Measuring the Performance of Post-Quantum Cryptography on Embedded Systems

A Major Qualifying Project Report
Submitted to the faculty of
WORCESTER POLYTECHNIC INSTITUTE
In partial fulfillment of the requirements for the degree of Bachelor of Science

By:
Elçin Önder
A. Ungerer

Date:
March 18, 2021

WPI Faculty Advisor:
Professor Yarkin Doroz

Abstract

Post-quantum cryptography is becoming more important as quantum computers get closer to becoming part of our daily lives. Understanding this need, the NIST organized a competition-based system in 2019 to find public key encryption and digital signature algorithms that could be used in future standardizations. Currently the competition is in the third round. Our project will consist of using and modifying the finalist's code to compare the timing between Linux and the Raspberry Pi 4 Model B.

Table of Contents

Table of Figures

Table of Tables

1. Introduction

Everyday there are many cyber attack attempts to access private information of individuals and organizations. We use cryptography to protect that information by embedding algorithms in our code to prevent attackers from reaching the intended data [1]. Post-quantum cryptography is an extension of that. It uses principles from quantum mechanics in order to encrypt data and transmit it securely so that it cannot be hacked [2], and is centered around algorithms that are designed to secure data in a quantum computer [3]. As post-quantum cryptography is being developed it is intended to become "cryptographic systems that are secure against both quantum and classical computers, and can interoperate with existing communications protocols and networks" [4]. All over the world, researchers are rushing to develop algorithms that could protect the public-key infrastructure under post-quantum cryptography.

With the increase of research done on quantum computers, the closer society gets to implementing it into our everyday lives. However, with this development, our security within and between devices that will need modifications to accommodate for the differing cybersecurity techniques that quantum and non-quantum computers use. Cybersecurity typically uses number theory-base cryptography to help us send and share information. With quantum computing, that concept will no longer work since the computers will be able to solve the basic number theory, thus potentially breaching privacy. Luckily, studies have shown that matrix manipulation cryptography is not able to be deciphered by quantum computers. [5]

The National Institute of Standards and Technology (NIST) understands that this could be an issue and is currently working on making quantum-safe cryptography more widely used by the industry and government. It is devoting its work to standardizing public-key encryption and key-establishment algorithms. Knowing that it took decades to implement current standards, there is no time to wait as the newer technology is steadily approaching [6]. Therefore, they have organized a competition-based system in 2019 to determine post-quantum cryptography methods that could be used in future standardizations and are compatible with both classical and quantum computers [7]. With this, the organization hopes that in a couple years, there will be some submissions that are usable to standardize in the industry, and that it will help promote safety and security in technology.

Our project will use ‘Encryption Algorithms’ and ‘Digital Signature Algorithms’ submissions that were named finalists and alternatives in the competition. Both of these are Public-key algorithms which is a cryptographic system that uses two keys: a public key and a private key. Public key is known to everyone and a private key which is known only to the recipient of the message. Encryption is a method in cryptography by which cryptographic keys, a piece of information, are exchanged between two parties [8], [9]. Digital signatures are the public-key primitives of message authentication using signatures to authenticate the message. It is a cryptographic value that is calculated from the data and a secret key known only by the signer. [10]

We will be testing the finalists’ algorithms submitted to the NIST competition through a Linux remote machine, then document if there are any additional libraries needed, their security levels and validate that the signatures are being properly read. Afterwards, we will compare embedded systems to find one that will be compatible with the programs. Finally, the programs will go through the chosen embedded system, which in our case will be the Raspberry Pi 4 Model B. Here, we will document timing for the encryption and digital signature algorithms, and record any necessary libraries or errors found. If necessary, we will have to restructure some of the existing code to allow for processing through the embedded system. At the end, the goal is to learn how the latest versions of post-quantum cryptography works on current embedded systems.

2. Background Information

2.1 Cryptography

Cryptography has roots throughout history where people used variations of it to encode and decode messages. A notable time it was used was during World War II when Alan Turning, an English mathematician, played a major role in speeding up the process of deciphering German communications. These days, cryptography still plays a major role in the way we communicate with others and is integrated in our technology in order to deliver a message from one device to another safely and privately.

We can define cryptography as a science of concealing communication to hide the meaning [11]. It is based on mathematical algorithms through the notion that they are complicated and hard to decipher by others [12]. By using cryptography in our day-to-day messaging, it can give us the security to do our daily tasks whether it be sending money for rent or having personal conversations with our friends. Cryptography is the basis of how we keep our communications safe and is the foundation of cybersecurity. [5] For our project, we will focus on looking into Public-Key Cryptography, specifically into Encryption and Digital Signature Algorithms.

2.1.1. Public Key Cryptography

Public-key cryptography, also known as asymmetrical cryptography is a newer developed cryptography that encompasses creating secret keys that can decode encoded messages. Mentioned previously, the cryptography that has been with us throughout history is symmetrical, which means that it uses the same key for encryption and decryption, but that can pose issues if the key itself is not secure, thus others can interfere, and simply the number of keys needed. To ease this situation, asymmetrical cryptography was created that first prioritizes creating a safe key on one end, then the receiver has a matching key. We can show RSA encryption as a public-key encryption example. RSA algorithm is used for digital signatures and public encryption. It is a simple algorithm based on modular exponentiation of large integers. The idea of RSA is straightforward.

Set-up Stage

1. Choose two large primes p and q.
2. Compute $n = p \cdot q$.
3. Compute $\Phi(n) = (p-1)(q-1)$.
4. Choose random b; $0 < b < \Phi(n)$, with $\gcd(b, \Phi(n)) = 1$.
   Note that b has inverse in $Z_{\Phi(n)}$.
5. Compute inverse $a = b^{-1} \bmod \Phi(n)$:
   $$b \cdot a \equiv 1 \bmod \Phi(n).$$
6. Public key: $k_{pub} = (n, b)$.
   Private key: $k_{pr} = (p, q, a)$.

Figure 1. Setting Up RSA [13]

First, we choose two large prime numbers *p* and *q* that are bigger than 2^512. For step 4, we choose a random number called *b* that is between 0 and the number we computed in the third step. Later, we take inverse mod to find which is one of the inputs of the private key. Last step shows us to find the key generation for public key and private key.

RSA's encryption is done by using a public key which is shown as kpub in Figure 2.

$$y = e_{k_{pub}}(x) = x^b \bmod n.$$
$$x \in Z_n = \{0, 1, \ldots, n-1\}.$$

Figure 2. RSA Encryption with Public Key [13]

The encrypted message is shown as x in Figure 2. y equation is to calculate the encrypted message. After finding y, we can find the decrypted message by using the equation shown in Figure 3 below.

$$x = d_{k_{pr}}(y) = y^a \bmod n.$$

Figure 3. RSA Decryption [13]

In the programs we will be using in our project, each one creates a key and uses algorithms to decode a message. This method has improved the way communications are delivered and the added algorithm in the mix makes it more complicated for other parties to access information. The programs we will use for this project involve Encryption and Digital Signature Algorithms. [8]

- **Encryption Algorithms:** This type of algorithm involves creating a public key that anyone can use to encrypt and a private key that can only be used between the parties. The private key is transferred to the other party using a one-way algorithm. The message is protected because the corresponding private key is needed to decrypt the message [9]
- **Digital Signature Algorithms:** This algorithm provides a different level of security that protects parties from each other. This is done by generating a signature on the end of a message. The signature is a means of authentication and is used to prove who wrote the message. It will need to be verified. This asymmetric algorithm is able to distinguish the actions of different party members, thus ensuring the authenticity of the message. [10]

2.2 Quantum Computers

Although a fully-functioning quantum computer does not exist yet, researchers are still looking into ways to advance that technology of a quantum computer and are coming up with safety measures for its debut into our daily lives [14].

Fundamentally, a quantum computer should be able to take multiple inputs at once and process them simultaneously while factoring in the other inputs, similar to how we process many different functions at once in our brains. This large amount of processing is done on a singular computer system. This makes quantum computing vastly different from classical computing. In classic computing, computers are only able to use one decision at a time in a logical and series

method [15]. The future of quantum computers suggest that the development could lead to major improvement in many areas including health and safety for it's capability to be used with Artificial Intelligence. [14]

The basis of quantum computing comes from quantum theory. Quantum theory is still in development, but summarily, it consists of looking into a quantum world that contrasts the physical world and science we see everyday. In the quantum world, there are three distinct properties:

- *Wave Particle Duality*: Where waves and particles would behave as the other would in physical science.
- *Quantum Tunneling*: The idea that matter can move from one point to another without the use of space.
- *Quantum Teleportation*: The idea that information can move from one point to another without space.

What quantum theory intends to explain is how matter and energy interact in the quantum world through mathematics, and is still being developed today. [16]

2.3 Problem: Classic Cryptography

Quantum computers work significantly differently than classic computers. As a result, updates to our cybersecurity are necessary to make sure that it is able to protect us from outside parties that try to intercept our information. But what makes our current system unusable? Let's discuss it.

Classic computers are protected by classic cryptography, also known as modern cryptography. The foundations in classic cryptography lies in number theory, which can include abstract algebra, elementary number theory, discrete logarithms and elliptic curve discrete logarithms to name a few [17]. To simplify it, modern algorithms will integrate number theory in order to make it challenging for attackers to solve. For example, a program can have it's security dependent on the ability to factor large integers (think about 2048-bit numbers) [5]. The system relies on the fact that there are no algorithms that can beat it, and aims to make it not only challenging, but infeasible to solve [18]. However, that may not always be the case and cannot be guaranteed [19].

We can see this particularly in the way classic cryptography interacts with quantum computers. Because quantum computers are able to take in many inputs while factoring in the others inputted, they are able to solve and break classical cryptography with ease. Knowing that classic cryptography took years to develop, it is clear that there is a potential for a security issue as quantum technology becomes more well-known and eventually integrated into our lives.

Next, is the comparison of the two proposed solutions for protecting quantum computers.

2.4 Solution: Quantum vs Post-Quantum Cryptography

With the increasing need for cybersecurity protections, two different solutions were found: quantum cryptography and post-quantum cryptography. In this portion, we will briefly discuss both of them and how each impacts quantum computing, then why post-quantum cryptography is more effective. Each statement is brief because there is still much research needed on both options as quantum computers are developed into a fully functioning machine.

Quantum cryptography uses a quantum channel in order to send quantum bits, the storage for communication, from one location to another. This process is ideal for short distances, but in long-distance communications, facilities such as quantum satellites and repeaters will be needed to allow for communication all over the world. [5]

On the other hand, post-quantum cryptography uses ciphers and matrix mathematics. Matrix mathematics is believed to be protected from quantum attacks in comparison to classically used methods (e.g. discrete mathematics and factoring) which quantum computers are able to decipher. Matrix manipulation provides more dimensions that will make it difficult for a quantum attack to occur despite the multiple processes it uses. [5], [12]

Although both options are useful to have, post-quantum cryptography provides for less facilities built and is able to reach long distances easily, albeit computation assumptions are necessary to get it working; therefore, it was decided by the National Institute of Standards and Technology (NIST) that there needs to be a process to standardize post-quantum cryptography so when the devices come out, we are prepared to handle quantum attacks.

2.5 NIST Safety Levels and Standards

To ensure safety of cryptography, the NIST has developed standards with increasing security levels that are based on key sizes, or Advanced Encryption Standard (AES) and has

function sizes or, Secure Hash Algorithm (SHA) methods [20]. They work as the following in the Table 1 below. This will be relevant to the next chapter where security standards will still need to be upheld in the post-quantum cryptography competition.

Level	Size	Comparable to
1	128-bit key	AES128
2	256-bit hash function	SHA256/SHA3-256
3	192-bit key	AES192
4	384-bit hash function	SHA384/SHA3-384
5	256-bit key	AES256

Table 1. NIST Standards for Security Levels, Size and Comparisons [21]

2.6 NIST Competition/Competitors

NIST requests algorithms in three categories: public-key encryption, key exchange or key encapsulation mechanisms (KEMs), and digital signatures. As mentioned before, the NIST believed that it was important to standardize post-quantum cryptography to be used in the public and has started implementing that process through a competition-based submission. The first set of criteria was the following:

- Options for reference and optimized C code.
- Known answer tests for encoding and decoding each algorithm.
- Written reports that describe the specifications of each algorithm.
- Intellectual property statements to allow for use. [7]

The competition is held in rounds, and currently the competition is in its third phase. The first phase included 82 package submissions that were put under minimal criteria for code implementation, known-answer test, specification document and intellectual property declarations. [22]

The next evaluation criteria used included “security”, “cost and performance”, and “algorithm and implementation characteristics.” Security involved providing proofs of security efforts, and estimating the security strengths. Cost and performance used the size of keys, text and signatures, how effective the key generation was in the private and public end and the

probability of failures. Finally, algorithm and implementation characteristics were considered that favored simpler designs that were multifaceted and could encompass multiple platforms. [7]

Using these criteria, in the second round, 26 algorithms were selected and the breakdown of these were 17 Public Key Encryption and 9 Digital Signatures. Currently, at the end of the third round, out of the 26 algorithms, 15 have advanced. Within the finalists, there are four Public-Key Encryption algorithms (Classic McEliece, CRYSTALS-Kyber, NTRU and SABER), and three Digital Signature algorithms (CRYSTALS-Dilithium, Rainbow and Falcon). For our project, we will be attempting to use the algorithms that have passed as the finalists to take timing measurements. [22]

2.6.1 Classic McEliece

Classic McEliece uses a One Way against Chosen Plaintext Attacks (OW-CPA) public key encryption (PKE) within a KEM to create a IND-CCA2 security level. Indistinguishability against chosen-ciphertext attacks (IND-CCA) secure KEM is a concept where the attacker would get a decryption oracle and be able to ask for as many decryptions as possible up to a challenge ciphertext [23]. The IND-CCA2 security level is the adaptive version, where it will allow queries even after the challenge ciphertext is given assuming it won't pass [23], [24].

The algorithm creates a keypair by generating a uniform random monic irreducible polynomial and selecting a uniform random sequence to compute a matrix. That matrix has the Gaussian elimination applied to generate a random s-bit string which is used as part of the private and public key. After, encapsulation is done by computation using the public key and encoding subroutine on a uniform random vector. This process is undone for decapsulation. [25]

Classic McEliece contains five versions: mceliece348864, mceliece460896, mceliece6688128, mceliece6960119, and mceliece8192128. The breakdown of the levels and parameter sizes are shown below in Table 2.

	NIST Security Level	Public Key (pk) size (bytes)	Secret Key (sk) size (bytes)	Ciphertext (ct) size (bytes)
mceliece348864	1	261120	6492	128
mceliece460896	3	524160	13608	188
mceliece6688128	5	1044992	13931	240
mceliece6960119	5	1047319	13948	226
mceliece8192128	5	1357824	14120	240

Table 2. Classic McEliece: Security Levels and Parameter Sizes [25]–[27]

2.6.2 CRYSTALS-Kyber

"Cryptographic Suite for Algebraic Lattices" (CRYSTALS) encompasses two cryptographic primitives: Kyber, and Dilithium. The Kyber algorithm consists of two parts: “an IND-CPA-secure public-key encryption scheme encrypting messages of a fixed length of 32 bytes” [23] and “a slightly tweaked Fujisaki–Okamoto (FO) transform to construct the IND-CCA2-secure KEM” [23].

The first part of this algorithm, the IND-CPA-secure public key, works where the algorithm is able to interact with any outside attacks, but the message received will be encoded with a random key every time so there is no distinction which key will decrypt the message [24]. In this case, the CRYSTALS-Kyber will use learning with errors (LWE) encryption scheme. Their scheme is similar to learning with errors over rings (R-LWE) where an attacker would look for the correct ring out of many random noisy rings that are reinforced with error factors, but instead they use the same ring with varying dimensions of ring module making it learning with errors over modules (MLWE) [23], [28].

The second part of the algorithm has IND-CCA2 security implemented through a variation of a FO transform, which combines a one-way secure asymmetric encryption scheme and a one-time secure encryption scheme [29], [30]. The variation includes hashing the public key into the random generator seed and shared key, ciphertext into the shared key, and use of Keccak-based functions [31].

CRYSTALS-Kyber has three different parameter sets: KYBER512, KYBER768 and KYBER1024. Specifications are shown in Table 3.

	NIST Security Level	Public Key (pk) size (bytes)	Secret Key (sk) size (bytes)	Ciphertext (ct) size (bytes)
KYBER512	2	736	1632	800
KYBER768	3	1088	2400	1152
KYBER1024	4	1440	3168	1540

Table 3. CRYSTALS-Kyber: Security Levels and Parameter Sizes [23]

2.6.3 NTRU

N-th degree Truncated polynomial Ring Units (NTRU) is an over 20 year old encryption scheme that was implemented for post-quantum cryptography to achieve IND-CCA2 security [32]. The NTRU algorithm originally uses polynomial algebra and reduction modulo for encryption and elementary probability theory for decryption [33]. With the original design, it is a partially correct probabilistic public key encryption scheme (partially correct PPKE), but can be made into a correct deterministic public key encryption scheme (DPKE) using transformations since it is lattice based [32], [34].

