Hardware Accelerator for NewHope Algorithm

Saarila Kenkare

Abstract

Encryption is an important tool used to protect data, and we need to ensure that the algorithms we use to verify that data is protected are secure. Some operations are considered hard, and many encryption algorithms are built around that assumption. However, some of these assumptions are broken when a quantum computer is doing the operations. Because of this, the National Institute of Standards and Technology has started a competition for people to develop encryption algorithms that they believe to be secure from attacks by a quantum computer. I analyzed and compared the NIST Competition’s Round 2 key exchange algorithms. I then selected one of these, NewHope, to implement in Chisel, a hardware design language. In this thesis, we discuss post-quantum cryptography, the algorithms that made it to Round 2 of the NIST competition, the NewHope algorithm, and my implementation of it.

1 Introduction

We rely on encryption for many of the world’s most important systems. From healthcare to financial records and much more, it is crucial that our data is protected. The algorithms used to protect this data are called cryptographic algorithms, which rely on certain assumptions; for example, the RSA algorithm relies on the assumption that factoring a number into its primes is hard for a computer to solve, so is safe from hackers [1]. However, many of these assumptions are untrue if the computer used to attack the algorithm was a quantum computer; in the RSA example, a quantum computer could solve factoring numbers into their primes relatively easily [2].

Most of the algorithms currently used for encryption across all systems are not quantum-proof. NIST started a competition for post-quantum cryptographic algorithms to pick standards, and various research teams from companies and universities have been submitting their algorithms that are free from quantum attacks [3].

For this thesis, I picked one specific algorithm submitted to this competition, NewHope [4], implemented it in Chisel [5], a hardware design language, and evaluated
it using post-synthesis tools. This thesis will discuss various PQC algorithms, my implementation of NewHope and its results.

2 Background

2.1 Post-Quantum Cryptography

Many of the encryption algorithms we use rely on problems that we assume are hard for computers. However, some of these problems are susceptible to quantum attacks. Where standard computers use bits, which can either store 0 or 1 as a value, quantum computers use qubits, which behave in a probabilistic manner, and know about the status of all other qubits in the system. This contributes to quantum computers being faster than normal computers. Therefore, if a hacker got access to a quantum computer, many of our systems would break. To aim to create new standards to address this problem, NIST has a competition currently in Round 2 for various companies, universities, and individuals to submit algorithms that they believe to be safe from attacks by quantum computers.

2.2 Key Exchange

A type of cryptographic algorithm represented in the NIST competition is a key exchange protocol. A specific type of key exchange protocol is a key agreement protocol, in which a sender and receiver exchange cryptographic keys to ensure that a secure communication channel can be established [6].

The three steps of a key exchange algorithm are key generation, key encapsulation, and key decapsulation. Figure 1 describes the flow among parameters of these three modules. Key generation outputs both a public key and a secret key. The public key is used in key encapsulation to create the shared secret and a ciphertext. The ciphertext and secret key are then used in key decapsulation to also create the shared secret. When the shared secrets match, the key exchange is complete.
2.3 NIST Round 2 Algorithms

NIST is accepting algorithms of a couple types: key exchange algorithms, which are discussed above, and signature algorithms, which allow a sender to “sign” a message before sending it to a receiver. This paper will discuss an implementation of NewHope, one of the NIST key exchange submissions, and this section will talk about all of the key exchange algorithms which qualified for Round 2 of the competition.

When deciding which algorithm to implement, I compared the performance of all of the Round 2 submissions to see which ones were significantly different. The below table has each algorithm’s public key size for the version of the algorithm
that meets Level 1 security standards, the equivalent of AES-128. The units for latency are Haswell cycles, using AVX2 (Advanced Vector instructions, allowing for optimization), divided into the three steps of a key exchange algorithm. The type of the algorithm is also mentioned, and explanations of what each of these mean will be discussed in the following sections. I also got the NIST submission source code for each algorithm and timed how long it took to run one instance of the three functions mentioned as a measure of wall clock time, to get data that I could compare to my own implementation later. There were three algorithms that I could not gather this data for due to an inability to run the submission code.

