

# BENCHMARKING PRE/POST-QUANTUM CRYPTOGRAPHY

BY

Cavan Phelan

C00249198

17 April 2023

## Contents

1. Introduction.....	1
2. Description of Submitted Project.....	1
2.1. User Interface .....	1
2.1.1. Home Page .....	1
2.1.2. Benchmarking Options Algorithm.....	2
2.1.3. Profiler Selection .....	5
2.2. Benchmarking .....	6
2.3. Graphing.....	7
2.4. Benchmarking the Algorithms .....	8
2.4.1. Benchmark Variables .....	8
2.4.2. Benchmark Parameters .....	9
2.4.3. Setup Class.....	9
2.4.4. Benchmark Class .....	10
2.5. Benchmark Results.....	10
2.5.1. Graphing Benchmark Results .....	11
2.6. Getting Algorithm Selection to Benchmark .....	13
2.7. Showcasing Algorithm Security.....	13
2.7.1. Saving Data to a File.....	14
3. Description of Conformance to Specification and Design .....	15
3.1. Technology Differences .....	15
3.2. Algorithm Differences.....	15
4. Description of Learning .....	16
4.1. Technical Learning.....	16
4.1.1. Using Maven.....	16
4.1.2. Using Java Micro-Benchmarking Harness .....	16
4.1.3. Implementing post-quantum algorithms with Bouncy Castle. ....	16
4.1.4. Coding in Java.....	16
4.2. Personal Learning.....	16
4.2.1. Patience .....	16
4.2.2. Allocating Time .....	17
5. Project Review .....	17
5.1. Would I approach it differently? .....	17

5.2. Technology Choices .....	17
5.3. Was My Project a Success? .....	21
6. Acknowledgements.....	22

# 1. Introduction

The aim of this document is to provide an overview of the progress of my project. It will include an analysis of what I had initially planned to do with what I have managed to do. It will also serve as a time of self-reflection on what I have learned during this project and if I were to start fresh, how I would change my work ethic and structure to have my project in a better condition.

It will describe the workings of my project, including some underlying code in which I am proud to have managed to get working.