NTRU's NIST submission included four sets of parameters: ntruhps2048509, ntruhrss701, ntruhps2048677 and ntruhps4096821 which are shown in Table 4. Looking at each of the names, we can see there are two versions of NTRU: hps and hrss where hps uses fixed-weight sample spaces and hrss uses arbitrary weight [33]–[35].

	NIST Security Level	Public Key (pk) size (bytes)	Secret Key (sk) size (bytes)	Ciphertext (ct) size (bytes)
ntruhps2048509	1	699	935	699
ntruhrss701	3	1138	1450	1138
ntruhps2048677	3	930	1234	930
ntru4096821	5	1230	1590	1230

Table 4. NTRU: Security Levels and Parameter Sizes [32]

2.6.4 SABER

SABER is another lattice-based algorithm that was submitted to NIST with IND-CCA2 security [36]. The algorithm consists of Saber.PKE and Saber.KEM. Saber.PKE is the public-key encryption algorithm that uses Module Learning with Rounding (MLWR) that is a variation of LWE [28], [36]. MLWR works differently from LWE where the program rounds the samples to create "noise" instead of adding error to them [28], [37]. However, the developers of SABER mentioned that Saber.PKE alone cannot fight off chosen-ciphertext attacks, so "Saber.PKE can be compiled into Saber.KEM using a post-quantum variant of the Fujisaki-Okamoto (FO) transformation" [36]. [30]

There are three levels of security, LightSABER, SABER and FireSABER which are described in Table 5.

	NIST Security Level	Public Key (pk) size (bytes)	Secret Key (sk) size (bytes)	Ciphertext (ct) size (bytes)
LightSABER	1	672	1568	736
SABER	3	992	2304	1088
FireSABER	5	1312	3040	1472

Table 5. SABER: Security Levels and Parameter Sizes [38]

2.6.5 CRYSTALS-Dilithium

CRYSTALS-Dilithium utilizes the "Fiat-Shamir with Aborts" approach that emphasizes security that is based on difficulty in finding short vectors in lattices. In the key generation portion, the algorithm consists of creating a matrix of polynomials, then sampling random secret key vectors and creating a public key from that. All of the mathematics used for this also uses a ring. To create the signature, the algorithm uses a vector of polynomials with coefficients so that the key could not be revealed and forged. For further protection, rejection sampling is used where "given the ability to sample according to some probability distribution *P*, one is asked to produce samples from some other distribution *S*" [39]. Finally for the verification, the high order bits are computed and accepted if the coefficients and hash is valid. [22], [40], [41]

CRYSTALS-Dilithium provides four sets of parameters shown in Table 6. They provide Strong Unforgeability under Chosen Message Attack (SUF-CMA) security. For UF-CMA

security, “the adversary gets the public key and has access to a signing oracle to sign messages of his choice. The adversary’s goal is to come up with a valid signature of a new message” [40]. SUF-CMA follows the same principle, but includes a false win for the adversary. [40]

	NIST Security Level	Public Key (pk) size (bytes)	Signature (sig) size (bytes)
DILITHIUM2	1	1184	2044
DILITHIUM3	2	1472	2701
DILITHIUM4	3	1760	3366

Table 6. Crystals-DILITHIUM: Security Levels and Parameter Sizes [40]

2.6.6 Rainbow

Rainbow is a multivariate public key cryptosystem (MPKC) which uses polynomials that are both non-linear and multivariate over an infinite field to generate a key for a message. Rainbow’s security is dependent on the difficulty to solve those polynomials. The signature is generated from hash functions and then computed by inverting the central map with choosing random values, Gaussian Elimination on polynomials and substitution of the random values. The signature is then verified using the hash function to find the value, then comparing to see if accepted. [42]

Rainbow is based on Existential Unforgeability under Chosen Message Attack (EUF-CMA) that upgrades UF-CMA security by applying a transform that uses a salt to generate the signature [42]. For the case of this project, we will only be working with Rainbow Classic, but there are other versions: Cyclic and Compressed Cyclic. There are three levels for Rainbow Classic: I, III and V, and their security levels and parameters are shown in Table 7.

	NIST Security Level	Public Key (pk) size (kB)	Secret Key (sk) size (kB)	Signature (sig) size (bytes)
Classic I	1 and 2	157.8	101.2	528
Classic III	3 and 4	861.4	611.3	1312
Classic V	5	1885.4	1375.5	1632

Table 7. Rainbow: Security Levels and Parameter Sizes [43]

2.6.7 FALCON

Fast Fourier lattice-based compact signatures over NTRU, or FALCON, is a lattice based algorithm based on compactness. For the key generation part, FALCON describes it in two parts: solving for NTRU and computing a FALCON tree. In the first part, NTRU is mapped over a ring in a tower of rings, and for the second part, the FALCON tree is used to process compact and fast signature generation. A FALCON tree uses recursion to break diagonal elements in a matrix to place in another one over small rings so that subtrees are created. In signature creation, the algorithm computes a hash value from the original message and a salt, which is similar to a password to hashes information in a singular direction [44]. The resulting hash value is then used with the secret key to create two short values that become the signature. The signature is verified at the end is calculated comparing the public key, message and signature to an acceptance bound. [45]

FALCON has three implementations, FALCON-256, FALCON-512 and FALCON-1024, but provides the public key and signature parameters for the latter two since they fulfill NIST security requirements. This is shown in Table 8.

	NIST Security Level	Public Key (pk) size (bytes)	Signature (sig) size (bytes)
FALCON-512	1	897	666
FALCON-1024	5	1793	1280

Table 8. FALCON: Security Levels and Parameter Sizes [45]

2.6 Embedded System Comparison

An embedded system is a microcontroller based, software driven, reliable, real-time control system. It is a combination of computer software and hardware which is either fixed in capability or programmable. Embedded systems are designed to perform a dedicated function, either as an independent system or as a part of a large system. For all its features we are implementing these algorithms to an embedded system and measure its encryption and decryption in real time. [46]

We had few criterias choosing our embedded system which included the embedded system to be able to process Linux and run Ubuntu. Additionally, we needed enough RAM so that the CPU could handle the compilation of the algorithms and run them faster. Another specification was to have a display port to see the results of the tests in the algorithms which show the time that takes to encapsulate and decapsulate a message. We took Texas Instruments EK-TM4C123GXL, Arduino Yún Rev 2, Raspberry Pi 3 Model B (Raspberry Pi 3B), and Raspberry Pi 4 Model B (Raspberry Pi 4B) into our consideration. The Texas Instruments EK-TM4C123GXL board wouldn't be feasible to process Linux. Comparing the rest of the embedded systems, Raspberry Pi Models are more widely common than the Arduino board. Therefore, more sources are provided in terms of downloading the Linux operating system and using the board in general. The Arduino board also has a low RAM and clock speed compared to Raspberry Pi boards. It has 64 MB RAM and clock speed at 400 MHz which could affect the process of work and the results significantly. Between the two Raspberry Pi boards, we decided to go with Raspberry Pi 4B mainly because it had a higher random-access memory (RAM) with 8 GB while Raspberry Pi 3B has 1 GB. Raspberry Pi 4B also has 1.5GHz which is a little higher than Raspberry 3B's speed. That was a priority since there are many algorithms to run and we wanted to measure and compare the run times. Otherwise, both models have the same amount of HDMI ports and same power ratings and sources which is 1.25 A at 5V with USB-C port. [47]–[50]

2.7 Raspberry Pi 4B

After comparing the embedded systems we researched, we decided to go with Raspberry Pi 4 Model B with 8GB RAM. We marked some important differences compared to other boards and Raspberry Pi being used more commonly also affected our decision since there are more

resources available about it. The Raspberry Pi 4 has a Quad core Cortex-A72 and 8GB SDRAM. It also has some USB ports and micro-HDMI ports that are enough for us to display the test results of the algorithms.The Raspberry Pi has a powerful processor and is eligible to install and run Ubuntu. Raspberry Pi 4 is more reliable when running the tests for algorithms than other boards we considered. [49]

3. Algorithm Implementation

To test the timing of post-quantum cryptography, we ran the key-establishment and digital signature algorithms on a remote Linux server hosted by WPI and a Raspberry Pi 4 to get the averages of each phase of the program. Unless specified within the section, each program is repeated 1000 times within the average.

3.1 Linux Remote Machine Results

To get the Linux run times, we used the WPI run servers on PuTTY and used the speed capturing files in each of the programs to get the run times for each step of the encryption and digital signature algorithms. Many of the original files used implementations that used CPU cycles to run. The raw data for that will be listed in Appendix A. For the purposes of comparing, we converted the cycles to seconds using the Linux clock cycles for our particular machine.

We are running Linux hosted on a server that uses the CPU model 45, specifically the Intel (R) Xeon (R) CPU E5-4617 0 @ 2.90GHz. It is running at ~1196.757 MHz, but has a minimum of 1200 MHz and a maximum of 3400 MHz. The architectural structure is x86_64 and there are 12 CPUs total in the machine. There is a total memory of ~264.1 kB.

3.1.1 Classic McEliece

Classic McEliece [51] was one run with the reference implementation at all levels. When first trying to run the code, we initially got an issue with missing files so we needed the following libraries to be added:

- xstlproc
- libssl-dev
- KeccackCodePackage [52]

With KeccakCodePackage, we generated the Keccak library, libkeccak.a, then shared it in the system directory with the header files. The libkeccak.a version was “generic64”. The original code did not have timing implementations, so we used the internal timer, which will be described more in Raspberry Pi 4 Implementation chapter. The modified code can be seen in Appendix B.1 The results of the code are shown in Table 9.

	mceliece 348864 (ms)	mceliece 460896 (ms)	mceliece 6688128 (ms)	mceliece 6960119 (ms)	mceliece 8192128 (ms)
Keypair	202.24	466.74	1305.24	1144.09	1135.47
Encapsulation	0.11	0.16	0.27	0.56	0.29
Decapsulation	29.36	72.85	136.13	132.07	164.47

Table 9. Classic McEliece: Average Measurements on Linux

3.1.2 CRYSTALS-Kyber

CRYSTALS-Kyber [53] was run with it's reference implementations at all levels. Below, on Table 10 is the measured results of the keypair, encapsulation, and decapsulation times for each level. To get these times, we downloaded the software package, and made the overall speed file, "test_speed.c", then ran it's branching files for each of the levels: "test_speed512", "test_speed768" and "test_speed1024". Each of those files took the average of 1000 repeats for each step. There were no code modifications needed to run.

	Kyber512 (ms)	Kyber768 (ms)	Kyber1024 (ms)
Keypair	0.14	0.25	0.37
Encapsulation	0.20	0.30	0.39
Decapsulation	0.24	0.35	0.44

Table 10. CRYSTALS-Kyber: Average Measurements on Linux

3.1.3 NTRU

To run NTRU [54], there were no specific libraries needed for installation not currently on the machine. When running this program, there were four implementations, two of them at NIST Level 3 standards. Table 11 displays the results of the average measurements for keypair generation, encapsulation and decapsulation in milliseconds. These measurements were found by running the reference implementation version of each of the files and taking the average of 1000 repeats for each step of the encryption algorithm.

	ntruhps2048509 (ms)	ntruhrss701 (ms)	ntruhps2048677 (ms)	ntruhps4096821 (ms)
Keypair	9.96	17.27	17.44	19.71
Encapsulation	0.60	0.83	0.85	1.23
Decapulation	1.55	2.45	2.28	3.33

Table 11. NTRU: Average Measurements on Linux

3.1.4 SABER

Similarly to the previous two programs, SABER [55] did not need extra files to run it's programs. To get the SABER measurements, we ran the "test_kex" file in the reference implementation of the algorithm. To get each of the levels, the SABER definition in "SABER_params.h" needed to be modified for each run, which displays the average time for keypair, encapsulation, and decapsulation for all levels. Table 12 below displays the average measurements for the keypair generation, encapsulation and decapsulation for 1000 repeats on each level.

	LightSABER (ms)	SABER (ms)	FireSABER (ms)
Keypair	0.17	0.21	0.35
Encapsulation	0.23	0.28	0.38
Decapsulation	0.23	0.28	0.41

Table 12. SABER: Average Measurements on Linux

3.1.5 CRYSTALS-Dilithium

CRYSTALS-Dilithium [56] was run with its reference implementations at levels one, two and three [40]. Below, on Table 13 is the measured results of the keypair, encapsulation, and decapsulation times for each level. To get these times, we downloaded the software package, and made the overall speed file, "test_speed.c", then ran it's branching files for each of the levels: "DILITHIUM2", "DILITHIUM3" and "DILITHIUM4" which correspond to "test_speed2",

“test_speed3”, “test_speed4”. Each of those files took the average of 1000 repeats for each step. There were no extra libraries needed to run this or code modifications needed.

.

	DILITHIUM2 (ms)	DILITHIUM3 (ms)	DILITHIUM4 (ms)
Keypair	0.29	0.46	0.62
Signature	1.45	2.21	2.02
Verification	0.28	0.41	0.57

Table 13. CRYSTALS-Dilithium: Average Measurements on Linux

3.1.6 Rainbow

Rainbow [57] runs in NIST security levels one and two which is represented by Classic I, three and four by Classic III, and five by Classic V. In the Table 14 below, we entered the timings we measured for generating keypair, signature, and verification. To do so, we added a C file called test-time.c which takes the “rainbow-genkey.c”, “rainbow-sign.c”, “rainbow-verify.c” functions and calculates the timing. That can be found in Appendix B.2. Each of these took an average of 1000 repeats. In order to choose which file to test, the necessary change was done through setting up the Makefile accordingly.

	Classic I (ms)	Classic III (ms)	Classic V (ms)
Keypair	9.64	93.161	263.87
Signature	4.27	15.57	23.87
Verification	0.009	0.26	0.04

Table 14. Rainbow: Average Measurements on Linux

3.1.7 FALCON

FALCON was run using its reference implementation for FALCON-256, FALCON-512 and FALCON-1024. FALCON uses their own timing method in ‘speed.c’ that measures one cycle, but trains the machine first with blanks before collecting that data. Unlike most of the other programs, FALCON does not use the CPU RAM cycles to get it’s measurements, but uses

benchmarks. Additionally, there are no other libraries needed. Table 15 below shows the results from running the program.

	FALCON-256 (µs)	FALCON-512 (µs)	FALCON-1024 (µs)
Keypair	4.34	9.47	26.26
Signature	58.90	122.40	253.77
Verification	191.64	384.91	776.59

Table 15. FALCON: Average Measurements on Linux

3.2 Raspberry Pi 4B Implementations

The Raspberry Pi 4 Model B has a Quad core Cortex-A72 (ARM v8) and 8GB SDRAM and runs at 1.5GHz [49]. To get the measurements, Linux Ubuntu was downloaded into a microSD card as a means to run the code, similarly to how we did it in the Linux implementation.

However, for the majority of the implementations, we struggled to get the timed measurements since many of the algorithms used the CPU's cycles based off a Linux machine, and that code was not compatible with a Raspberry Pi, which uses a different kind of CPU. To fix this, we used the system time (sys/time) and the function "gettimeofday()" to collect the time difference between after and before a step of the algorithm ran, and ran that in a loop to get the average for 1000 cycles. Additionally, the Makefile had to be edited to remove instances of the original CPU cycle measurement code or else the program would fail to run. Any different implementations than this will be discussed in it's program-specific section.

3.2.1 Classic McEliece

Classic McEliece [51] was run with the reference implementation at all levels. When first trying to run the code, we also initially got an issue with missing files so we installed the same libraries as needed for the Linux implementation. The generated Keccak library, libkeccak.a, was shared in the system directory with the header files. However, the libkeccak.a version was "reference" because "generic64" was not compatible with a Raspberry Pi 4. This is because the

CPU architecture is different. The code itself did not change between the Linux and Raspberry Pi 4B implementations, so it can be found in Appendix A.1.

	mceliece 348864 (ms)	mceliece 460896 (ms)	mceliece 6688128 (ms)	mceliece 6960119 (ms)	mceliece 8192128 (ms)
Keypair	404.63	2134.48	15018.66	12300.12	13558.86
Encapsulation	0.49	0.83	1.36	2.44	1.42
Decapsulation	0.05	110.57	209.51	202.78	0.26

Table 16. Classic McEliece: Average Measurements on Raspberry Pi 4B

3.2.2 CRYSTALS-Kyber

Similarly to its Linux implementation, CRYSTALS-Kyber [53] was run with its reference implementations at levels one, three and five. Table 17 displays the measured results of each step of the key-establishment algorithm, the keypair, encapsulation, and decapsulation times, at each level. Prior to running, we downloaded the software package, and re-made the overall speed file, “test_speed.c”, to include the time/sys program and edited the make file so that it would not use the CPU cycles from the RAM that causes errors. Running the new implementation, then created branching files for each of the levels: “test_speed512”, “test_speed768” and “test_speed1024” which took the average of 1000 repeats. The files with modified code are in Appendix C.1.