As shown in the table, there is a wide range of public key sizes and latencies across various algorithms, sometimes with a tradeoff. For example, SIKE [7] has the second smallest public key size of any algorithm submitted; however, it has one of the largest latencies. NewHope, on the other hand, has the smallest total latency across the whole algorithm and still has a mid-range public key size. Based on these statistics and the fact that I was interested in the math behind NewHope, I chose to implement this algorithm.
<table>
<thead>
<tr>
<th>Algorithm Name</th>
<th>Type of Algorithm</th>
<th>Public Key Size</th>
<th>Latency</th>
<th>Wall Clock Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRYSTALS-KYBER [8]</td>
<td>Lattice</td>
<td>800 bytes</td>
<td>33K KG, 49K KE, 40K KD</td>
<td>0.041s</td>
</tr>
<tr>
<td>FrodoKEM [9]</td>
<td>Lattice</td>
<td>9616 bytes</td>
<td>1.3M KG, 1.8M KE, 1.7 KD</td>
<td>0.0041s</td>
</tr>
<tr>
<td>LAC [10]</td>
<td>Lattice</td>
<td>544 bytes</td>
<td>59K KG, 89K KE, 140K KD</td>
<td>0.0012s</td>
</tr>
<tr>
<td>NewHope [11]</td>
<td>Lattice</td>
<td>928 bytes</td>
<td>58K KG, 88K KE, 20K KD</td>
<td>0.025s</td>
</tr>
<tr>
<td>NTRU [12]</td>
<td>Lattice</td>
<td>699 bytes</td>
<td>12M KG, 760K KE, 1.9M KD*</td>
<td>0.127s</td>
</tr>
<tr>
<td>NTRU Prime [13]</td>
<td>Lattice</td>
<td>994 bytes</td>
<td>46K KG, 59K KE, 750K KD*</td>
<td>0.184s</td>
</tr>
<tr>
<td>Round5 [14]</td>
<td>Lattice</td>
<td>634 bytes</td>
<td>47K KG, 66K KE, 37K KD</td>
<td>0.944s</td>
</tr>
<tr>
<td>SABER [15]</td>
<td>Lattice</td>
<td>672 bytes</td>
<td>101K KG, 125K KE, 129K KD</td>
<td>0.00168s</td>
</tr>
<tr>
<td>Three Bears [16]</td>
<td>Lattice</td>
<td>804 bytes</td>
<td>41K KG, 62K KE, 28K KD*</td>
<td>0.00632s</td>
</tr>
<tr>
<td>BIKE [17]</td>
<td>Code-Based</td>
<td>2540 bytes</td>
<td>600K KG, 220K KE, 2220K KD*</td>
<td>0.00692s</td>
</tr>
<tr>
<td>Classic McEliece [18]</td>
<td>Code-Based</td>
<td>261120 bytes</td>
<td>1.5M KG, 2K KE, 15K KD*</td>
<td>N/A</td>
</tr>
<tr>
<td>HQC [19]</td>
<td>Code-Based</td>
<td>3024 bytes</td>
<td>175K KG, 286K KE, 486K KD</td>
<td>N/A</td>
</tr>
<tr>
<td>LEDAcrypt [20]</td>
<td>Code-Based</td>
<td>1368 bytes</td>
<td>1.2M KG, 1M KE, 850K KD</td>
<td>N/A</td>
</tr>
<tr>
<td>NTS-KEM [21]</td>
<td>Code-Based</td>
<td>319488 bytes</td>
<td>234K KG, 118K KE, 246K KD*</td>
<td>0.291s</td>
</tr>
<tr>
<td>ROLLO [22]</td>
<td>Code-Based</td>
<td>696 bytes</td>
<td>17M KG, 1.3M KE, 4.4M KD*</td>
<td>0.00327s</td>
</tr>
<tr>
<td>RQC [23]</td>
<td>Code-Based</td>
<td>1834 bytes</td>
<td>370K KG, 530K KE, 2.58M KD</td>
<td>0.00765s</td>
</tr>
<tr>
<td>SIKE</td>
<td>Supersingular Elliptic Curve Isogeny</td>
<td>564 bytes</td>
<td>718M KG, 1175M KE, 1254M KD*</td>
<td>0.450s</td>
</tr>
</tbody>
</table>