# 2. Description of Submitted Project

## 2.1. User Interface

### 2.1.1. Home Page

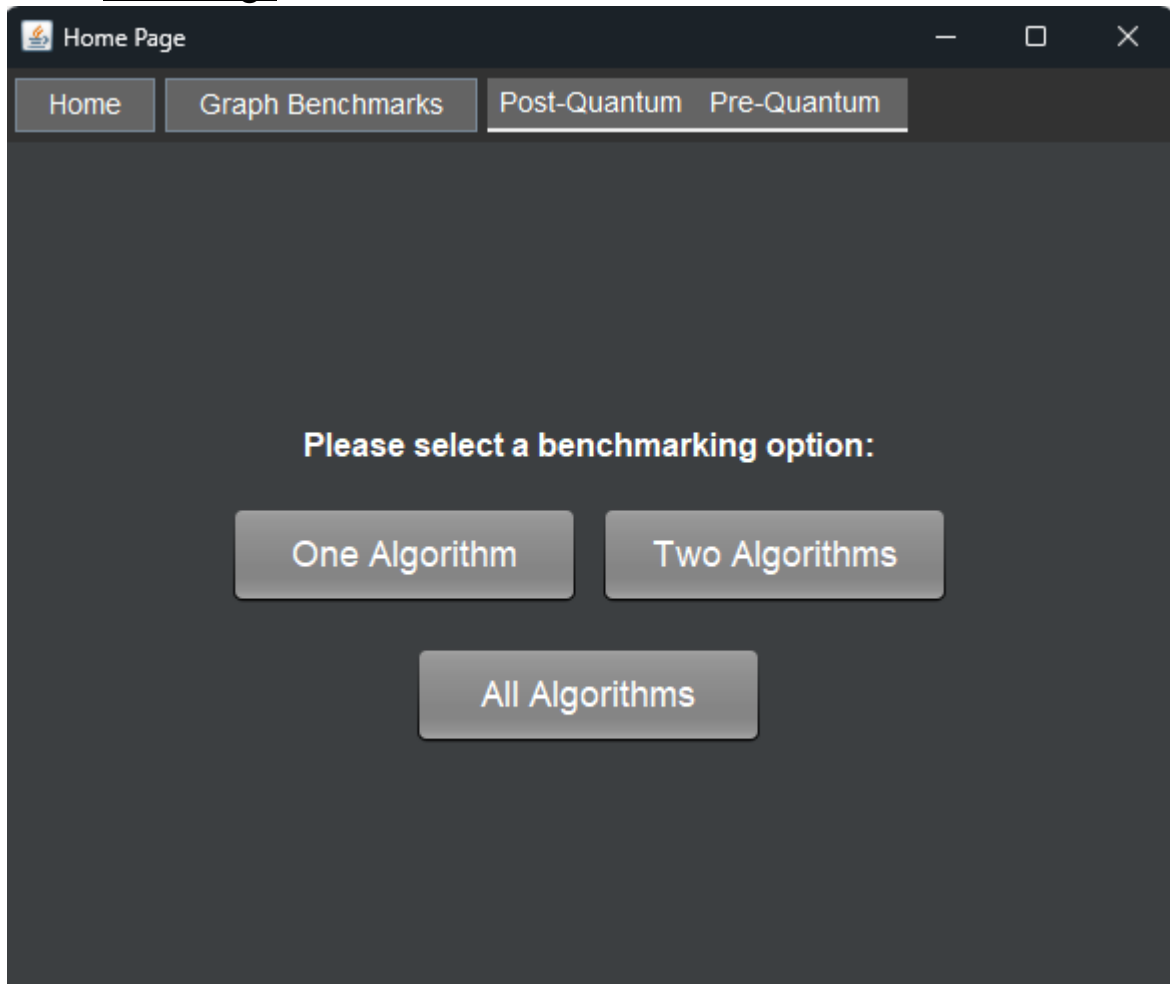


Figure 1: Home menu

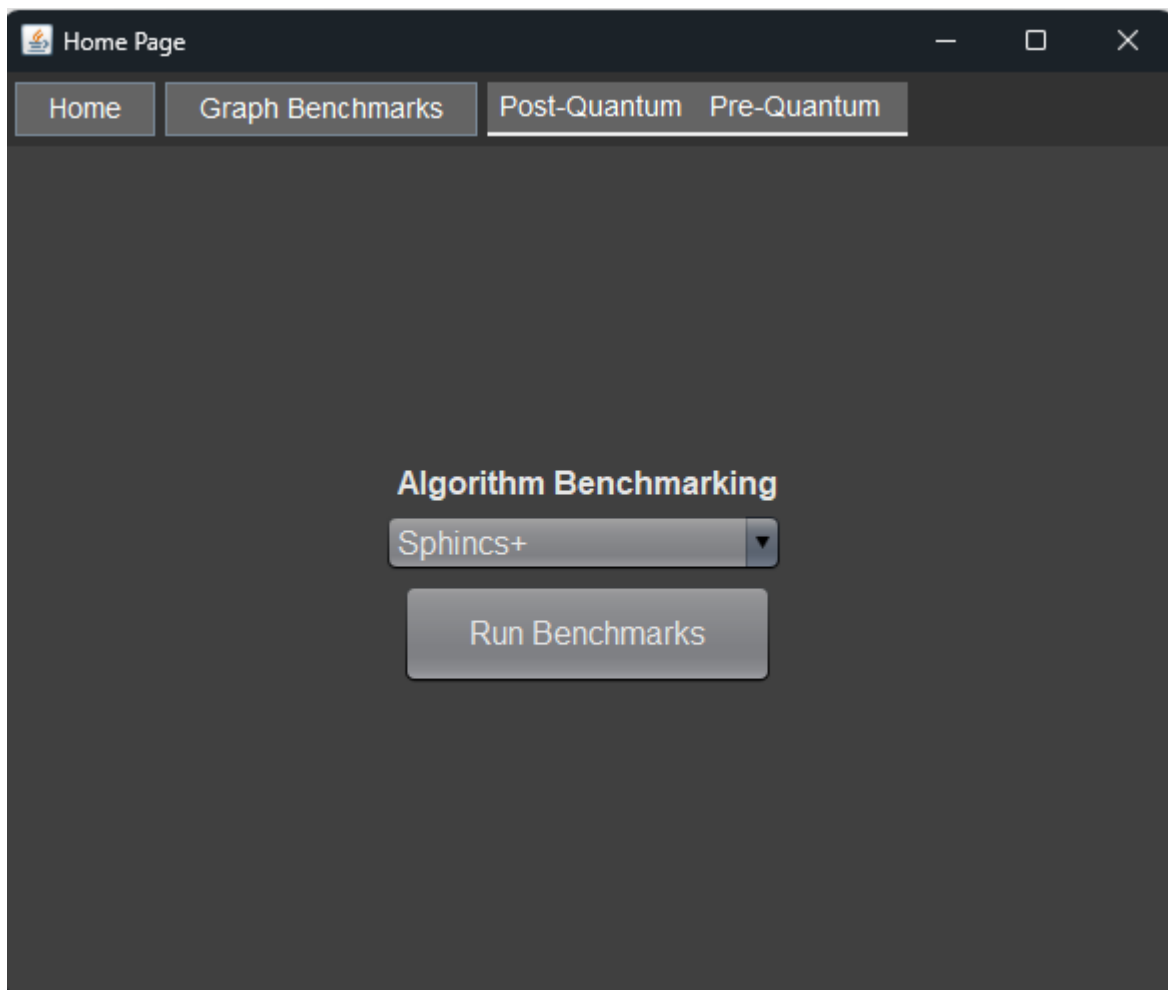
The above is the initial home menu when starting the application. The user will be presented with multiple button options to run a certain number of algorithms.

The user also has the option to start graphing completed benchmarks.

The post-quantum and pre-quantum buttons provide dropdowns linking the user to the source developer pages of each algorithm.

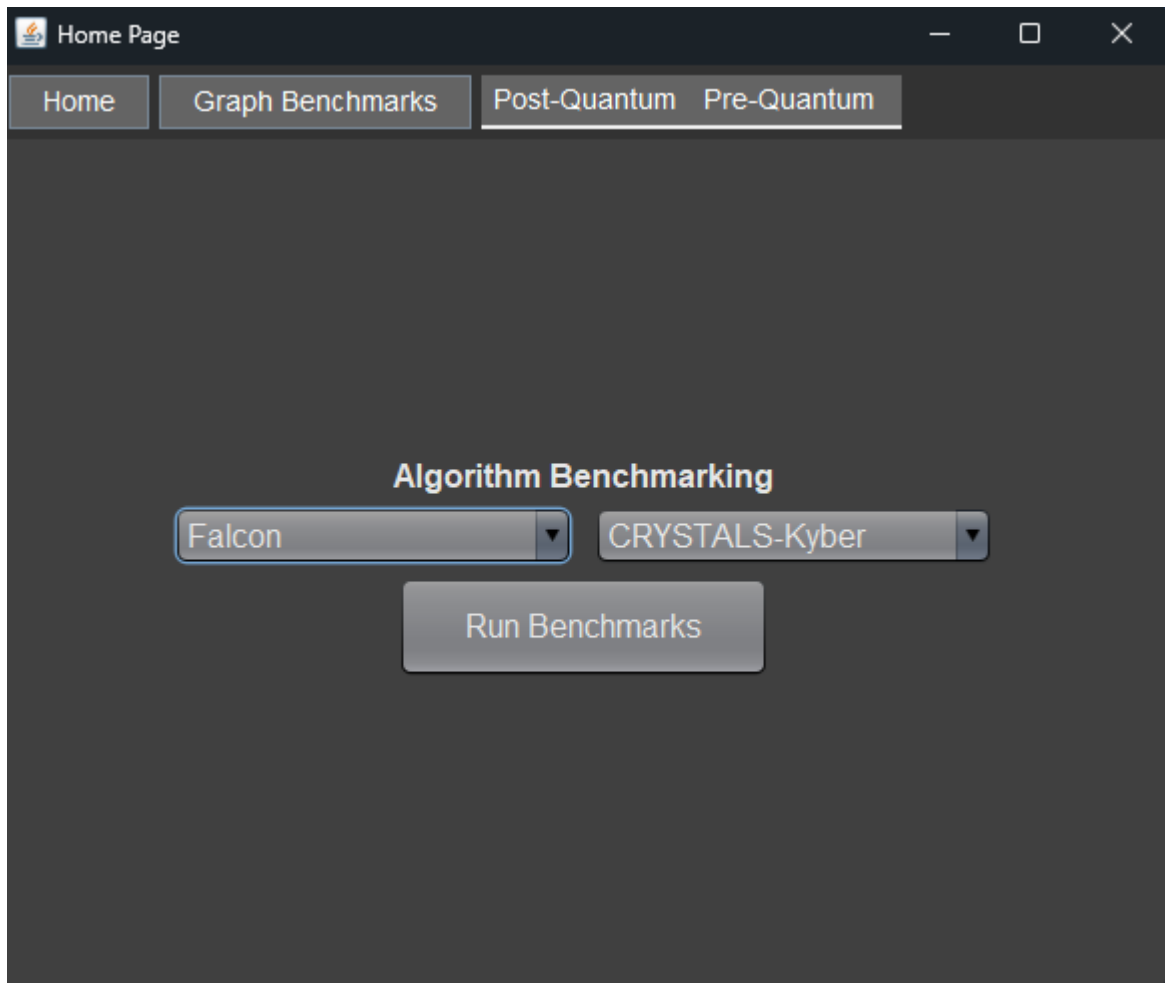
### 2.1.2. Benchmarking Options Algorithm

If the user selects the 'One Algorithm' button they will be presented on a new page containing a dropdown of which algorithm they wish to benchmark. Once a user has selected their algorithm, they may hit run, but before the benchmark begins, they can enable some profilers for extra debugging and information.