.

	kyber512 (ms)	kyber768 (ms)	kyber1024 (ms)
Keypair	0.10	0.17	0.25
Encapsulation	0.12	0.19	0.29
Decapsulation	0.14	0.22	0.32

Table 17. CRYSTALS-Kyber: Average Measurements on Raspberry Pi 4B

3.2.3 NTRU

In the Raspberry Pi runthrough of NTRU [54], there were no specific libraries needed for installation. In NTRU, there were four implementations, two of them at NIST Level 3 standards. Table 18 displays the results of the average measurements for keypair generation, encapsulation

and decapsulation in milliseconds with 1000 repeats. These measurements were found by creating a new speed file that utilized the “gettimeofday()” command in each of the reference implementations, updating each makefile, then running each one to get our timing. The files with modified code are in Appendix C.2.

	ntruhps2048509 (ms)	ntruhrss701 (ms)	ntruhps2048677 (ms)	ntruhps4096821 (ms)
Keypair	3.29	5.85	5.81	8.30
Encapsulation	0.20	0.15	0.28	0.35
Decapulation	0.15	0.25	0.23	0.31

Table 18. NTRU: Average Measurements on Raspberry Pi 4B

3.2.4 SABER

SABER [55] did not need extra files to run it’s programs. To get the SABER measurements, we created a new speed file based on their “test_kex” file in the reference implementation of the algorithm. Similar to the Linux implementation, to get each of the levels, the SABER definition in “SABER_params.h” needed to be modified for each run. Another thing that needed to change was the makefile to remove the CPU cycle code that took measurements from the RAM and caused issues. At the end, we were able to display the average time for keypair, encapsulation, and decapsulation for all levels. Table 19 shows the average measurements of 1000 repeats for the keypair generation, encapsulation and decapsulation for each level. The files with modified code are in Appendix C.3.

.

	LightSABER (ms)	SABER (ms)	FireSABER (ms)
Keypair	0.1	0.18	0.29
Encapsulation	0.12	0.21	0.33
Decapsulation	0.13	0.23	0.37

Table 19. SABER: Average Measurements on Raspberry Pi 4B

3.2.5 CRYSTALS-Dilithium

To be consistent, CRYSTALS-Dilithium [56] was run with it's reference implementations with no additional libraries needed. To get these times, we downloaded the software package, and re-made the overall speed file, "test_speed.c", and the Makefile. After, we ran it's branching files for each of the levels: "DILITHIUM2", "DILITHIUM3" and "DILITHIUM4" which correspond to "test_speed2", "test_speed3", "test_speed4". Table 20 shows the measured results of the keypair, encapsulation, and decapsulation times for each level through taking an average of 1000 repeats for each step. The files with modified code are in Appendix C.4.

	DILITHIUM2 (ms)	DILITHIUM3 (ms)	DILITHIUM4 (ms)
Keypair	0.23	0.34	0.45
Signature	1.30	1.35	1.81
Verification	0.02	0.25	0.47

Table 20. CRYSTALS-Dilithium: Average Measurements on Raspberry Pi 4B

3.2.6 Rainbow

Like its Linux implementation, Rainbow [57] runs in NIST security levels one and two which is represented by Classic I, three and four by Classic III, and five by Classic V. In the Table 21 below, we entered the timings we measured for generating keypair, signature, and verification. To do so, we added a C file called test-time.c which takes the "rainbow-genkey.c", "rainbow-sign.c", "rainbow-verify.c" functions and calculates the timing. Each of these took an average of 1000 repeats on Raspberry Pi 4B as well.

	Classic I (ms)	Classic III (ms)	Classic V (ms)
Keypair	32.38	36.00	993.90
Signature	12.48	56.15	88.33
Verification	0.02	0.07	0.14

Table 21. Rainbow: Average Measurements on Raspberry Pi 4B

3.2.7 FALCON

With the Raspberry Pi, FALCON [58] was also run using its reference implementation for FALCON-256, FALCON-512 and FALCON-1024. For FALCON, they use their own timing method that uses one cycle, but trains the machine first with blanks before collecting that data. FALCON does not initially use the CPU RAM cycles so we did not need to create any new code for this program, additionally no libraries were needed. Table 22 belows shows the result of the measurements.

.

	FALCON-256 (µs)	FALCON-512 (µs)	FALCON-1024 (µs)
Keypair	11.59	23.43	68.79
Signature	157.29	333.58	704.59
Verification	501.12	1042.00	2261.03

Table 22. FALCON: Average Measurements on Raspberry Pi 4B

4. Comparison

Looking at the average measurements for each algorithm, we saw that in general, when the security level increases, it takes longer for them to generate keypair, and doing the encapsulation and decapsulation, or signature and verification. When we compared the results from both timings, we realized that most of the algorithms run faster on the Raspberry Pi 4 Model B than the Linux server. From Encryption, Classic McEliece, and from Digital Signature, FALCON, and Rainbow which are the exceptions. For the other algorithms, they run faster on the Raspberry Pi 4B. We compared Encryption and Digital Signature separately for each program. We then took the ratios of each algorithm. We divided the higher timings to smaller ones to see how many times faster the smaller timing ran. At the end, we ran a comparison of how much faster each program ran within Linux or Raspberry Pi 4B by finding the speed factor based on the highest speed.

4.1 Encryption Algorithm Comparison

As mentioned before, Classic McEliece, CRYSTALS-Kyber, NTRU, SABER are under the Encryption Algorithms. First, we are going to compare the timings of each algorithm with the different security levels addressed to.

	mceliece 348864	mceliece 460896	mceliece 6688128	mceliece 6960119	mceliece 8192128
Keypair	2.00	4.57	11.51	10.75	11.94
Encapsulation	4.49	5.05	4.98	4.32	4.98
Decapsulation	1.60	1.58	1.54	1.54	1.56

Table 23. Classic McEliece Timings Ratio

Looking at Classic McEliece's timing on Table 23, we concluded that it runs faster on the Linux machine than Raspberry Pi 4B. For the security levels 1 and 3 (mceliece348864 and mceliece460896), we see that the encapsulation timing has a bigger difference compared to others. Comparing all security level 5s (mceliece6688128, mceliece6960119 and

mceliece8192128), the biggest ratio occurs during generating keypair while the smallest happens during decapsulation.

	kyber512	kyber768	kyber1024
Keypair	1.42	1.5	1.46
Encapsulation	1.62	1.57	1.37
Decapsulation	1.67	1.58	1.37

Table 24. CRYSTALS-Kyber Timings Ratio

CRYSTALS-Kyber's timings, on Table 24, are around 1.5 times longer on Linux compared to its timings on the Raspberry Pi 4B. This is the case for all of its security levels. Here, we see Raspberry Pi 4B being faster in running this algorithm. The highest difference in ratio is 1.67 for kyber512 decapsulation and the lowest is 1.37 for the timing of kyber1024 decapsulation. While this is one of the small differences for an algorithm, we see a bigger difference for NTRU.

	ntruhps2048509	ntruhrss701	ntruhps2048677	ntruhps4096821
Keypair	3.03	2.95	3.00	2.37
Encapsulation	3.05	5.65	3.08	3.48
Decapulation	10.42	10.02	9.84	10.68

Table 25: NTRU Timings Ratio

Comparing Table 11 and Table 18, NTRU takes 9.96 ms to generate keypair on Linux while it takes only 3.29 ms for its security level 1. We can see the same difference in encapsulation and decapsulation timings not only in security level 1, but also in other security levels. If we look at Table 25, the timings that the Linux machine gave us are almost triple of what we measured in Raspberry Pi 4 for all of its security levels. However, for all the decapsulation timings of each NTRUs run about 10 times faster on Raspberry Pi 4B than the

Linux machine. For the decapsulation timings, the ratio increases for all the security levels which is 10.24.

	LightSABER	SABER	FireSABER
Keypair	1.75	1.19	1.23
Encapsulation	1.93	1.26	1.14
Decapsulation	1.83	1.20	1.12

Table 26. SABER Timings Ratio

SABER also has a small difference compared to NTRU. The smallest ratio for SABER is 1.12 from FireSABER for decapsulation timings. On the other hand, the biggest difference is 1.93 from LightSABER which is from the encapsulation timings of LightSABER. This is seen in Table 26.

We also created a comparison table below (Table 27) to see visually how each key-encryption timings differ from each other. To be able to calculate the speed factors more efficiently, we converted Classic McEliece results from s to ms so that they are all in the same unit. Our comparison table below is based on the NIST security level 3 which is common for all. Since NTRU had two files in security level 3, we chose the one with the biggest public size. For this comparison, we used mceliece460896, ntruhrss701, kyber768, and SABER.

	Level 3 Keypair		Level 3 Encapsulation		Level 3 Decapsulation		Public Key Size
	Time (ms)	Speed Factor	Time (ms)	Speed Factor	Time (ms)	Speed Factor	(bytes)
Cl.McEliece	466.74	1x	0.16	5x	72.85	1x	524160
NTRU	17.27	27x	0.83	1x	2.45	30x	1138
CRY.-Kyber	0.25	1,867x	0.30	3x	0.35	208x	1088
SABER	0.21	2,223x	0.28	3x	0.28	260x	992

Table 27. Encryption Comparison on Linux Machine

In Table 27, we compared the Key-Encryption algorithms for NIST security level 3. One of the first things that is remarkable is that among all these algorithms run on the Linux machine in the table above, we see that Classic McEliece is the one that takes longer to run and occupies the most space with its public key size with 524160. With that size of a post-quantum algorithm, it would be hard to use it for an embedded system with a small RAM size such as MSP430 with 512 kB flash memory. SABER, on the other hand, is the one that runs faster and has the smallest public key size with 992 bytes. When we compare the timing of Classic McEliece's generating keypair and SABER's, we see that SABER is 2,223 times faster. Looking at encapsulation timings, it is noticeable that NTRU with public key size 1138 bytes which is the second greatest, takes almost five times longer than Classic McEliece which is the fastest doing the encapsulation. Classic McEliece is again the one that runs the slowest during decapsulation and SABER is again the fastest. SABER runs about 260 times faster than Classic McEliece.

The timings change for the same algorithms on Raspberry Pi4B. However, we see that Classic McEliece is still the slowest in general.

	Level 3 Keypair		Level 3 Encapsulation		Level 3 Decapsulation		Public Key Size
	Time (ms)	Speed Factor	Time (ms)	Speed Factor	Time (ms)	Speed Factor	(bytes)
Cl.McEliece	2134.48	1x	0.83	1x	110.57	1x	524160
NTRU	5.85	365x	0.15	6x	0.25	442x	1138
SABER	0.18	11,856x	0.21	4x	0.23	481x	992
CRY.-Kyber	0.17	12,556x	0.19	4x	0.22	503x	1088

Table 28. Encryption Comparison on Raspberry Pi 4B

Looking at Table 28, we realized that Classic McEliece runs the slowest on Raspberry Pi 4B among all in each category for NIST security level 3. CRYSTALS-Kyber is the fastest in generating keypair and decapsulation key. CRYSTALS-Kyber runs around 12,556 faster than Classic McEliece which is immediately recognizable. We see that the speed factor is less high for

decapsulation which is 503. There is not a noticeable difference for the encapsulation timings, however, we see that NTRU runs approximately six times faster than Classic McEliece.

Finally, we compared these algorithms' highest security levels. While this is the security level 5 for Classic McEliece, SABER, and NTRU, for CRYSTALS-Kyber it is security level 4. Since Classic McEliece has three available algorithms for the security level 5, we chose the one that has the largest public key size which is mceliece6960119. We are using the timing measurements for KYBER1024, ntru4096821, and FireSABER. The results are shown in Table 29.

	Level 5 Keypair Level 4 for CRY.-Kyber		Level 5 Encapsulation Level 4 for CRY.-Kyber		Level 5 Decapsulation Level 4 for CRY.-Kyber		Public Key Size
	Time (ms)	Speed Factor	Time (ms)	Speed Factor	Time (ms)	Speed Factor	(bytes)
Cl.McEliece	1144.09	1x	0.56	2x	132.07	1x	1047319
NTRU	19.71	58x	1.23	1x	3.33	40x	1230
SABER	0.35	3,269x	0.38	3x	0.41	322x	1312
CRY.-Kyber	0.37	3,092x	0.39	3x	0.44	300x	1440

Table 29. High Security Level Encryption Comparison on Linux

From Table 29, we see that Classic McEliece's level 5 algorithm is again the one that has the largest public key size. It is larger than its other security level public key sizes and has a big difference with NTRU, SABER and CRYSTALS-Kyber. Also, Classic McEliece, like security level 3 comparison in the Table 27, runs the slowest during the generation of keypair and decapsulation. By looking at the speed factor under the keypair, SABER is about 3,269 times faster than Classic McEliece which is also the biggest difference in this table among the speed factors. While Classic McEliece runs in 132.07 ms to decapsulate, SABER, on the other hand, runs in 0.41 ms which is 322 times faster. We don't see a big difference in algorithms' encapsulation timings. NTRU takes up the most time however doing the encapsulation while the slowest is CRYSTALS-Kyber. However, SABER runs almost about the same time with CRYSTALS-Kyber, which shows us that there is no big difference when they are compared.

CRYSTALS-Kyber is the only one with security level 4 yet, still not the fastest in most of the categories above. We see that SABER is the fastest among all these categories.

	Level 5 Keypair Level 4 for CRY.-Kyber		Level 5 Encapsulation Level 4 for CRY.-Kyber		Level 5 Decapsulation Level 4 for CRY.-Kyber		Public Key Size
	Time (ms)	Speed Factor	Time (ms)	Speed Factor	Time (ms)	Speed Factor	(bytes)
Cl.McEliece	12,300.12	1x	2.44	1x	202.78	1x	1047319
NTRU	8.3	1,482x	0.35	7x	0.31	654x	1230
SABER	0.29	42,414x	0.33	7x	0.37	548x	1312
CRY.-Kyber	0.25	49,200x	0.29	8x	0.32	634x	1440

Table 30. High Security Level Encryption Comparison on Raspberry Pi 4B

When we look at the same algorithms' timings on Raspberry Pi 4B in Table 30 for their highest security level, McEliece is again the slowest to generate a keypair, encapsulate and decapsulate key. The biggest difference strikes us under the keypair category. While CRYSTALS-Kyber only takes 0.25 ms to generate, Classic McEliece runs 49,200 times longer which gives us the average result of 12,300.12 ms. For encapsulation, we see that NTRU, SABER, and CRYSTALS-Kyber run in less than 1 ms. Classic McEliece is 8 times longer to encapsulate than the CRYSTALS-Kyber which is relatively smaller than other speed factors but higher than the factor on the Table 29.

4.2 Digital Signature Algorithm Comparison

We also observed some differences with the Digital Signature algorithms. We again first took the timing ratios of each tested in Linux and Raspberry Pi 4B and did the comparison. Later, we compared NIST security level 1 results of the algorithms with each other by showing in the table the speed factors for keypair, signature, and verification.

	DILITHIUM2	DILITHIUM3	DILITHIUM4
Keypair	1.29	1.37	1.37
Signature	1.12	1.63	1.12
Verification	11.62	1.67	1.2

Table 31. CRYSTALS-Dilithium Timings Ratio

CRYSTALS-Dilithium, on Table 31, also has some big differences in its timings on Linux versus Raspberry Pi 4B. While the lowest difference is between the signature timing of DILITHIUM4 which is measured 2.02 ms on Linux and 1.806 ms on Raspberry Pi 4B. While these timings ratio is 1.12, which is one of the smallest differences and the highest timing difference is observed in DILITHIUM2 during its verification with a ratio of 11.62.

	Classic I	Classic III	Classic V
Keypair	3360	390	3770
Signature	2930	3610	3700
Verification	2220	280	3300

Table 32. Rainbow Timings Ratio

By looking at the table above (Table 32) for Rainbow, in general each Raspberry Pi4B is slower than the Linux machine. However, it is possible to observe the Linux machine being faster for generating keypair and doing verification for Rainbow Classic III. From Table 32, we see that Rainbow takes the shortest time creating keypair for Classic V in Linux and runs 3770 times faster than it runs on Raspberry Pi 4B.

	FALCON-256	FALCON-512	FALCON-1024
Keypair	2.67	2.47	2.62
Signature	2.67	2.73	2.78
Verification	2.61	2.71	2.91

Table 33. FALCON Timings Ratio

FALCON, shown in Table 33, runs faster in the Linux machine than the Raspberry Pi 4B. Its 256-bit security level generates keypair in 4.34 μs in the Linux machine while it takes 11.59 μs in Raspberry Pi 4. While it is taking 58.9 μs in order to create the signature key on the Linux machine, on Raspberry Pi 4B, it takes 157.29 μs which is more than twice of the time that took on Linux. Verification time taken in both also has the same difference. More we look into it, it is possible to observe the same pattern for FALCON's other security levels as well. The highest ratio is between the timing of the decapsulation of FALCON-1024 and the lowest difference is between the timings of the generation of keypair for the second security level.

For Digital Signature, we also created a table to see the difference by including the speed factor for keypair, signature, and verification. We took the timings for Rainbow Classic I, DILITHIUM2, and FALCON-512. It is shown below on Table 34.

