio JNI code example

Performance Comparison of Succinct NewHope Implementations to Baseline					
Highest Performance Measurement On Application Start					
	512-CCA	**1024-CCA**	**512-CPA**	**1024-CPA**	**Baseline**
CPU Usage(%)	65.0	62.8	63.0	63.0	60.9
Memory Usage(MB)	176.0	170.6	157.8	166.4	159.8
Energy Usage	medium	medium	medium	medium	medium
Performance Range During Application Runtime					
	512-CCA	**1024-CCA**	**512-CPA**	**1024-CPA**	**Baseline**
CPU Usage(%)	11.0 - 20.0	11.0 - 20.9	10.0 - 20.0	9.0 - 18.0	10.0 - 21.0
Memory Usage(MB)	63.3 - 76.2	69.8 - 83.7	62.5 - 74.4	65.0 - 83.6	53.0 - 69.1
Energy Usage	light - none	light - none	light - none	light - none	light - none

energy. However, the memory that was used varies. The difference between the memory consumption on application startup between NewHope and the baseline are as follows: NewHope512-CPA-KEM consumed 2 Megabytes (MB) less, NewHope512-CCA-KEM consumed 16.2 MB more, NewHope1024-CPA-KEM consumed 6.6 MB more, and NewHope1024-CCA-KEM consumed 10.8 MB more than the baseline. These differences were expected due to the differences in the size of codebases and the size of the native library that had to be loaded within the application. Android Profiler describes energy usage as either low, medium, or high. On startup, all NewHope implementations use approximately the same amount of energy as the baseline.

Next, was the analysis of how the application used resources throughout runtime. The range of CPU usage between NewHope implementations and the baseline demonstrated that the same amount of CPU was being utilized throughout the application.

This exhibits that utilizing NewHope within an application does not alter the amount of CPU usage being used anymore than any other native functions would. The same can be said for energy usage. It is shown that the applications used a light amount of energy or no energy at all depending on the time between function samplings. However, the amount of memory being utilized by the application differed but not by an unpredictable amount. According to the memory trace, the difference in memory usage came from the amount of code that was place on the phone and the number of variables that had been declared. This showed that NewHope did not create an over-consumption of memory.

4.9 Statistical Analysis of NewHope Configurations and Baseline Application

Through the analysis of the data samples collect several hypotheses were made and this section is dedicated to testing them. The hypotheses made are:

1. The average runtime of native code was not effected by the amount of Java code being executed within the mobile application

2. The average CPU usage of NewHope implementations is effected by the amount of Java code being executed within the mobile application

3. The average CPU usage is not effected by implementing a NewHope KEM scheme

These hypotheses are tested using either a two-sample two-tailed T-test or a two-sample one-tailed T-test. The first hypothesis is tested by performing two-sample two-tailed T-tests using native NewHope scheme runtimes among all of the NewHope implementations with each OnClick variation. The following hypothesis is tested

by performing two-sample one-tailed T-tests using CPU usage against NewHope implementations with a verbose OnClick function to the other implementations with succinct and no output OnClick functions. The last hypothesis is tested using CPU usages and a combination of two-sample T-tests of all NewHope implementations with each OnClick variation to the JNI baseline.

In this section there a few variables that are used and need to be defined. Throughout this section H_0 and H_A refer to the null hypothesis and the alternative hypothesis, respectively. There are also average variables where M_b, M_v, M_s, and M_n refer to the averages for the baseline, and NewHope OnClick verbose, succinct, and no output variations, respectively.

4.9.1 Native Scheme Runtime T-Tests Between NewHope Configurations

Table 11 shows the p-values that resulted from running a two-sample two-tailed T-test on the runtimes of NewHope native schemes with a verbose and succinct OnClick functions. This T-test is designed such that the

$$H_0 : M_v = M_s$$

$$H_A : M_v \neq M_s$$

. This table shows that for a significance value, $\alpha = 0.05$ all of the average native scheme runtimes were shown to be statistically equal. Also, with a higher significance level all NewHope implementations, excluding NewHope512-CPA, were still demonstrated to be statistically equal.

The next test ran was a two-sample two-tailed T-test between the verbose and no output onclick functions for each NewHope implementation. The resulting p-

Table 11: Comparison of average runtime native NewHope KEM scheme of verbose OnClick event condition and succinct OnClick event condition

Two-Sample Two-Tailed T-test Results Between Verbose and Succinct Average NewHope Scheme Runtime	
NewHope Implementation	**p-values**
NewHope512-CCA-KEM	0.4957
NewHope1024-CCA-KEM	0.8152
NewHope512-CPA-KEM	0.0533
NewHope1024-CPA-KEM	0.2329

values are shown in Table 12. This T-test was completed with the following null and alternative hypotheses.

$$H_0 : M_v = M_n$$

$$H_A : M_v \neq M_n$$

The resulting p-values indicated that the average native scheme runtimes were statistically equal.

Table 12: Comparison of average runtime native NewHope KEM scheme of verbose OnClick event condition and no output OnClick event condition

Two-Sample Two-Tailed T-test Results Between Verbose and No Output Average NewHope Scheme Runtime	
NewHope Implementation	**p-values**
NewHope512-CCA-KEM	0.3560
NewHope1024-CCA-KEM	0.1002
NewHope512-CPA-KEM	0.8294
NewHope1024-CPA-KEM	0.2212

The last two-sample two-tailed T-test that was done using native scheme runtimes of succinct and no output OnClick NewHope implementations. This test was done with the following hypothesis:

$$H_0 : M_s = M_n$$

$$H_A : M_s \neq M_n$$

The resulting p-values are displayed in Table 13. The p-values indicate that for a significance value of $\alpha < .05$ all the NewHope implementations are shown to have statistically equal averages.

Table 13: Comparison of average runtime native NewHope KEM scheme of succinct OnClick event condition and no output OnClick event condition

Two-Sample Two-Tailed T-test Results Between Succinct and No Output Average NewHope Scheme Runtime	
NewHope Implementation	**p-values**
NewHope512-CCA-KEM	0.7034
NewHope1024-CCA-KEM	0.3136
NewHope512-CPA-KEM	0.0461
NewHope1024-CPA-KEM	0.9077

Through these T-tests it was proven that the average native scheme runtime is not effected by the amount of Java code being implemented in the mobile application. This was accomplished by statistically comparing the native runtimes with NewHope mobile implementations that contained varying amounts of Java code.

4.9.2 CPU Average T-Tests Between NewHope Configurations and Baseline Application

To compare the CPU averages between the verbose OnClick event condition and the baseline application a two-sample one-tailed t-test was selected with the following hypotheses:

$$H_0 : M_b = M_v$$

$$H_A : M_b < M_v$$

The results from the comparison between the verbose OnClick event condition and the baseline application are displayed in Table 14. Within this table, it can be seen that regardless of significance value selection H_0 will be rejected and H_A will be accepted; this implies that the average CPU usage of the baseline application was less than

the average CPU usage of the verbose OnClick event condition. This indicated that the NewHope verbose implementation consumes more CPU than the JNI baseline application. However, with this information it was not possible to draw a conclusion on if the native NewHope KEM scheme was impacting CPU usage.

Table 14: Comparison of average CPU usage of verbose OnClick event condition and baseline application

Two-Sample One-Tailed T-test Results Between Verbose and Baseline Average CPU Usages	
NewHope Implementation	**p-values**
NewHope512-CCA-KEM	<0.0001
NewHope1024-CCA-KEM	<0.0001
NewHope512-CPA-KEM	<0.0001
NewHope1024-CPA-KEM	<0.0001

The next T-test that was ran was cpu usage two-sample two-tailed T-test under the hypotheses

$$H_0 : M_s = M_b$$

$$H_A : M_s \neq M_b$$

The results from this test are displayed in Table 15. The resulted p-values indicate that the average CPU consumption between succinct NewHope implementations and baseline application are statistically equal given a significance level $\alpha = 0.01$. A higher significance level would indicate that all NewHope implementations excluding NewHope1024-CCA would still have statistically equal CPU usages. This gave confidence that the CPU usage of the verbose implementations were higher due to the amount of Java code and not caused by the NewHope scheme. The test results also indicated that the CPU consumption is equal to an application implementing a JNI and the NewHope native scheme is not causing any additional use of the CPU.

To gain more confidence that the NewHope native scheme does add any additional usage of the CPU than other application using a JNI a two-sample two-tailed T-test

Table 15: Comparison of average CPU usage of succinct OnClick event condition and baseline application

Two-Sample One-Tailed T-test Results Between Succinct and Baseline Average CPU Usages	
NewHope Implementation	**p-values**
NewHope512-CCA-KEM	0.4767
NewHope1024-CCA-KEM	0.0488
NewHope512-CPA-KEM	0.4328
NewHope1024-CPA-KEM	0.6191

was ran using the CPU usage of the baseline and no output NewHope implementations. This test was ran with the following null and alternative hypothese

$$H_0 : M_n = M_b$$

$$H_A : M_n \neq M_b$$

Table 16 provides the p-values that resulted from the T-test. The p-values that resulted indicated that for a significance level, $\alpha = 0.01$, three of the NewHope implementations were shown to have statistically equal average CPU usage. The two NewHope implementations with a p-value less than 0.1, NewHope512-CPA and NewHope1024-CPA, have average CPU usages that are less than the baseline average CPU usage.

Table 16: Comparison of average CPU usage of no output OnClick event condition and baseline application

Two-Sample One-Tailed T-test Results Between No Output and Baseline Average CPU Usages	
NewHope Implementation	**p-values**
NewHope512-CCA-KEM	0.2205
NewHope1024-CCA-KEM	0.3993
NewHope512-CPA-KEM	0.0028
NewHope1024-CPA-KEM	0.0379

These tests were used to show that the NewHope scheme does not cause any

additional consumption of the CPU resource than any other JNI application.

4.9.3 CPU Average T-Tests Between NewHope Configurations

A series of T-tests were performed to compare the CPU usages between each NewHope implementations with varying OnClick functions. These results were all accomplished using a two-tailed one-tailed T-test using the CPU usages.

The first test was done using the succinct and verbose NewHope implementations with the following hypotheses

$$H_0 : M_v = M_s$$

$$H_A : M_s < M_v$$

. The results are displayed in Table 17 and the p-values that are displayed indicated that the average CPU usage of the succinct NewHope implementations are less than the verbose implementations.

Table 17: Comparison of average CPU usage of verbose OnClick event condition and succinct OnClick event condition

Two-Sample One-Tailed T-test Results Between Verbose and Succinct Average CPU Usages	
NewHope Implementation	**p-values**
NewHope512-CCA-KEM	<0.0001
NewHope1024-CCA-KEM	<0.0001
NewHope512-CPA-KEM	<0.0001
NewHope1024-CPA-KEM	<0.0001

The next T-test was accomplished using the verbose and no output NewHope implementations with the hypotheses

$$H_0 : M_v = M_n$$

$$H_A : M_n < M_v$$

The resulting p-values are displayed in Table 18. The resulting p-values indicated that the average CPU usages of no output implementations are statistically proven to be less than the verbose implementations.

Table 18: Comparison of average CPU usage of verbose OnClick event condition and no output OnClick event condition

Two-Sample One-Tailed T-test Results Between Verbose and No Output Average CPU Usages	
NewHope Implementation	**p-values**
NewHope512-CCA-KEM	<0.0001
NewHope1024-CCA-KEM	<0.0001
NewHope512-CPA-KEM	<0.0001
NewHope1024-CPA-KEM	<0.0001

The final T-test that was ran was comparing the CPU usage of succinct and no output implementations. This test was done under the following hypotheses

$$H_0 : M_s = M_n$$

$$H_A : M_n < M_s$$

. The resulting p-values are displayed in Table 19. The p-values indicated that the average CPU usage of the no output implementations is statistically lower than the succinct implementations.

Table 19: Comparison of average CPU usage of succinct OnClick event condition and no output OnClick event condition

Two-Sample One-Tailed T-test Results Between Succinct and No Output Average CPU Usages	
NewHope Implementation	**p-values**
NewHope512-CCA-KEM	0.020985
NewHope1024-CCA-KEM	0.001973
NewHope512-CPA-KEM	0.000044
NewHope1024-CPA-KEM	0.002473

These T-tests helped solidify the conclusion that the native NewHope schemes did not have a negative effect on the CPU usage of the application. This indicated that when implementing NewHope on a mobile application there was no impact on CPU usage and did not display any CPU resource constraints. In fact, it was displayed that the amount of Java code implemented in the NewHope mobile implementation had more of an impact on the consumption of CPU resources than the native NewHope scheme.

4.10 Comparison to Previous Work

The work done by Malina et al. [73] yielded several results that was used for comparison to the results that were obtained throughout this research effort. However, it is important to preface this section with the differences between their NewHope implementation and the succinct NewHope implementations built for this effort. Malina et at. [73] implemented NewHope-Usenix on an Android mobile device with a 32-bit processor. However, this implementation was done using NewHope which is based on NewHope-Simple. New-Hope-Simple is a variant of NewHope-Usenix, but instead of using a reconciliation-based approach of the ring learning with errors (RLWE) scheme it utilizes an encryption-based approach. Another difference is that the Android device selected for this effort had a 64-bit processor. Malina et al. [73] utilized a Java codebase referenced in [10]. Since its publication in 2018, access to the codebase has either been removed or relocated. This differs from the implementations achieved throughout this paper being that this implementation was done using a C/JNI codebase. Another significant difference is that the NewHope implementation by Malina et al. [73] claims to have a 206 bit-level security. However, in this effort the NewHope512 and NewHope1024 implementations are claimed to achieve a 201 bit-level security and a 233 bit-level security, respectively. With all the differences

aside, the results by Malina et al. [73] were still used for comparison; Due to the lack of available research on PQC KEMs being implemented on mobile phones. Ideally, a more similar implementation would be used for comparison if available.

The first results that were examined were the total number of bytes that needed to be sent in order achieve key establishment. Malina et al. [73] provided a comparison between all of the PQC KEMs that they implemented, it displays the number of bytes that need be transmitted to achieve key creation. Using Malina et al. [73] data a comparison between their implementation and this research efforts NewHope implementations is provided in Figure 9.