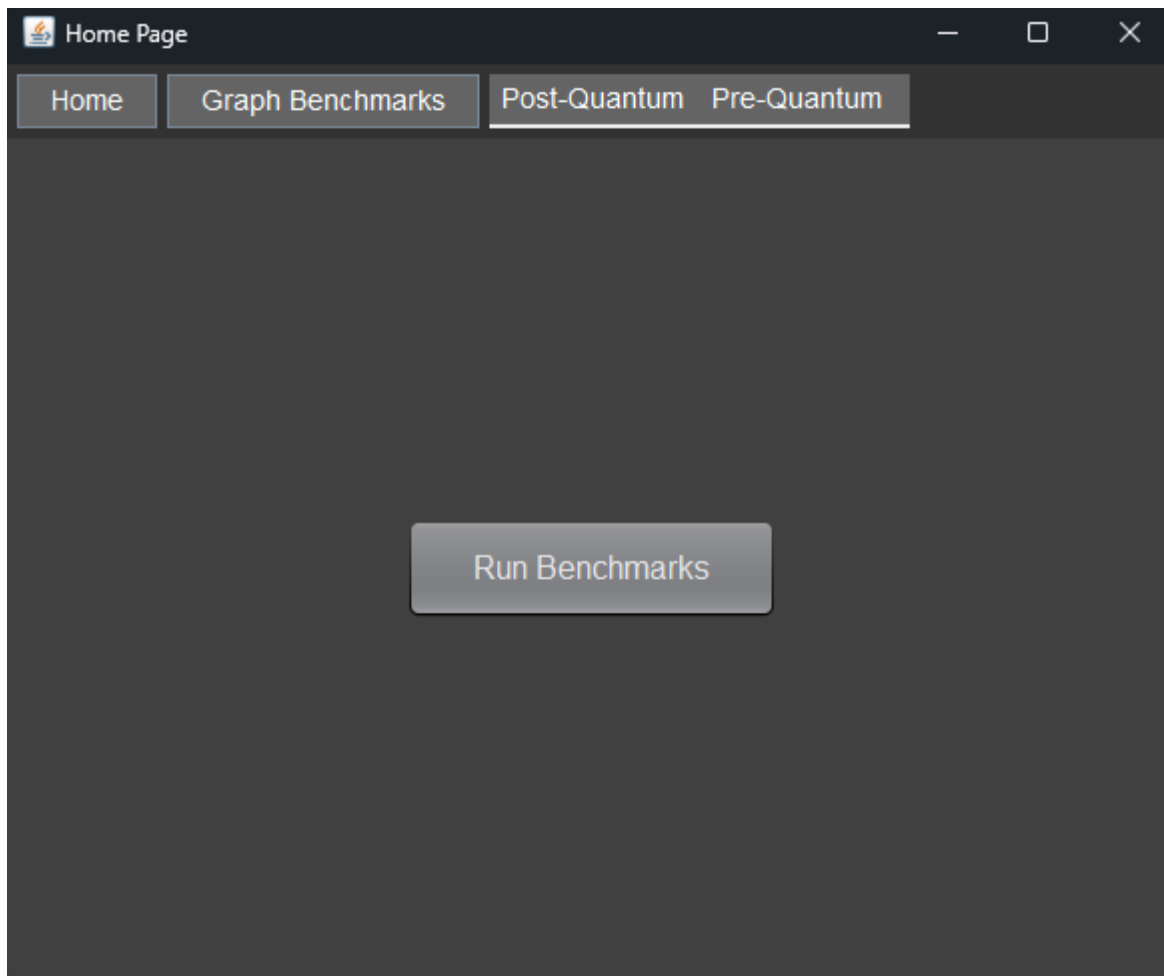
*Figure 2: Benchmarking 1 algorithm*

If the user selects the 'Two Algorithms' button they will be presented on a new page containing two dropdowns of which algorithms, they wish to benchmark.



*Figure 3: Benchmarking two algorithms*

If the user wants to run every benchmark and click “All benchmarks” they will be directed to just a single button to begin the benchmarks.



*Figure 4: All benchmarking options*

### 2.1.3. Profiler Selection

Before any of the benchmarks are run, clicking the 'run' buttons will prompt the users if they want to enable any of the profilers. This will pop up for both algorithms if you want to benchmark two, or if you are benchmarking all algorithms, this option will apply to all benchmarks.

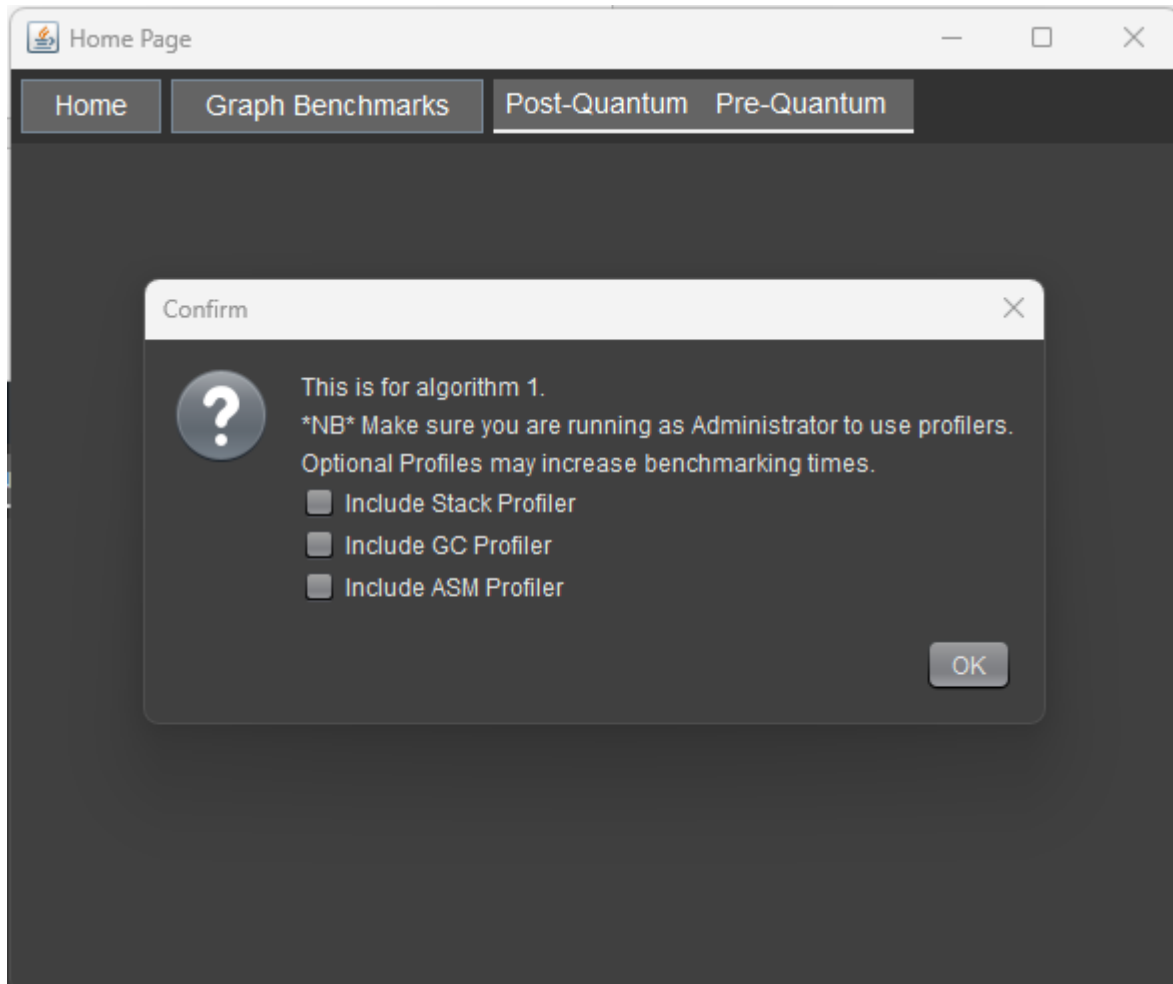


Figure 5: Profiler options

The options are as follows:

**Stack Profiler** – Samples the call stack of the running benchmark. It helps identify which methods in the call stack are consuming the most time. The profiler collects data on the method calls and the time spent in each method. This can help in identifying performance bottlenecks in the benchmark code.

**Garbage Collector (GC) Profiler** - Profiling the behaviour of the Java Garbage Collector during the benchmark execution. It provides information on the time taken by the GC, the number of GC cycles, and the amount of memory allocated and deallocated.

**ASM Profiler** - The ASM (Abstract Syntax Tree) Profiler in JMH is a profiler that uses the ASM library to instrument the bytecode of the benchmark method. It helps in measuring the number of instructions executed, the time taken by each instruction, and the number of branches and loops executed during the benchmark run.



Once the user has confirmed the benchmarks will begin to run. It is to be noted that these may increase benchmarking times.