*The algorithm did not have an implementation using AVX2 instructions completed, so the reference implementation’s latency was used.
2.4 Algorithm Types

There are several general types of cryptographic algorithms submitted to NIST, and three specific types for key-exchange mechanisms: lattice-based algorithms, code-based algorithms, and supersingular elliptic curve isogeny algorithms. While the specific mathematical problem an algorithm is based on can differ, algorithms of the same type will involve similar basic concepts that they are built upon. The following sections will discuss the three types of key-exchange algorithms submitted to NIST.

2.4.1 Lattice Algorithms

Nine out of the seventeen NIST Round 2 key-exchange algorithms are lattice-based. In mathematics, a lattice is comprised of all linear combinations with integer coefficients of the basis vectors of any subset of the real numbers, and lattice-based cryptographic algorithms involve these constructions [24]. Some well-studied computational lattice problems do not have an efficient solution, so many lattice-based constructions are considered to be resistant to quantum attacks.

2.4.2 Code-Based Algorithms

Seven out of the seventeen NIST Round 2 key-exchange algorithms are code-based. Code-based cryptography relies on the hardness of decoding a linear error correcting code. One of the most well-known of this class of algorithms was proposed by Robert McEliece, and was the first asymmetric encryption algorithm to use randomization. While it never gained much acceptance in the cryptographic community, it is believed to be quantum-resistant and has been accepted as an NIST Round 2 algorithm. While I did not implement a code-based algorithms, this class of algorithms is considered promising for finding quantum-proof solutions to the key-exchange problem.

2.4.3 Supersingular Elliptic Curve Isogeny Algorithms

One of the seventeen NIST Round 2 key-exchange algorithms is SIKE, or Supersingular Isogeny Key Encapsulation. This is a post-quantum version of the Diffie-Hellman key exchange [25]. In this protocol, two parties publicly agree on a starting value, encrypt it with their own private key, then publicly share the encrypted values. Next, each party encrypts the received value again with its own private key, and the two results should be the same. SIKE uses a similar concept, but is based on a supersingular isogeny graph, determined by choosing a small and large prime number and considering the class of all elliptic curves defined over the finite field. Unlike
Ring-LWE algorithms, SIKE supports perfect forward secrecy, which protects past exchanges from future compromises of the system.

2.5 NewHope

NewHope, created by Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe, uses the Ring-Learning with Errors problem for its key exchange protocol. It has made it past Round 2 of the NIST competition and has been implemented in multiple different languages, and even adapted by Google as part of a post-quantum cryptography experiment. In this section, I will discuss the background of NewHope’s protocol and its applications.

2.5.1 Learning With Errors

NewHope relies on the Ring-Learning With Errors problem, a modification of the Learning With Errors problem. Learning With Errors is a computational problem for which no quantum attacks are known [26]. To start, a secret key $s$, which is a matrix, and a second value $e$ are chosen. Then, a vector of values called $A[]$ are chosen, and compute the equation $B[] = A[] \ast s + e$. Overall, the LWE problem is finding the solution to

$$B[] = A[] \ast s + e$$

where $A$ and $B$ are known. $A$ and $B$ are the public key and the matrix $s$ becomes the secret key.

Lattice problems are optimization problems relating to lattices, comprised of all linear combinations with integer coefficients of the basis vectors of any subset of the real numbers. They are NP-hard, and the ones classified as worst-case hard can be used for extremely secure cryptographic schemes as they are assumed to be quantum-resistant. Assuming quantum resistance of lattice problems, the LWE problem is also quantum resistant and is as hard to solve as the worst-case hard lattice problems.