	Level 1 Keypair		Level 1 Signature		Level 1 Verification		Public Key Size
	Time (ms)	Speed Factor	Time (ms)	Speed Factor	Time (ms)	Speed Factor	(bytes)
Rainbow	9.64	1x	4.27	1x	0.01	38x	157800
CRY.-Dilithium	0.29	33x	1.45	3x	0.28	1x	1184
FALCON	0.01	964x	0.12	36x	0.38	1x	897

Table 34. Digital Signature Comparison on Linux

To compare Digital Signature algorithms in the same security level, we chose Rainbow Classic I, DILITHIUM2, and FALCON-1024. When we compare the digital signature algorithms' NIST security level 1, which is common for all, we see that Rainbow takes more

time generating keypair and signature than others. For this part of comparison, we converted FALCON's timings from μs to ms in order to see the difference clearer. With 0.01 ms, FALCON takes about 964 times faster to generate keypair than Rainbow and is the fastest among three. We again see that FALCON is also the fastest in creating signatures. With 0.12 ms, it is 36 times faster than Rainbow which takes 4.27 ms. While Rainbow takes the most time in generating keypair and creating signature, it is also the fastest in verification with 0.01 ms. However, Rainbow Classic I, which runs the security level 1 has the largest public key size among all yet, it is still less than Classic McEliece.

	Level 1 Keypair		Level 1 Signature		Level 1 Verification		Public Key Size
	Time (ms)	Speed Factor	Time (ms)	Speed Factor	Time (ms)	Speed Factor	(bytes)
Rainbow Clas	32.38	1x	12.48	1x	0.02	5x	157800
CRY.-Dilithium	0.23	167x	1.30	10x	0.02	5x	1184
FALCON	0.02	1,619x	0.33	38x	1.04	1x	897

Table 35. Digital Signature Comparison on Raspberry Pi 4B

When we run the same algorithms on Raspberry Pi4B, we observed the similar results which are seen in Table 33. Rainbow is again the slowest to generate keypair and create signatures while it is one of the fastest in verification with CRYSTALS-Dilithium. On the contrary, FALCON, again, is the fastest to generate keypair and create signatures. We see that the speed factor increased between Rainbow and FALCON. While FALCON was 964 times faster than Rainbow to generate keypair in Linux, running it on Raspberry Pi 4B, it increased to 1,619. This is similar to the speed factor of creating a signature. While there is no big difference between them, it is still remarkable seeing differences increasing.

Like we did for key-encryption algorithms, we also compared digital signature algorithms' timings run in their highest security levels. For this, we used DILITHIUM4, Rainbow Classic V, and FALCON-1024. While the highest security level for Rainbow and FALCON is 5, CRYSTALS-Dilithium is level 3.

	Level 5 Keypair Level 3 for CRY.-Dilithium		Level 5 Signature Level 3 for CRY.-Dilithium		Level 5 Verification Level 3 for CRY.-Dilithium		Public Key Size
	Time (ms)	Speed Factor	Time (ms)	Speed Factor	Time (ms)	Speed Factor	(bytes)
CRY.-Dilithium	0.62	426x	2.02	12x	0.57	1x	1760
Rainbow	263.87	1x	23.87	1x	0.04	20x	1632
FALCON	0.03	8,796x	0.26	92x	0.78	1x	1280

Table 36. High Security Level Digital Signature Comparison on Linux

Comparing the high security level algorithms for digital signature, we can say that Rainbow takes the most time to generate keypair and signature key while FALCON is the fastest in these categories. Rainbow runs approximately 8,796 times slower than FALCON to generate For the signature, the fastest and slowest are again Rainbow and FALCON. Rainbow runs in 23.87 ms while FALCON only takes 0.26 ms. When we check the speed factor, we see that FALCON is faster by 92 times. However, for the verification we see that it is the opposite. In this case, we see that Rainbow is faster in verification. It is important to mention again that CRYSTALS-Dilithium's security level is 3 which is lower than the other two so this might have affected the results too.

	Level 5 Keypair Level 3 for CRY.-Dilithium		Level 5 Signature Level 3 for CRY.-Dilithium		Level 5 Verification Level 3 for CRY.-Dilithium		Public Key Size
	Time (ms)	Speed Factor	Time (ms)	Speed Factor	Time (ms)	Speed Factor	(bytes)
CRY.-Dilithium	0.45	2,209x	1.81	49x	0.47	5x	1760
Rainbow	993.9	1x	88.33	1x	0.14	16x	1632
FALCON	0.07	14,199x	0.7	126x	2.3	1x	1280

Table 37. High Security Level Digital Signature Comparison on Raspberry Pi 4B

Similarly to the comparison made in Table 36, algorithm timings from Raspberry Pi 4B show us that Rainbow is again the slowest to generate keypair and signature while FALCON is the fastest. We see that the speed factor for these two categories increased when we run it on Raspberry Pi 4B. FALCON became 14,199 faster than Rainbow to generate keypair on Raspberry Pi 4B. The speed factor also increased for the signature. We see that FALCON runs here 126 times faster than Rainbow. However, for the verification, like in Table 36, FALCON became the slowest among them all.

5. Findings and Conclusion

Through comparing each of the sets of code we have for the Linux and Raspberry Pi 4B implementations, we see that most of the code runs faster on the Raspberry Pi 4B in comparison to Linux. There were some factors that contributed to this. First, the speeds of the CPU. Linux ran at ~1197 MHz while the Raspberry Pi runs at ~1.5 GHz. A faster CPU speed for the Raspberry Pi 4B will decrease the amount of time it takes to run each program. Another factor would be the age of the CPU. The Linux machine is run on an older server while the Raspberry Pi 4B is a newer model. Being a newer model, even with similar CPU clock cycles does impact how well the CPU will perform when being run on the program.

However, there are two clear outliers. The programs FALCON, and Classic McEliece, had the speeds reversed where the Raspberry Pi implementation took longer than the Linux implementation. We looked into some factors that could contribute to that. One of them may be the way the FALCON measures speed. Unlike the other programs, FALCON's original program was compatible with both Linux and Raspberry Pi, so we did not have to be concerned about different methods of measurement being an issue, but it varied in that FALCON ran all security levels of the program within one program iteration.

For Classic McEliece, there was a significant increase in time when creating the keypair on Raspberry Pi 4B. Part of this issue could be the way it was generated. Classic McEliece had the largest sizes for public keys at each security level within encryption algorithms. Because of its size, it takes significantly longer to generate the key. The Raspberry Pi 4B likely did not have the same CPU power to process such large bits of information, and took a longer time running than the Linux.

With more time permitted, we wanted to look into multiple embedded systems, specifically ones with singular or dual cores, however, we had difficulty finding one that had the capacity to run Linux Ubuntu, and that was necessary to operate the program. We would also have liked to look into the alternative place winners since they also had interesting post-quantum algorithms with much potential to become a standardization.

Bibliography

[1] "Post-quantum Cryptography," *Microsoft Research*. https://www.microsoft.com/en-us/research/project/post-quantum-cryptography/ (accessed Mar. 07, 2021).

[2] D. R. T. D. R. is a freelance writer, content developer who lives in Kennebunk, and Maine., "What Is the Difference Between Quantum Cryptography and Post-Quantum Cryptography?," *Technology Solutions That Drive Government*. https://fedtechmagazine.com/article/2020/03/what-difference-between-quantum-cryptography-and-post-quantum-cryptography-perfcon (accessed Mar. 07, 2021).

[3] "Post-Quantum Cryptography Definition & Meaning," *Webopedia*, Dec. 11, 2020. https://www.webopedia.com/definitions/post-quantum-cryptography-definition-meaning/ (accessed Mar. 07, 2021).

[4] I. T. L. Computer Security Division, "Post-Quantum Cryptography | CSRC | CSRC," *CSRC | NIST*, Jan. 03, 2017. https://content.csrc.e1c.nist.gov/projects/post-quantum-cryptography (accessed Mar. 07, 2021).

[5] M. Mosca, "Cybersecurity in an era with quantum computers: will we be ready?," 1075, 2015. Accessed: Oct. 26, 2020. [Online]. Available: http://eprint.iacr.org/2015/1075.

[6] L. Chen *et al.*, "Report on Post-Quantum Cryptography," National Institute of Standards and Technology, NIST IR 8105, Apr. 2016. doi: 10.6028/NIST.IR.8105.

[7] G. Alagic *et al.*, "Status report on the first round of the NIST post-quantum cryptography standardization process," National Institute of Standards and Technology, Gaithersburg, MD, NIST IR 8240, Jan. 2019. doi: 10.6028/NIST.IR.8240.

[8] C. Paar and J. Pelzl, "Introduction to Public-Key Cryptography," in *Understanding Cryptography: A Textbook for Students and Practitioners*, C. Paar and J. Pelzl, Eds. Berlin, Heidelberg: Springer, 2010, pp. 149–171.

[9] C. Paar and J. Pelzl, "Key Establishment," in *Understanding Cryptography: A Textbook for Students and Practitioners*, C. Paar and J. Pelzl, Eds. Berlin, Heidelberg: Springer, 2010, pp. 331–357.

[10] C. Paar and J. Pelzl, "Digital Signatures," in *Understanding Cryptography: A Textbook for Students and Practitioners*, C. Paar and J. Pelzl, Eds. Berlin, Heidelberg: Springer, 2010, pp. 259–292.

[11] C. Paar and J. Pelzl, "Introduction to Cryptography and Data Security," in *Understanding Cryptography: A Textbook for Students and Practitioners*, C. Paar and J. Pelzl, Eds. Berlin, Heidelberg: Springer, 2010, pp. 1–27.

[12] P. W. Shor, "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer," *ArXivquant-Ph9508027*, Jan. 1996, doi: 10.1137/S0097539795293172.

[13] Köksal Muş, "Chapter 8: RSA," presented at the Lecture for WPI's ECE4802 Course.

[14] E. B. Kania and J. K. Costello, "The Second Quantum Revolution," Center for a New American Security, 2018. Accessed: Oct. 26, 2020. [Online]. Available: http://www.jstor.org/stable/resrep20450.5.

[15] G. M. Palma, K.-A. Suominen, and A. K. Ekert, "Quantum Computers and Dissipation," *Proc. Math. Phys. Eng. Sci.*, vol. 452, no. 1946, pp. 567–584, 1996.

[16] Y. Wang, "Quantum Computation and Quantum Information," *Stat. Sci.*, vol. 27, no. 3, pp. 373–394, 2012.

[17] S. Y. Yan, *Computational Number Theory and Modern Cryptography*. John Wiley & Sons, 2013.
[18] O. Goldreich, "The Foundations of Modern Cryptography," in *Modern Cryptography, Probabilistic Proofs and Pseudorandomness*, vol. 17, Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 1–37.
[19] H. Zbinden, H. Bechmann-pasquinucci, N. Gisin, and G. Ribordy, "Quantum cryptography," in *Applied Physics B Lasers and Optics*, Springer-Verlag, 1998, pp. 743–748.
[20] N. A. Fauziah, E. H. Rachmawanto, D. R. I. M. Setiadi, and C. A. Sari, "Design and Implementation of AES and SHA-256 Cryptography for Securing Multimedia File over Android Chat Application," in *2018 International Seminar on Research of Information Technology and Intelligent Systems (ISRITI)*, Nov. 2018, pp. 146–151, doi: 10.1109/ISRITI.2018.8864485.
[21] "Submission Requirements and Evaluation Criteria for the Post-Quantum Cryptography Standardization Process." NIST, [Online]. Available: https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/call-for-proposals.
[22] D. Moody *et al.*, "Status report on the second round of the NIST post-quantum cryptography standardization process," National Institute of Standards and Technology, Gaithersburg, MD, NIST IR 8309, Jul. 2020. doi: 10.6028/NIST.IR.8309.
[23] R. Avanzi *et al.*, "CRYSTALS-KYBER Algorithm Specifications And Supporting Documentation," p. 32, 2017.
[24] J. Katz and Y. Lindell, *Introduction to modern cryptography : principles and protocols*. Boca Raton : CRC PRESS, 2007.
[25] Daniel J. Bernstein *et al.*, "Classic McEliece: conservative code-based cryptography," Nov. 2017. [Online]. Available: https://classic.mceliece.org/nist/mceliece-20171129.pdf.
[26] Martin R. Albrecht *et al.*, "Classic McEliece: conservative code-based cryptography," Oct. 2020. [Online]. Available: https://classic.mceliece.org/nist/mceliece-20201010.pdf.
[27] Daniel J. Bernstein *et al.*, "Classic McEliece: conservative code-based cryptography," Mar. 2019. [Online]. Available: https://classic.mceliece.org/nist/mceliece-20190331.pdf.
[28] O. Regev, "Learning with Errors over Rings," in *Algorithmic Number Theory*, vol. 6197, G. Hanrot, F. Morain, and E. Thomé, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 3–3.
[29] D. Hofheinz, K. Hövelmanns, and E. Kiltz, "A Modular Analysis of the Fujisaki-Okamoto Transformation," in *Theory of Cryptography*, Cham, 2017, pp. 341–371, doi: 10.1007/978-3-319-70500-2_12.
[30] E. E. Targhi and D. Unruh, "Quantum Security of the Fujisaki-Okamoto and OAEP Transforms," 1210, 2015. Accessed: Mar. 04, 2021. [Online]. Available: http://eprint.iacr.org/2015/1210.
[31] P. Schwabe, "CRYSTALS-Kyber," Apr. 12, 2018, [Online]. Available: https://pq-crystals.org/kyber/data/slides-nistpqc18-schwabe.pdf.
[32] C. Chen *et al.*, "NTRU Algorithm Specifications And Supporting Documentation," 2019.
[33] J. Hoffstein, J. Pipher, and J. H. Silverman, "NTRU: A ring-based public key cryptosystem," in *Algorithmic Number Theory*, vol. 1423, J. P. Buhler, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 267–288.
[34] Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman, "NTRU: A new high speed public key cryptosystem," Brown University. [Online]. Available:

https://web.securityinnovation.com/hubfs/files/ntru-orig.pdf.
[35] A. Hülsing, J. Rijneveld, J. Schanck, and P. Schwabe, "High-speed key encapsulation from NTRU," p. 21.
[36] M. V. Beirendonck, J.-P. D'Anvers, A. Karmakar, J. Balasch, and I. Verbauwhede, "A Side-Channel Resistant Implementation of SABER," 733, 2020. Accessed: Oct. 26, 2020. [Online]. Available: http://eprint.iacr.org/2020/733.
[37] J. Alwen, S. Krenn, K. Pietrzak, and D. Wichs, "Learning with Rounding, Revisited: New Reduction, Properties and Applications," 098, 2013. Accessed: Mar. 04, 2021. [Online]. Available: http://eprint.iacr.org/2013/098.
[38] "SABER: LWR-based KEM." https://www.esat.kuleuven.be/cosic/pqcrypto/saber/ (accessed Mar. 04, 2021).
[39] M. Ozols, M. Roetteler, and J. Roland, "Quantum rejection sampling," *Proc. 3rd Innov. Theor. Comput. Sci. Conf. - ITCS 12*, pp. 290–308, 2012, doi: 10.1145/2090236.2090261.
[40] L. Ducas *et al.*, "CRYSTALS-Dilithium Algorithm Specifications And Supporting Documentation," p. 32, Mar. 2019.
[41] V. Lyubashevsky, "Fiat-Shamir with Aborts: Applications to Lattice and Factoring-Based Signatures," in *Advances in Cryptology – ASIACRYPT 2009*, vol. 5912, M. Matsui, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 598–616.
[42] Jintai Ding *et al.*, "Rainbow - Algorithm Specification and Documentation," The 3rd Round Proposal.
[43] "PQCRainbow." https://www.pqcrainbow.org/ (accessed Feb. 21, 2021).
[44] P. Gauravaram, "Security Analysis of salt||password Hashes," in *2012 International Conference on Advanced Computer Science Applications and Technologies (ACSAT)*, Nov. 2012, pp. 25–30, doi: 10.1109/ACSAT.2012.49.
[45] P.-A. Fouque *et al.*, "Falcon: Fast-Fourier Lattice-based Compact Signatures over NTRU," p. 67.
[46] "What is an Embedded System? Definition and FAQs | OmniSci." https://www.omnisci.com/technical-glossary/embedded-systems (accessed Feb. 14, 2021).
[47] "Arduino Yún Rev 2 | Arduino Official Store." https://store.arduino.cc/usa/arduino-yun-rev-2 (accessed Feb. 28, 2021).
[48] The Raspberry Pi Foundation, "Buy a Raspberry Pi 3 Model B," *Raspberry Pi*. https://www.raspberrypi.org/products/raspberry-pi-3-model-b/ (accessed Feb. 28, 2021).
[49] "Raspberry Pi 4 Computer Model B Product Brief." Raspberry Pi Trading Ltd., Jan. 2021, [Online]. Available: https://datasheets.raspberrypi.org/rpi4/raspberry-pi-4-product-brief.pdf.
[50] "Tiva™ TM4C123GH6PM Microcontroller Data Sheet." Texas Instruments Incorporated, Jun. 12, 2014.
[51] Lawrence E. Bassham *et al.* (2020), *mceliece-20201010* (NIST Round 3) [Source Code]. https://classic.mceliece.org/nist/mceliece-20201010.tar.gz.
[52] Ko- (2018), *Ko-/KeccakCodePackage* (forked from XKCP/XKCP) [Source Code]. https://github.com/Ko-/KeccakCodePackage.
[53] Roberto Avanzi *et al.* (2020), *pq-crystals/kyber* (NIST Round 3) [Source Code]. https://github.com/pq-crystals/kyber.
[54] J. Schanck et al. (2020), jschanck/ntru (NIST Round 3) [Source Code]. https://github.com/jschanck/ntru.
[55] Jan-Pieter D'Anvers *et al.* (2020), *KULeuven-COSIC/SABER* (NIST Round 3) [Source

Code]. KU Leuven - COSIC. https://github.com/KULeuven-COSIC/SABER.
[56] Léo Ducas *et al.* (2020), *pq-crystals/dilithium*. (NIST Round 3) [Source Code]. https://github.com/pq-crystals/dilithium.
[57] Jintai Ding *et al.* (2020), *fast-crypto-lab/rainbow-submission-round2* (NIST Round 3) [Source Code]. Fast Crypto Lab. https://github.com/fast-crypto-lab/rainbow-submission-round2.
[58] Pierre-Alain Fouque *et al.* (2020), *FALCON* (NIST Round 3) [Source Code]. https://falcon-sign.info/falcon-round3.zip.