## 2.2. Benchmarking

As the benchmarks are beginning, it will provide you information on the benchmark parameters such as iteration amount and length, threads being used, and which benchmark is being run.

```
Running benchmarks for: CRYSTALS-Kyber
# JMH version: 1.36
# VM version: JDK 20, Java HotSpot(TM) 64-Bit Server VM, 20+36-2344
# VM invoker: C:\Program Files\Java\jdk-20\bin\java.exe
# VM options: -javaagent:C:\Applications\IntelliJ IDEA 2022.3.2\lib\idea
# Blackhole mode: compiler (auto-detected, use -Djmh.blackhole.autoDetect
# Warmup: 1 iterations, 1 s each
# Measurement: 1 iterations, 1 s each
# Timeout: 10 min per iteration
# Threads: 1 thread, will synchronize iterations
# Benchmark mode: Average time, time/op
# Benchmark: Post_Quantum.Kyber.k1024AesEncapsulatedPrivateKeyGen
```

Figure 6: Benchmarking Information

Below we can see the benchmarks in real-time being completed. It will let us know how far into the benchmark we are, and the speed of the benchmarks being run, and which iterations are being done. Note that in this example I am using only 1 warmup and measurement iteration which will not provide a broad enough range to be accurate, this is only for demonstration.

```
# Run progress: 96.67% complete, ETA 00:00:03
# Fork: 1 of 1
# Warmup Iteration 1: 57274.707 ns/op
Iteration 1: 49107.899 ns/op

Result "Post_Quantum.Kyber.k768WrapKey":
49107.899 ns/op
```

Once the benchmark is fully complete it will post all the results to the console. Here we can see the benchmarks being run, the mode of the benchmarks which is the average time in this case, and the score of the benchmarks.

Benchmark	Mode	Cnt	Score	Error	Units
Post_Quantum.Kyber.k1024AesEncapsulatedPrivateKeyGen	avgt		131615.606		ns/op
Post_Quantum.Kyber.k1024AesEncapsulatedPublicKeyGen	avgt		128486.333		ns/op
Post_Quantum.Kyber.k1024AesKeyGen	avgt		115908.639		ns/op
Post_Quantum.Kyber.k1024AesUnwrapKey	avgt		132425.758		ns/op
Post_Quantum.Kyber.k1024AesWrapKey	avgt		123660.210		ns/op
Post_Quantum.Kyber.k1024EncapsulatedPrivateKeyGen	avgt		73377.167		ns/op
Post_Quantum.Kyber.k1024EncapsulatedPublicKeyGen	avgt		69422.995		ns/op
Post_Quantum.Kyber.k1024KeyGen	avgt		60206.260		ns/op
Post_Quantum.Kyber.k1024UnwrapKey	avgt		78263.775		ns/op
Post_Quantum.Kyber.k1024WrapKey	avgt		66611.854		ns/op
Post_Quantum.Kyber.k512AesEncapsulatedPrivateKeyGen	avgt		50812.837		ns/op
Post_Quantum.Kyber.k512AesEncapsulatedPublicKeyGen	avgt		48481.465		ns/op
Post_Quantum.Kyber.k512AesKeyGen	avgt		41381.858		ns/op
Post_Quantum.Kyber.k512AesUnwrapKey	avgt		51930.785		ns/op

Figure 7: Benchmark Results

### 2.3. Graphing

The user has the option to include their benchmarks in a graph. Once the user clicks to display the graph, it will initially be empty, but upon clicking “Add File” they can add a benchmark to the graph like below.

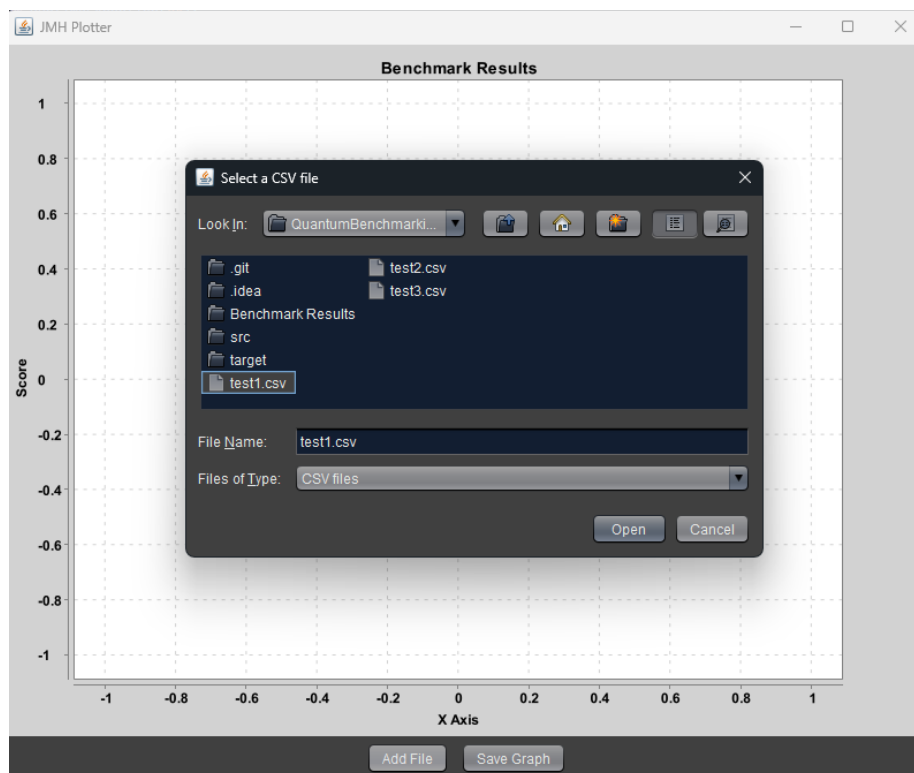


Figure 8: Choosing a file to graph.

The user can then add as many benchmarks as they want to graph, and it will look like the following.

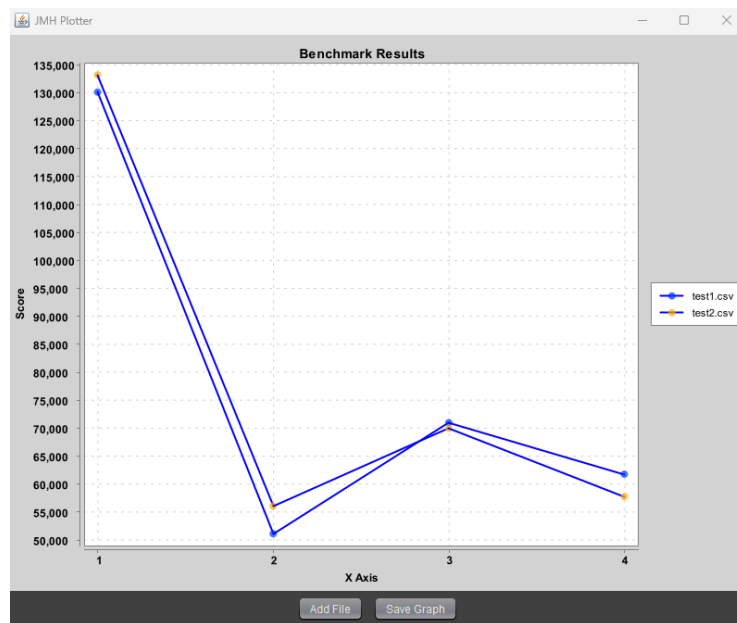


Figure 9: Graphed Benchmarks

The user can now save the graph by clicking the ‘Save Graph’ button and choosing where to save the file.

## 2.4. Benchmarking the Algorithms

There are multiple components to setting up and configuring benchmarks, these components being the benchmark variable, benchmark parameters, the setup class, and the benchmark classes.

### 2.4.1. Benchmark Variables

These dictate the benchmarking modes, benchmarking types and how long the benchmarks are. For all algorithms, I use the same benchmarking variables.

```
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.NANOSECONDS)
@Warmup(iterations = 3, time = 3)
@Measurement(iterations = 5, time = 3)
@Fork(1)
@State(Scope.Benchmark)
public class Falcon {
```

Figure 10: Benchmark variables

The current benchmark mode is calculating the average time of the benchmarks and that will result in the score of that benchmark rather than its quickest or slowest score.

The output time unit refers to what unit the benchmarks will be shown as, so in this case they will be shown in nanoseconds.