2.5.2 Ring Learning With Errors Key Exchange

The foundation of the NewHope algorithm is the computational RLWE problem, which is the Learning with Errors problem specialized to polynomial rings [27]. The set of all polynomials with coefficients from a finite field, where polynomials can be summed and multiplied, creates an infinite polynomial ring. A major advantage that RLWE has over LWE in cryptography is key size: for a 128-bit security scheme, an
RLWE cryptographic algorithm would use a public key about seven thousand bits in length while a LWE cryptographic scheme would use a public key about 49 million bits in length. In general, RLWE key sizes are about the size of the square root of LWE key sizes. However, RLWE key sizes are still larger than those of RSA and Diffie-Hellman, two public key schemes susceptible to quantum computer attacks.

2.5.3 Algorithm Details

The creators of the NewHope algorithm made decisions in designing the algorithm. One of these is how the base vector is generated, using the SHAKE algorithm from the Secure Hash Algorithm family. This prevents a “back-door” value from being used, so the algorithm is more secure. This is a difference from Diffie-Hellman which can be compromised through the Logjam attack due to many implementations using the same pregenerated value. In addition, RLWE schemes before NewHope did error correction one coefficient of the polynomial at a time, where NewHope does it 2 or 4 coefficients at a time using high-dimension geometry. Because of this, there is a higher rate of decryption success.

2.5.4 NewHope Applications

NewHope’s creators provided a reference implementation done in C with their NIST submission. In addition, there have been various other implementations, which will be discussed here.

An implementation has been done of NewHope in Java by a developer not affiliated with the creators of NewHope [28]. In addition, a paper has been published about an implementation of NewHope-Simple, a variant of the algorithm, done on Xilinx FPGA [29]. The paper’s conclusion is that NewHope-Simple runs well on hardware devices and is promising as a quantum key exchange.

In 2016, Google decided to work on protecting Google Chrome against quantum computers by implementing a post-quantum cryptographic algorithm in an experimental version of the browser [30]. In Chrome Canary, a testing ground for the browser, a version of the NewHope algorithm was enabled. The Google team working on this chose NewHope because it seemed to be the most promising to them at the time. This version of the algorithm was called CECPQ1, and it combined NewHope with X25519, which ensured that if NewHope broke, the existing security of the browser would still exist. CECPQ1 was not intended to be a standard, but rather, a means of gathering information about post-quantum cryptography. This feature was available in Chrome beta and was later disabled and succeeded by CECPQ2 in 2019, which did not use NewHope.
3 Accelerator Implementation

3.1 Code

My repository containing the Chisel code is here: [https://github.com/saarilakenkare/NewHope_Generator](https://github.com/saarilakenkare/NewHope_Generator). All classes but the main, NewHope, are not reliant on any non-standard libraries. The main class is integrated with an RoCC accelerator, as explained later in section 3.3.

3.2 Chisel

My version of the NewHope algorithm was done in Chisel. Chisel is an open-source hardware design language built on top of Scala by UC Berkeley, so the code can be written in an object-oriented manner. When compiled, Chisel generates Verilog code, which can be used for analysis.

3.3 RoCC Accelerator

The accelerator developed is an RoCC accelerator following the RISC-V architecture. RISC-V is a new Instruction Set Architecture intended for education and research. One implementation of this is called Rocket, and RoCC is an interface designed to attach accelerators to Rocket. This is one way to write a program with RISC-V. Using the RoCC interface, a user can provide inputs to the accelerator and receive an output. The accelerator developer will receive commands and corresponding source and destination registers. In addition, the user can provide addresses to variables, and the accelerator can read from and write to memory using a special request/response interface.

I wrote an implementation of NewHope as an RoCC accelerator in Chipyard, a UCB framework for defining hardware. I used the language Chisel, a hardware design language built on Scala. A user can write a C program sending instructions to the accelerator, which can be compiled to a binary that runs with a simulator of the accelerator. The instruction includes a signal to start the full key exchange process.

NewHope relies heavily on polynomial operations, and so polynomials are passed from module to module as Vectors of Size 512 elements, each 16 bits wide. Each element can be directly accessed and manipulated. The algorithm also relies on several constant arrays necessary for transformations, and these are all stored as constant Scala arrays, which ended up compiling faster than storing them as constant Chisel vectors. The separate parts of the algorithm, such as a function to add
polynomials and a function which computes an NTT transform, are implemented in individual Chisel modules.