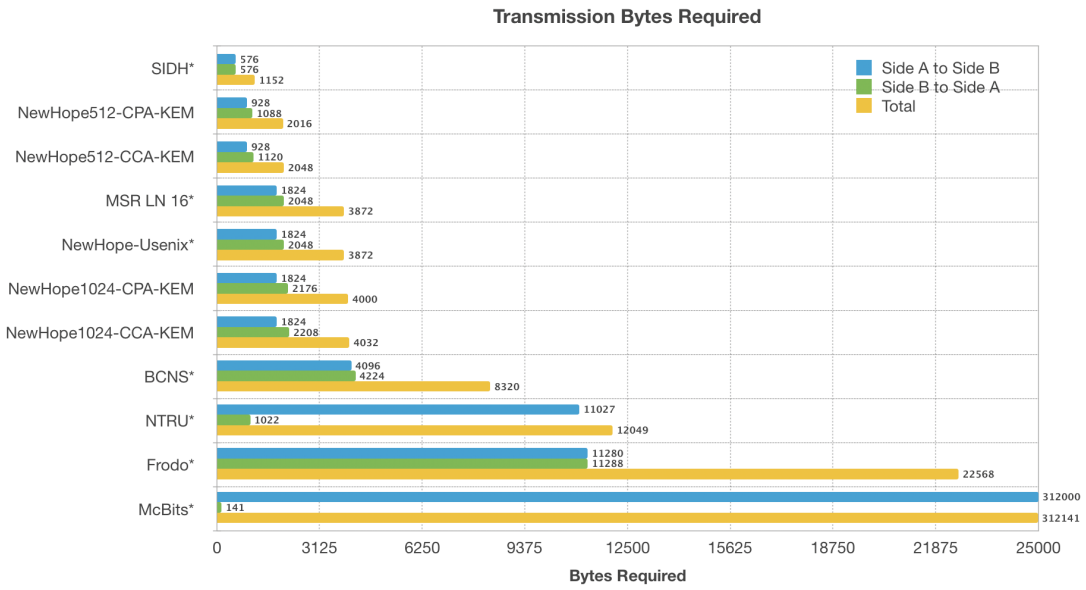

Figure 9: Comparison of the total number of bytes sent to achieve key establishment between NewHope mobile implementation to the work by Malina et al. [73](*)

It can be seen through examination of Figure 9 that the NewHope-Usenix implemented by Malina et al. [73] achieves a claimed 206 bit-level security by transmitting a total of 3872 bytes and it was one of the lowest number in comparison to their other PQC KEM's. The only KEM that had a lower total number of bytes was SIDH with a total of 1152 bytes and a 128-bit level security. This mobile implemen-

tation achieved a 101-bit level security through 2016 or 2048 bytes, and a 233-bit level security through 4000 or 4032 bytes. The total transmission byte requirement differences between the NewHope-Usenix implementation and this research efforts NewHope implementations was either 128 bytes or 160 bytes depending on bit-level security. Malina et al. [73] implementation achieved a lower bit-level security, which makes it hard to compare. However, within the NewHope documentation [8] they addressed the difference in bytes transmitted between NewHope-Usenix and NewHope. Within this documentation they state that since the bandwidth requirement difference is small and complexity to implement a reconciliation-based scheme is high the use of a less complex encryption-based scheme would be the ideal selection [8]. It is also important to notice that NewHope1024 achieved a claimed higher bit-level security compared to all the PQC KEM's referenced in Figure 9, and does so with less bandwidth than all but three (including NewHope-Usenix).

Malina et al. [73] also provide data on the runtime of the two PQC KEM schemes that were implemented on a mobile device. Using that data and the data on the runtime of the NewHope schemes seen in this research effort Figure 10. In this figure, it is shown that the MSR LN16 scheme, with a presumed post-quantum 128-bit level security, completed in a total of 1.72 ms, which is significantly faster than that of their NewHope-Usenix implementation. Also, comparing these results to the succinct NewHope implementations it was shown that NewHope completes its KEM scheme at least 90 ms faster than NewHope-Usenix. While achieving a higher claimed bit-level security, these NewHope implementations also complete their scheme within less than 11 ms of MSR LN16. The differences between the scheme completion time of MSR LN16 and NewHope are as follows: NewHope512-CCA-KEM is slower by 4.932 ms, NewHope1024-CCA-KEM is slower by 3.438 ms, NewHope512-CPA-KEM is slower by 10.18 ms, and NewHope1024-CPA-KEM is slower by 6.096 ms.

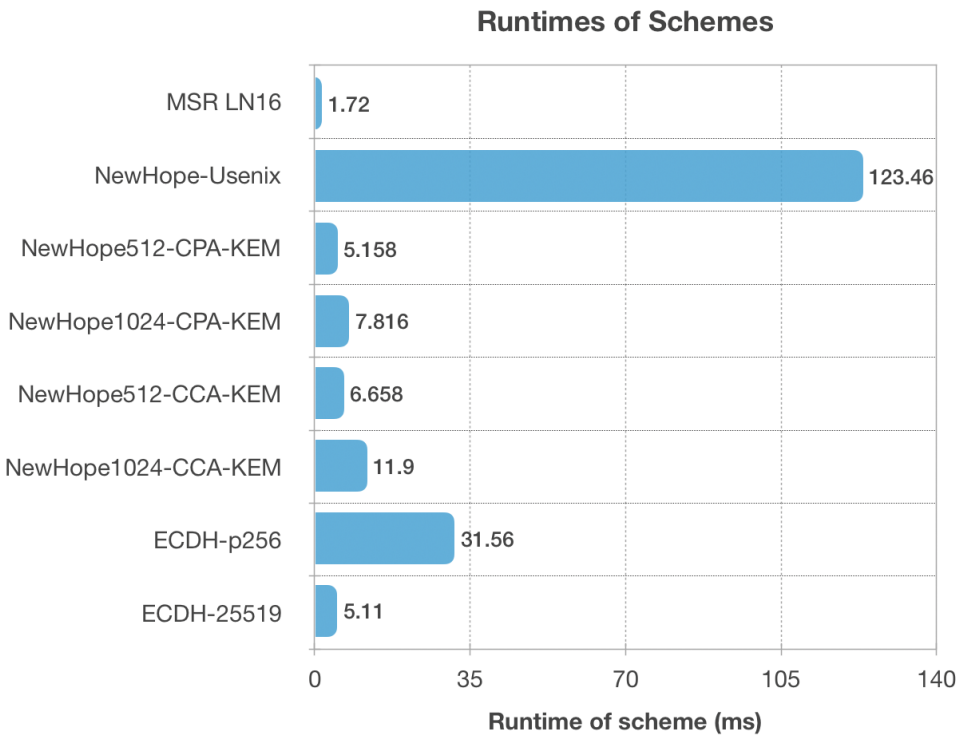

Figure 10: Comparison of the average runtime of schemes between NewHope mobile implementation to the work by Malina et al. [73](*)

Finally, Malina et al. [73] provide a comparison between NewHope-Usenix and two Elliptic-Curve Diffie-Hellman (ECDH) key exchange schemes. Figure 10 shows that the runtime of NewHope-Usenix was slower than both ECDH key exchanges. NewHope schemes complete their runtime faster than NewHope-Usenix and ECDH-p256. Also, NewHope512-CPA-KEM on average completed 0.048 ms slower then ECDH-25519. This demonstrates that NewHope is comparable to classic key establishment schemes such as ECDH schemes.

Ultimately, NewHope appeared to outperform NewHope-Usenix when implemented on a mobile device. However, it is important to consider that when implemented on an Android device with a 64-bit processor, significant improvements could be achieved. Also, NewHope could have performed better due to the chosen language. If NewHope-Usenix was implemented using a C/JNI codebase that could also improve its results.

Throughout this section it was demonstrated that NewHope has a competitive bandwidth compared to other PQC KEM's, has a superior or comparable runtime to other PQC KEM's and classic key exchange schemes, and has a variety of different implementations that when chosen properly could better fit implementation needs. The conclusions that have been achieved in this section are similar to the conclusions that Malina et al. [73] came to about NewHope-Usenix.

4.11 Chapter Summary

Section 4.2, the results from all of the correctness tests were given. All four NewHope KEM mobile implementations were verified to be implemented correctly.

Section 4.3 provided the test results from the JNI baseline. These results were utilized to determine that implementation of the NewHope PQC KEM scheme does not cause an application to consume more CPU than a basic JNI application.

Section 4.4, Section 4.5, and Section 4.6 provided the results for verbose, succinct, and no output NewHope implementations respectively. These results were displayed and compared in Section 4.7. These results were important for comparing the succinct and no output NewHope implementations to the verbose NewHope implementations to determine if the amount of Java code effected the amount of CPU that was consumed by the application. Additionally, the runtime results were utilized in determining that the scheme runtime remained consistent despite the amount of Java code being implemented.

In Section 4.8 all NewHope KEM mobile implementations with succinct OnClick functions were compared to the baseline results based on the amount of resources used on application startup and throughout application runtime. Ultimately, the conclusion is that the usage of central processing unit (CPU) and energy of NewHope KEMs are approximately equal to the JNI baseline. The only resource usage that was

not equal was memory usage but, as described in this section, this is to be expected based on the amount of code and the amount of computations that occur.

The statistical analysis done on the results was provided in Section 4.9. In this section, hypotheses were created by comparing the NewHope implementations. This section addressed those hypotheses through a series of T-tests. Ultimately, it was concluded that the native scheme runtime was not effect by the amount of Java code that was implemented in the mobile application and the amount of Java code caused an increase in CPU usage. The final conclusion was that the CPU usage of succinct and no output NewHope implementations had average CPU usages that were statistically equal to the JNI baseline; this implies that implementing the NewHope PQC KEM consumes the same amount of CPU as an application implementing a JNI.

Lastly, in Section 4.10 each NewHope KEM implementations are compared to the work done by Malina et al. [73]. Through these comparisons it is concluded that NewHope KEMs are comparable to other PQC KEMs and to current key exchange cryptosystems. Additionally, the NewHope PQC KEM performed better than Malina et al. [73] NewHope-Usenix mobile implementation. However, there are some daunting differences between implementations that make these comparisons carry less weight.

V. Conclusions

5.1 Chapter Overview

This chapter is dedicated to presenting the conclusions obtained from this research effort. Section 5.2 outlines the main conclusions achieved. Section 5.3 presents the future work that could stem from this research effort. Lastly, Section 5.4 outlines the contributions this research has for the current research field.

5.2 Conclusions

Throughout this effort all the NewHope PQC KEMs were implemented on a mobile phone with three different OnClick conditions. NewHope was selected from the group of Round 2 National Institute of Standards and Technology (NIST) PQC Standardization Project due to its small key sizes and simple implementation. It was also selected because it gains its hardness from the RLWE problem which is a cryptographic primitive that gained concrete hardness in its underlying problem.

The NewHope implementations were achieved through the utilization of a native C library and a JNI. The NewHope implementations performed as anticipated, due to the NewHope-CCA KEM's adding extra security, against the known CCA targeting key reuse, it was anticipated that the NewHope-CCA implementations would utilize more resources and take longer to complete. So, when it comes to comparing which NewHope implementation is better for mobile implementation, it is important to evaluate the target devices requirements and the constraints that are present. With a target device of a google pixel, NewHope KEMs perform well and only need to be questioned on which NewHope implementation best fits the cryptographic purposes needed.

Through statistical analysis of the results obtained from each NewHope instanti-

ation conclusions about average CPU usage and scheme runtimes were made. It was proven that the average native scheme runtime was not effected by the amount of Java code being implemented within the application. The average scheme runtimes were tested amongst all NewHope implemntation through the use of multiple two-sample two-tailed T-test to demonstrate that all had equal average scheme runtimes. This implies that the average scheme runtimes are anticipated to run consistently at the same rate. This could be effected dependant on device constraints, but a device similar to a Google Pixel could anticipate scheme runtimes similiar to this research effort. Additionally, the native NewHope KEM scheme did not cause any additional CPU consumption than implementing a JNI. This was done by comparing CPU usages from all NewHope implementations to the JNI baseline through two-sample T-tests. The results from the T-tests demonstrated that average CPU usage of NewHope implementations with graphics usage similiar to the baseline, had average CPU usages that were equal to those of the baseline. Finally, these T-tests also demonstrated that implementation of more Java code did lead to an increase in average CPU usage.

The NewHope Implementations completed quickly and were comparable to the runtimes of the schemes presented in the work by Malina et al. [73]. However, the NewHope implementations runtimes did not compare to the baseline but that was expected based on the amount computations that have to be completed. When evaluating the CPU usage and energy usage of the NewHope KEMs, they were comparable the tested baseline both on startup and during application runtime.

Overall, mobile implementations of PQC KEMs are feasible to implement and perform well through the use of a native C library and a JNI. The NewHope implementations did not overly consume any of the devices resources and acted in a predicted manner. These implementations have exhibited that PQC KEM are feasible and competitive when implemented on devices. When compared to the work

done by Malina et al. [73] it was demonstrated that NewHope implementations have the ability to be competitive not only against other PQC schemes but it is also competitive with current cryptography.

5.3 Future Work

Due to the scope of the project and continued research available in the field, the following is a list of future work:

- Provide a more in-depth analysis of the C/C++ function traces to analyze the amount of scheme runtime spent on each individual algorithm
- Utilizing the produced C/JNI NewHope codebase, network the NewHope mobile implementation to further test the performance of the scheme
- Implement other Round 2 PQC KEM submissions on a mobile device to analyze and fully assess the performance of the second round PQC submissions
- Implement NewHope with a Java implementation to test the best method of implementation for PQC algorithms
- Implement and test NewHope KEMs on other platforms to further assess the flexibility of the NewHope PQC KEM

5.4 Contributions

This research effort contributed meaningful analysis on the feasibility and implementation expectations that can be expected when implementing a PQC KEM on a mobile device. Also, contributions were made on the analysis of the PQC KEMs currently in Round 2 of NIST's Standardization Project and the limitations or security risks they might present when implemented on a mobile phone. This research effort also contributed a detailed look into how PQC KEMs can be implemented in an Android application through a native C library using a JNI. This effort also produced important data benchmarks that can be used to compare other PQC KEM algorithms to.