Appendix A: Raw Data for Linux Implementations

Repeats = 1000 cycles (for each program)

1. CRYSTALS-Kyber (units are CPU cycles)

	Level 1 (kyber512)	Level 3 (kyber768)	Level 5 (kyber1024)
Keypair	median: 168180 average: 168650	median: 295976 average: 297788	median: 466652 average: 438888
Encapsulation	median: 242528 average: 239032	median: 363480 average: 364821	median: 468861 average: 471802
Decapsulation	median: 289392 average: 284423	median: 414340 average: 415964	median: 526357 average: 528385

2. NTRU (units are CPU cycles)

	Level 1 (2048509)	Level 3 (701)	Level 3 (2048677)	Level 5 (4096821)
Keypair	median: 10876050 average: 11912411	median: 20070928 average: 20669804	median: 19498166 average: 20866148	median: 23313031 average: 23581402
Encapsulation	median: 708538 average: 710936	median: 992059 average: 994478	median: 1017747 average: 1021620	median: 1458174 average: 1463140
Decapulation	median: 1844876 average: 1857940	median: 2924214 average: 2936811	median: 2712136 average: 2729136	median: 3967731 average: 3986273

3. SABER (units are in CPU cycles)

	Level 2 (LightSABER)	Level 3 (SABER)	Level 4 (FireSABER)
Keypair	Average times key_pair: 207730	Average times key_pair: 256479	Average times key_pair: 423619
Encapsulation	Average times enc:	Average times enc:	Average times enc:

	270520	330302	453343
Decapsulation	Average times dec: 279593	Average times dec: 335166	Average times dec: 489642

4. CRYSTALS-Dilithium (units are in CPU cycles)

	Level 2 (DILITHIUM2)	Level 3 (DILITHIUM3)	Level 4 (DILITHIUM4)
Keypair	median: 367436 average: 350920	median: 553308 average: 555878	median: 738912 average: 743010
Signature	median: 1370959 average: 1734800	median: 2030460 average: 2641179	median: 1944139 average: 2414323
Verification	median: 338247 average: 340612	median: 487551 average: 489695	median: 662992 average: 667353

Appendix B: Code Modifications for Linux

B.1 Classic McEliece

/mceliece-20201010/Reference_Implementation/kem/mceliece348864/build

For this program, mceliece348864 was referenced, but this code was applied as well to mceliece460896, mceliece6688128, mceliece6960119 and mceliece8192128. [51]

```
#!/bin/sh
gcc -O3 -march=native -mtune=native -Wall -I. -Isubroutines -DKAT -
L /home/xxxxx/KeccakCodePackage/bin/generic64/ -$
```

/mceliece-20201010/Reference_Implementation/kem/mceliece348864/nist/kat_kem.c

For this program, mceliece348864 was referenced, but this code was applied as well to mceliece460896, mceliece6688128, mceliece6960119 and mceliece8192128. Here, KATNUM indicates the repeats and is set to 1000. [51]

```
/*
   PQCgenKAT_kem.c
   Created by Bassham, Lawrence E (Fed) on 8/29/17.
   Copyright © 2017 Bassham, Lawrence E (Fed). All rights reserved.
   + mods from djb: see KATNOTES
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "rng.h"
#include "crypto_kem.h"
#include <sys/time.h>

#define KAT_SUCCESS          0
#define KAT_FILE_OPEN_ERROR -1
#define KAT_CRYPTO_FAILURE  -4

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

unsigned char entropy_input[48];
unsigned char seed[KATNUM][48];

int
main()
{
        struct timeval start;
        struct timeval end;
        struct timeval total;
        struct timeval total_new_key;
        struct timeval total_prev_key;
        struct timeval total_new_enc;
```

```
    struct timeval total_prev_enc;
    struct timeval total_new_dec;
    struct timeval total_prev_dec;

    double total_sum_key = 0.0;
    double total_sum_enc = 0.0;
    double total_sum_dec = 0.0;

FILE                *fp_req, *fp_rsp;
int                 ret_val;
int i;
unsigned char *ct = 0;
unsigned char *ss = 0;
unsigned char *ss1 = 0;
unsigned char *pk = 0;
unsigned char *sk = 0;

for (i=0; i<48; i++)
    entropy_input[i] = i;
randombytes_init(entropy_input, NULL, 256);

for (i=0; i<KATNUM; i++)
    randombytes(seed[i], 48);

fp_req = fdopen(8, "w");
if (!fp_req)
    return KAT_FILE_OPEN_ERROR;

for (i=0; i<KATNUM; i++) {
    fprintf(fp_req, "count = %d\n", i);
    fprintBstr(fp_req, "seed = ", seed[i], 48);
    fprintf(fp_req, "pk =\n");
    fprintf(fp_req, "sk =\n");
    fprintf(fp_req, "ct =\n");
    fprintf(fp_req, "ss =\n\n");
}

fp_rsp = fdopen(9, "w");
if (!fp_rsp)
    return KAT_FILE_OPEN_ERROR;

fprintf(fp_rsp, "# kem/%s\n\n", crypto_kem_PRIMITIVE);

for (i=0; i<KATNUM; i++) {
    if (!ct) ct = malloc(crypto_kem_CIPHERTEXTBYTES);
    if (!ct) abort();
    if (!ss) ss = malloc(crypto_kem_BYTES);
    if (!ss) abort();
    if (!ss1) ss1 = malloc(crypto_kem_BYTES);
    if (!ss1) abort();
    if (!pk) pk = malloc(crypto_kem_PUBLICKEYBYTES);
    if (!pk) abort();
    if (!sk) sk = malloc(crypto_kem_SECRETKEYBYTES);
    if (!sk) abort();

    randombytes_init(seed[i], NULL, 256);
```

```
fprintf(fp_rsp, "count = %d\n", i);
fprintBstr(fp_rsp, "seed = ", seed[i], 48);

// Start of Keypair

gettimeofday(&start, NULL);
if ( (ret_val = crypto_kem_keypair(pk, sk)) != 0) {
    fprintf(stderr, "crypto_kem_keypair returned <%d>\n", ret_val);
    return KAT_CRYPTO_FAILURE;
}
gettimeofday(&end, NULL);
timersub(&end, &start, &total);

if (i==0) {
        total_prev_key = total;
}
else {
        timeradd(&total, &total_prev_key, &total_new_key);
        total_prev_key = total_new_key;
}

fprintBstr(fp_rsp, "pk = ", pk, crypto_kem_PUBLICKEYBYTES);
fprintBstr(fp_rsp, "sk = ", sk, crypto_kem_SECRETKEYBYTES);

// Start of Encryption
gettimeofday(&start, NULL);
if ( (ret_val = crypto_kem_enc(ct, ss, pk)) != 0) {
    fprintf(stderr, "crypto_kem_enc returned <%d>\n", ret_val);
    return KAT_CRYPTO_FAILURE;
}
gettimeofday(&end, NULL);
timersub(&end, &start, &total);

if (i==0) {
        total_prev_enc = total;
}
else {
        timeradd(&total, &total_prev_enc, &total_new_enc);
        total_prev_enc = total_new_enc;
}

fprintBstr(fp_rsp, "ct = ", ct, crypto_kem_CIPHERTEXTBYTES);
fprintBstr(fp_rsp, "ss = ", ss, crypto_kem_BYTES);

fprintf(fp_rsp, "\n");

// Start of Decryption
gettimeofday(&start, NULL);
if ( (ret_val = crypto_kem_dec(ss1, ct, sk)) != 0) {
    fprintf(stderr, "crypto_kem_dec returned <%d>\n", ret_val);
    return KAT_CRYPTO_FAILURE;
}
gettimeofday(&end, NULL);
timersub(&end, &start, &total);
```

```
        if (i==0) {
                total_prev_dec = total;
        }
        else {
                timeradd(&total, &total_prev_dec, &total_new_dec);
                total_prev_dec = total_new_dec;
        }

        if ( memcmp(ss, ss1, crypto_kem_BYTES) ) {
            fprintf(stderr, "crypto_kem_dec returned bad 'ss' value\n");
            return KAT_CRYPTO_FAILURE;
        }
    }


        total_sum_key = (double)total_new_key.tv_sec +
((double)total_new_key.tv_usec * 0.000001);
        total_sum_key /= KATNUM;

        total_sum_enc = (double)total_new_enc.tv_sec +
((double)total_new_enc.tv_usec * 0.000001);
        total_sum_enc /= KATNUM;

        total_sum_dec = (double)total_new_dec.tv_sec +
((double)total_new_dec.tv_usec * 0.000001);
        total_sum_dec /= KATNUM;

        printf("Repeat is: %1d\n", KATNUM);
        printf("Average times key_pair: %fs\n", total_sum_key);
        printf("Average times enc: %fs\n", total_sum_enc);
        printf("Average times dec: %fs\n", total_sum_dec);

    return KAT_SUCCESS;
}

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");
}
```

B.2 Rainbow

rainbow-submission-round2/Reference_Implementation/test-time.c [57]

```
///  @file test-time.c
///  @brief A command-line tool for timing.
///

//#define _BSD_SOURCE
#define _DEFAULT_SOURCE

#include <stdio.h>
#include <stdint.h>

#include "rainbow_config.h"

#include "utils.h"

#include "api.h"

#include <sys/time.h>
#include <stdlib.h>
#include <time.h>


#define NTESTS 1000

int main()
{
 // unsigned char key_a[32], key_b[32];
  //poly r, a, b;
  //unsigned char* sl = (unsigned char*) malloc(NTESTS*CRYPTO_ALGNAME);
  unsigned char* pks = (unsigned char*) malloc(NTESTS*CRYPTO_PUBLICKEYBYTES);
  unsigned char* sks = (unsigned char*) malloc(NTESTS*CRYPTO_SECRETKEYBYTES);
  //unsigned long long t[NTESTS];
 // uint16_t a1 = 0;
  int i;
  printf("-- api --\n\n");

//  for(i=0; i<NTESTS; i++)
//  {
    struct timeval start;
    struct timeval end;
    struct timeval total;
    struct timeval total_new;
    struct timeval total_prev;
    double total_sum = 0.0;
    unsigned long long smlen = 0;
    unsigned long long mlen = 0;
    unsigned char * msg = NULL;
    unsigned char *_sl = malloc( CRYPTO_ALGNAME );
    uint8_t *_sk = (uint8_t*)malloc( CRYPTO_SECRETKEYBYTES );
    uint8_t * pk = (uint8_t *) malloc( CRYPTO_PUBLICKEYBYTES );
    unsigned char * signature = malloc( mlen + CRYPTO_BYTES );
```

```
    /*
    * KEYPAIR
    */
    for (i=0; i<NTESTS; i++) {
      gettimeofday (&start, NULL);
      crypto_sign_keypair(pks+CRYPTO_PUBLICKEYBYTES,
sks+CRYPTO_SECRETKEYBYTES);
        gettimeofday (&end, NULL);
      timersub(&end, &start, &total); // needed &
      // printf("keypair time taken: %d.%06ds\n", total.tv_sec,
total.tv_usec);

      if (i == 0) {
          total_prev = total;
      }
      else
      {
          timeradd(&total, &total_prev, &total_new);
          total_prev = total_new;
      }
    }
    total_sum = (double)total_new.tv_sec + ((double)total_new.tv_usec *
0.000001);
    total_sum /= NTESTS;
    printf("keypair time taken avg: %fs\n", total_sum);

    /*
    * ENC
    */
    total_sum = 0.0; // reset total sum value;
    for (i=0; i<NTESTS; i++) {
      gettimeofday (&start, NULL);
      //insert the function where it uses the signature
      crypto_sign( signature, &smlen, msg , mlen , _sk );
        gettimeofday (&end, NULL);
      timersub(&end, &start, &total); // needed &
      // printf("encoder time taken: %d.%06ds\n", total.tv_sec,
total.tv_usec);

      if (i == 0) {
          total_prev = total;
      }
      else
      {
          timeradd(&total, &total_prev, &total_new);
          total_prev = total_new;
      }
    }
    total_sum = (double)total_new.tv_sec + ((double)total_new.tv_usec *
0.000001);
    total_sum /= NTESTS;
    printf("creating signature time taken avg: %fs\n", total_sum);


     /*
    * verification
```

```
    */
    total_sum = 0.0; // reset total sum value;
    printf("%f\n", total_sum);
    for (i=0; i<NTESTS; i++) {
      gettimeofday (&start, NULL);
      //printf( "%s\n", CRYPTO_ALGNAME );
      //insert the function where it uses the verification
      crypto_sign_open( msg , &mlen , signature , mlen + CRYPTO_BYTES, pk );

        gettimeofday (&end, NULL);
      timersub(&end, &start, &total);


      if (i == 0) {
          total_prev = total;
      }
      else
      {
          timeradd(&total, &total_prev, &total_new);
          total_prev = total_new;
      }
    }
    total_sum = (double)total_new.tv_sec + ((double)total_new.tv_usec *
0.000001);
    total_sum /= NTESTS;
    printf("doing verification time taken avg: %fs\n", total_sum);

    return 0;
  }
```

rainbow-submission-round2/Reference_Implementation/Makefile [57]

```
CC=    gcc
LD=    gcc


#
# The variable `$PROJ_DIR' controls the variant(corresponding to a specific
directory) will be built.
# To build a specific variant, set the $PROJ_DIR to a specific name of the
directory.
#
# For example of building the `Ia_Classic' variant,
# one have to set $PROJ_DIR = Ia_Classic.
# The makefile will compile codes in the `Ia_Classic' directory only.
# No code changes have to be made.
#
# All possible variants are listed as followings.
#

ifndef PROJ_DIR
#PROJ_DIR = Ia_Classic
#PROJ_DIR = Ia_Circumzenithal
#PROJ_DIR = Ia_Compressed
#PROJ_DIR = IIIc_Classic
```

```
#PROJ_DIR = IIIc_Circumzenithal
#PROJ_DIR = IIIc_Compressed
PROJ_DIR = Vc_Classic
#PROJ_DIR = Vc_Circumzenithal
#PROJ_DIR = Vc_Compressed
endif


CFLAGS= -O3 -std=c11 -Wall -Wextra -fno-omit-frame-pointer
INCPATH= -I/usr/local/include -I/opt/local/include -I/usr/include
-I$(PROJ_DIR)
LDFLAGS=
LIBPATH= -L/usr/local/lib -L/opt/local/lib -L/usr/lib
LIBS=    -lcrypto


ifeq ($(shell pwd | tail -c 5),avx2)
CFLAGS += -mavx2
CXXFLAGS += -mavx2
endif

ifeq ($(shell pwd | tail -c 6),ssse3)
CFLAGS += -mssse3
CXXFLAGS += -mssse3
endif



SRCS = $(wildcard $(PROJ_DIR)/*.c)
SRCS_O = $(SRCS:.c=.o)
SRCS_O_ND = $(subst $(PROJ_DIR)/,,$(SRCS_O))

OBJ = $(SRCS_O_ND)

EXE= rainbow-genkey rainbow-sign rainbow-verify PQCgenKAT_sign test-time


CSRC= $(wildcard *.c)


ifdef DEBUG
        CFLAGS=  -D_DEBUG_ -g -O1 -mavx2 -std=c99 -Wall -Wextra
-fsanitize=address -fno-omit-frame-pointer
        CXXFLAGS= -D_DEBUG_ -g -O1 -mavx2 -Wall -Wextra -fno-exceptions
-fno-rtti -nostdinc++
endif

ifdef GPROF
      CFLAGS += -pg
      CXXFLAGS += -pg
      LDFLAGS += -pg
endif

.PHONY: all tests tables clean

all: $(OBJ) $(EXE)
```

```
%-test: $(OBJ) %-test.o
	$(LD) $(LDFLAGS) $(LIBPATH) -o $@ $^ $(LIBS)

%-benchmark: $(OBJ) %-benchmark.o
	$(LD) $(LDFLAGS) $(LIBPATH) -o $@ $^ $(LIBS)

rainbow-genkey: $(OBJ) rainbow-genkey.o
	$(LD) $(LDFLAGS) $(LIBPATH) -o $@ $^ $(LIBS)

rainbow-sign: $(OBJ) rainbow-sign.o
	$(LD) $(LDFLAGS) $(LIBPATH) -o $@ $^ $(LIBS)

rainbow-verify: $(OBJ) rainbow-verify.o
	$(LD) $(LDFLAGS) $(LIBPATH) -o $@ $^ $(LIBS)

PQCgenKAT_sign: $(OBJ) PQCgenKAT_sign.o
	$(LD) $(LDFLAGS) $(LIBPATH) -o $@ $^ $(LIBS)
test-time: $(OBJ) test-time.o
	$(LD) $(LDFLAGS) $(LIBPATH) -o $@ $^ $(LIBS)

%.o: %.c
	$(CC) $(CFLAGS) $(INCPATH) -c $<

%.o: $(PROJ_DIR)/%.c
	$(CC) $(CFLAGS) $(INCPATH) -c $<

clean:
	rm *.o *-test *-benchmark rainbow-genkey rainbow-sign rainbow-verify
PQCgenKAT_sign test-time;
```