The warmup is a warmup iteration of the benchmark to optimise the benchmark before

ranking the official score. I have three warmup iterations that run for three seconds each before moving on to the measurement iterations. These are the recorded iterations, which is why I run them for more iterations to get a more accurate score.

The fork indicates how many times the benchmarks will start over, in this case, it will run through one set of their warmups and five measurements.

The state is configured to the benchmark scopes, meaning that each benchmark will have a fresh state for benchmarking.

### 2.4.2. Benchmark Parameters

Parameters allow me to specify if I want to run multiple different instances of a variable such as a key length of plaintext size. In the image below, I am initialising a static int with the parameters 256, 512, 1024 and 2048. This is then used to create plaintext of different sizes. The benchmarks will start on a 256-byte plaintext and execute every benchmark, instead of completing, it will start again, but move to the next parameter until all parameters are complete. This allows me to test multiple key sizes and plaintext sizes without having to code benchmarks for each size.

```
@Param({"256", "512", "1024", "2048"})
static int plaintextSize;
```

Figure 11: Benchmark variables

### 2.4.3. Setup Class

This is where most of the initialising for the benchmarks takes place. I decided to make static variables and define them inside the setup, to minimise the time spent doing it during benchmarks so I can focus more on the algorithm's goals.

```
@Setup
public void setup() throws Exception {
    // Setting up starting variables
    Security.addProvider(new BouncyCastlePQCProvider());
    plaintext = new byte[plaintextSize];
    new SecureRandom().nextBytes(plaintext);
    // Creating KPGs for KPs
    f512KPG = KeyPairGenerator.getInstance( algorithm: "Falcon", provider: "BCPQC"); f512KPG.initialize(FalconParameterSpec.falcon_512, new SecureRandom());
    f1024KPG = KeyPairGenerator.getInstance( algorithm: "Falcon", provider: "BCPQC"); f1024KPG.initialize(FalconParameterSpec.falcon_1024, new SecureRandom());
    // Creating KPs
    falcon512KP = falcon512KeyGeneration(); falcon1024KP = falcon1024KeyGeneration();
    // Creating Sig instances
    f512Sig = Signature.getInstance( algorithm: "Falcon-512", provider: "BCPQC"); f1024Sig = Signature.getInstance( algorithm: "Falcon-1024", provider: "BCPQC");
    // Using variables to call KPG class to go into verify() without impacting benchmarks
    falcon512Signature = falcon512Sign(); falcon1024Signature = falcon1024Sign();
}
```

Figure 12: Benchmark setup

In the above image I am creating the plaintext and randomising the bits, I am then creating key pair generators with different security levels provided by the algorithm to then create a key pair. Rather than rely on the benchmarks to call on one another, inside the setup I would assign a variable to run the benchmarks to get assigned variables. It is important to note that these don't run the benchmarks as the setup is fully initialised before the benchmarks can take place. The setup class is annotated with the @Setup annotation.

#### 2.4.4. Benchmark Class

This is where the real benchmarking takes place, where it will execute and keep track of how fast the operations defined take.

```
@Benchmark
public byte[] falcon512Sign() throws Exception {
    f512Sig.initSign(falcon512KP.getPrivate(), new SecureRandom());
    f512Sig.update(plaintext, off: 0, plaintext.length);
    return f512Sig.sign();
}
```

Figure 13: Falcon Sign Example

In the example above, we are using `falcon512Sign()` to create a signature. Inside this benchmark, we want to calculate how long it takes to initialise the cipher and create the signature. Since I didn't want benchmarks calling other benchmarks, I ended up allowing the benchmarks to return values, which is good due to it being more efficient on the code, and it allows me to assign a variable to run these benchmarks to reuse them in other benchmarks, without the benchmarks running yet since it's all assigned in the setup phase.

#### 2.5. Benchmark Results

By default, the benchmarks are run, and the results are just outputted to the console, but to better compare algorithms I would need to save these easily. When a benchmark is selected it runs through a switch statement until it finds its algorithm name, then I am adding on to the JMH options to include a results file to be saved at a specified location.

```
case "Sphincs+" → {
    try {
        SphincsPlus.main(new String[0]);
    } catch (Exception ex) {
        throw new RuntimeException(ex);
    }
    builder.include(SphincsPlus.class.getSimpleName())
        .result(filename: "Benchmark Results/Post-Quantum/SphincsPlus Benchmarks/SphincsPlus_Benchmarks.csv");
    Options options = builder.build();
    try {
        new Runner(options).run();
    } catch (RunnerException ex) {
        throw new RuntimeException(ex);
    }
}
```

Figure 14: Sphincs+ CSV

In the above, when “Sphincs+” is found, I can specify that I want to save the file, and where I want it saved. For the profilers, I have Boolean statements that are triggered, which will add to the benchmark builder to add the profiler.

```
// Check if profilers are created
if (includeStackProfiler) {
    builder.addProfiler(StackProfiler.class);
}
if (includeStackProfiler?) {
```

Figure 15: Adding Profiler

The results are being saved to a CSV file as they’re compatible with JMH and Excel. The results save to the file in the same format as of which the printout is to the console.

	A	B	C	D	E	F	G
1	Benchmark	Mode	Threads	Samples	Score	Score Error (99.9%)	Unit
2	Testing.Cavan.k512EncapsulatedKeyGen	avgt	16	1	130104.3	NaN	ns/op
3	Testing.Cavan.k512KeyGen	avgt	16	1	51122	NaN	ns/op
4	Testing.Cavan.k512UnwrapKey	avgt	16	1	70978.07	NaN	ns/op
5	Testing.Cavan.k512WrapKey	avgt	16	1	61773.21	NaN	ns/op

Figure 16: CSV File

### 2.5.1. Graphing Benchmark Results

To graph the results, it would require me to parse the result files, search for the score column and then update the graph with the scores every time a new graph is added.

I started by creating a blank canvas with no data, as I can update the canvas when needed.

```
// Create initial chart
chart = new XYChartBuilder()
    .width(800)
    .height(600)
    .title("Benchmark Results")
    .xAxisTitle("X Axis")
    .yAxisTitle("Score")
    .build();
```

Figure 17: Creating Blank Graph

Once a user selected results to graph, I parsed the file looking for the 'score' and if it was found it was extracting all the data until it hit an empty cell again and then stop. The scores are stored in an array to layer and are painted on the chart.

```
for (int i = 0; i < headerRow.length; i++) {
    if (headerRow[i].equalsIgnoreCase( anotherString: "score")) {
        scoreColumnIndex = i;
        break;
    }
}

if (scoreColumnIndex != -1) {
    int rowIndex = 1;
    for (String[] record : records) {
        xData.add((double) rowIndex);
        yData.add(Double.parseDouble(record[scoreColumnIndex]));
        rowIndex++;
    }
} else {
    System.err.println("Score column not found.");
}

addSeriesToChart(xData, yData, selectedFile.getName(), chart, chartPanel);
```

Figure 18: File Parse

This would then call the function I created which repaints the graph with the updated arrays and plots them on the graph.

```
public static void updateChart(JPanel chartPanel) { chartPanel.repaint(); }
```

Figure 19: Update Chard Method

Saving the graph was simply using a Bitmap Encoder to save the file to a path specified by the user.

```
try {
    BitmapEncoder.saveBitmap(chart, outputFile.getAbsolutePath(), BitmapEncoder.BitmapFormat.PNG);
    System.out.println("Graph saved successfully.");
}
```