Appendix A. MainActivity.java

```
package com.example.newhope512cca;

import androidx.appcompat.app.AppCompatActivity;

import android.os.Bundle;
import android.text.method.ScrollingMovementMethod;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.TextView;
import java.util.Arrays;

public class MainActivity extends AppCompatActivity {
    //Declaring the KAT values into the arrays

    //Hard coded KAT Array values goes here
    //However they are too large to be formatted correctly

    //Loading the 'native-lib' library on application startup.
    static {
        System.loadLibrary("native-lib");
    }
    //Setting current KEM values from C code
    private int count;
    private String seed = "";
    private String ct = "";
    private String ss = "";
    private String pk = "";
    private String sk = "";

    //Creating new arrays to store the 100 values in to test correctness
    private String[] receivedSeedValues = new String[100];
    private String[] receivedPkValues = new String[100];
    private String[] receivedSkValues = new String[100];
    private String[] receivedCtValues = new String[100];
    private String[] receivedSsValues = new String[100];

    //Declaring output string to avoid the use of passing hardcoded
    string to TextView
    private String outputString;
    //Defining onCreate function
    @Override
```

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);



    //Initializing the app buttons
    Button createKey = findViewById(R.id.createKey);
    Button clearKeys = findViewById(R.id.clearKeys);
    final Button correctnessTest = findViewById(R.id.correctnessTest);

    //Initializing the app TextView
    final TextView textView = findViewById(R.id.keyText);
    //Adding Scrolling capabilities to TextView
    textView.setMovementMethod(new ScrollingMovementMethod());


    OnClickListener setKeyText = new OnClickListener() {
        @Override
        public void onClick(View v) {
            RequestString();

            outputString = "key creation complete\n SS :" + ss;
            textView.setText(outputString);
        }
    };
    createKey.setOnClickListener(setKeyText);

    OnClickListener clearKeyText = new OnClickListener() {
        @Override
        public void onClick(View v) {

            textView.setText("");
        }
    };
    clearKeys.setOnClickListener(clearKeyText);

    OnClickListener correctnessTestText = new OnClickListener() {
        @Override
        public void onClick(View v) {

            correctnessTest(receivedSeedValues, receivedPkValues,
            receivedSkValues, receivedCtValues, receivedSsValues);
```

```
Boolean seedValuesCorrect = Arrays.equals(seedValues,
receivedSeedValues);
Boolean pkValuesCorrect = Arrays.equals(pkValues,
receivedPkValues);
Boolean skValuesCorrect = Arrays.equals(skValues,
receivedSkValues);
Boolean ctValuesCorrect = Arrays.equals(ctValues,
receivedCtValues);
Boolean ssValuesCorrect = Arrays.equals(ssValues,
receivedSsValues);
if(seedValuesCorrect && pkValuesCorrect &&
skValuesCorrect && ctValuesCorrect &&
        ssValuesCorrect)
{
    outputString = "\nCorrectness Check Passed\n";
    textView.append(outputString);
}
else
{
    outputString = "Correctness check failed\n";
    textView.append(outputString);
    if(!seedValuesCorrect){
        textView.append("Differences in seeds at: \n");
        findArrayDifferences(textView, seedValues,
        receivedSeedValues);
    }
    else
    {
        textView.append("Seed values are correct!\n");
    }
    if(!pkValuesCorrect){
        textView.append("Differences in pk's at: \n");
        findArrayDifferences(textView, pkValues,
        receivedPkValues);
    }
    if(!skValuesCorrect){
        textView.append("Differences in sk's at: \n");
        findArrayDifferences(textView, skValues,
        receivedSkValues);
    }
    if(!ctValuesCorrect){
        textView.append("Differences in ct's at: \n");
        findArrayDifferences(textView, ctValues,
```

```
                    receivedCtValues);
                }
                if(!ssValuesCorrect){
                    textView.append("Differences in ss's at: \n");
                    findArrayDifferences(textView, ssValues,
                    receivedSsValues);
                }
            }
        }
    };
    correctnessTest.setOnClickListener(correctnessTestText);

}
public void findArrayDifferences(TextView view, String[] array1,
String[] array2)
{
    if(array1.length == 0 || array2.length == 0)
    {
        view.append("Empty array --> no values received \n");
    }
    else if(array1.length == array2.length)
    {
        for(int i = 0; i < array1.length; i++)
        {
            if(array1[i] != array2[i])
            {
                view.append("count: " + i + "\n");
            }
        }
    }
    else
    {
        view.append("Arrays not same length\n");
    }

}

/**
 * A native method that is implemented by the 'native-lib' native
 library,
 * which is packaged with this application.
 */
public native void RequestString();
```

```
    public native void correctnessTest(String[] receivedSeedValues,
    String[] receivedPkValues, String[] receivedSkValues, String[]
    receivedCtValues, String[] receivedSsValues);
}
```

Appendix B. MainActivityDriver.c

This appendix contains the `MainActivityDriver.c` from the NewHope512-CCA-KEM mobile implementation. All other NewHope mobile implementations contain this same source file but with a different kTag variable.

```
//
// Created by Jessica Switzler on 2020-01-14.
//
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include "rng.h"
#include "api.h"
#include <jni.h>
#include <android/log.h>
#include <time.h>


// Defining the log statements to use when running/testing code
static const char* kTAG = "NewHope512CCA";
#define LOGI(...) \
  ((void)__android_log_print(ANDROID_LOG_INFO, kTAG, __VA_ARGS__))
#define LOGW(...) \
  ((void)__android_log_print(ANDROID_LOG_WARN, kTAG, __VA_ARGS__))
#define LOGE(...) \
  ((void)__android_log_print(ANDROID_LOG_ERROR, kTAG, __VA_ARGS__))

// Declaring functions that are defined below their use
void setString(JNIEnv *env, jobject this, const char *fieldName,
               const char *outputString);
void string2hexString(unsigned char* input, unsigned char* output,
                      unsigned long long len);
int getDigitValue(char digit);
int getPairValue(char digit1, char digit2);
void hexStrToStr(unsigned char *dst, const char *src, int size);

// Java function used to create a singular key
JNIEXPORT void JNICALL
Java_com_example_newhope512cca_MainActivity_RequestString(JNIEnv
        *env, jobject this)
{
    unsigned char       seed[48];
    unsigned char       entropy_input[48];
    unsigned char       ct[CRYPTO_CIPHERTEXTBYTES],
                        ss[CRYPTO_BYTES],
                        ss1 [CRYPTO_BYTES];
    unsigned char       pk[CRYPTO_PUBLICKEYBYTES],
                        sk[CRYPTO_SECRETKEYBYTES];
    int                 ret_val;
```

```
//Creating temporary values to be utilized in copying values to
    Java
unsigned char seedCopy[48];
unsigned char ssCopy[CRYPTO_BYTES];
unsigned char ctCopy[CRYPTO_CIPHERTEXTBYTES];
unsigned char pkCopy[CRYPTO_PUBLICKEYBYTES];
unsigned char skCopy[CRYPTO_SECRETKEYBYTES];

//Used to create a random value to feed in for the randombytes
    so the seed doesn't always come
//out the same
srand(time(0));

//Initialize the entropy input for seed creation
for (int i=0; i<48; i++)
{
    entropy_input[i] = rand()%256;
}

//Initialize the buffer that will be used to get the seed
randombytes_init(entropy_input, NULL, 256);

//Get 48 bytes off of the randombytes buffer to be used as a
    seed
randombytes(seed, 48);

//Set the Java seed value to be the same as the C seed value so
    it may be used in java application
string2hexString(seed, seedCopy, 48);
setString(env, this, "seed", (const char*)seedCopy);

//Intialize the randombytes buffer using the seed created for
    key creation
//Note if the seed is the same the key will be the same, seed
    needs to be randomized for security
randombytes_init(seed, NULL, 256);

// Generate the public/private keypair
if ( (ret_val = crypto_kem_keypair(pk, sk)) != 0) {
    LOGE("crypto_kem_keypair returned <%d>\n", ret_val);
    LOGE("KAT_CRYPTO_FAILURE");
}

//Set the java PK and SK values so it may be utilized in
    application
string2hexString(pk, pkCopy, CRYPTO_PUBLICKEYBYTES);
setString(env, this, "pk", (const char*)pkCopy);
string2hexString(sk, skCopy, CRYPTO_SECRETKEYBYTES);
setString(env, this, "sk", (const char*)skCopy);

if ( (ret_val = crypto_kem_enc(ct, ss, pk)) != 0) {
    LOGE("crypto_kem_enc returned <%d>\n", ret_val);
```

```
        LOGE("KAT_CRYPTO_FAILURE");
    }

    if ( (ret_val = crypto_kem_dec(ss1, ct, sk)) != 0) {
        LOGE("crypto_kem_dec returned <%d>\n", ret_val);
        LOGE("KAT_CRYPTO_FAILURE");
    }

    if ( memcmp(ss, ss1, CRYPTO_BYTES) ) {
        LOGE("crypto_kem_dec returned bad 'ss' value\n");
        LOGE("KAT_CRYPTO_FAILURE");
    }

    LOGI("KAT_SUCCESS");

    //Set the java CT and SS value so it may be utilized in
       application
    string2hexString(ct, ctCopy , CRYPTO_CIPHERTEXTBYTES);
    setString(env, this, "ct", (const char*)ctCopy);
    string2hexString(ss, ssCopy , CRYPTO_BYTES);
    setString(env, this, "ss", (const char*)ssCopy);
}

// Java function used to run the correctness test to verify the
   correctness
    of implementation
JNIEXPORT void JNICALL
   Java_com_example_newhope512cca_MainActivity_correctnessTest(
        JNIEnv *env,jobject  this,
        jobjectArray receivedSeedValues,
        jobjectArray receivedPkValues,
        jobjectArray receivedSkValues,
        jobjectArray receivedCtValues,
        jobjectArray receivedSsValues)
{
    unsigned char       seed[48];
    unsigned char       entropy_input[48];
    unsigned char       ct[CRYPTO_CIPHERTEXTBYTES],
                        ss[CRYPTO_BYTES],
                        ss1[CRYPTO_BYTES];
    unsigned char       pk[CRYPTO_PUBLICKEYBYTES],
                        sk[CRYPTO_SECRETKEYBYTES];
    int                 ret_val;

    //Temporary values utilized to create java arrays to check
       application for correctness
    unsigned char tempSeed[48];
    unsigned char tempSk[CRYPTO_SECRETKEYBYTES];
    unsigned char tempPk[CRYPTO_PUBLICKEYBYTES];
    unsigned char tempCt[CRYPTO_CIPHERTEXTBYTES];
    unsigned char tempSs[CRYPTO_BYTES];
```

```
//Comments for similiar lines of code will not be repeated, look
    at function above prior to
//looking at this function to fully understand the code


for (int i=0; i<48; i++)
    entropy_input[i] = i;

//Obtaining arraylength to ensure loop iterates through the
   number of KAT values
jsize arrayLen = (*env)->GetArrayLength(env, receivedSeedValues)
   ;

randombytes_init(entropy_input, NULL, 256);
for (int i=0; i<arrayLen; i++)
{
    randombytes(seed, 48);

    //Setting the Java receivedSeedValues array for comparison
       to the hardcoded KAT values
    string2hexString(seed, tempSeed, 48);
    jstring seedArrayValues = (*env)->NewStringUTF(env,
                                (const char*)tempSeed);
    (*env)->SetObjectArrayElement(env, receivedSeedValues, i,
       seedArrayValues);
}


for(int i=0; i<arrayLen; i++)
{
    //Initialize a jsize value so that it may be used in the
       GetObjectArrayElement function
    jsize javaLoop = i;
    //Get the seed value and convert it to back to its original
       string
    jstring javaArrayElement = (*env)->GetObjectArrayElement(
                            env,receivedSeedValues, javaLoop);
    const char *seedFromArray = (*env)->GetStringUTFChars(
                                env, javaArrayElement, NULL);
    hexStrToStr(tempSeed, seedFromArray, strlen(seedFromArray));

    randombytes_init(tempSeed, NULL, 256);

    // Generate the public/private keypair
    if ( (ret_val = crypto_kem_keypair(pk, sk)) != 0) {
        LOGE("crypto_kem_keypair returned <%d>\n", ret_val);
        LOGE("KAT_CRYPTO_FAILURE");
    }


    if ( (ret_val = crypto_kem_enc(ct, ss, pk)) != 0) {
```

```
            LOGE("crypto_kem_enc returned <%d>\n", ret_val);
            LOGE("KAT_CRYPTO_FAILURE");
        }



        if ( (ret_val = crypto_kem_dec(ss1, ct, sk)) != 0) {
            LOGE("crypto_kem_dec returned <%d>\n", ret_val);
            LOGE("KAT_CRYPTO_FAILURE");
        }

        if ( memcmp(ss, ss1, CRYPTO_BYTES) ) {
            LOGE("crypto_kem_dec returned bad 'ss' value\n");
            LOGE("KAT_CRYPTO_FAILURE");
        }

        LOGI("KAT_SUCCESS");
        //Add PK and SK values to their respective arrays for
            comparison
        string2hexString(pk, tempPk, CRYPTO_PUBLICKEYBYTES);
        jstring str = (*env)->NewStringUTF(env,(const char*)tempPk);
        (*env)->SetObjectArrayElement(env, receivedPkValues, i, str)
            ;

        string2hexString(sk, tempSk, CRYPTO_SECRETKEYBYTES);
        str = (*env)->NewStringUTF(env,(const char*)tempSk);
        (*env)->SetObjectArrayElement(env, receivedSkValues, i,
                                                        str);

        //Add CT and SS values to their respective arrays for
            comparison
        string2hexString(ct, tempCt, CRYPTO_CIPHERTEXTBYTES);
        str = (*env)->NewStringUTF(env,(const char*)tempCt);
        (*env)->SetObjectArrayElement(env, receivedCtValues, i,
                                                          str);
        string2hexString(ss, tempSs, CRYPTO_BYTES);
        str = (*env)->NewStringUTF(env,(const char*)tempSs);
        (*env)->SetObjectArrayElement(env, receivedSsValues, i,
                                                          str);
    }

}

// JNI helper function used to set java values using c value
void setString(JNIEnv *env, jobject this, const char *fieldName,
                const char *outputString)
{
    jclass thisClass = (*env)->GetObjectClass(env, this);
    jfieldID stringField = (*env)->GetFieldID(env, thisClass,
        fieldName, "Ljava/lang/String;");
    jstring javaString = (*env)->GetObjectField(env, this,
        stringField);
```

```
    const char *newString = (*env)->GetStringUTFChars(env,
       javaString, NULL);
    (*env)->ReleaseStringUTFChars(env, javaString, newString);
    javaString = (*env)->NewStringUTF(env, outputString);
    (*env)->SetObjectField(env, this, stringField, javaString);
}

// Helper function used to create a hex string from a string
void string2hexString(unsigned char* input, unsigned char* output,
                      unsigned long long len)
{
    int loop = 0;
    int i = 0;
    while(loop < len)
    {
        sprintf((char*)(output+i),"%02X", input[loop]);
        loop+=1;
        i+=2;
    }
}

// Helper function used to convert a hex string to string
int getDigitValue(char digit)
{
    int n = digit-'0';
    if (n>=0 && n<10) { return n; }
    if (digit=='A') { return 10; }
    if (digit=='B') { return 11; }
    if (digit=='C') { return 12; }
    if (digit=='D') { return 13; }
    if (digit=='E') { return 14; }
    if (digit=='F') { return 15; }
    return -1;
}

// Helper function used to convert a hex string to string
int getPairValue(char digit1, char digit2)
{
    int v1 = getDigitValue(digit1);
    int v2 = getDigitValue(digit2);
    return (v1==-1 || v2==-1) ? -1 : v1 * 16 + v2;
}

// Function to convert a hex string to string, used for seed so only
    an int is needed for size
void hexStrToStr(unsigned char *dst, const char *src, int size)
{
    int isGood = 0;
    if (size % 2 == 0) {
        isGood = 1;
        for (int i=0; i<size; i+=2) {
```

```
            int pv = getPairValue(src[i], src[i+1]);
            if (pv==-1) { isGood = 0; break; }
            *dst = pv;
            dst++;
        }
    }
    *dst = 0;
}
```

Appendix C. Original api.h

This appendix contains the `api.h` file from the original NewHope codebase and was used to find the key and ciphertext sizes.

```
#ifndef API_H
#define API_H

#include "params.h"

#define CRYPTO_SECRETKEYBYTES  NEWHOPE_CCAKEM_SECRETKEYBYTES
#define CRYPTO_PUBLICKEYBYTES  NEWHOPE_CCAKEM_PUBLICKEYBYTES
#define CRYPTO_CIPHERTEXTBYTES NEWHOPE_CCAKEM_CIPHERTEXTBYTES
#define CRYPTO_BYTES           NEWHOPE_SYMBYTES

#if   (NEWHOPE_N == 512)
#define CRYPTO_ALGNAME "NewHope512-CCAKEM"
#elif (NEWHOPE_N == 1024)
#define CRYPTO_ALGNAME "NewHope1024-CCAKEM"
#else
#error "NEWHOPE_N must be either 512 or 1024"
#endif

int crypto_kem_keypair(unsigned char *pk, unsigned char *sk);

int crypto_kem_enc(unsigned char *ct, unsigned char *ss, const
   unsigned char *pk);

int crypto_kem_dec(unsigned char *ss, const unsigned char *ct, const
    unsigned char *sk);

#endif
```

Appendix D. Original params.h

This appendix contains the `params.h` file from the original NewHope codebase and was used to find the key and ciphertext sizes.