Appendix C: Code Modifications for Raspberry Pi 4

C.1 CRYSTALS-Kyber

kyber\ref\test_speed.c [53]

```
#include <stddef.h>
#include <stdint.h>
#include <stdlib.h>
#include <stdio.h>
#include "api.h"
#include "kex.h"
#include "params.h"
#include "indcpa.h"
#include "polyvec.h"
#include "poly.h"
#include "cpucycles.h"
#include "speed_print.h"
#include <sys/time.h>

#define NTESTS 1000

uint64_t t[NTESTS];
uint8_t seed[KYBER_SYMBYTES] = {0};

int main()
{
  unsigned int i;
  unsigned char pk[CRYPTO_PUBLICKEYBYTES];
  unsigned char sk[CRYPTO_SECRETKEYBYTES];
  unsigned char ct[CRYPTO_CIPHERTEXTBYTES];
  unsigned char key[CRYPTO_BYTES];
  unsigned char kexsenda[KEX_AKE_SENDABYTES];
  unsigned char kexsendb[KEX_AKE_SENDBBYTES];
  unsigned char kexkey[KEX_SSBYTES];
  polyvec matrix[KYBER_K];
  poly ap;



    struct timeval start;
    struct timeval end;
    struct timeval total;
    struct timeval total_new;
    struct timeval total_prev;
    double total_sum = 0.0;

/** KEYPAIR **/
  for(i=0;i<NTESTS;i++) {
    gettimeofday (&start, NULL);
    crypto_kem_keypair(pk, sk);
    gettimeofday (&end, NULL);
    timersub(&end, &start, &total); // needed &

    // printf("keypair time taken: %d.%06ds\n", total.tv_sec, total.tv_usec);
```

```
    if (i == 0) {
      total_prev = total;
    }
    else
    {
      timeradd(&total, &total_prev, &total_new);
      total_prev = total_new;
    }
  }
    total_sum = (double)total_new.tv_sec + ((double)total_new.tv_usec *
0.000001);
    total_sum /= NTESTS;
    printf("keypair time taken avg: %fs\n", total_sum);

  //print_results("kyber_keypair: ", t, NTESTS);

/** ENCODE **/
  total_sum = 0.0; // reset total sum value;
  for(i=0;i<NTESTS;i++) {
    gettimeofday (&start, NULL);
    crypto_kem_enc(ct, key, pk);
  gettimeofday (&end, NULL);
      timersub(&end, &start, &total); // needed &

      // printf("encoder time taken: %d.%06ds\n", total.tv_sec,
total.tv_usec);

      if (i == 0) {
          total_prev = total;
      }
      else
      {
          timeradd(&total, &total_prev, &total_new);
          total_prev = total_new;
      }
    }
    total_sum = (double)total_new.tv_sec + ((double)total_new.tv_usec *
0.000001);
    total_sum /= NTESTS;
    printf("encoder time taken avg: %fs\n", total_sum);
  //print_results("kyber_encaps: ", t, NTESTS);

/** DECODE **/
  total_sum = 0.0; // reset total sum value;
  for(i=0;i<NTESTS;i++) {
    gettimeofday (&start, NULL);
    crypto_kem_dec(key, ct, sk);
   gettimeofday (&end, NULL);
      timersub(&end, &start, &total); // needed &

      // printf("decoder time taken: %d.%06ds\n", total.tv_sec,
total.tv_usec);

      if (i == 0) {
          total_prev = total;
```

```
      }
      else
      {
          timeradd(&total, &total_prev, &total_new);
          total_prev = total_new;
      }
    }
    total_sum = (double)total_new.tv_sec + ((double)total_new.tv_usec *
0.000001);
    total_sum /= NTESTS;
    printf("decoder time taken avg: %fs\n", total_sum);
  //print_results("kyber_decaps: ", t, NTESTS);

  return 0;
}
```

kyber\ref\Makefile [53]

```
CC ?= /usr/bin/cc
CFLAGS += -Wall -Wextra -Wpedantic -Wmissing-prototypes -Wredundant-decls \
  -Wshadow -Wpointer-arith -O3 -fomit-frame-pointer
NISTFLAGS += -Wno-unused-result -O3
RM = /bin/rm

SOURCES = kem.c indcpa.c polyvec.c poly.c reduce.c ntt.c cbd.c verify.c
SOURCESKECCAK = $(SOURCES) fips202.c symmetric-shake.c
SOURCESNINETIES = $(SOURCES) sha256.c sha512.c aes256ctr.c symmetric-aes.c
HEADERS = params.h api.h indcpa.h polyvec.h poly.h reduce.h ntt.h cbd.h \
  verify.h symmetric.h randombytes.h
HEADERSKECCAK = $(HEADERS) fips202.h
HEADERSNINETIES = $(HEADERS) aes256ctr.h sha2.h

.PHONY: all speed shared clean

all: \
  test_kyber512 \
  test_kyber768 \
  test_kyber1024 \
  test_kex512 \
  test_kex768 \
  test_kex1024 \
  test_vectors512 \
  test_vectors768 \
  test_vectors1024 \
  test_kyber512-90s \
  test_kyber768-90s \
  test_kyber1024-90s \
  test_kex512-90s \
  test_kex768-90s \
  test_kex1024-90s \
  test_vectors512-90s \
  test_vectors768-90s \
  test_vectors1024-90s \
  PQCgenKAT_kem
```

```
speed: \
  test_speed512 \
  test_speed768 \
  test_speed1024 \
  test_speed512-90s \
  test_speed768-90s \
  test_speed1024-90s

shared: \
  libpqcrystals_kyber512_ref.so \
  libpqcrystals_kyber768_ref.so \
  libpqcrystals_kyber1024_ref.so \
  libpqcrystals_kyber512-90s_ref.so \
  libpqcrystals_kyber768-90s_ref.so \
  libpqcrystals_kyber1024-90s_ref.so \
  libpqcrystals_fips202_ref.so \
  libpqcrystals_sha2_ref.so

libpqcrystals_fips202_ref.so: fips202.c fips202.h
	$(CC) -shared -fPIC $(CFLAGS) fips202.c -o libpqcrystals_fips202_ref.so

libpqcrystals_sha2_ref.so: sha256.c sha512.c sha2.h
	$(CC) -shared -fPIC $(CFLAGS) sha256.c sha512.c -o
libpqcrystals_sha2_ref.so

libpqcrystals_kyber512_ref.so: $(SOURCES) $(HEADERS) symmetric-shake.c
	$(CC) -shared -fPIC $(CFLAGS) -DKYBER_K=2 $(SOURCES) symmetric-shake.c
-o libpqcrystals_kyber512_ref.so

libpqcrystals_kyber768_ref.so: $(SOURCES) $(HEADERS) symmetric-shake.c
	$(CC) -shared -fPIC $(CFLAGS) -DKYBER_K=3 $(SOURCES) symmetric-shake.c
-o libpqcrystals_kyber768_ref.so

libpqcrystals_kyber1024_ref.so: $(SOURCES) $(HEADERS) symmetric-shake.c
	$(CC) -shared -fPIC $(CFLAGS) -DKYBER_K=4 $(SOURCES) symmetric-shake.c
-o libpqcrystals_kyber1024_ref.so

test_kyber512: $(SOURCESKECCAK) $(HEADERSKECCAK) test_kyber.c randombytes.c
	$(CC) $(CFLAGS) -DKYBER_K=2 $(SOURCESKECCAK) randombytes.c test_kyber.c
-o test_kyber512

test_kyber768: $(SOURCESKECCAK) $(HEADERSKECCAK) test_kyber.c randombytes.c
	$(CC) $(CFLAGS) -DKYBER_K=3 $(SOURCESKECCAK) randombytes.c test_kyber.c
-o test_kyber768

test_kyber1024: $(SOURCESKECCAK) $(HEADERSKECCAK) test_kyber.c randombytes.c
	$(CC) $(CFLAGS) -DKYBER_K=4 $(SOURCESKECCAK) randombytes.c test_kyber.c
-o test_kyber1024

test_kex512: $(SOURCESKECCAK) $(HEADERSKECCAK) test_kex.c randombytes.c kex.c
kex.h
	$(CC) $(CFLAGS) -DKYBER_K=2 $(SOURCESKECCAK) randombytes.c kex.c
test_kex.c -o test_kex512

test_kex768: $(SOURCESKECCAK) $(HEADERSKECCAK) test_kex.c randombytes.c kex.c
kex.h
```

```
	$(CC) $(CFLAGS) -DKYBER_K=3 $(SOURCESKECCAK) randombytes.c kex.c
test_kex.c -o test_kex768

test_kex1024: $(SOURCESKECCAK) $(HEADERSKECCAK) test_kex.c randombytes.c kex.c
kex.h
	$(CC) $(CFLAGS) -DKYBER_K=4 $(SOURCESKECCAK) randombytes.c kex.c
test_kex.c -o test_kex1024

test_vectors512: $(SOURCESKECCAK) $(HEADERSKECCAK) test_vectors.c
	$(CC) $(CFLAGS) -DKYBER_K=2 $(SOURCESKECCAK) test_vectors.c -o
test_vectors512

test_vectors768: $(SOURCESKECCAK) $(HEADERSKECCAK) test_vectors.c
	$(CC) $(CFLAGS) -DKYBER_K=3 $(SOURCESKECCAK) test_vectors.c -o
test_vectors768

test_vectors1024: $(SOURCESKECCAK) $(HEADERSKECCAK) test_vectors.c
	$(CC) $(CFLAGS) -DKYBER_K=4 $(SOURCESKECCAK) test_vectors.c -o
test_vectors1024

test_speed512: $(SOURCESKECCAK) $(HEADERSKECCAK) test_speed.c randombytes.c
kex.c kex.h
	$(CC) $(CFLAGS) -DKYBER_K=2 $(SOURCESKECCAK) randombytes.c kex.c
test_speed.c -o test_speed512

test_speed768: $(SOURCESKECCAK) $(HEADERSKECCAK) test_speed.c randombytes.c
kex.c kex.h
	$(CC) $(CFLAGS) -DKYBER_K=3 $(SOURCESKECCAK) randombytes.c kex.c
test_speed.c -o test_speed768

test_speed1024: $(SOURCESKECCAK) $(HEADERSKECCAK) test_speed.c randombytes.c
kex.c kex.h
	$(CC) $(CFLAGS) -DKYBER_K=4 $(SOURCESKECCAK) randombytes.c kex.c
test_speed.c -o test_speed1024

libpqcrystals_kyber512-90s_ref.so: $(SOURCES) $(HEADERS) symmetric-aes.c
	$(CC) -shared -fPIC $(CFLAGS) -DKYBER_K=2 -DKYBER_90S $(SOURCES)
symmetric-aes.c -o libpqcrystals_kyber512-90s_ref.so

libpqcrystals_kyber768-90s_ref.so: $(SOURCES) $(HEADERS) symmetric-aes.c
	$(CC) -shared -fPIC $(CFLAGS) -DKYBER_K=3 -DKYBER_90S $(SOURCES)
symmetric-aes.c -o libpqcrystals_kyber768-90s_ref.so

libpqcrystals_kyber1024-90s_ref.so: $(SOURCES) $(HEADERS) symmetric-aes.c
	$(CC) -shared -fPIC $(CFLAGS) -DKYBER_K=4 -DKYBER_90S $(SOURCES)
symmetric-aes.c -o libpqcrystals_kyber1024-90s_ref.so

test_kyber512-90s: $(SOURCESNINETIES) $(HEADERSNINETIES) test_kyber.c
randombytes.c
	$(CC) $(CFLAGS) -D KYBER_90S -DKYBER_K=2 $(SOURCESNINETIES)
randombytes.c test_kyber.c -o test_kyber512-90s

test_kyber768-90s: $(SOURCESNINETIES) $(HEADERSNINETIES) test_kyber.c
randombytes.c
	$(CC) $(CFLAGS) -D KYBER_90S -DKYBER_K=3 $(SOURCESNINETIES)
randombytes.c test_kyber.c -o test_kyber768-90s
```

```
test_kyber1024-90s: $(SOURCESNINETIES) $(HEADERSNINETIES) test_kyber.c
randombytes.c
	$(CC) $(CFLAGS) -D KYBER_90S -DKYBER_K=4 $(SOURCESNINETIES)
randombytes.c test_kyber.c -o test_kyber1024-90s

test_kex512-90s: $(SOURCESNINETIES) $(HEADERSNINETIES) test_kex.c
randombytes.c fips202.c fips202.h kex.c kex.h
	$(CC) $(CFLAGS) -D KYBER_90S -DKYBER_K=2 $(SOURCESNINETIES)
randombytes.c fips202.c kex.c test_kex.c -o test_kex512-90s

test_kex768-90s: $(SOURCESNINETIES) $(HEADERSNINETIES) test_kex.c
randombytes.c fips202.c fips202.h kex.c kex.h
	$(CC) $(CFLAGS) -D KYBER_90S -DKYBER_K=3 $(SOURCESNINETIES)
randombytes.c fips202.c kex.c test_kex.c -o test_kex768-90s

test_kex1024-90s: $(SOURCESNINETIES) $(HEADERSNINETIES) test_kex.c
randombytes.c fips202.c fips202.h kex.c kex.h
	$(CC) $(CFLAGS) -D KYBER_90S -DKYBER_K=4 $(SOURCESNINETIES)
randombytes.c fips202.c kex.c test_kex.c -o test_kex1024-90s

test_vectors512-90s: $(SOURCESNINETIES) $(HEADERSNINETIES) test_vectors.c
	$(CC) $(CFLAGS) -D KYBER_90S -DKYBER_K=2 $(SOURCESNINETIES)
test_vectors.c -o test_vectors512-90s

test_vectors768-90s: $(SOURCESNINETIES) $(HEADERSNINETIES) test_vectors.c
	$(CC) $(CFLAGS) -D KYBER_90S -DKYBER_K=3 $(SOURCESNINETIES)
test_vectors.c -o test_vectors768-90s

test_vectors1024-90s: $(SOURCESNINETIES) $(HEADERSNINETIES) test_vectors.c
	$(CC) $(CFLAGS) -D KYBER_90S -DKYBER_K=4 $(SOURCESNINETIES)
test_vectors.c -o test_vectors1024-90s

test_speed512-90s: $(SOURCESNINETIES) $(HEADERSNINETIES) test_speed.c
randombytes.c kex.c kex.h fips202.c fips202.h
	$(CC) $(CFLAGS) -D KYBER_90S -DKYBER_K=2 $(SOURCESNINETIES)
randombytes.c kex.c fips202.c test_speed.c -o test_speed512-90s

test_speed768-90s: $(SOURCESNINETIES) $(HEADERSNINETIES) test_speed.c
randombytes.c kex.c kex.h fips202.c fips202.h
	$(CC) $(CFLAGS) -D KYBER_90S -DKYBER_K=3 $(SOURCESNINETIES)
randombytes.c kex.c fips202.c test_speed.c -o test_speed768-90s

test_speed1024-90s: $(SOURCESNINETIES) $(HEADERSNINETIES) test_speed.c
randombytes.c kex.c kex.h fips202.c fips202.h
	$(CC) $(CFLAGS) -D KYBER_90S -DKYBER_K=4 $(SOURCESNINETIES)
randombytes.c kex.c fips202.c test_speed.c -o test_speed1024-90s

PQCgenKAT_kem: $(SOURCESKECCAK) $(HEADERSKECCAK) PQCgenKAT_kem.c rng.c rng.h
	$(CC) $(NISTFLAGS) -o $@ $(SOURCESKECCAK) -I. rng.c PQCgenKAT_kem.c
$(LDFLAGS) -lcrypto

clean:
	-$(RM) -rf *.gcno *.gcda *.lcov *.o *.so
	-$(RM) -rf test_kyber512
	-$(RM) -rf test_kyber768
```

```
	-$(RM) -rf test_kyber1024
	-$(RM) -rf test_kex512
	-$(RM) -rf test_kex768
	-$(RM) -rf test_kex1024
	-$(RM) -rf test_vectors512
	-$(RM) -rf test_vectors768
	-$(RM) -rf test_vectors1024
	-$(RM) -rf test_speed512
	-$(RM) -rf test_speed768
	-$(RM) -rf test_speed1024
	-$(RM) -rf test_kyber512-90s
	-$(RM) -rf test_kyber768-90s
	-$(RM) -rf test_kyber1024-90s
	-$(RM) -rf test_kex512-90s
	-$(RM) -rf test_kex768-90s
	-$(RM) -rf test_kex1024-90s
	-$(RM) -rf test_vectors512-90s
	-$(RM) -rf test_vectors768-90s
	-$(RM) -rf test_vectors1024-90s
	-$(RM) -rf test_speed512-90s
	-$(RM) -rf test_speed768-90s
	-$(RM) -rf test_speed1024-90s
	-$(RM) -rf PQCgenKAT_kem
```