Figure 20: Saving Graph

## 2.6. Getting Algorithm Selection to Benchmark

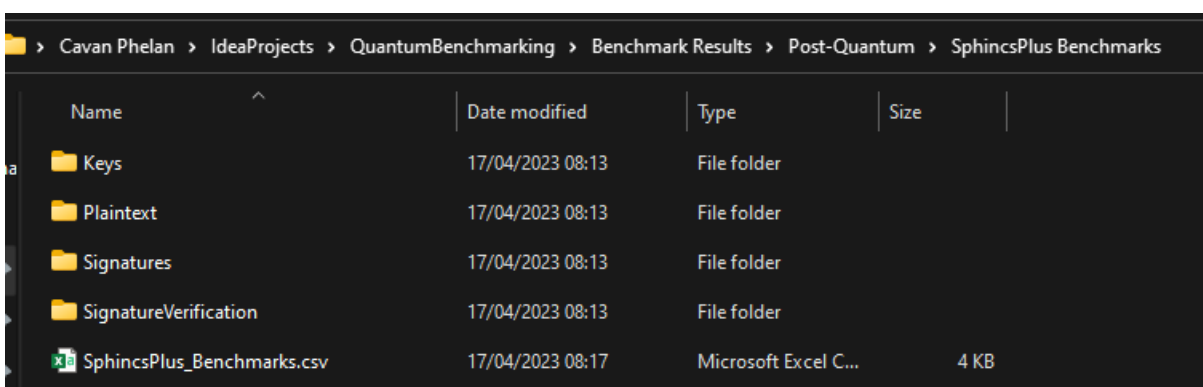
I stored the names of each algorithm into a string array, with the names equalling the options provided in the dropdown box to the user. The selected algorithm name would then run through a switch statement and stop when it found a match. It would then run the main method of the algorithm before then running and saving the benchmarks.

```
switch (Objects.requireNonNull(algorithm1)) {
    case "Falcon" → {
        try {
            Falcon.main(new String[0]);
        } catch (Exception ex) {
            throw new RuntimeException(ex);
        }
        builder.include(Falcon.class.getSimpleName())
            .result( filename: "Benchmark Results/Post-Quantum/Falcon Benchmarks/Falcon_Benchmarks.csv");
        Options options = builder.build();
        try {
            new Runner(options).run();
        } catch (RunnerException ex) {
            throw new RuntimeException(ex);
        }
    }
}
```

Figure 21: Switch Statement to run benchmarks.

## 2.7. Showcasing Algorithm Security

Trying to showcase the security differences such as key or signature lengths wasn't possible due to the benchmarks being isolated and reset. This led to me implementing the algorithms outside of the benchmarks so that I could save parameters such as encoded/decoded keys and signatures. That is why in the switch statement above, I am executing the main method of that algorithm first to quickly generate the keys and other parameters. The main method is creating the folder structures and saving the files and their contents.



Name	Date modified	Type	Size
Keys	17/04/2023 08:13	File folder	
Plaintext	17/04/2023 08:13	File folder	
Signatures	17/04/2023 08:13	File folder	
SignatureVerification	17/04/2023 08:13	File folder	
SphincsPlus_Benchmarks.csv	17/04/2023 08:17	Microsoft Excel C...	4 KB

Figure 22: Folder Layout





### 3. Description of Conformance to Specification and Design

I believe a lot has changed from the original Functional Specification I had designed. I believe the structure of the UI stayed someone similar but the further I progressed into the project, the more I discovered and changed what I was using and the way I was using them.

I will first discuss the changes in technologies, then the algorithms in which I wanted to benchmark and then any other tasks I've since added.

#### 3.1. Technology Differences

**Bouncy-Castle** - Although I did specify the use of Bouncy-Castle, I didn't specify that I was mostly using the beta Post-Quantum Cryptography (PQC) of Bouncy-Castle, which is required to implement post-quantum algorithms.

**Java Universal Network/Graph (JUNG)** – I had initially planned to use this to graph my benchmarks to get a visual representation of the benchmarked data. I was having some difficulties in getting this to work so I decided to use **X-Chart** instead, which was a much easier tool to use.

**Maven** – I found using Eclipse a little hard, as it's not visually pleasing and connecting the right files can lead to a lot of issues, to begin with. Instead of just using Java, I used Maven which helped me automatically build and compile my project and helped me organise it in a better manner.

**OpenCSV** – I hadn't researched how to parse files initially, but when it finally came to that point in the project, I tried a couple of CSV file readers, with this being the only one to work.

#### 3.2. Algorithm Differences

I initially planned to benchmark five pre-quantum and five post-quantum algorithms.

Pre-Quantum: AES, MD5, SHA-256, RSA, El Gamal.

Post-Quantum: CRYSTALS-Kyber, CRYSTALS-Dilithium, Sphincs+, SIKE, PICNIC.

I instead ended up benchmarking five pre-quantum algorithms and seven post-quantum algorithms.

Pre-Quantum: AES, RSA, TwoFish, SHA-256, SHA-3

Post-Quantum: CRYSTALS-Kyber, CRYSTALS-Dilithium, Sphincs+, BIKE, PICNIC, Falcon, Rainbow

**MD5** – This is a broken algorithm but is still used for checksums, but I felt I could implement a more relative algorithm instead.

**El Gamal** – There isn't anything wrong with El Gamal, it's just that **TwoFish** seemed like a more interesting option to benchmark.

**SHA-3** – Instead of MD5, I felt like the relatively new SHA-3 would be a good algorithm to cover as it will probably become the next new hashing standard to be used worldwide.

**SIKE** – SIKE was initially implemented in the Bouncy Castle library but was then removed due to the algorithm being attacked successfully. Instead, I went for **BIKE** as at the time, there was a limited number of post-quantum algorithms to choose from.

**Falcon & Rainbow** – I couldn't decide which algorithms to replace with SIKE, so I initially picked Rainbow but there was a bug within the Bouncy Castle library, so I had to turn to Falcon. After a few months, there was a bug fix, so I decided to implement Rainbow anyway because of the name.

All of the other algorithms that I have kept, as of different backgrounds and types so keeping a variety of them help with benchmarking them.

## 4. Description of Learning

### 4.1. Technical Learning

#### 4.1.1. Using Maven

I initially only started using Maven because of its popularity of it when I searched for how to start this project. This led to me not knowing how to use it, especially the pom file, where you link all, your project dependencies too. After a lot of frustration, I finally learned how to utilise Maven and use it to further organise my project and keep track of tasks.

#### 4.1.2. Using Java Micro-Benchmarking Harness

I thought benchmarking would be quite easy, but unfortunately, no one has much information on benchmarking algorithms like I am, so I was stuck with hardcoding every single piece of my code. Instead of researching online, I instead started trying different methods that came to mind, finally coming to the system I have now where I can create static variables and assign them outside benchmarks to maximize performance impact. It also allows me to prevent the benchmarks from calling one another in other to proceed, instead, I can run them inside the setup to initialise static variables to include in the benchmarks.

#### 4.1.3. Implementing post-quantum algorithms with Bouncy Castle.

Another issue I was having was the lack of information on implementing these algorithms with BC. There is next to no information on this, so I had to extensively research to find information on the implementation of the algorithms,

#### 4.1.4. Coding in Java

Having not coded in Java in two years, it was safe to say I was rusty on how to even start coding a simple program. After some research and a lot of practice, I was somewhat fluent in Java and got into a good uninterrupted workflow.

All of these contribute to increasing my technical abilities. Not only has it increased my knowledge of Java, but I've also learned a lot about cryptographic algorithms and their implementations and limitation. I've learned the restrictions when benchmarking algorithms and potential optimisation I need to make to get more accurate results. Since a lot of my work is coding, my typing speed and accuracy have improved too.

### 4.2. Personal Learning

#### 4.2.1. Patience

Having to look at complex arithmetic and equations took a lot of energy out of me, and often more than not I would need to re-read sentences over and over to just wrap my head around certain concepts. Being patient allowed me to finally take on the information needed.

### 4.2.2. Allocating Time

I quickly learned the importance of setting myself up to work on the project for a couple of hours a day to maintain a steady workflow throughout the year. I also overestimated how long each task would take me to achieve, just to give myself enough time in case I needed that extra work and not have to worry about time constraints.