```
#ifndef PARAMS_H
#define PARAMS_H

#ifndef NEWHOPE_N
#define NEWHOPE_N 512
#endif

#define NEWHOPE_Q 12289
#define NEWHOPE_K 8           /* used in noise sampling */

#define NEWHOPE_SYMBYTES 32   /* size of shared key, seeds/coins,
    and hashes */

#define NEWHOPE_POLYBYTES             ((14*NEWHOPE_N)/8)
#define NEWHOPE_POLYCOMPRESSEDBYTES   (( 3*NEWHOPE_N)/8)

#define NEWHOPE_CPAPKE_PUBLICKEYBYTES  (NEWHOPE_POLYBYTES +
    NEWHOPE_SYMBYTES)
#define NEWHOPE_CPAPKE_SECRETKEYBYTES  (NEWHOPE_POLYBYTES)
#define NEWHOPE_CPAPKE_CIPHERTEXTBYTES (NEWHOPE_POLYBYTES +
    NEWHOPE_POLYCOMPRESSEDBYTES)

#define NEWHOPE_CPAKEM_PUBLICKEYBYTES NEWHOPE_CPAPKE_PUBLICKEYBYTES
#define NEWHOPE_CPAKEM_SECRETKEYBYTES NEWHOPE_CPAPKE_SECRETKEYBYTES
#define NEWHOPE_CPAKEM_CIPHERTEXTBYTES
    NEWHOPE_CPAPKE_CIPHERTEXTBYTES

#define NEWHOPE_CCAKEM_PUBLICKEYBYTES NEWHOPE_CPAPKE_PUBLICKEYBYTES
#define NEWHOPE_CCAKEM_SECRETKEYBYTES (NEWHOPE_CPAPKE_SECRETKEYBYTES
     + NEWHOPE_CPAPKE_PUBLICKEYBYTES + 2*NEWHOPE_SYMBYTES)
#define NEWHOPE_CCAKEM_CIPHERTEXTBYTES (
    NEWHOPE_CPAPKE_CIPHERTEXTBYTES + NEWHOPE_SYMBYTES)  /* Second
    part is for Targhi-Unruh */

#endif
```

Appendix E. CMakeLists.txt

This appendix contains the which is the configuration file that Gradle uses to build native source makefiles.

```
# Sets the minimum version of CMake required to build the native
    library.
cmake_minimum_required(VERSION 3.4.1)

# Includes directories that source files might require as
    dependencies
include_directories(/Users/jessicaswitzler/Library/Android/Sdk/ndk/
                    openssl-1.1.1c/include/)
include_directories(/Users/jessicaswitzler/Library/Android/Sdk/ndk/
                    openssl-1.1.1c/)

# Creates and names a library, sets it as either STATIC
# or SHARED, and provides the relative paths to its source code.
# You can define multiple libraries, and CMake builds them for you.
# Gradle automatically packages shared libraries with your APK.
add_library(
        # Sets the name of the library.
        native-lib
        # Sets the library as a shared library.
        SHARED
        # Provides a relative path to your source file(s).
        cpapke.c fips202.c kem.c ntt.c poly.c precomp.c reduce.c
        rng.c verify.c MainActivityDriver.c)

add_library(
        imported-crypto-lib
        STATIC
        IMPORTED )

set_target_properties(
        # Specifies the target library.
        imported-crypto-lib
        # Specifies the location to import the library from
        PROPERTIES IMPORTED_LOCATION
        # Provides the path to the library you want to import.
        /Users/jessicaswitzler/Library/Android/Sdk/ndk/
        openssl-1.1.1c/libcrypto.a)

# Searches for a specified prebuilt library and stores the path as a
# variable. Because CMake includes system libraries in the search
    path by
# default, you only need to specify the name of the public NDK
    library
# you want to add. CMake verifies that the library exists before
# completing its build.
find_library(
        # Names the library
        log-lib
```

```
        # Specifies the name of the NDK library that
        # you want CMake to locate.
        log)

# Specifies libraries CMake should link to your target library. You
# can link multiple libraries, such as libraries you define in this
# build script, prebuilt third-party libraries, or system libraries.
target_link_libraries(
        # Specifies the target library.
        native-lib
        # Links the target library to the log library
        # included in the NDK.
        imported-crypto-lib
        ${log-lib})
```

Appendix F. Application Build Gradle Settings

```
apply plugin: 'com.android.application'

android {
    compileSdkVersion 28
    buildToolsVersion "29.0.2"
    defaultConfig {
        applicationId "com.example.newhope512cca"
        minSdkVersion 23
        targetSdkVersion 24
        versionCode 1
        versionName "1.0"
        testInstrumentationRunner "androidx.test.runner.
            AndroidJUnitRunner"
        externalNativeBuild {
            cmake {
                cppFlags ""
                cFlags ""
            }
        }
        sourceSets {
            main {
                jniLibs.srcDirs '/home/switzbeats/Android/Sdk/ndk/
                    openssl-1.1.1c/lib/x86'
            }
        }
        defaultConfig {
            ndk {
                abiFilters 'arm64-v8a'
            }
        }
    }
    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android-
                optimize.txt'), 'proguard-rules.pro'
        }
    }
    externalNativeBuild {
        cmake {
            path "src/main/cpp/CMakeLists.txt"
            version "3.10.2"
        }
    }
}

dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])
    implementation 'androidx.appcompat:appcompat:1.1.0'
    implementation 'androidx.constraintlayout:constraintlayout:1.1.3
```

```
        '
    testImplementation 'junit:junit:4.12'
    androidTestImplementation 'androidx.test.ext:junit:1.1.1'
    androidTestImplementation 'androidx.test.espresso:espresso-core
        :3.2.0'
}
```

Appendix G. Modified PQCgenKAT_kem.c

This appendix contains the modified `PQCgenKAT_kem.c` file that was used to construct the hardcoded Java arrays containing the KAT values. These hardcoded arrays were utilized to verify the correctness of each mobile implementation.

```
//
//  PQCgenKAT_kem.c
//
//  Created by Bassham, Lawrence E (Fed) on 8/29/17.
//  Copyright Â© 2017 Bassham, Lawrence E (Fed). All rights reserved
    .
//
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include "rng.h"
#include "api.h"

#define MAX_MARKER_LEN          50
#define KAT_SUCCESS          0
#define KAT_FILE_OPEN_ERROR -1
#define KAT_DATA_ERROR      -3
#define KAT_CRYPTO_FAILURE  -4

int             FindMarker(FILE *infile, const char *marker);
int             ReadHex(FILE *infile, unsigned char *A, int Length,
    char *str);
void    fprintBstr(FILE *fp, char *S, unsigned char *A, unsigned
    long long L);
void    printBstr(char *S, unsigned char *A, unsigned long long L);
void    string2hexString(unsigned char* input, char* output,
    unsigned long long len);

int
main()
{
    char                fn_req[32], fn_rsp[32];
    FILE                *fp_req, *fp_rsp;
    unsigned char       seed[48];
    unsigned char       entropy_input[48];
    unsigned char       ct[CRYPTO_CIPHERTEXTBYTES], ss[CRYPTO_BYTES
        ], ss1[CRYPTO_BYTES];
    int                 count;
    int                 done;
    unsigned char       pk[CRYPTO_PUBLICKEYBYTES], sk[
        CRYPTO_SECRETKEYBYTES];
    int                 ret_val;

    //ADD TEMP VARIABLES TO STORE OUTPUT
    //TEMP VALUES ARE CHAR* BECAUSE IT IS NEEDED FOR sprintf
```

```
char                tempSeed[48];
char                tempCt[CRYPTO_CIPHERTEXTBYTES], tempSs[
    CRYPTO_BYTES], tempPk[CRYPTO_PUBLICKEYBYTES], tempSk[
    CRYPTO_SECRETKEYBYTES];
//END ADD TEMP VARIABLES TO STORE OUTPUT

// Create the REQUEST file
sprintf(fn_req, "PQCkemKAT_%d.req", CRYPTO_SECRETKEYBYTES);
if ( (fp_req = fopen(fn_req, "w")) == NULL ) {
    printf("Couldn't open <%s> for write\n", fn_req);
    return KAT_FILE_OPEN_ERROR;
}
sprintf(fn_rsp, "PQCkemKAT_%d.rsp", CRYPTO_SECRETKEYBYTES);
if ( (fp_rsp = fopen(fn_rsp, "w")) == NULL ) {
    printf("Couldn't open <%s> for write\n", fn_rsp);
    return KAT_FILE_OPEN_ERROR;
}

for (int i=0; i<48; i++)
    entropy_input[i] = i;

randombytes_init(entropy_input, NULL, 256);
for (int i=0; i<100; i++) {
    fprintf(fp_req, "count = %d\n", i);
    randombytes(seed, 48);

    fprintBstr(fp_req, "seed = ", seed, 48);
    fprintf(fp_req, "pk =\n");
    fprintf(fp_req, "sk =\n");
    fprintf(fp_req, "ct =\n");
    fprintf(fp_req, "ss =\n\n");

    //NOTE: THROUGHOUT THIS EDITED MAIN THERE ARE MULTIPLE
       SECTIONS THAT ARE REQUIRED TO BE COMMENTED OUT
       SELECTIVELY TO COMPARE THE OUTPUT TO THE REQUEST
    //AND RESPONSE FILES. IN ORDER TO SELECT WHICH ONES SHOULD
       BE COMMENTED OUT IT THE KAT REQ AND RSP FILES SHOULD BE
       EVALUATED. ON TOP OF THAT WHEN RUNNING
    //THE PRODUCED EXECUTABLES SHOULD BE RAN AND THE CONSOLE
       OUTPUT SHOULD BE SAVED TO A FILE FOR A TERMINAL CMP
       COMMAND TO BE RAN TO COMPARE THE TWO FILES AND
    //CHECK FOR EQUIVALENCY. THE SAME PROCEDURE SHOULD BE
       FOLLOWED TO PRODUCE THE ARRAY VALUES FOR THE JAVA CODE TO
        WORK FOR THE TEST CORRECTNESS BUTTON.

    //ADDED PRINT STATEMENT TO TEST OUTPUT FOR CORRECTNESS
    //printf("count = %d\n", i);
    //string2hexString(seed, tempSeed, 48);
    //printf("seed = %s\n", tempSeed);
    //printf("pk =\n");
    //printf("sk =\n");
    //printf("ct =\n");
```

```
    //printf("ss =\n\n");
    //CAN PRODUCE OUTPUT FOR ARRAYS BY COMMENTING OUT THE ABOVE
       string2hexString STATEMENT AND UNCOMMENTING OUT THE CODE
       BELOW SELECTIVELY
    //printf("\"%s\", \n", tempSeed);
    //END OF ADDED PRINT STATEMENT
}
fclose(fp_req);

//Create the RESPONSE file based on what's in the REQUEST file
if ( (fp_req = fopen(fn_req, "r")) == NULL ) {
    printf("Couldn't open <%s> for read\n", fn_req);
    return KAT_FILE_OPEN_ERROR;
}

fprintf(fp_rsp, "# %s\n\n", CRYPTO_ALGNAME);

//ADDED PRINT STATEMENT TO TEST OUTPUT FOR CORRECTNESS
//printf("# %s\n\n", CRYPTO_ALGNAME);
//END OF ADDED PRINT STATEMENT

done = 0;
do {
    if ( FindMarker(fp_req, "count = ") )
        fscanf(fp_req, "%d", &count);
    else {
        done = 1;
        break;
    }
    fprintf(fp_rsp, "count = %d\n", count);

    //ADDED PRINT STATEMENT TO TEST OUTPUT FOR CORRECTNESS
    //printf("count = %d\n", count);
    //END OF ADDED PRINT STATEMENT

    if ( !ReadHex(fp_req, seed, 48, "seed = ") ) {
        printf("ERROR: unable to read 'seed' from <%s>\n",
           fn_req);
        return KAT_DATA_ERROR;
    }
    fprintBstr(fp_rsp, "seed = ", seed, 48);

    //ADDED PRINT STATEMENT TO TEST FOR CORRECTNESS
    //printBstr("seed = ", seed, 48);
    //END OF ADDED PRINT STATEMENT

    randombytes_init(seed, NULL, 256);

    // Generate the public/private keypair
    if ( (ret_val = crypto_kem_keypair(pk, sk)) != 0) {
        printf("crypto_kem_keypair returned <%d>\n", ret_val);
        return KAT_CRYPTO_FAILURE;
```

```
}
fprintBstr(fp_rsp, "pk = ", pk, CRYPTO_PUBLICKEYBYTES);
fprintBstr(fp_rsp, "sk = ", sk, CRYPTO_SECRETKEYBYTES);


//ADDED PRINT STATEMENT TO TEST OUTPUT FOR CORRECTNESS
//string2hexString(pk, tempPk, CRYPTO_PUBLICKEYBYTES);
//printf("pk = %s\n", tempPk);
//CAN PRODUCE OUTPUT FOR ARRAYS BY COMMENTING OUT THE ABOVE
    string2hexString STATEMENT AND UNCOMMENTING OUT THE CODE
    BELOW SELECTIVELY
//printf("\"%s\", \n", tempPk);
//END OF ADDED PRINT STATEMENT

//ADDED PRINT STATEMENT TO TEST OUTPUT FOR CORRECTNESS
//string2hexString(sk, tempSk, CRYPTO_SECRETKEYBYTES);
//printf("sk = %s\n", tempSk);
//CAN PRODUCE OUTPUT FOR ARRAYS BY COMMENTING OUT THE ABOVE
    string2hexString STATEMENT AND UNCOMMENTING OUT THE CODE
    BELOW SELECTIVELY
//printf("\"%s\", \n", tempSk);
//END OF ADDED PRINT STATEMENT


if ( (ret_val = crypto_kem_enc(ct, ss, pk)) != 0) {
    printf("crypto_kem_enc returned <%d>\n", ret_val);
    return KAT_CRYPTO_FAILURE;
}
fprintBstr(fp_rsp, "ct = ", ct, CRYPTO_CIPHERTEXTBYTES);
fprintBstr(fp_rsp, "ss = ", ss, CRYPTO_BYTES);
fprintf(fp_rsp, "\n");

//NORMALLY WOULD PUT TEST OUTPUT FOR VALUES CT AND SS AND
    CREATE ARRAY HERE. HOWEVER, THERE IS AN ISSUE WITH
    PLACING THE string2hexstring(ct) HERE IT
//WOULD CAUSE AN ERROR IN THE PROGRAMMING EXITING WITH "
    crypto_kem_dec returned bad 'ss' value\n", DESPITE THE
    STRINGS BEING EQUIVALENT TO THOSE OF THE
//ORIGINAL PROGRAM, 'cmp'ing ALL OUTPUT VALUES TO THOSE OF
    THE KAT PROVIDED THAT THERE WAS NO VALUES DIFFERENT IT
    WASN'T UNTIL THE ERROR THAT THERE WAS A
//DIFFERENCE IN STRING VALUES. I THOUGHT IT MIGHT BE CAUSED
    BY ADDING A NULL TERMINATED TO THE END OF THE char *.
    HOWEVER, NONE OF THE OTHER STRINGS HAD
//HAVE CAUSED AN ERROR. ALSO THE ERROR ONLY APPEARED WITH
    string2hexstring(ct) NOT ANY OF THE OTHER
    string2hexstring() CALLS. IT IS FOR THAT REASON THAT THE
//VALUES ARE PRINTED AT THE END. THIS DOESN'T CHANGE THE
    CORRECTNESS OF THE IMPLEMENTATION, WHEN ADDING IT TO THIS
     LOCATION THE OUTPUT IS IDENTICAL TO THAT
//OF THE KAT FILE.
```