C.2 NTRU

ntru\ref-hps2048509\test\time_test.c

For this program, ref-hps2048509 was referenced, but this code was applied as well to ref-hps2048677, ref-hps4096821 and ref-hrss701. [54]

```
#include "../kem.h"
#include "../params.h"
// Not needed - #include "../cpucycles.h"
#include "../randombytes.h"
#include "../poly.h"
#include "../sample.h"
#include <stdlib.h>
#include <stdio.h>

// new
#include <sys/time.h>

#define NTESTS 1000

int main()
{
  unsigned char key_a[32], key_b[32];
  poly r, a, b;
  unsigned char* pks = (unsigned char*) malloc(NTESTS*NTRU_PUBLICKEYBYTES);
  unsigned char* sks = (unsigned char*) malloc(NTESTS*NTRU_SECRETKEYBYTES);
  unsigned char* cts = (unsigned char*) malloc(NTESTS*NTRU_CIPHERTEXTBYTES);
  // Not needed - unsigned char fgbytes[NTRU_SAMPLE_FG_BYTES];
  // Not needed - unsigned char rmbytes[NTRU_SAMPLE_RM_BYTES];
  unsigned long long t[NTESTS];
  uint16_t a1 = 0;
  int i;

  printf("-- api --\n\n");

//  for(i=0; i<NTESTS; i++)
//  {
    struct timeval start;
    struct timeval end;
    struct timeval total;
    struct timeval total_new;
    struct timeval total_prev;
    double total_sum = 0.0;


    /*
    * KEYPAIR
    */

    for (i=0; i<NTESTS; i++) {
      gettimeofday (&start, NULL);
      crypto_kem_keypair(pks+NTRU_PUBLICKEYBYTES, sks+NTRU_SECRETKEYBYTES);
        gettimeofday (&end, NULL);
```

```
      timersub(&end, &start, &total); // needed &

      // printf("keypair time taken: %d.%06ds\n", total.tv_sec,
total.tv_usec);

      if (i == 0) {
          total_prev = total;
      }
      else
      {
          timeradd(&total, &total_prev, &total_new);
          total_prev = total_new;
      }
    }
    total_sum = (double)total_new.tv_sec + ((double)total_new.tv_usec *
0.000001);
    total_sum /= NTESTS;
    printf("keypair time taken avg: %fs\n", total_sum);

    /*
    * ENCODER
    */
    total_sum = 0.0; // reset total sum value;

    for (i=0; i<NTESTS; i++) {
      gettimeofday (&start, NULL);
      crypto_kem_enc(cts+i*NTRU_CIPHERTEXTBYTES, key_b,
pks+i*NTRU_PUBLICKEYBYTES);
        gettimeofday (&end, NULL);
      timersub(&end, &start, &total); // needed &

      // printf("encoder time taken: %d.%06ds\n", total.tv_sec,
total.tv_usec);

      if (i == 0) {
          total_prev = total;
      }
      else
      {
          timeradd(&total, &total_prev, &total_new);
          total_prev = total_new;
      }
    }
    total_sum = (double)total_new.tv_sec + ((double)total_new.tv_usec *
0.000001);
    total_sum /= NTESTS;
    printf("encoder time taken avg: %fs\n", total_sum);

    /*
    * DECODE
    */
     total_sum = 0.0; // reset total sum value;

    for (i=0; i<NTESTS; i++) {
      gettimeofday (&start, NULL);
      crypto_kem_dec(key_a, cts+i*NTRU_CIPHERTEXTBYTES,
```

```
sks+i*NTRU_SECRETKEYBYTES);
        gettimeofday (&end, NULL);
      timersub(&end, &start, &total); // needed &

      // printf("decoder time taken: %d.%06ds\n", total.tv_sec,
total.tv_usec);

      if (i == 0) {
          total_prev = total;
      }
      else
      {
          timeradd(&total, &total_prev, &total_new);
          total_prev = total_new;
      }
    }
    total_sum = (double)total_new.tv_sec + ((double)total_new.tv_usec *
0.000001);
    total_sum /= NTESTS;
    printf("decoder time taken avg: %fs\n", total_sum);

    return 0;
  }
```

ntru\ref-hps2048509\Makefile

For this program, ref-hps2048509 was referenced, but this code was applied as well to ref-hps2048677, ref-hps4096821 and ref-hrss701. [54]

```
NTRU_NAMESPACE ?= ntru_

CC ?= /usr/bin/cc
CFLAGS = -O3 -fomit-frame-pointer -march=native -fPIC -fPIE -pie
CFLAGS += -Wall -Wextra -Wpedantic
CFLAGS += -DCRYPTO_NAMESPACE\(s\)=${NTRU_NAMESPACE}\#\#s

SRC = cmov.c \
      crypto_sort_int32.c \
      fips202.c \
      kem.c \
      owcpa.c \
      pack3.c \
      packq.c \
      poly.c \
      poly_lift.c \
      poly_mod.c \
      poly_r2_inv.c \
      poly_rq_mul.c \
      poly_s3_inv.c \
      sample.c \
      sample_iid.c
```

```
HDR = cmov.h \
      crypto_hash_sha3256.h \
      crypto_sort_int32.h \
      kem.h \
      owcpa.h \
      params.h \
      poly.h \
      sample.h

SRC_URAND = $(SRC) randombytes.c
HDR_URAND = $(HDR) randombytes.h

SRC_KAT = $(SRC) rng.c PQCgenKAT_kem.c
HDR_KAT = $(HDR) rng.h api.h

all: test/decap \
     test/encap \
     test/keypair \
     test/speed \
     test/test_gp_compat \
     test/test_ntru \
     test/test_owcpa \
     test/test_pack \
     test/test_time

test/test_gp_compat: $(SRC_URAND) $(HDR_URAND) test/test_gp_compat.c
      $(CC) $(CFLAGS) -o $@ $(SRC_URAND) test/test_gp_compat.c

test/test_ntru: $(SRC_URAND) $(HDR_URAND) test/test_ntru.c
      $(CC) $(CFLAGS) -o $@ $(SRC_URAND) test/test_ntru.c

test/test_time: $(SRC_URAND) $(HDR_URAND) test/test_time.c
      $(CC) $(CFLAGS) -o $@ $(SRC_URAND) test/test_time.c

test/test_owcpa: $(SRC_URAND) $(HDR_URAND) test/test_owcpa.c
      $(CC) $(CFLAGS) -o $@ $(SRC_URAND) test/test_owcpa.c

test/test_pack: $(SRC_URAND) $(HDR_URAND) test/test_pack.c
      $(CC) $(CFLAGS) -o $@ $(SRC_URAND) test/test_pack.c

test/speed: $(SRC_URAND) $(HDR_URAND) cpucycles.h cpucycles.c test/speed.c
      $(CC) $(CFLAGS) -o $@ $(SRC_URAND) cpucycles.c test/speed.c

test/gen_owcpa_vecs: test/gen_owcpa_vecs.c
      $(CC) $(CFLAGS) -o $@ $(SRC_URAND) test/gen_owcpa_vecs.c

test/encap: $(SRC_URAND) $(HDR_URAND) test/encap.c
      $(CC) $(CFLAGS) -o $@ $(SRC_URAND) test/encap.c

test/decap: $(SRC_URAND) $(HDR_URAND) test/decap.c
      $(CC) $(CFLAGS) -o $@ $(SRC_URAND) test/decap.c

test/keypair: $(SRC_URAND) $(HDR_URAND) test/keypair.c
      $(CC) $(CFLAGS) -o $@ $(SRC_URAND) test/keypair.c

PQCgenKAT_kem: $(SRC_KAT) $(HDR_KAT)
```

```
	$(CC) $(CFLAGS) -o $@ $(SRC_KAT) -lcrypto $(LDFLAGS)

.PHONY: clean test

test: all
	./test/speed
	./test/test_gp_compat | gp -q
	./test/test_ntru
	./test/test_pack
	./test/test_time

clean:
	-$(RM) *.o
	-$(RM) -r test/decap
	-$(RM) -r test/encap
	-$(RM) -r test/keypair
	-$(RM) -r test/speed
	-$(RM) -r test/test_gp_compat
	-$(RM) -r test/test_ntru
	-$(RM) -r test/test_pack
	-$(RM) -r test/test_time
	-$(RM) PQCgenKAT_kem
	-$(RM) PQCkemKAT_*.req
	-$(RM) PQCkemKAT_*.rsp
```

C.3 SABER

SABER\Reference_Implementation_KEM\test\test_time.c [55]

```
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <time.h>
#include <sys/time.h>
#include <string.h>

#include "../api.h"
#include "../poly.h"
#include "../rng.h"
#include "../SABER_indcpa.h"
#include "../verify.h"


uint64_t clock1,clock2;
uint64_t clock_kp_mv,clock_cl_mv, clock_kp_sm, clock_cl_sm;

static int test_kem_cca()
{


  uint8_t pk[SABER_PUBLICKEYBYTES];
  uint8_t sk[SABER_SECRETKEYBYTES];
  uint8_t c[SABER_BYTES_CCA_DEC];
  uint8_t k_a[SABER_KEYBYTES], k_b[SABER_KEYBYTES];

  unsigned char entropy_input[48];

  uint64_t i, j, repeat;
  repeat=1000;
  uint64_t CLOCK1,CLOCK2;
  uint64_t CLOCK_kp,CLOCK_enc,CLOCK_dec;

      CLOCK1 = 0;
        CLOCK2 = 0;
      CLOCK_kp = CLOCK_enc = CLOCK_dec = 0;
      clock_kp_mv=clock_cl_mv=0;
      clock_kp_sm = clock_cl_sm = 0;



      time_t t;
      // Intializes random number generator
      srand((unsigned) time(&t));

      for (i=0; i<48; i++){
            //entropy_input[i] = rand()%256;
            entropy_input[i] = i;
      }
      randombytes_init(entropy_input, NULL, 256);
```

```
      printf("SABER_INDCPA_PUBLICKEYBYTES=%d\n", SABER_INDCPA_PUBLICKEYBYTES);
      printf("SABER_INDCPA_SECRETKEYBYTES=%d\n", SABER_INDCPA_SECRETKEYBYTES);
      printf("SABER_PUBLICKEYBYTES=%d\n", SABER_PUBLICKEYBYTES);
      printf("SABER_SECRETKEYBYTES=%d\n", SABER_SECRETKEYBYTES);
      printf("SABER_KEYBYTES=%d\n", SABER_KEYBYTES);
      printf("SABER_HASHBYTES=%d\n", SABER_HASHBYTES);
      printf("SABER_BYTES_CCA_DEC=%d\n", SABER_BYTES_CCA_DEC);
      printf("\n");

      struct timeval start;
      struct timeval end;
      struct timeval total;
      struct timeval total_new_key;
      struct timeval total_prev_key;
      struct timeval total_new_enc;
      struct timeval total_prev_enc;
      struct timeval total_new_dec;
      struct timeval total_prev_dec;

      double total_sum_key = 0.0;
      double total_sum_enc = 0.0;
      double total_sum_dec = 0.0;


      for(i=0; i<repeat; i++)
      {
          //printf("i : %lu\n",i);

          //Generation of secret key sk and public key pk pair
          gettimeofday (&start, NULL);
          crypto_kem_keypair(pk, sk);
          gettimeofday (&end, NULL);

          timersub(&end, &start, &total);
          if (i == 0) {
            total_prev_key = total;
          }
          else
          {
              timeradd(&total, &total_prev_key, &total_new_key);
              total_prev_key = total_new_key;
          }



          //Key-Encapsulation call; input: pk; output: ciphertext c,
shared-secret k_a;
          gettimeofday (&start, NULL);
          crypto_kem_enc(c, k_a, pk);
          gettimeofday (&end, NULL);

          timersub(&end, &start, &total);
          if (i == 0) {
            total_prev_enc = total;
          }
```

```
          else
          {
              timeradd(&total, &total_prev_enc, &total_new_enc);
              total_prev_enc = total_new_enc;
          }


          //Key-Decapsulation call; input: sk, c; output: shared-secret k_b;

          gettimeofday (&start, NULL);
          crypto_kem_dec(k_b, c, sk);
          gettimeofday (&end, NULL);

          timersub(&end, &start, &total);
          if (i == 0) {
            total_prev_dec = total;
          }
          else
          {
              timeradd(&total, &total_prev_dec, &total_new_dec);
              total_prev_dec = total_new_dec;
          }

          // Functional verification: check if k_a == k_b?
          for(j=0; j<SABER_KEYBYTES; j++)
          {
            //printf("%u \t %u\n", k_a[j], k_b[j]);
            if(k_a[j] != k_b[j])
            {
                  printf("----- ERR CCA KEM ------\n");
                  return 0;
                  break;
            }
          }
            //printf("\n");
      }

        total_sum_key = (double)total_new_key.tv_sec +
((double)total_new_key.tv_usec * 0.000001);
      total_sum_key /= repeat;

      total_sum_enc = (double)total_new_enc.tv_sec +
((double)total_new_enc.tv_usec * 0.000001);
      total_sum_enc /= repeat;


        total_sum_dec = (double)total_new_dec.tv_sec +
((double)total_new_dec.tv_usec * 0.000001);
      total_sum_dec /= repeat;

      printf("Repeat is : %ld\n",repeat);
      printf("Average times key_pair: \t %fs \n",total_sum_key);
      printf("Average times enc: \t %fs \n",total_sum_enc);
      printf("Average times dec: \t %fs \n",total_sum_dec);
      return 0;
}
```

```
int main()
{
	test_kem_cca();
	return 0;
}
```

SABER\Reference_Implementation_KEM\Makefile

```
CC            = /usr/bin/gcc
CFLAGS  = -Wall -Wextra -Wmissing-prototypes -Wredundant-decls\
	-O3 -fomit-frame-pointer -march=native
NISTFLAGS  = -Wno-unused-result -O3 -fomit-frame-pointer -march=native
-std=c99
CLANG   = clang -march=native -O3 -fomit-frame-pointer -fwrapv
-Qunused-arguments
RM            = /bin/rm


all: test/PQCgenKAT_kem \
     test/test_kex \
     test/kem \

SOURCES = pack_unpack.c poly.c fips202.c verify.c cbd.c SABER_indcpa.c kem.c
HEADERS = SABER_params.h pack_unpack.h poly.h rng.h fips202.h verify.h cbd.h
SABER_indcpa.h

test/test_kex: $(SOURCES) $(HEADERS) rng.o test/test_kex.c
	$(CC) $(CFLAGS) -o $@ $(SOURCES) rng.o test/test_kex.c -lcrypto

test/test_time: $(SOURCES) $(HEADERS) rng.o test/test_time.c
	$(CC) $(CFLAGS) -o $@ $(SOURCES) rng.o test/test_time.c -lcrypto

test/PQCgenKAT_kem: $(SOURCES) $(HEADERS) rng.o test/PQCgenKAT_kem.c
	$(CC) $(NISTFLAGS) -o $@ $(SOURCES) rng.o test/PQCgenKAT_kem.c -lcrypto

test/kem: $(SOURCES) $(HEADERS) rng.o test/kem.c
	$(CC) $(CFLAGS) -o $@ $(SOURCES) rng.o test/kem.c -lcrypto

rng.o: rng.c
	$(CC) $(NISTFLAGS) -c rng.c -lcrypto -o $@

# fips202.o: fips202.c
#	$(CLANG) -c $^ -o $@

.PHONY: clean test

test:
	./test/test_kex
	./test/test_time
	./test/PQCgenKAT_kem
	./test/kem
```

```
clean:
	-$(RM) -f *.o
	-$(RM) -rf test/test_kex
	-$(RM) -rf test/test_time
	-$(RM) -rf test/kem
	-$(RM) -rf test/PQCgenKAT_kem
	-$(RM) -f *.req
	-$(RM) -f *.rsp
```