## 5. Project Review

### 5.1. Would I approach it differently?

I feel as though overall my project is where I wanted it to be and that it's in good shape. One of my biggest mistakes was leaving it quite late to start on the post-quantum algorithms. I left them until last and I started to struggle with time as I had to pressure myself into learning the post-quantum algorithms and problems. Instead, I would have started with learning and writing up about the post-quantum algorithms at the start of the year when it is quieter and stress-free, this means you would get the hard, intensive work out of the way first. I didn't account for the lack of good information, so I wasted a lot of time on that. I underestimated the complexity of understanding and implementing the post-quantum algorithms.

### 5.2. Technology Choices

I initially believed my technology choice at the start was what was best for my project, but that quickly changed once I started trying to integrate them into my project. I started running into issues and had to try multiple similar plugins. Reviewing my final technology choices now, I believe I found a good balance that works and allows me to implement my project without issues.

### 5.3. Testing Results

Here is a summary of my findings, showcasing the results of each algorithm. Since some algorithms can have up to two-hundred benchmarks done for five categories, I will provide the average time of each category and not even the benchmark instance.

I will be using three warmup iterations for one second each, and five measurement iterations for one second each. These benchmarks would be processed for longer, but the PICNIC algorithm alone took two hours for benchmarking five iterations for five seconds each and my parents weren't too happy with the electricity bill.

Disclaimer: I am missing the SPHINCS+ benchmarks as I ran out of time to organise them to display.

#### 5.3.3. Falcon Results

These are the benchmarks for the two Falcon parameters. We can clearly see that increasing the security of this algorithm comes with large impacts on performance, which may make it unfeasible to use in most cases, the Falcon-512 parameter is probably worth it more as it's faster and still provides a high level of security.

Benchmark	Score	
Post_Quantum.Falcon.falcon512KeyGeneration	32692506.41	
Post_Quantum.Falcon.falcon512Sign	2871012.979	
Post_Quantum.Falcon.falcon512Verify	173128.6221	Total Score
		35736648
Benchmark	Score	
Post_Quantum.Falcon.falcon1024KeyGeneration	91770858.89	
Post_Quantum.Falcon.falcon1024Sign	6054446.962	
Post_Quantum.Falcon.falcon1024Verify	316811.9648	Total Score
		98142117.8

Figure 27: Falcon Results

### 5.3.4. Picnic Results

As we can see, the higher security we try to use in Picnic, the longer it takes to run the algorithms, however, the differences between L1FS and L5FULL aren't that bad, and it might be worth considering testing out all of these algorithms to find the best fit.

Benchmark	Score		Benchmark	Score	
Post_Quantum.Picnic.L1fsKeyGeneration	273252.3		Post_Quantum.Picnic.L1fullKeyGeneration	273243.5099	
Post_Quantum.Picnic.L1fsSign	461866749.4		Post_Quantum.Picnic.L1fullSign	460719535.3	
Post_Quantum.Picnic.L1fsVerify	301141355.6	Total Score	Post_Quantum.Picnic.L1fullVerify	302114730.1	Total Score
		763281357			763107509
Benchmark	Score		Benchmark	Score	
Post_Quantum.Picnic.L3fsKeyGeneration	170997.7844		Post_Quantum.Picnic.L3fullKeyGeneration	268406.9664	
Post_Quantum.Picnic.L3fsSign	463380244.6		Post_Quantum.Picnic.L3fullSign	464675217.4	
Post_Quantum.Picnic.L3fsVerify	298997479	Total Score	Post_Quantum.Picnic.L3fullVerify	374485955	Total Score
		762548721			839429579
Benchmark	Score		Benchmark	Score	
Post_Quantum.Picnic.L5fsKeyGeneration	323928.2014		Post_Quantum.Picnic.L5fullKeyGeneration	320469.6858	
Post_Quantum.Picnic.L5fsSign	554963172.2		Post_Quantum.Picnic.L5fullSign	540609928.4	
Post_Quantum.Picnic.L5fsVerify	355569496.5	Total Score	Post_Quantum.Picnic.L5fullVerify	359834299.3	Total Score
		910856597			900764697

### 5.3.5. CRYSTALS-Dilithium Results

We can quickly see that Dilithium with AES takes a lot longer than on its own, but with hashing, you do get a very high level of security, although at this rate you're probably best off using AES on its own rather than Dilithium, or even Dilithium on its own to save performance.

Benchmark	Score		Benchmark	Score	
Post_Quantum.Dilithium.d2AesKeyGeneration	666157		Post_Quantum.Dilithium.d2KeyGeneration	326004.7	
Post_Quantum.Dilithium.d2AesPrivateKeyRecovery	78395.13		Post_Quantum.Dilithium.d2PrivateKeyRecovery	81975.73	
Post_Quantum.Dilithium.d2AesPublicKeyRecovery	56412.26		Post_Quantum.Dilithium.d2PublicKeyRecovery	53294.81	
Post_Quantum.Dilithium.d2AesSign	1869907		Post_Quantum.Dilithium.d2Sign	1561200	
Post_Quantum.Dilithium.d2AesVerify	797980.3	Total Score	Post_Quantum.Dilithium.d2Verify	393273.7	Total Score
		3468851.69			2415748.94
Benchmark	Score		Benchmark	Score	
Post_Quantum.Dilithium.d3AesKeyGeneration	1359256		Post_Quantum.Dilithium.d3KeyGeneration	972176.6	
Post_Quantum.Dilithium.d3AesPrivateKeyRecovery	121800		Post_Quantum.Dilithium.d3PrivateKeyRecovery	174450	
Post_Quantum.Dilithium.d3AesPublicKeyRecovery	71796.84		Post_Quantum.Dilithium.d3PublicKeyRecovery	87989.34	
Post_Quantum.Dilithium.d3AesSign	4825641		Post_Quantum.Dilithium.d3Sign	3204359	
Post_Quantum.Dilithium.d3AesVerify	1734934	Total Score	Post_Quantum.Dilithium.d3Verify	800432.2	Total Score
		8113427.84			5239407.14
Benchmark	Score		Benchmark	Score	
Post_Quantum.Dilithium.d5AesKeyGeneration	2850128		Post_Quantum.Dilithium.d5KeyGeneration	964902	
Post_Quantum.Dilithium.d5AesPrivateKeyRecovery	178026.6		Post_Quantum.Dilithium.d5PrivateKeyRecovery	144286	
Post_Quantum.Dilithium.d5AesPublicKeyRecovery	97976.52		Post_Quantum.Dilithium.d5PublicKeyRecovery	65805.66	
Post_Quantum.Dilithium.d5AesSign	6291543		Post_Quantum.Dilithium.d5Sign	2511678	
Post_Quantum.Dilithium.d5AesVerify	2713209	Total Score	Post_Quantum.Dilithium.d5Verify	975009.8	Total Score
		12130883.1			4661681.46

### 5.3.6. CRYSTALS-Kyber Results

Unlike Dilithium, we don't see as much of a performance gap when using Kyber with AES, although here the Kyber-768Aes benchmark is higher than what is expected from the other benchmarks. This could be due to the awkward key size and the need for padding to wrap the key. Despite this, Kyber is surprisingly efficient.