```
        if ( (ret_val = crypto_kem_dec(ss1, ct, sk)) != 0) {
            printf("crypto_kem_dec returned <%d>\n", ret_val);
            return KAT_CRYPTO_FAILURE;
        }

        if ( memcmp(ss, ss1, CRYPTO_BYTES) ) {
            printf("crypto_kem_dec returned bad 'ss' value\n");
            printBstr("\n\nFAILED ss: ", ss, CRYPTO_BYTES);
            printBstr("\nFAILED ss1: ", ss1, CRYPTO_BYTES);
            return KAT_CRYPTO_FAILURE;
        }

        //ADDED PRINT STATEMENT TO TEST OUTPUT FOR CORRECTNESS
        //string2hexString(ct, tempCt, CRYPTO_CIPHERTEXTBYTES);
        //printf("ct = %s\n", tempCt);
        //CAN PRODUCE OUTPUT FOR ARRAYS BY COMMENTING OUT THE ABOVE
           string2hexString STATEMENT AND UNCOMMENTING OUT THE CODE
           BELOW SELECTIVELY
        //printf("\"%s\", \n", tempCt);
        //END OF ADDED PRINT STATEMENT

        //ADDED PRINT STATEMENT TO TEST OUTPUT FOR CORRECTNESS
        //string2hexString(ss, tempSs, CRYPTO_BYTES);
        //printf("ss = %s\n", tempSs);
        //CAN PRODUCE OUTPUT FOR ARRAYS BY COMMENTING OUT THE ABOVE
           string2hexString STATEMENT AND UNCOMMENTING OUT THE CODE
           BELOW SELECTIVELY
        //printf("\"%s\", \n", tempSs);
        //END OF ADDED PRINT STATEMENT

        //ADDED PRINT STATEMENT TO TEST OUTPUT FOR CORRECTNESS
        //printf("\n");
        //END OF ADDED PRINT STATEMENT

    } while ( !done );

    fclose(fp_req);
    fclose(fp_rsp);

    return KAT_SUCCESS;
}

//
// ALLOW TO READ HEXADECIMAL ENTRY (KEYS, DATA, TEXT, etc.)
//
//
// ALLOW TO READ HEXADECIMAL ENTRY (KEYS, DATA, TEXT, etc.)
//
int
FindMarker(FILE *infile, const char *marker)
{
        char    line[MAX_MARKER_LEN];
```

```
        int             i, len;
        int curr_line;

        len = (int)strlen(marker);
        if ( len > MAX_MARKER_LEN-1 )
                len = MAX_MARKER_LEN-1;

        for ( i=0; i<len; i++ )
          {
            curr_line = fgetc(infile);
            line[i] = curr_line;
            if (curr_line == EOF )
              return 0;
          }
        line[len] = '\0';

        while ( 1 ) {
                if ( !strncmp(line, marker, len) )
                        return 1;

                for ( i=0; i<len-1; i++ )
                        line[i] = line[i+1];
                curr_line = fgetc(infile);
                line[len-1] = curr_line;
                if (curr_line == EOF )
                    return 0;
                line[len] = '\0';
        }

        // shouldn't get here
        return 0;
}

//
// ALLOW TO READ HEXADECIMAL ENTRY (KEYS, DATA, TEXT, etc.)
//
int
ReadHex(FILE *infile, unsigned char *A, int Length, char *str)
{
        int                     i, ch, started;
        unsigned char   ich;

        if ( Length == 0 ) {
                A[0] = 0x00;
                return 1;
        }
        memset(A, 0x00, Length);
        started = 0;
        if ( FindMarker(infile, str) )
                while ( (ch = fgetc(infile)) != EOF ) {
                        if ( !isxdigit(ch) ) {
                                if ( !started ) {
```

```
                                        if ( ch == '\n' )
                                                break;
                                        else
                                                continue;
                                }
                                else
                                        break;
                        }
                        started = 1;
                        if ( (ch >= '0') && (ch <= '9') )
                                ich = ch - '0';
                        else if ( (ch >= 'A') && (ch <= 'F') )
                                ich = ch - 'A' + 10;
                        else if ( (ch >= 'a') && (ch <= 'f') )
                                ich = ch - 'a' + 10;
                else // shouldn't ever get here
                        ich = 0;

                        for ( i=0; i<Length-1; i++ )
                                A[i] = (A[i] << 4) | (A[i+1] >> 4);
                        A[Length-1] = (A[Length-1] << 4) | ich;
                }
        else
                return 0;

        return 1;
}

void
fprintBstr(FILE *fp, char *S, unsigned char *A, unsigned long long L
    )
{
        unsigned long long  i;

        fprintf(fp, "%s", S);

        for ( i=0; i<L; i++ )
                fprintf(fp, "%02X", A[i]);

        if ( L == 0 )
                fprintf(fp, "00");

        fprintf(fp, "\n");
}

//ADDED FUNCTION TO ALTER fprintBstr() SO THAT IT WORKS FOR CONSOLE
    OUTPUT
//COULD USE JUST THIS FUNCTION FOR ARRAY VALUE CREATIONS HOWEVER TO
    ALTER STRINGS WOULD NEED TO OUT 1 MORE PARAMETER
//ALTERED VERSION FOR ARRAY CREATION WOULD LOOK LIKE void printBstr(
    char *S, unsigned char *A, unsigned long long L, char *E)
```

```
//WHERE char *E WOULD BE FOR ENDING CHARACTERS NEEDING TO BE ADDED.
    HOWEVER, THIS FUNCTION WONT WORK IN JNI TO CREATE JAVASTRINGS
//SINCE IT DOESNT PRODUCE A SINGLUAR STRING INSTEAD IT PRODUCES
    CONSOLE OUTPUT
void printBstr(char *S, unsigned char *A, unsigned long long L)
{
    unsigned long long i;

    printf("%s", S);

    for(i=0; i<L; i++)
    {
        printf("%02X", A[i]);
    }

    if(L==0)
    {
        printf("00");
    }

    printf("\n");