### 3.4 Key Generation

Key generation starts with a randomly generated seed. Using shake256, a variation on SHA-3, this is expanded, and various polynomial algorithms are done to result in a public key and secret key. The core of these algorithms are the NTT transforms and operations like adding and multiplying polynomials, which can be done simply in Chisel through Vectors. Figure 2 shows the full order of operations in key generation, and the psuedocode below shows the steps of the algorithm. It uses NTT transforms and vector transformations, as well as linear operations.

NewHope Key Generation

begin
  seed := random(64)
  expanded := shake256(64, seed)
  public_seed := expanded[0 : 31]
  noise_seed := expanded[32 : 63]
  poly_a := gen_a(public_seed)
  poly_s := poly_sample(noise_seed, 0)
  poly_s := poly_ntt(poly_s)
  poly_c := poly_sample(noise_seed, 1)
  poly_c := poly_ntt(poly_c)
  poly_b := poly_a \ast poly_s + poly_c
  secret_key := serialize_poly(poly_s)
  public_key := serialize_public_key(poly_b, public_seed)
end
3.5 Key Encapsulation

Key encapsulation generates the shared secret and the ciphertext. The shared secret is generated by the shake256 algorithm on a random sequence, while the ciphertext is created through an algorithm made up of several polynomial components, similar to key generation, manipulated through NTT transforms and samples, and combined through addition and multiplication operations. The ciphertext is used in decapsulation, while the shared secret is used to verify the key exchange worked.

NewHope Key Encapsulation
begin
  seed := random(64)
  expanded := shake256(64, seed)
  poly_v := message_to_polynomial(expanded)
  poly, public_seed := decode_public_key(public_key)
\[
\text{poly}_a := \text{gen}_a(\text{public\_seed}) \\
\text{poly}_s := \text{poly\_ntt}(\text{poly\_sample}(\text{expanded}, 0)) \\
\text{poly}_e := \text{poly\_ntt}(\text{poly\_sample}(\text{expanded}, 1)) \\
\text{poly}_e' := \text{poly\_sample}(\text{expanded}, 2) \\
\text{poly}_u := \text{multiply\_polys}(\text{poly}_a, \text{poly}_s) \\
\text{poly}_u := \text{add\_polys}(\text{poly}_u, \text{poly}_e) \\
\text{poly}_v' := \text{poly\_inv\_ntt}(\text{poly}_b \ast \text{poly}_s) + \text{poly}_e' + \text{poly}_v \\
\text{ciphertext} := \text{encode\_cipher}(\text{poly}_u, \text{poly}_v') \\
\text{shared\_secret} := \text{shake256}(32, \text{expanded})
\]

\text{end}

Figure 3: New Hope Key Encapsulation Flow
3.6 Key Decapsulation

Key decapsulation uses the ciphertext from key encapsulation and the secret key from key generation to generate the shared secret, which should match the shared secret from key encapsulation. This relies on algorithms that “undo” some of the operations in the past two steps; for example, decompressing and decoding a polynomial and an NTT inverse transform.

NewHope Key Decapsulation

\[
\begin{align*}
\text{poly}_s & := \text{decode\_polynomial}(\text{secret\_key}) \\
\text{poly}_u & := \text{decode\_polynomial}(\text{ciphertext}) \\
\text{poly}_v & := \text{decompress\_polynomial}(\text{ciphertext}[-32]) \\
\text{message} & := \text{polynomial\_to\_message}(\text{poly\_nvtt}(\text{poly}_s \ast \text{poly}_u) - \text{poly}_v) \\
\text{shared\_secret} & := \text{shake256}(32, \text{message}) 
\end{align*}
\]

Figure 4: New Hope Key Decapsulation Flow
3.7 Verification

The creators of NewHope provided a reference implementation in C, and I also developed my own C implementation to better understand the algorithm before working in Chisel. Using these two tools, I verified correctness of the algorithm by ensuring that the same input of a random seed would lead to the same outputs.

3.8 Results

Upon compilation of Chisel code, Chipyard generates the equivalent Verilog code. I used this code along with Vivado’s post-synthesis analysis tools to gather information. The specific board used was the Virtex UltraScale+ FPGA VCU118 evaluation kit.