C.4 CRYSTALS-Dilithium

dilithium\ref\test\test_speed.c [56]

```
#include <stdint.h>
#include "../sign.h"
#include "../poly.h"
#include "../polyvec.h"
#include "../params.h"
#include "cpucycles.h"
#include "speed_print.h"
#include <sys/time.h>

#define NTESTS 1000

uint64_t t[NTESTS];

int main(void)
{
  unsigned int i;
  size_t smlen;
  uint8_t pk[CRYPTO_PUBLICKEYBYTES];
  uint8_t sk[CRYPTO_SECRETKEYBYTES];
  uint8_t sm[CRYPTO_BYTES + CRHBYTES];
  uint8_t seed[CRHBYTES];
  polyvecl mat[K];
  poly *a = &mat[0].vec[0];
  poly *b = &mat[0].vec[1];
  poly *c = &mat[0].vec[2];

    struct timeval start;
    struct timeval end;
    struct timeval total;
    struct timeval total_new;
    struct timeval total_prev;
    double total_sum = 0.0;

    /** KEYPAIR **/
  for(i = 0; i < NTESTS; ++i) {
    gettimeofday (&start, NULL);
    crypto_sign_keypair(pk, sk);
    gettimeofday (&end, NULL);
    timersub(&end, &start, &total); // needed &

    // printf("keypair time taken: %d.%06ds\n", total.tv_sec, total.tv_usec);

    if (i == 0) {
      total_prev = total;
    }
    else
    {
      timeradd(&total, &total_prev, &total_new);
      total_prev = total_new;
    }
  }
```

```
    total_sum = (double)total_new.tv_sec + ((double)total_new.tv_usec *
0.000001);
    total_sum /= NTESTS;
    printf("keypair time taken avg: %fs\n", total_sum);
  // print_results("Keypair:", t, NTESTS);

   /** SIGN **/
   total_sum = 0.0; // reset total sum value;
  for(i = 0; i < NTESTS; ++i) {
    gettimeofday (&start, NULL);
    crypto_sign(sm, &smlen, sm, CRHBYTES, sk);
    gettimeofday (&end, NULL);
      timersub(&end, &start, &total); // needed &

      // printf("sign time taken: %d.%06ds\n", total.tv_sec, total.tv_usec);

      if (i == 0) {
          total_prev = total;
      }
      else
      {
          timeradd(&total, &total_prev, &total_new);
          total_prev = total_new;
      }
    }
    total_sum = (double)total_new.tv_sec + ((double)total_new.tv_usec *
0.000001);
    total_sum /= NTESTS;
    printf("sign time taken avg: %fs\n", total_sum);
  //print_results("Sign:", t, NTESTS);


/** VERIFY **/
total_sum = 0.0; // reset total sum value;
  for(i = 0; i < NTESTS; ++i) {
    gettimeofday (&start, NULL);
    crypto_sign_verify(sm, CRYPTO_BYTES, sm + CRYPTO_BYTES, CRHBYTES, pk);
   gettimeofday (&end, NULL);
      timersub(&end, &start, &total); // needed &

      // printf("verify time taken: %d.%06ds\n", total.tv_sec, total.tv_usec);

      if (i == 0) {
          total_prev = total;
      }
      else
      {
          timeradd(&total, &total_prev, &total_new);
          total_prev = total_new;
      }
    }
    total_sum = (double)total_new.tv_sec + ((double)total_new.tv_usec *
0.000001);
    total_sum /= NTESTS;
    printf("verify time taken avg: %fs\n", total_sum);
  //print_results("Verify:", t, NTESTS);
```

```
  return 0;
}
```

dilithium\ref\Makefile [56]

```
CC ?= /usr/bin/cc
CFLAGS += -Wall -Wextra -Wpedantic -Wmissing-prototypes -Wredundant-decls \
  -Wshadow -Wvla -Wpointer-arith -O3
NISTFLAGS += -Wno-unused-result -O3
SOURCES = sign.c packing.c polyvec.c poly.c ntt.c reduce.c rounding.c
HEADERS = config.h params.h api.h sign.h packing.h polyvec.h poly.h ntt.h \
  reduce.h rounding.h symmetric.h randombytes.h
KECCAK_SOURCES = $(SOURCES) fips202.c symmetric-shake.c
KECCAK_HEADERS = $(HEADERS) fips202.h
AES_SOURCES = $(SOURCES) fips202.c aes256ctr.c symmetric-aes.c
AES_HEADERS = $(HEADERS) fips202.h aes256ctr.h

.PHONY: all speed shared clean

all: \
  test/test_dilithium2 \
  test/test_dilithium3 \
  test/test_dilithium4 \
  test/test_dilithium2aes \
  test/test_dilithium3aes \
  test/test_dilithium4aes \
  test/test_vectors2 \
  test/test_vectors3 \
  test/test_vectors4 \
  test/test_vectors2aes \
  test/test_vectors3aes \
  test/test_vectors4aes \
  PQCgenKAT_sign2 \
  PQCgenKAT_sign3 \
  PQCgenKAT_sign4 \
  PQCgenKAT_sign2aes \
  PQCgenKAT_sign3aes \
  PQCgenKAT_sign4aes

speed: \
  test/test_mul \
  test/test_speed2 \
  test/test_speed3 \
  test/test_speed4 \
  test/test_speed2aes \
  test/test_speed3aes \
  test/test_speed4aes

shared: \
  libpqcrystals_dilithium2_ref.so \
  libpqcrystals_dilithium3_ref.so \
  libpqcrystals_dilithium4_ref.so \
  libpqcrystals_dilithium2aes_ref.so \
```

```
  libpqcrystals_dilithium3aes_ref.so \
  libpqcrystals_dilithium4aes_ref.so \
  libpqcrystals_fips202_ref.so \
  libpqcrystals_aes256ctr_ref.so

libpqcrystals_fips202_ref.so: fips202.c fips202.h
	$(CC) -shared -fPIC $(CFLAGS) -o $@ $<

libpqcrystals_aes256ctr_ref.so: aes256ctr.c aes256ctr.h
	$(CC) -shared -fPIC $(CFLAGS) -o $@ $<

libpqcrystals_dilithium2_ref.so: $(SOURCES) $(HEADERS) symmetric-shake.c
	$(CC) -shared -fPIC $(CFLAGS) -DDILITHIUM_MODE=2 \
	  -o $@ $(SOURCES) symmetric-shake.c

libpqcrystals_dilithium3_ref.so: $(SOURCES) $(HEADERS) symmetric-shake.c
	$(CC) -shared -fPIC $(CFLAGS) -DDILITHIUM_MODE=3 \
	  -o $@ $(SOURCES) symmetric-shake.c

libpqcrystals_dilithium4_ref.so: $(SOURCES) $(HEADERS) symmetric-shake.c
	$(CC) -shared -fPIC $(CFLAGS) -DDILITHIUM_MODE=4 \
	  -o $@ $(SOURCES) symmetric-shake.c

libpqcrystals_dilithium2aes_ref.so: $(SOURCES) $(HEADERS) symmetric-aes.c
	$(CC) -shared -fPIC $(CFLAGS) -DDILITHIUM_MODE=2 -DDILITHIUM_USE_AES \
	   -o $@ $(SOURCES) symmetric-aes.c

libpqcrystals_dilithium3aes_ref.so: $(SOURCES) $(HEADERS) symmetric-aes.c
	$(CC) -shared -fPIC $(CFLAGS) -DDILITHIUM_MODE=3 -DDILITHIUM_USE_AES \
	   -o $@ $(SOURCES) symmetric-aes.c

libpqcrystals_dilithium4aes_ref.so: $(SOURCES) $(HEADERS) symmetric-aes.c
	$(CC) -shared -fPIC $(CFLAGS) -DDILITHIUM_MODE=4 -DDILITHIUM_USE_AES \
	   -o $@ $(SOURCES) symmetric-aes.c

test/test_dilithium2: test/test_dilithium.c randombytes.c $(KECCAK_SOURCES) \
  $(KECCAK_HEADERS)
	$(CC) $(CFLAGS) -DDILITHIUM_MODE=2 \
	  -o $@ $< randombytes.c $(KECCAK_SOURCES)

test/test_dilithium3: test/test_dilithium.c randombytes.c $(KECCAK_SOURCES) \
  $(KECCAK_HEADERS)
	$(CC) $(CFLAGS) -DDILITHIUM_MODE=3 \
	  -o $@ $< randombytes.c $(KECCAK_SOURCES)

test/test_dilithium4: test/test_dilithium.c randombytes.c $(KECCAK_SOURCES) \
  $(KECCAK_HEADERS)
	$(CC) $(CFLAGS) -DDILITHIUM_MODE=4 \
	  -o $@ $< randombytes.c $(KECCAK_SOURCES)

test/test_dilithium2aes: test/test_dilithium.c randombytes.c $(AES_SOURCES) \
  $(AES_HEADERS)
	$(CC) $(CFLAGS) -DDILITHIUM_MODE=2 -DDILITHIUM_USE_AES \
	  -o $@ $< randombytes.c $(AES_SOURCES)

test/test_dilithium3aes: test/test_dilithium.c randombytes.c $(AES_SOURCES) \
```

```
  $(AES_HEADERS)
	$(CC) $(CFLAGS) -DDILITHIUM_MODE=3 -DDILITHIUM_USE_AES \
	  -o $@ $< randombytes.c $(AES_SOURCES)

test/test_dilithium4aes: test/test_dilithium.c randombytes.c $(AES_SOURCES) \
  $(AES_HEADERS)
	$(CC) $(CFLAGS) -DDILITHIUM_MODE=4 -DDILITHIUM_USE_AES \
	  -o $@ $< randombytes.c $(AES_SOURCES)

test/test_vectors2: test/test_vectors.c rng.c rng.h $(KECCAK_SOURCES) \
  $(KECCAK_HEADERS)
	$(CC) $(NISTFLAGS) -DDILITHIUM_MODE=2 \
	  -o $@ $< rng.c $(KECCAK_SOURCES) $(LDFLAGS) -lcrypto

test/test_vectors3: test/test_vectors.c rng.c rng.h $(KECCAK_SOURCES) \
  $(KECCAK_HEADERS)
	$(CC) $(NISTFLAGS) -DDILITHIUM_MODE=3 \
	  -o $@ $< rng.c $(KECCAK_SOURCES) $(LDFLAGS) -lcrypto

test/test_vectors4: test/test_vectors.c rng.c rng.h $(KECCAK_SOURCES) \
  $(KECCAK_HEADERS)
	$(CC) $(NISTFLAGS) -DDILITHIUM_MODE=4 \
	  -o $@ $< rng.c $(KECCAK_SOURCES) $(LDFLAGS) -lcrypto

test/test_vectors2aes: test/test_vectors.c rng.c rng.h $(AES_SOURCES) \
  $(AES_HEADERS)
	$(CC) $(NISTFLAGS) -DDILITHIUM_MODE=2 -DDILITHIUM_USE_AES \
	  -o $@ $< rng.c $(AES_SOURCES) $(LDFLAGS) -lcrypto

test/test_vectors3aes: test/test_vectors.c rng.c rng.h $(AES_SOURCES) \
  $(AES_HEADERS)
	$(CC) $(NISTFLAGS) -DDILITHIUM_MODE=3 -DDILITHIUM_USE_AES \
	  -o $@ $< rng.c $(AES_SOURCES) $(LDFLAGS) -lcrypto

test/test_vectors4aes: test/test_vectors.c rng.c rng.h $(AES_SOURCES) \
  $(AES_HEADERS)
	$(CC) $(NISTFLAGS) -DDILITHIUM_MODE=4 -DDILITHIUM_USE_AES \
	  -o $@ $< rng.c $(AES_SOURCES) $(LDFLAGS) -lcrypto

test/test_speed2: test/test_speed.c randombytes.c $(KECCAK_SOURCES) \
  $(KECCAK_HEADERS)
	$(CC) $(CFLAGS) -DDILITHIUM_MODE=2 \
	  -o $@ $< randombytes.c \
	  $(KECCAK_SOURCES)

test/test_speed3: test/test_speed.c randombytes.c $(KECCAK_SOURCES) \
  $(KECCAK_HEADERS)
	$(CC) $(CFLAGS) -DDILITHIUM_MODE=3 \
	  -o $@ $< randombytes.c \
	  $(KECCAK_SOURCES)

test/test_speed4: test/test_speed.c randombytes.c $(KECCAK_SOURCES) \
  $(KECCAK_HEADERS)
	$(CC) $(CFLAGS) -DDILITHIUM_MODE=4 \
	  -o $@ $< randombytes.c \
	  $(KECCAK_SOURCES)
```

```
test/test_speed2aes: test/test_speed.c randombytes.c $(AES_SOURCES)
$(AES_HEADERS)
	$(CC) $(CFLAGS) -DDILITHIUM_MODE=2 -DDILITHIUM_USE_AES \
	  -o $@ $< randombytes.c \
	  $(AES_SOURCES)

test/test_speed3aes: test/test_speed.c randombytes.c $(AES_SOURCES)
$(AES_HEADERS)
	$(CC) $(CFLAGS) -DDILITHIUM_MODE=3 -DDILITHIUM_USE_AES \
	  -o $@ $< randombytes.c \
	  $(AES_SOURCES)

test/test_speed4aes: test/test_speed.c randombytes.c $(AES_SOURCES)
$(AES_HEADERS)
	$(CC) $(CFLAGS) -DDILITHIUM_MODE=4 -DDILITHIUM_USE_AES \
	  -o $@ $< randombytes.c \
	  $(AES_SOURCES)

test/test_mul: test/test_mul.c randombytes.c $(KECCAK_SOURCES)
$(KECCAK_HEADERS)
	$(CC) $(CFLAGS) -UDBENCH -o $@ $< randombytes.c $(KECCAK_SOURCES)

PQCgenKAT_sign2: PQCgenKAT_sign.c rng.c rng.h $(KECCAK_SOURCES) \
  $(KECCAK_HEADERS)
	$(CC) $(NISTFLAGS) -DDILITHIUM_MODE=2 \
	  -o $@ $< rng.c $(KECCAK_SOURCES) $(LDFLAGS) -lcrypto

PQCgenKAT_sign3: PQCgenKAT_sign.c rng.c rng.h $(KECCAK_SOURCES) \
  $(KECCAK_HEADERS)
	$(CC) $(NISTFLAGS) -DDILITHIUM_MODE=3 \
	  -o $@ $< rng.c $(KECCAK_SOURCES) $(LDFLAGS) -lcrypto

PQCgenKAT_sign4: PQCgenKAT_sign.c rng.c rng.h $(KECCAK_SOURCES) \
  $(KECCAK_HEADERS)
	$(CC) $(NISTFLAGS) -DDILITHIUM_MODE=4 \
	  -o $@ $< rng.c $(KECCAK_SOURCES) $(LDFLAGS) -lcrypto

PQCgenKAT_sign2aes: PQCgenKAT_sign.c rng.c rng.h $(AES_SOURCES) \
  $(AES_HEADERS)
	$(CC) $(NISTFLAGS) -DDILITHIUM_MODE=2 -DDILITHIUM_USE_AES \
	  -o $@ $< rng.c $(AES_SOURCES) $(LDFLAGS) -lcrypto

PQCgenKAT_sign3aes: PQCgenKAT_sign.c rng.c rng.h $(AES_SOURCES) \
  $(AES_HEADERS)
	$(CC) $(NISTFLAGS) -DDILITHIUM_MODE=3 -DDILITHIUM_USE_AES \
	  -o $@ $< rng.c $(AES_SOURCES) $(LDFLAGS) -lcrypto

PQCgenKAT_sign4aes: PQCgenKAT_sign.c rng.c rng.h $(AES_SOURCES) \
  $(AES_HEADERS)
	$(CC) $(NISTFLAGS) -DDILITHIUM_MODE=4 -DDILITHIUM_USE_AES \
	  -o $@ $< rng.c $(AES_SOURCES) $(LDFLAGS) -lcrypto

clean:
	rm -f *~ test/*~ *.gcno *.gcda *.lcov
	rm -f libpqcrystals_dilithium2_ref.so
```

```
	rm -f libpqcrystals_dilithium3_ref.so
	rm -f libpqcrystals_dilithium4_ref.so
	rm -f libpqcrystals_dilithium2aes_ref.so
	rm -f libpqcrystals_dilithium3aes_ref.so
	rm -f libpqcrystals_dilithium4aes_ref.so
	rm -f libpqcrystals_fips202_ref.so
	rm -f libpqcrystals_aes256ctr_ref.so
	rm -f test/test_dilithium2
	rm -f test/test_dilithium3
	rm -f test/test_dilithium4
	rm -f test/test_dilithium2aes
	rm -f test/test_dilithium3aes
	rm -f test/test_dilithium4aes
	rm -f test/test_vectors2
	rm -f test/test_vectors3
	rm -f test/test_vectors4
	rm -f test/test_vectors2aes
	rm -f test/test_vectors3aes
	rm -f test/test_vectors4aes
	rm -f test/test_speed2
	rm -f test/test_speed3
	rm -f test/test_speed4
	rm -f test/test_speed2aes
	rm -f test/test_speed3aes
	rm -f test/test_speed4aes
	rm -f test/test_mul
	rm -f PQCgenKAT_sign2
	rm -f PQCgenKAT_sign3
	rm -f PQCgenKAT_sign4
	rm -f PQCgenKAT_sign2aes
	rm -f PQCgenKAT_sign3aes
	rm -f PQCgenKAT_sign4aes
```