}

//ADDED FUNCTION TO CREATE HEX STRING FOR OUTPUT VALUES
void string2hexString(unsigned char* input, char* output, unsigned
    long long len)
{
    int loop = 0;
    int i = 0;
    //printf("\n\nINPUT STRING INTO STRING TO HEX STRING BEFORE LOOP
        IS: %s \n", input);
    while(loop < len)
    {
        sprintf((char*)(output+i),"%02X", input[loop]);
        loop+=1;
        i+=2;
    }
    //insert NULL at the end of the output string
    output[i++] = '\0';
    //printf("\nINPUT STRING INTO STRING TO HEX STRING AFTER LOOP IS
        : %s\n\n", input);
}
//END ADDED FUNCTION
```

Appendix H. NewHope Sample Tables

Table 20: Make Keys samples for for NewHope512-CCA with a Verbose OnClick Function

NewHope512-CCA Results With a Verbose OnClick Function		
Native Function Time(ms)	**OnClick Function Time(ms)**	**Highest CPU Usage(%)**
6.15	105.96	21.0%
13.26	111.53	26.0%
4.66	91.51	28.7%
7.34	103.11	25.0%
5.55	91.45	22.9%
8.32	111.16	26.8%
6.28	87.71	24.8%
6.09	88.24	22.0%
5.47	77.68	20.9%
6.71	86.46	24.9%
6.16	89.36	16.0%
5.92	81.03	16.0%
9.66	88.30	18.9%
8.81	84.11	17.0%
17.37	96.95	18.0%
6.16	79.60	17.0%
2.83	68.01	18.0%
5.19	78.45	18.9%
5.29	69.52	20.0%
6.25	93.68	21.9%
6.39	76.47	16.0%
4.00	77.66	18.9%
6.26	74.27	19.9%
8.90	100.07	19.0%
8.70	85.42	22.9%
8.54	96.16	21.9%
6.37	87.5	19.9%
6.88	80.57	23.9%
5.71	92.25	19.0%
4.91	88.07	21.9%
7.86	83.57	19.9%
8.27	86.46	21.9%
8.77	86.06	22.9%

Continued on next page

Table 20 – continued from previous page

Native Function Time(ms)	OnClick Function Time(ms)	Highest CPU Usage(%)
6.45	79.77	20.0%
6.24	78.80	21.0%
7.51	84.67	22.8%
2.47	83.30	17.9%
7.83	79.06	17.9%
10.74	93.68	20.9%
4.78	86.53	21.8%
6.68	111.13	22.0%
4.74	109.50	19.9%
5.29	98.99	27.8%
4.71	91.70	23.0%
8.54	99.78	20.0%
6.76	85.34	23.9%
5.16	85.25	21.9%
7.12	112.52	23.9%
11.11	97.35	20.9%
5.91	75.96	18.0%

Table 21: Make Keys samples for NewHope1024-CCA With a Verbose OnClick Function

NewHope1024-CCA Results With a Verbose OnClick Function		
Native Function Time(ms)	**OnClick Function Time(ms)**	**Highest CPU Usage(%)**
11.51	193.72	23.9%
11.83	182.68	30.7%
9.62	206.13	25.5%
7.63	142.59	30.0%
14.19	168.74	18.9%
12.33	166.07	24.8%
10.07	154.82	28.8%
10.61	154.04	26.0%
12.36	160.99	30.3%
11.79	138.23	22.9%
11.64	152.54	27.7%
9.43	144.4	25.6%
10.04	145.54	24.4%
10.55	142.9	23.8%
9.90	130.09	24.9%
15.82	152.83	25.0%
9.09	142.09	29.9%
14.39	153.27	29.0%
13.53	158.81	18.0%
13.03	142.51	21.0%
10.69	152.92	26.0%
9.63	158.44	22.0%
9.94	144.71	28.0%
17.75	166.32	28.9%
11.49	157.31	27.0%
12.12	147.64	27.0%
12.01	160.17	18.0%
10.57	147.2	27.0%
11.27	151.15	22.0%
18.98	160.19	26.9%
10.02	148.54	27.9%
9.45	146.14	20.0%
11.69	146.57	24.0%
9.00	145.09	25.0%
16.06	159.19	21.9%
		Continued on next page

Table 21 – continued from previous page

Native Function Time(ms)	OnClick Function Time(ms)	Highest CPU Usage(%)
10.76	135.57	24.0%
13.02	145.21	19.0%
11.63	151.88	26.9%
10.5	143.13	20.0%
13.85	159.68	22.0%
13.12	149.65	24.9%
11.61	136.54	18.0%
16.55	168.08	25.9%
9.75	140.61	27.0%
10.57	143.86	21.0%
10.35	195.42	34.7%
11.25	189.62	29.8%
11.31	185.14	34.9%
9.32	159.82	26.8%
14.38	153.41	29.9%

Table 22: Make Keys samples for NewHope512-CPA With a Verbose OnClick Function

NewHope512-CPA Results With a Verbose OnClick Function		
Native Function Time(ms)	**OnClick Function Time(ms)**	**Highest CPU Usage(%)**
6.50	99.15	27.0%
5.56	84.07	24.9%
6.39	94.10	28.0%
4.46	66.22	23.9%
3.94	72.50	19.0%
3.65	62.69	21.9%
7.31	82.25	24.8%
4.91	66.50	18.0%
5.30	77.89	24.0%
2.59	62.99	20.0%
6.00	68.12	17.9%
4.32	68.20	20.0%
3.65	73.04	19.9%
4.31	82.87	21.0%
3.83	62.69	17.0%
4.70	67.69	20.0%
4.09	73.79	20.9%
3.62	60.81	17.9%
4.20	68.67	20.9%
7.27	71.86	18.0%
4.28	61.67	23.8%
3.96	59.15	19.0%
4.21	64.21	21.9%
4.24	65.69	19.9%
5.60	109.53	25.9%
5.26	72.04	12.9%
4.77	65.77	20.9%
2.52	65.11	18.0%
3.75	83.53	21.9%
3.63	61.6	19.0%
3.98	69.36	15.0%
4.22	66.25	23.9%
4.51	59.92	16.0%
2.54	64.13	19.9%
6.73	72.11	20.9%
		Continued on next page

Table 22 – continued from previous page

Native Function Time(ms)	OnClick Function Time(ms)	Highest CPU Usage(%)
3.78	58.27	15.9%
3.03	59.23	17.9%
5.72	73.69	13.0%
3.45	62.43	17.0%
4.10	58.51	17.0%
4.28	66.69	16.0%
4.32	67.99	18.9%
8.17	71.3	18.9%
6.57	69.64	18.9%
5.26	73.48	19.9%
2.69	60.01	15.0%
1.47	64.53	18.0%
3.61	60.42	17.9%
4.47	85.81	23.9%
7.69	84.11	17.0%

Table 23: Make Keys samples for NewHope1024-CPA With a Verbose OnClick Function

NewHope1024-CPA Results With a Verbose OnClick Function		
Native Function Time(ms)	**OnClick Function Time(ms)**	**Highest CPU Usage(%)**
10.79	159.01	26.9%
10.65	127.79	27.9%
7.82	118.67	22.9%
6.34	119.88	26.0%
12.15	136.43	26.9%
6.79	116.12	27.8%
8.42	119.63	26.0%
9.76	125.71	25.9%
6.63	108.7	23.9%
7.50	113.22	24.9%
6.41	115.81	17.0%
12.97	120.23	18.0%
6.86	108.48	20.0%
11.36	121.71	23.8%
8.53	122.25	22.9%
10.73	121.01	22.5%
11.03	114.97	19.0%
7.20	116.29	24.9%
8.43	118.82	19.9%
6.46	112.09	24.0%
7.43	109.18	21.9%
6.17	120.09	23.8%
7.08	106.11	19.0%
6.53	109.05	18.0%
8.23	105.3	20.9%
5.28	112.06	24.9%
4.82	112.62	24.0%
9.68	119.80	25.0%
8.10	111.55	23.0%
6.29	123.56	23.9%
7.68	115.13	25.0%
9.49	115.99	25.9%
6.99	111.28	22.9%
5.88	110.36	23.8%
8.57	114.56	18.0%
Continued on next page		

Table 23 – continued from previous page

Native Function Time(ms)	OnClick Function Time(ms)	Highest CPU Usage(%)
6.65	115.39	16.9%
7.20	108.33	22.9%
7.10	118.78	21.9%
7.61	111.11	21.8%
8.27	124.24	22.9%
6.07	109.12	23.0%
9.83	113.67	23.0%
10.98	116.75	21.9%
7.97	121.57	24.9%
6.99	116.22	22.9%
7.37	174.38	28.9%
10.96	145.46	25.0%
6.74	132.17	25.9%
14.89	137.71	30.9%
9.65	122.46	26.9%

Table 24: Make Keys samples for NewHope512-CCA With a Succinct OnClick Function

NewHope512-CCA Results With a Succinct OnClick Function		
Native Function Time(ms)	**OnClick Function Time(ms)**	**Highest CPU Usage(%)**
5.90	18.59	16.0
6.72	12.77	20.0
4.78	10.85	14.0
8.64	15.47	12.0
7.67	18.71	13.0
6.19	12.84	13.0
6.10	14.77	13.0
7.42	18.05	12.0
7.39	12.44	12.0
6.64	12.14	12.0
5.66	12.57	16.0
8.36	17.08	14.0
6.42	16.65	13.0
9.86	18.95	13.0
6.48	11.97	15.0
6.94	12.16	13.0
5.91	13.24	15.0
6.42	13.04	14.0
6.40	10.23	18.0
2.71	8.23	11.0
7.00	12.26	11.0
6.36	13.30	11.0
8.95	17.58	14.0
6.54	11.91	13.0
11.08	20.74	12.0
7.21	18.08	12.0
9.19	15.70	15.0
8.33	13.78	12.0
8.44	14.71	12.0
7.26	17.72	12.0
6.11	15.18	17.0
6.56	10.40	20.0
5.18	10.36	11.0
6.95	14.03	16.0
2.71	11.71	16.0

Continued on next page

Table 24 – continued from previous page

Native Function Time(ms)	OnClick Function Time(ms)	Highest CPU Usage(%)
8.16	16.45	13.0
6.77	10.55	13.0
6.81	12.33	12.0
6.31	11.88	12.0
7.22	14.64	13.0
5.65	11.21	12.0
7.24	11.62	12.0
6.09	11.18	17.0
5.24	12.70	13.0
3.58	12.37	12.0
7.47	11.62	12.0
5.29	12.58	13.0
6.13	9.84	12.0
6.28	13.05	12.0
4.18	12.42	14.0

Table 25: Make Keys samples for NewHope1024-CCA With a Succinct OnClick Function

NewHope1024-CCA Results With a Succinct OnClick Function		
Native Function Time(ms)	**OnClick Function Time(ms)**	**Highest CPU Usage(%)**
11.72	19.76	15.0
15.08	17.73	14.0
15.02	19.40	15.9
13.81	22.99	16.0
3.95	9.27	15.0
14.77	24.96	17.0
14.51	26.10	14.0
16.40	21.66	16.9
13.98	21.25	12.0
11.52	18.39	14.0
15.91	21.31	13.0
15.71	21.31	15.0
6.31	13.02	16.0
13.43	18.65	17.0
10.12	15.76	15.0
14.92	20.09	13.0
12.07	19.89	14.0
9.75	16.07	17.0
10.05	17.94	20.9
14.35	20.55	13.0
5.13	15.8	16.0
14.60	19.74	14.0
10.10	15.41	12.0
10.17	15.85	12.0
11.18	17.33	15.0
11.47	21.52	11.0
13.36	21.41	11.0
9.36	19.00	12.0
10.8	17.87	14.0
7.18	12.60	13.0
9.34	13.89	11.0
10.41	17.22	13.0
12.16	17.58	12.0
12.56	22.51	12.0
6.99	10.32	14.0
Continued on next page		

Table 25 – continued from previous page

Native Function Time(ms)	OnClick Function Time(ms)	Highest CPU Usage(%)
15.47	19.63	16.0
17.06	26.58	15.0
15.68	19.75	11.0
17.70	25.25	13.0
11.85	17.26	13.0
4.73	11.70	12.0
11.85	19.08	17.0
10.36	17.29	15.0
14.27	25.37	17.0
10.46	15.87	15.0
11.35	16.51	13.0
3.87	10.91	12.0
20.13	25.35	14.0
11.78	19.27	13.0
10.26	20.84	11.0

Table 26: Make Keys samples for NewHope512-CPA With a Succinct OnClick Function

NewHope512-CPA Results With a Succinct OnClick Function		
Native Function Time(ms)	**OnClick Function Time(ms)**	**Highest CPU Usage(%)**
6.96	14.32	13.0
6.65	16.88	14.9
3.93	7.88	15.0
6.43	11.32	16.0
4.50	11.89	14.0
3.48	10.75	11.0
4.32	9.87	13.9
3.75	14.70	15.0
3.68	12.17	13.0
9.02	14.23	13.0
3.60	13.90	15.0
3.93	11.76	13.0
4.08	9.48	14.0
4.84	11.59	17.0
5.29	8.71	16.0
2.70	11.32	15.0
3.94	11.05	15.0
4.77	10.38	20.0
4.41	8.18	12.0
5.41	15.14	13.0
3.46	9.26	15.0
6.83	11.76	12.0
7.59	13.61	13.9
7.62	15.47	12.0
7.26	12.6	12.0
6.15	11.48	14.0
7.32	12.69	12.0
5.06	10.27	12.0
5.12	9.22	11.0
4.60	9.09	12.0
5.44	10.93	13.0
4.24	10.02	14.0
5.28	13.43	13.0
5.33	16.80	14.0
5.19	14.41	14.0

Continued on next page

Table 26 – continued from previous page

Native Function Time(ms)	OnClick Function Time(ms)	Highest CPU Usage(%)
4.66	10.19	15.0
6.92	11.18	13.0
4.45	14.07	10.0
3.06	8.85	14.0
3.83	8.05	11.0
6.36	17.36	18.0
3.70	7.58	12.0
8.42	22.55	14.0
4.93	8.87	11.0
5.72	11.28	13.0
3.02	9.50	14.0
4.52	10.1	11.0
7.96	12.17	13.0
3.65	18.13	11.0
4.53	8.50	13.0

Table 27: Make Keys samples for NewHope1024-CPA With a Succinct OnClick Function

NewHope1024-CPA Results With a Succinct OnClick Function		
Native Function Time(ms)	**OnClick Function Time(ms)**	**Highest CPU Usage(%)**
8.58	17.15	14.0
7.49	13.16	14.0
8.73	11.95	13.0
6.00	11.45	13.0
8.48	12.22	15.0
7.01	12.10	12.0
6.97	11.96	15.9
9.94	15.61	17.0
7.91	14.67	13.0
7.85	15.30	13.0
7.59	13.49	14.0
6.70	12.36	12.0
9.81	16.68	14.0
7.76	10.51	17.0
7.39	11.30	14.0
7.55	16.23	13.0
7.73	13.02	19.0
8.41	12.15	11.0
8.36	13.70	13.0
7.30	12.56	11.0
6.17	11.39	15.0
10.77	16.33	11.0
7.40	17.28	15.9
6.49	11.88	13.0
7.21	11.55	11.0
6.61	13.57	12.0
6.75	14.11	15.0
7.44	16.04	12.0
8.66	12.63	14.0
6.09	11.53	18.0
10.10	13.98	14.0
6.99	12.28	13.0
8.53	16.22	13.0
8.37	21.15	14.0
5.53	10.62	12.0

Continued on next page

Table 27 – continued from previous page

Native Function Time(ms)	OnClick Function Time(ms)	Highest CPU Usage(%)
7.41	13.03	14.0
6.83	12.83	14.0
6.41	13.16	11.0
6.08	16.56	12.0
12.18	20.26	15.0
7.49	12.66	12.0
12.14	17.4	13.0
5.39	12.38	12.0
9.59	17.21	15.0
4.24	11.34	14.0
7.10	11.07	13.0
8.32	12.34	10.0
11.79	18.77	14.0
8.75	14.55	9.0
6.40	10.44	11.0

Table 28: Results From Running Make Keys for NewHope512-CCA With a No Output OnClick Function

NewHope512-CCA Results With a No Output OnClick Function		
Native Function Time(ms)	**OnClick Function Time(ms)**	**Highest CPU Usage(%)**
8.48	8.48	15
6.22	6.22	16
6.54	6.54	11
5.59	5.59	16
4.78	4.78	11
7.09	8.3	15
7.88	7.88	17
1.34	1.34	14
7.68	7.68	13
4.86	4.86	13
7.24	7.24	14
8.74	8.74	14
7.37	7.37	11
7.01	7.01	15
8.8	8.8	13
3.7	3.7	13
6.62	6.62	14
2.94	2.94	11
6.52	6.52	12
5.3	5.3	14
6.31	6.31	13
8.3	8.3	14
2.78	2.78	11
5.85	5.85	10
9.08	9.08	11
2.49	2.49	12
9.25	9.25	13
8.78	8.78	11
6.98	6.98	13
5.38	5.38	8
8.67	8.67	13
5.55	5.55	12
8.89	8.89	12
4.97	4.97	12
5.35	5.35	15

Continued on next page

Table 28 – continued from previous page

Native Function Time(ms)	OnClick Function Time(ms)	Highest CPU Usage(%)
5.21	5.21	11
7.24	7.24	15
5.16	5.16	10
5.15	5.15	12
2.6	2.6	11
6.22	6.22	14
11.31	11.31	12
10.06	10.06	11
7.54	7.54	11
5.61	5.61	11
10.36	10.36	12
7.04	7.04	12
9.07	9.07	13
5.26	5.26	12
4.57	4.57	14

Table 29: Make Keys samples for NewHope1024-CCA With a No Output OnClick Function

NewHope1024-CCA Results With a No Output OnClick Function		
Native Function Time(ms)	**OnClick Function Time(ms)**	**Highest CPU Usage(%)**
13.15	13.15	20
16.16	16.16	14
11.84	11.84	14
12.93	12.93	14
12.74	12.74	12
15.07	15.07	14
10.64	10.64	14
14.9	14.9	16
13.51	13.51	16
11.35	11.35	12
11.19	11.19	12
9.88	9.88	13
15.3	15.3	12
12.34	12.34	17
15.81	15.81	15
10.85	10.85	11
9.85	9.85	17
15.66	15.66	12
8.09	8.09	12
11.12	11.12	12
15.91	15.91	11
12.17	12.17	12
10.91	10.91	12
8.8	8.8	12
8.61	8.61	11
12.74	12.74	14
13.57	13.57	13
10.49	10.49	13
12.49	12.49	11
14.66	14.66	11
11.27	11.27	13
13.27	13.27	11
13.47	13.47	14
11.79	11.79	14
10.74	10.74	10

Continued on next page

Table 29 – continued from previous page

Native Function Time(ms)	OnClick Function Time(ms)	Highest CPU Usage(%)
9.53	9.53	14
12.83	12.83	11
14.77	14.77	12
9.45	9.45	13
14.51	14.51	10
11.03	11.03	12
13.26	13.26	10
9.7	9.7	12
12.62	12.62	11
12.31	12.31	12
12.27	12.27	13
14.65	14.65	12
10.63	10.63	11
10.56	10.56	11
11.04	11.04	16

Table 30: Make Keys samples for NewHope512-CPA With a No Output OnClick Function

NewHope512-CPA Results With a No Output OnClick Function		
Native Function Time(ms)	**OnClick Function Time(ms)**	**Highest CPU Usage(%)**
7.49	7.49	16
5.09	5.09	14
7.19	7.19	12
4.82	4.82	14
4.2	4.2	12
2.78	2.78	11
2.7	2.7	14
6.07	6.07	14
5.31	5.31	15
5.01	5.01	11
4.84	4.84	12
4.01	4.01	12
4.2	4.2	11
2.77	2.77	10
3.94	3.94	12
3.53	3.53	11
2.76	4.15	15
5.15	5.15	19
3.9	3.9	11
3.75	3.75	12
3.45	3.45	10
4.43	4.43	10
5.2	5.2	10
6.31	6.31	14
4.65	4.65	11
4.28	4.28	10
3.84	3.84	11
4.88	4.88	11
5.62	5.62	10
4.02	4.02	12
4.02	4.02	12
5.11	5.11	11
4.03	4.03	12
3.8	3.8	11
2.47	2.47	11
		Continued on next page

Table 30 – continued from previous page

Native Function Time(ms)	OnClick Function Time(ms)	Highest CPU Usage(%)
3.97	3.97	12
5.21	5.21	12.9
3.8	3.8	10
2.82	2.82	14
3.61	3.61	8
12.26	12.26	12
7.15	7.15	11
2.65	2.65	12
3.71	3.71	10
2.43	2.43	11
4.11	4.11	14
3.02	3.02	12
6.06	6.06	10
4.89	4.89	11.9
4.81	4.81	10

Table 31: Make Keys samples for NewHope1024-CPA With a No Output OnClick Function

NewHope1024-CPA Results With a No Output OnClick Function		
Native Function Time(ms)	**OnClick Function Time(ms)**	**Highest CPU Usage(%)**
7.22	7.22	15
7.28	7.28	13
9.84	9.84	13
2.74	6.01	14
5.69	5.69	13
7.55	7.55	12
10.46	10.46	13
6.15	6.15	15
9.01	9.01	13
7.7	7.7	12
6.75	6.75	11
12.44	12.44	14
7.26	7.26	12
8.83	8.83	14
6.65	6.65	14
8.9	8.9	11
6.87	6.87	16
10.7	10.7	13
6.39	6.39	15
8.27	8.27	13
9.01	9.01	12
4.17	4.17	11
6.11	6.11	13
11.81	11.81	13
8.35	8.35	11
10.21	10.21	12.9
8.13	8.13	11
4.92	4.92	11
8.45	8.45	14
7.15	7.15	12
6.75	6.75	14
6.4	6.4	11.9
6.97	6.97	14
6.83	6.83	11
8.29	8.29	11
		Continued on next page

Table 31 – continued from previous page

Native Function Time(ms)	OnClick Function Time(ms)	Highest CPU Usage(%)
9.32	9.32	12
7.52	7.52	9
6.64	6.64	13
6.21	6.21	10
7.39	7.39	12
7.3	7.3	11
9.15	9.15	11
8.96	8.96	11
4.79	4.79	10
10.15	10.15	12
10.03	10.03	11