For analysis, I separated the algorithm into the three natural components described previously: key generation, key encapsulation, and key decapsulation, as this is how the original NewHope paper analyzed it. My goal with gathering this data was to compare it to the simple version of the algorithm implemented on FPGA while understanding which parts of the algorithm are bottlenecks for area and performance. I ran analysis on each component twice: once optimizing for area, and once for timing. The following are the results.

<table>
<thead>
<tr>
<th>Component</th>
<th>Optimization</th>
<th>Frequency</th>
<th>LUT Count</th>
<th>LUT Board Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Generation</td>
<td>Timing</td>
<td>82.95 MHz</td>
<td>629658</td>
<td>53.26%</td>
</tr>
<tr>
<td>Key Encapsulation</td>
<td>Timing</td>
<td>62.49 MHz</td>
<td>1134546</td>
<td>95.97%</td>
</tr>
<tr>
<td>Key Decapsulation</td>
<td>Timing</td>
<td>83.47 MHz</td>
<td>508935</td>
<td>43.05%</td>
</tr>
<tr>
<td>Key Generation</td>
<td>Area</td>
<td>97.2 MHz</td>
<td>197461</td>
<td>16.70%</td>
</tr>
<tr>
<td>Key Encapsulation</td>
<td>Area</td>
<td>94.83 MHz</td>
<td>639850</td>
<td>54.12%</td>
</tr>
<tr>
<td>Key Decapsulation</td>
<td>Area</td>
<td>450 MHz</td>
<td>409</td>
<td>0.03%</td>
</tr>
</tbody>
</table>

In addition to these statistics, running the entire simulator took 169 seconds, orders of magnitude above the wall clock times for the C implementations of key generation algorithms mentioned previously. However, most of this time is spent setting up the RoCC system; it takes several seconds for the simulator to even get the starting signal, so this comparison isn’t head-to-head with the earlier wall clock times. As previously mentioned, there is a modified version of NewHope, called NewHope-Simple, which was implemented on a low-cost Xilinx FPGA. Although this algorithm is different than the implementation discussed here, their resource consumption was about 10,000 LUT, significantly smaller than this implementation,
and their frequency was 125 MHz. The simple version of the algorithm uses different error reconciliation, which is less complex. In addition, the creators optimized their implementation for area.

4 Conclusion

One of the goals of this project was to explore post-quantum cryptography. There are several algorithms currently in the running for NIST's competition, and I found no space with comparison of the area and timing statistics for each of these. My background work aims to remedy that.

My other goal was to use Chisel to implement NewHope and learn about both that hardware design language and the key exchange algorithm in the process. The module taking up the most area in my design is the NTT transform, and the NewHope-Simple FPGA design reached the same conclusion. One downside of Chisel is that the only random number generator is based off of a shift register, so using a random seed did not yield the same results as my C implementation. Instead, I had to hardcode base values to verify my implementation.

For future work, I would want to modify my Chisel implementation to be parametrized to have more control over timing optimizations. In addition, I would like to spend more time understanding the generated Verilog, as this is what all of my reported numbers are based off of.

5 Acknowledgments

First of all, I would like to thank my thesis advisor, Dr. Mohit Tiwari, for all of his help and guidance throughout this process. I would also like to thank Dr. Calvin Lin and Dr. Robert Van De Geijn for serving on my committee and giving me feedback on this thesis.

I would like to thank my academic advisor, Doug Roberts, for always going above and beyond when I needed help with absolutely anything. I would also like to thank Austin Harris for assisting me in getting set up with the tools I was using for this project.

Finally, I would like to thank my father for fostering my love of science and my mother for never allowing me to give up.
References


[10] X Lu, Y Liu, Z Zhang, D Jia, H Xue, J He, B Li, K Wang. LAC: Practical Ring-LWE Based Public-Key Encryption with Byte-Level Modulus. IACR Cryptol. ePrint Arch., 2018


[28] GitHub, Art3misOne, newhope. [https://github.com/Art3misOne/newhope](https://github.com/Art3misOne/newhope)