5.86	5.86	12
5.88	5.88	11.9
10.09	10.09	13.9
9.18	9.18	10

Appendix I. Additional NewHope Implementation Tables

All of the NewHope mobile implementations were tested using three different OnClick functions. The tables within this appendix display the average, minimum, and maximum for OnClick event runtime, native NewHope scheme runtime, and highest CPU usage.

First, the results from the verbose OnClick event are shown.

Table 32: NewHope512-CCA results from create keys test runs with a verbose OnClick function

NewHope512-CCA Create Key Results With Verbose OnClick			
	Average	**Minimum**	**Maximum**
Native Function Time(ms)	6.941	2.47	13.26
OnClick Function Time(ms)	89.034	68.01	111.53
Highest CPU Usage(%)	21.13	16.0	28.7

Table 33: NewHope1024-CCA results from create keys test runs with a verbose OnClick function

NewHope-1024-CCA Create Key Results With Verbose OnClick			
	Average	**Minimum**	**Maximum**
Native Function Time(ms)	11.691	7.63	18.98
OnClick Function Time(ms)	155.644	130.09	206.13
Highest CPU Usage(%)	25.35	18.0	34.9

Table 34: NewHope512-CPA results from create keys test runs with a verbose OnClick function

NewHope-512-CPA Create Key Results With Verbose OnClick			
	Average	**Minimum**	**Maximum**
Native Function Time(ms)	4.588	2.52	8.17
OnClick Function Time(ms)	70.447	58.27	109.53
Highest CPU Usage(%)	19.782	13.0	28.0

Next, the results from the succinct OnClick event are shown.

Finally, the results from the no output OnClick event are shown.

Table 35: NewHope1024-CPA results from create keys test runs with a verbose OnClick function

NewHope-1024-CPA Create Key Results With Verbose OnClick			
	Average	**Minimum**	**Maximum**
Native Function Time(ms)	8.267	4.82	12.97
OnClick Function Time(ms)	119.616	105.30	174.38
Highest CPU Usage(%)	23.42	17.0	27.9

Table 36: NewHope512-CCA results from Create Keys sample runs with A succinct OnClick function

NewHope512-CCA Create Key Results With Succinct OnClick			
	Average	**Minimum**	**Maximum**
Native Function Time(ms)	6.658	2.71	11.08
OnClick Function Time(ms)	13.653	8.23	20.74
Highest CPU Usage(%)	13.500	11.00	20.00

Table 37: NewHope1024-CCA results from Create Keys sample runs with a succinct OnClick function

NewHope1024-CCA Create Key Results With Succinct OnClick			
	Average	**Minimum**	**Maximum**
Native Function Time(ms)	11.900	3.87	20.13
OnClick Function Time(ms)	18.694	9.27	26.58
Highest CPU Usage(%)	14.054	11.00	20.90

Table 38: NewHope512-CPA results from Create Keys sample runs with a succinct OnClick function

NewHope512-CPA Create Key Results With Succinct OnClick			
	Average	**Minimum**	**Maximum**
Native Function Time(ms)	5.158	2.70	9.02
OnClick Function Time(ms)	11.899	7.88	22.55
Highest CPU Usage(%)	13.514	10.00	20.00

Table 39: NewHope1024-CPA results from Create Keys sample runs with a succinct OnClick function

NewHope1024-CPA Create Key Results With Succinct OnClick			
	Average	**Minimum**	**Maximum**
Native Function Time(ms)	7.816	4.24	12.18
OnClick Function Time(ms)	13.843	10.44	21.15
Highest CPU Usage(%)	13.396	9.00	18.00

Table 40: NewHope512-CCA results from create keys sample runs with a no output OnClick function

NewHope512-CCA Create Key Results With No Output OnClick			
	Average	**Minimum**	**Maximum**
Native Function Time(ms)	6.5146	1.34	11.31
OnClick Function Time(ms)	6.5388	1.34	11.31
Highest CPU Usage(%)	12.68	8.00	17.00

Table 41: NewHope1024-CCA results from create keys sample runs with a no output OnClick function

NewHope1024-CCA Create Key Results With No Output OnClick			
	Average	**Minimum**	**Maximum**
Native Function Time(ms)	12.2486	8.09	16.16
OnClick Function Time(ms)	12.2486	8.09	16.16
Highest CPU Usage(%)	12.82	10.00	20.00

Table 42: NewHope512-CPA results from create keys sample runs with a no output OnClick function

NewHope512-CPA Create Key Results With No Output OnClick			
	Average	**Minimum**	**Maximum**
Native Function Time(ms)	4.5224	2.43	12.26
OnClick Function Time(ms)	4.5502	2.43	12.26
Highest CPU Usage(%)	11.896	8.00	19.00

Table 43: NewHope1024-CPA results from create keys sample runs with a no output OnClick function

NewHope1024-CPA Create Key Results With No Output OnClick			
	Average	**Minimum**	**Maximum**
Native Function Time(ms)	7.7744	2.74	12.44
OnClick Function Time(ms)	7.8398	4.17	12.44
Highest CPU Usage(%)	12.372	9.00	16.00

Acronyms

ABI application binary interface. 71, 75

AES Advanced Encryption Standard. 13

APK Android Package. 57, 72, 73, 80, 81

ASIC application-specific integrated circuit. 21

BDD bounded distance decoding. 40

BKZ Block Korkine-Zolotarev. 35, 36, 38

CCA chosen ciphertext attack. 4, 18, 24, 40, 46, 47, 68, 89, 111

CMA chosen message attack. 18

CPA chosen plaintext attack. 4, 18, 24, 40, 46, 47, 68

CPU central processing unit. iv, 5, 109, 1

CVP closest vector problem. viii, 27, 28, 31, 32, 33, 38, 45, 57

DHKE Diffie-Hellman key exchange. 2, 13, 20, 26, 40

DLP data loss prevention. 49

DSA digital signature algorithm. 13

DSVP decision shortest vector problem. 28

ECC elliptic-curve cryptography. 2, 13, 20

ECDH Elliptic-Curve Diffie-Hellman. 13, 21, 40, 107

ECDSA elliptic-curve digital signature algorithm. 13

FHE fully-homomorphic encryption. 41, 42

FPGA field-programmable gate array. 21

HLS high-level synthesis. 21

ICS industrial control systems. 1

IoT internet of things. 1, 8, 19, 20, 21, 22, 61

IRSD Ideal Rank Syndrome Decoding. 23, 24

JNI Java Native Interface. 3, 5, 8, 57, 58, 59, 60, 62, 63, 71, 72, 75, 78, 83, 84, 86, 87, 88, 93, 100, 101, 102, 109, 110, 111, 112, 113

JVM Java Virtual Machine. 58

KAT known answer tests. 5, 16, 24, 46, 66, 67, 68, 75, 76, 78, 79, 85, 87, 132

KEM key encapsulation mechanism. iv, v, ix, 2, 3, 4, 5, 6, 8, 18, 21, 23, 24, 26, 40, 46, 47, 48, 61, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 76, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 90, 97, 98, 99, 100, 105, 106, 107, 109, 110, 111, 112, 113, 1

LPWN learning parity with noise. 32

LWE learning with errors. 28, 32, 33, 34, 38, 39, 40, 43, 45, 47, 65, 66

MB Megabytes. 95

MLWE Module Learning With Errors. 23

MLWR Module Learning With Rounding. 23, 24

ms milliseconds. 88, 107

NDK Native Development Kit. 59, 71, 72, 73, 74

NIST National Institute of Standards and Technology. iv, ix, 1, 2, 3, 4, 5, 14, 47, 63, 65, 66, 72, 111, 113, 1

NTT Number Theoretic Transform. 34, 43, 46, 52, 55, 66, 68

PKE public-key encryption. 2, 13, 16, 18, 42, 47, 49, 65, 68, 79

PQC post-quantum cryptography. iv, v, ix, 1, 2, 3, 4, 5, 6, 8, 12, 14, 15, 16, 19, 20, 21, 22, 23, 24, 26, 40, 42, 46, 60, 61, 63, 64, 65, 71, 72, 84, 86, 87, 105, 106, 107, 109, 110, 111, 112, 113, 1

PRNG pseudorandom generator. 49

PUF physically unclonable functions. 22

QSR quantum search reduction. 36

RAM random access memory. 20

RLWE ring learning with errors. 4, 8, 20, 27, 28, 33, 34, 38, 39, 40, 42, 43, 45, 46, 52, 53, 57, 61, 65, 66, 68, 105, 111

RLWR Ring Learning With Rounding. 23

ROM random oracle model. 49

RSA Rivest-Shamir-Adleman. 2, 13, 26

RSR random sampling reduction. 35, 36

SCA side-channel attack. 16, 18, 24, 39, 43, 44, 45

SIS shortest integer problem. 28

SPA simple power analysis. 43

SVP shortest vector problem. viii, 28, 30, 31, 32, 37, 40, 45

TLS Transport Layer Security. 20, 40

USVP unique shortest vector problem. 28, 40

VM Virtual Machine. 58, 59

XOF extendable-output functions. 49

Bibliography

1. *A faster lattice reduction method using quantum search, author=Ludwig, Christoph*, in International Symposium on Algorithms and Computation, Springer, 2003, pp. 199–208.

2. D. Adrian, K. Bhargavan, Z. Durumeric, P. Gaudry, M. Green, J. A. Halderman, N. Heninger, D. Springall, E. Thomé, L. Valenta, et al., *Imperfect forward secrecy: How Diffie-Hellman fails in practice*, in Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, ACM, 2015, pp. 5–17.

3. S. Akleylek, N. Bindel, J. Buchmann, J. Krämer, and G. A. Marson, *An efficient lattice-based signature scheme with provably secure instantiation*, in International Conference on Cryptology in Africa, Springer, 2016, pp. 44–60.

4. S. H. Al-Bakri, M. Mat Kiah, A. Zaidan, B. Zaidan, and G. M. Alam, *Securing peer-to-peer mobile communications using public key cryptography: New security strategy*, International Journal of the Physical Sciences, 6 (2011), pp. 930–938.

5. G. Alagic, G. Alagic, J. Alperin-Sheriff, D. Apon, D. Cooper, Q. Dang, Y.-K. Liu, C. Miller, D. Moody, R. Peralta, et al., *Status report on the first round of the NIST post-quantum cryptography standardization process*, US Department of Commerce, National Institute of Standards and Technology, 2019.

6. M. Albrecht, C. Cid, K. Paterson, C. Tjhai, and M. Tomlinson, *NTS-KEM, 2019*, `https://nts-kem.io`.

7. E. Alkim, R. Avanzi, J. Bos, L. Ducas, A. de la Piedra, T. Pöppelmann, P. Schwabe, and D. Stebila, *Newhope-algorithm Specifications and Supporting Documentation*, Second Round NIST PQC Project Submission Document, (2019).

8. E. Alkim, R. Avanzi, J. Bos, L. Ducas, A. de la Piedra, T. Poppelmann, P. Schwabe, D. Stebila, M. R. Albrecht, E. Orsini, et al., *NewHope: algorithm specifications and supporting documentation*, `https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions`, (2018).

9. E. Alkim, J. W. Bos, L. Ducas, P. Longa, I. Mironov, M. Naehrig, V. Nikolaenko, C. Peikert, A. Raghunathan, D. Stebila, et al., *FrodoKEM*, 2019.

10. E. Alkim, L. Ducas, T. Pöppelmann, and P. Schwabe, *Post-quantum key exchange - a new hope*, in 25th USENIX Security Symposium USENIX Security 16), 2016, pp. 327–343.

11. Android, *Getting Started with the NDK: Android NDK: Android Developers.* `https://developer.android.com/ndk/guides`, 2020.

12. B. Applebaum, *Cryptographic hardness of random local functions*, Computational complexity, 25 (2016), pp. 667–722.

13. N. Aragon, P. S. Barreto, S. Bettaieb, F. Worldline, L. Bidoux, O. Blazy, P. Gaborit, T. Güneysu, C. A. Melchor, R. Misoczki, et al., *Bike: Bit Flipping Key Encapsulation Round 2 Submission.*

14. R. Avanzi, J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehlé, *CRYSTALS-*

Kyber Algorithm Specifications And Supporting Documentation, Submission to the NIST post-quantum project, 9 (2017), p. 11.

15. H. Baan, S. Bhattacharya, S. Fluhrer, O. Garcia-Morchon, T. Laarhoven, R. Player, R. Rietman, M.-J. O. Saarinen, L. Tolhuizen, J. L. Torre-Arce, et al., *Round5: KEM and PKE based on (ring) learning with rounding*, Round5 submission to NIST PQC standardization (Second Round), (2019).

16. L. Babai, *On Lovász lattice reduction and the nearest lattice point problem*, Combinatorica, 6 (1986), pp. 1–13.

17. W. Backes and S. Wetzel, *A parallel LLL using POSIX threads*, Department of computer science, Stevens Institute of Technology, (2008).

18. M. Baldi, A. Barenghi, F. Chiaraluce, G. Pelosi, and P. Santini, *LEDAcrypt: Low-density parity-check code-based cryptographic systems*, NIST round, 2 (2019).

19. K. Basu, D. Soni, M. Nabeel, and R. Karri, *NIST Post-Quantum Cryptography-A Hardware Evaluation Study*.

20. D. J. Bernstein, *Introduction to post-quantum cryptography*, in Post-quantum cryptography, Springer, 2009, pp. 1–14.

21. D. J. Bernstein, T. Chou, C. Chuengsatiansup, A. Hülsing, E. Lambooij, T. Lange, R. Niederhagen, and C. Van Vredendaal, *How to Manipulate Curve Standards: A White Paper for the Black Hat*, in International Conference on Research in Security Standardisation, Springer, 2015, pp. 109–139.

22. D. J. Bernstein, T. Chou, T. Lange, R. Misoczki, R. Niederhagen, E. Persichetti, P. Schwabe, J. Szefer, and W. Wang, *Classic McEliece: conservative code-based cryptography*, (2019).

23. D. J. Bernstein, C. Chuengsatiansup, T. Lange, and C. van Vredendaal, *NTRU Prime: reducing attack surface at low cost*, in International Conference on Selected Areas in Cryptography, Springer, 2017, pp. 235–260.

24. D. J. Bernstein, T. Lange, and C. van Vredendaal, *NTRU Prime*.

25. D. J. Bernstein, T. Lange, and C. van Vredendaal, *NTRU Prime: round 2*, (2019).

26. G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, *Keccak*, in Advances in Cryptology – EUROCRYPT 2013, T. Johansson and P. Q. Nguyen, eds., Berlin, Heidelberg, 2013, Springer Berlin Heidelberg, pp. 313–314.

27. J. Bi, X. Meng, and L. Han, *Cryptanalysis of two knapsack-type public-key cryptosystems*, in 2010 International Conference on Computer Application and System Modeling (ICCASM 2010), vol. 9, IEEE, 2010, pp. V9–623.

28. J.-F. Biasse and F. Song, *Efficient quantum algorithms for computing class groups and solving the principal ideal problem in arbitrary degree number fields*, in Proceedings of the twenty-seventh annual ACM-SIAM symposium on Discrete algorithms, Society for Industrial and Applied Mathematics, 2016, pp. 893–902.

29. N. Bindel, J. Buchmann, and J. Krämer, *Lattice-based signature schemes and their sensitivity to fault attacks*, in 2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC), IEEE, 2016, pp. 63–77.

30. J. W. Bos, C. Costello, M. Naehrig, and D. Stebila, *Post-quantum key exchange for the TLS protocol from the ring learning with errors problem*, in 2015 IEEE Symposium on Security and Privacy, IEEE, 2015, pp. 553–570.

31. J. Buchmann, F. Göpfert, T. Güneysu, T. Oder, and T. Pöppelmann, *High-performance and lightweight lattice-based public-key encryption*, in Proceedings of the 2nd ACM International Workshop on IoT Privacy, Trust, and Security, ACM, 2016, pp. 2–9.

32. M. Campagna, C. Costello, B. Hess, A. Jalali, B. Koziel, B. LaMacchia, P. Longa, M. Naehrig, J. Renes, D. Urbanik, et al., *Supersingular Isogeny Key Encapsulation*, (2019).

33. P. Campbell, M. Groves, and D. Shepherd, *Soliloquy: A cautionary tale*, in ETSI 2nd Quantum-Safe Crypto Workshop, vol. 3, 2014, pp. 1–9.

34. L. Chen, S. Jordan, Y.-K. Liu, D. Moody, R. Peralta, R. Perlner, and D. Smith-Tone, *Report on post-quantum cryptography*, US Department of Commerce, National Institute of Standards and Technology, 2016.

35. C. Cheng, R. Lu, A. Petzoldt, and T. Takagi, *Securing the Internet of Things in a quantum world*, IEEE Communications Magazine, 55 (2017), pp. 116–120.

36. R. Cramer, L. Ducas, C. Peikert, and O. Regev, *Recovering short generators of principal ideals in cyclotomic rings*, in Annual International Conference on the Theory and Applications of Cryptographic Techniques, Springer, 2016, pp. 559–585.

37. M. H. Devoret and R. J. Schoelkopf, *Superconducting circuits for quantum information: an outlook*, Science, 339 (2013), pp. 1169–1174.

38. D. P. DiVincenzo, *The physical implementation of quantum computation*, Fortschritte der Physik: Progress of Physics, 48 (2000), pp. 771–783.

39. L. Ducas, A. Durmus, T. Lepoint, and V. Lyubashevsky, *Lattice signatures and bimodal Gaussians*, in Annual Cryptology Conference, Springer, 2013, pp. 40–56.

40. L. Ducas, V. Lyubashevsky, and T. Prest, *Efficient identity-based encryption over NTRU lattices*, in International Conference on the Theory and Application of Cryptology and Information Security, Springer, 2014, pp. 22–41.

41. M. Dyakonov, *When will useful quantum computers be constructed? Not in the foreseeable future, this physicist argues. Here's why: The case against: Quantum computing*, IEEE Spectrum, 56 (2019), pp. 24–29.

42. C. Easttom, *An Analysis of Leading Lattice-Based Asymmetric Cryptographic Primitives*, in 2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC), IEEE, 2019, pp. 0811–0818.

43. T. Espitau, P.-A. Fouque, B. Gerard, and M. Tibouchi, *Loop-abort faults on lattice-based signature schemes and key exchange protocols*, IEEE Transactions on Computers, 67 (2018), pp. 1535–1549.

44. U. Fincke and M. Pohst, *Improved methods for calculating vectors of short length in a lattice, including a complexity analysis*, Mathematics of computation, 44 (1985), pp. 463–471.

45. S. R. Fluhrer, *Cryptanalysis of ring-LWE based key exchange with key share reuse*, IACR Cryptology ePrint Archive, 2016 (2016), p. 85.

46. S. D. Galbraith, *Mathematics of public key cryptography*, Cambridge University Press, 2012.

47. S. D. Galbraith, *Space-efficient variants of cryptosystems based on learning with errors*, https://www.math.auckland.ac.nz/~sgal018/compact-LWE.pdf, (2013).

48. C. Gentry and S. Halevi, *Implementing gentryâĂŹs fully-homomorphic encryption scheme*, in Annual international conference on the theory and applications of cryptographic techniques, Springer, 2011, pp. 129–148.

49. A. Ghasemmehdi and E. Agrell, *Faster recursions in sphere decoding*, IEEE Transactions on Information Theory, 57 (2011), pp. 3530–3536.

50. S. Ghosh, R. Misoczki, and M. R. Sastry, *Lightweight Post-Quantum-Secure Digital Signature Approach for IoT Motes*.

51. Gradle, Inc., *Community Home*. https://docs.gradle.org/current/userguide/what_is_gradle.html#five_things.

52. L. K. Grover, *A fast quantum mechanical algorithm for database search*, arXiv preprint quantum, (1996).

53. T. Güneysu, V. Lyubashevsky, and T. Pöppelmann, *Practical lattice-based cryptography: A signature scheme for embedded systems*, in International Workshop on Cryptographic Hardware and Embedded Systems, Springer, 2012, pp. 530–547.

54. M. Hamburg, *Post-quantum cryptography proposal: ThreeBears*, NIST Post-Quantum Cryptography Standardization, (2019).

55. A. W. Harrow and A. Montanaro, *Quantum computational supremacy*, Nature, 549 (2017), pp. 203–209.

56. J. Hoffstein, J. Pipher, J. M. Schanck, J. H. Silverman, and W. Whyte, *Practical signatures from the partial Fourier recovery problem*, in International Conference on Applied Cryptography and Network Security, Springer, 2014, pp. 476–493.

57. J. Hoffstein, J. Pipher, and J. H. Silverman, *NTRU: A ring-based public key cryptosystem*, in International Algorithmic Number Theory Symposium, Springer, 1998, pp. 267–288.

58. J. Howe, A. Khalid, M. Martinoli, F. Regazzoni, and E. Oswald, *Fault Attack Countermeasures for Error Samplers in Lattice-Based Cryptography*, in 2019 IEEE International Symposium on Circuits and Systems (ISCAS), IEEE, 2019, pp. 1–5.

59. M. Hung, *Leading the IoT, gartner insights on how to lead in a connected world*, Gartner Research, (2017), pp. 1–29.

60. IBM, *Three Types of Quantum Computing - IBM Infographic*. `https://www.flickr.com/photos/ibm_research_zurich/22963781794`, 2015.

61. J. Kelly, *A Preview of Bristlecone - Google's New Quantum Processor*. `https://ai.googleblog.com/2018/03/a-preview-of-bristlecone-googles-new.html`, 2019.

62. A. Khalid, C. Rafferty, J. Howe, S. Brannigan, W. Liu, and M. OâĂŹNeill, *Error Samplers for Lattice-Based Cryptography-Challenges, Vulnerabilities and Solutions*, in 2018 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS), IEEE, 2018, pp. 411–414.

63. Kitware, *Overview*. `https://cmake.org/overview/`.

64. P.-C. Kuo and C.-M. Cheng, *Lattice-based cryptanalysisâĂŤHow to estimate the security parameter of lattice-based cryptosystem*, in 2014 IEEE International Conference on Consumer Electronics-Taiwan, IEEE, 2014, pp. 53–54.

65. T. Laarhoven, M. Mosca, and J. Van De Pol, *Finding shortest lattice vectors faster using quantum search*, Designs, Codes and Cryptography, 77 (2015), pp. 375–400.

66. T. Laarhoven, J. van de Pol, and B. de Weger, *Solving Hard Lattice Problems and the Security of Lattice-Based Cryptosystems*, IACR Cryptology EPrint Archive, 2012 (2012), p. 533.

67. A. K. Lenstra, *Lattices and factorization of polynomials over algebraic number fields*, in European Computer Algebra Conference, Springer, 1982, pp. 32–39.

68. P. Longa and M. Naehrig, *Speeding up the number theoretic transform for faster ideal lattice-based cryptography*, in International Conference on Cryptology and Network Security, Springer, 2016, pp. 124–139.

69. X. Lu, Y. Liu, D. Jia, H. Xue, J. He, Z. Zhang, Z. Liu, H. Yang, B. Li, and K. Wang, *LAC: Lattice-based Cryptosystems*, URL: https://csrc. nist. gov/projects/post-quantum-cryptography/round-2-submissions. Citations in this document, 1.

70. V. Lyubashevsky, C. Peikert, and O. Regev, *On ideal lattices and learning with errors over rings*, in Annual International Conference on the Theory and Applications of Cryptographic Techniques, Springer, 2010, pp. 1–23.

71. V. Lyubashevsky, C. Peikert, and O. Regev, *A toolkit for ring-LWE cryptography*, in Annual International Conference on the Theory and Applications of Cryptographic Techniques, Springer, 2013, pp. 35–54.

72. A. T. M. Watari and S. Ano, *Documentation, Java Native Interface Specification.*

73. L. Malina, L. Popelova, P. Dzurenda, J. Hajny, and Z. Martinasek, *On Feasibility of Post-Quantum Cryptography on Small Devices*, IFAC-PapersOnLine, 51 (2018), pp. 462–467.

74. A. Mariano, T. Laarhoven, F. Correia, M. Rodrigues, and G. Falcao, *A practical view of the state-of-the-art of lattice-based cryptanalysis*, IEEE Access, 5 (2017), pp. 24184–24202.

75. C. A. Melchor, N. Aragon, S. Bettaieb, L. Bidoux, O. Blazy, A. Couvreur, J.-C. Deneuville, P. Gaborit, A. Hauteville, and G. Zémor, *Rank Quasi-Cyclic (RQC).*

76. C. A. Melchor, N. Aragon, S. Bettaieb, L. Bidoux, O. Blazy, J.-C. Deneuville, P. Gaborit, A. Hauteville, O. Ruatta, J.-P. Tillich, et al., *ROLLO-Rank-Ouroboros, LAKE LOCKER*, (2019).

77. D. Micciancio, *CSE206A: Lattices algorithms and applications (spring 2014), 2014*, `http://cseweb.ucsd.edu/classes/sp14/cse206A-a`.

78. A. W. Mohsen, A. M. Bahaa-Eldin, and M. A. Sobh, *Lattice-based cryptography*, in 2017 12th International Conference on Computer Engineering and Systems (ICCES), IEEE, 2017, pp. 462–467.

79. D. Moody, *Post-Quantum Cryptography Standardization: Announcement and outline of NIST's Call for Submissions*, 2016.

80. C. Nay, *IBM Opens Quantum Computation Center in New York; Brings World's Largest Fleet of Quantum Computing Systems Online, Unveils New*

53-Qubit Quantum System for Broad Use. https://newsroom.ibm.com/2019-09-18-IBM-Opens-Quantum-Computation-Center-in-New-York, 2019.

81. P. Q. Nguyen and J. Stern, *The two faces of lattices in cryptology*, in International Cryptography and Lattices Conference, Springer, 2001, pp. 146–180.

82. M. A. Nielson and I. L. Chuang, *Quantum computation and quantum information*, (2000).

83. NIST, *Call for Post-Quantum Standardization Project Algorithms.* https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf, 2019.

84. M. O'Neill et al., *Insecurity by design: Today's IoT device security problem*, Engineering, 2 (2016), pp. 48–49.

85. A. Park and D.-G. Han, *Chosen ciphertext simple power analysis on software 8-bit implementation of Ring-LWE encryption*, in 2016 IEEE Asian Hardware-Oriented Security and Trust (AsianHOST), IEEE, 2016, pp. 1–6.

86. T. Pathak, *Apple Maintains Lead in Premium Smartphone Segment, OnePlus Enters Top Five Brands for the First Time in 2018 - Counterpoint Research.* http://bit.ly/2VMSHQP, 2019.

87. C. Peikert, O. Regev, and N. Stephens-Davidowitz, *Pseudorandomness of ring-LWE for any ring and modulus*, in Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, ACM, 2017, pp. 461–473.

88. O. Regev, *Lattice-based cryptography*, in Annual International Cryptology Conference, Springer, 2006, pp. 131–141.

89. O. Regev, *On lattices, learning with errors, random linear codes, and cryptography*, Journal of the ACM (JACM), 56 (2009), p. 34.

90. E. Rescorla, *The Transport Layer Security (TLS) Protocol Version 1.3–draft-ietf-tls-tls13-05*, 2015.

91. S. K. Routray, M. K. Jha, L. Sharma, R. Nyamangoudar, A. Javali, and S. Sarkar, *Quantum cryptography for IoT: APerspective*, in 2017 International Conference on IoT and Application (ICIOT), IEEE, 2017, pp. 1–4.

92. S. S. Roy, F. Vercauteren, N. Mentens, D. D. Chen, and I. Verbauwhede, *Compact ring-LWE cryptoprocessor*, in International Workshop on Cryptographic Hardware and Embedded Systems, Springer, 2014, pp. 371–391.

93. J. Sanders, *TechRepublic: D-Wave announces 5,000-qubit fifth generation quantum annealer*, 2019.

94. C.-P. Schnorr and M. Euchner, *Lattice basis reduction: Improved practical algorithms and solving subset sum problems*, Mathematical programming, 66 (1994), pp. 181–199.

95. B. Schumacher, *Sending entanglement through noisy quantum channels*, Physical Review A, 54 (1996), p. 2614.

96. N. Sha, *standard: Permutation-based hash and extendable-output functions, 2015*, DOI, 3AD, (3).

97. P. W. Shor, *Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer*, SIAM review, 41 (1999), pp. 303–332.

98. V. Singh and A. Chopra, *Even More Practical Key Exchanges for the Internet using Lattice Cryptography*, IACR Cryptology ePrint Archive, 2015 (2015), p. 1120.

99. R. Srivastava, I. Choi, T. Cook, and N. U. E. Team, *The commercial prospects for quantum computing*, Networked Quantum Information Technologies, (2016).

100. D. E. Standard, *National Institute of Standards and Technology, FIPS PUB 46-2 (December 1993)*, `http://www.itl.nist.gov/fipspubs/fip46-2.html`.

101. D. Stehlé and R. Steinfeld, *Making NTRU as secure as worst-case problems over ideal lattices*, in Annual International Conference on the Theory and Applications of Cryptographic Techniques, Springer, 2011, pp. 27–47.

102. T. Strahl and R. Johansson, *Post-quantum Secure Communication on a Low Performance IoT Platform*, (2016).

103. T. N. Tan and H. Lee, *High-Secure Fingerprint Authentication System using Ring-LWE Cryptography*, IEEE Access, 7 (2019), pp. 23379–23387.

104. K. und Computeralgebra, *The BKZ Simulation Algorithm*, (2013).

105. P. van Emde Boas, *Another NP-complete problem and the complexity of computing short vectors in a lattice*, Tecnical Report, Department of Mathmatics, University of Amsterdam, (1981).

106. I. F. Vercauteren, *SABER: Mod-LWR based KEM (Round 2 Submission)*.

107. B. Wang, H. Liu, and Y. Hu, *Cryptanalysis of a knapsack public key cryptosystem*, in 2009 Fifth International Conference on Information Assurance and Security, vol. 2, IEEE, 2009, pp. 49–52.

108. T. Wunderer, M. Burger, and G. N. Nguyen, *Parallelizing the Hybrid Lattice-Reduction and Meet-in-the-Middle Attack*, in 2018 IEEE International Conference on Computational Science and Engineering (CSE), IEEE, 2018, pp. 185–193.

109. J. Xu, L. Hu, S. Sun, and Y. Xie, *Cryptanalysis of countermeasures against multiple transmission attacks on NTRU*, IET Communications, 8 (2014), pp. 2142–2146.

110. S. Zhou, C. Yin, and H. Xu, *Research progress of lattice bases reduction algorithms*, in 2012 International Conference on Computer Science and Electronics Engineering, vol. 1, IEEE, 2012, pp. 550–553.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From — To)*
26–03–2020	Master's Thesis	Sept 2018 — Mar 2020

4. TITLE AND SUBTITLE

NewHope: A Mobile Implementation of a Post-Quantum Crytographic Key Encapsulation Mechanism

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Jessica A. Switzler GS-11

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way
WPAFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT-ENG-MS-20-M-063

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Intentionally Left Blank

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

DISTRIBUTION STATEMENT A:
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

NIST anticipates the appearance of large-scale quantum computers by 2036 [34], which will threaten widely used asymmetric algorithms, NIST launched a Post-Quantum Cryptography Standardization Project to find quantum-secure alternatives. NewHope PQC KEM is the only Round 2 candidate to simultaneously achieve small key values through the use of a security problem with sufficient confidence its security, while mitigating any known vulnerabilities.
This research contributes to NIST project's overall goal by assessing the platform flexibility and resource requirements of NewHope KEMs on an Android mobile device. The resource requirements analyzed are transmission size as well as scheme runtime, CPU, memory, and energy usage. Results from each NewHope KEM instantiations are compared amongst each other, to a baseline application, and to results from previous work. NewHope PQC KEM was demonstrated to have sufficient flexibility for mobile implementation, competitive performance with other PQC KEMs, and to have competitive scheme runtime with current key exchange algorithms.

15. SUBJECT TERMS

post-quantum cryptographic key encapsulation mechanism, mobile device security, lattice-based cryptography, asymmetric cryptography

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Dr. Laurence D. Merkle, AFIT/ENG
U	U	U	UU	197	19b. TELEPHONE NUMBER *(include area code)* (937)255-3636 x4526; laurence.merkle@afit.edu