Benchmark	Score		Benchmark	Score
Post_Quantum.Kyber.k1024AesEncapsulatedPrivateKeyGen	19037.61953		Post_Quantum.Kyber.k1024EncapsulatedPrivateKeyGen	133299.1
Post_Quantum.Kyber.k1024AesEncapsulatedPublicKeyGen	273122.7178		Post_Quantum.Kyber.k1024EncapsulatedPublicKeyGen	124591.3
Post_Quantum.Kyber.k1024AesKeyGen	21570.82276		Post_Quantum.Kyber.k1024KeyGen	107380.1
Post_Quantum.Kyber.k1024AesUnwrapKey	150021.5359		Post_Quantum.Kyber.k1024UnwrapKey	149130.6
Post_Quantum.Kyber.k1024AesWrapKey	120077.615	Total Score	Post_Quantum.Kyber.k1024WrapKey	122276.9
		583830.311		
				636677.9366
Benchmark	Score		Benchmark	Score
Post_Quantum.Kyber.k512AesEncapsulatedPrivateKeyGen	95848.43041		Post_Quantum.Kyber.k512EncapsulatedPrivateKeyGen	45342.46
Post_Quantum.Kyber.k512AesEncapsulatedPublicKeyGen	102465.2261		Post_Quantum.Kyber.k512EncapsulatedPublicKeyGen	41920.67
Post_Quantum.Kyber.k512AesKeyGen	80603.32677		Post_Quantum.Kyber.k512KeyGen	42759.67
Post_Quantum.Kyber.k512AesUnwrapKey	107428.7574		Post_Quantum.Kyber.k512UnwrapKey	64727.43
Post_Quantum.Kyber.k512AesWrapKey	65334.00308	Total Score	Post_Quantum.Kyber.k512WrapKey	53453.2
		451679.7437		
				248203.4229
Benchmark	Score		Benchmark	Score
Post_Quantum.Kyber.k768AesEncapsulatedPrivateKeyGen	159246.824		Post_Quantum.Kyber.k768EncapsulatedPrivateKeyGen	89649.88
Post_Quantum.Kyber.k768AesEncapsulatedPublicKeyGen	146878.8827		Post_Quantum.Kyber.k768EncapsulatedPublicKeyGen	84215.36
Post_Quantum.Kyber.k768AesKeyGen	130934.8335		Post_Quantum.Kyber.k768KeyGen	72974.52
Post_Quantum.Kyber.k768AesUnwrapKey	156686.959		Post_Quantum.Kyber.k768UnwrapKey	99351.92
Post_Quantum.Kyber.k768AesWrapKey	144180.7919	Total Score	Post_Quantum.Kyber.k768WrapKey	93620.51
		737928.2911		
				439812.1954

### 5.3.7. BIKE Results

We can quickly compare the BIKE results to the Kyber results, which provide the same functions, but obviously implemented in different ways. We can see that BIKE is way more expensive than Kyber, but BIKE is notorious for being slow, but with this comes a high level of security as the key sizes are large, but a performance hit this high does not seem worth it as of now.

Benchmark	Score		Benchmark	Score
Post_Quantum.BIKE.bike128KeyEncapsulation	22571169.08		Post_Quantum.BIKE.bike192KeyEncapsulation	57360313
Post_Quantum.BIKE.bike128KeyGenerator	6826869.641		Post_Quantum.BIKE.bike192KeyGenerator	28921416
Post_Quantum.BIKE.bike128UnwrapKey	19230001.3		Post_Quantum.BIKE.bike192UnwrapKey	54915226
Post_Quantum.BIKE.bike128WrapKey	458926.3029	Total Score	Post_Quantum.BIKE.bike192WrapKey	1597239
		49086966.32		
				142794193.8
Benchmark	Score			
Post_Quantum.BIKE.bike256KeyEncapsulation	124749974.3			
Post_Quantum.BIKE.bike256KeyGenerator	82065015.38			
Post_Quantum.BIKE.bike256UnwrapKey	128277031.3			
Post_Quantum.BIKE.bike256WrapKey	3551253.851	Total Score		
		338643274.8		

### 5.3.8. Rainbow Results

As we can see that the performances here vary a lot, especially between Rainbow 3 and 5, but this is expected as the key and signatures sizes are a lot bigger. However, I think in terms of performance and security, Rainbow is a well-rounded candidate.

Benchmark	Score	Benchmark	Score
Post_Quantum.Rainbow.r3CircumKeyGeneration	284729168.8	Post_Quantum.Rainbow.r5CircumKeyGeneration	633205787.5
Post_Quantum.Rainbow.r3CircumSign	27046828.14	Post_Quantum.Rainbow.r5CircumSign	104439471
Post_Quantum.Rainbow.r3CircumVerify	37491379.38	Post_Quantum.Rainbow.r5CircumVerify	83281482.37
	Total Score		Total Score
	349267376		820926740.9
Benchmark	Score	Benchmark	Score
Post_Quantum.Rainbow.r3ClassicKeyGeneration	285948237.5	Post_Quantum.Rainbow.r5ClassicKeyGeneration	761535762.5
Post_Quantum.Rainbow.r3ClassicSign	26513792.44	Post_Quantum.Rainbow.r5ClassicSign	100381782.5
Post_Quantum.Rainbow.r3ClassicVerify	6623978.263	Post_Quantum.Rainbow.r5ClassicVerify	12398746.31
	Total Score		Total Score
	319086008		874316291.3
Benchmark	Score	Benchmark	Score
Post_Quantum.Rainbow.r3CompKeyGeneration	407933433.3	Post_Quantum.Rainbow.r5CompKeyGeneration	897303887.5
Post_Quantum.Rainbow.r3CompSign	21385732.33	Post_Quantum.Rainbow.r5CompSign	99008836.36
Post_Quantum.Rainbow.r3CompVerify	30173169.87	Post_Quantum.Rainbow.r5CompVerify	82197356.25
	Total Score		Total Score
	459492336		1078510080

### 5.3.9. AES Counter Results

Disclaimer: I didn't randomise the plaintext during the benchmark, so encryption is a lot faster than it should be. Besides this, it is one of the most used algorithms to date, but unfortunately will be rendered useless by quantum computing.

Pre_Quantum.AES_CTR.keyGeneration	36897.84	
Pre_Quantum.AES_CTR.encryption	116.9319	
Pre_Quantum.AES_CTR.decryption	133.0846	Total Score
		37147.86

### 5.3.10. SHA-256 with ECDSA Results

Already we can see why hash functions, even though classic algorithms, will still play a high part in quantum cryptography, and why a lot of post-quantum algorithms utilise SHA-2 or SHA-3 in some way. Since there is no decryption or reversals, hashing is extremely fast, whilst still providing good security levels. However, it may be recommended to start using higher keys once quantum computers release to combat the potential danger,

Benchmark	Score	
Pre_Quantum.SHA256_ECDSA.ecdsaSign	2249005	
Pre_Quantum.SHA256_ECDSA.ecdsaVerify	2631913	
Pre_Quantum.SHA256_ECDSA.keyGeneration	2201062	
Pre_Quantum.SHA256_ECDSA.sha256Hashing	55273.1	Total Score
		7137253.1

### 5.3.11. RSA Results

As we can see, RSA is a relatively fast algorithm, as long as it's not dealing with digital signatures. RSA is quick with encryption, but quick slow decrypting and usually is used with other algorithms to speed this up.

Benchmark	Score	
Pre_Quantum.RSA.generateKey	5486734795	
Pre_Quantum.RSA.encrypt	336486.0697	
Pre_Quantum.RSA.decrypt	18076452.7	
Pre_Quantum.RSA.sign	18179199.5	
Pre_Quantum.RSA.verify	396890.7093	Total Score
		<b>5523723824</b>

### 5.3.12. Sha-3 Results

We can see the benefits of using SHA-3 over the SHA-2 family like SHA-256. The SHA-3 algorithm is using a 512-bit key, so not only provides more security but also maintains the high performance of SHA-2. SHA-2 is still more popular due to the new age of SHA-3, but in the future SHA-3 will probably be implemented nearly everywhere.

Benchmark	Score	
Pre_Quantum.Sha3.sha3Hashing	55936.75	
Pre_Quantum.Sha3.sha3KeyGeneration	2255704	
Pre_Quantum.Sha3.sha3Sign	2239423	
Pre_Quantum.Sha3.sha3Verify	2632245	Total Score
		<b>7183308</b>

### 5.3.13. TwoFish Results

TwoFish is a reliable algorithm that is very quick due to its high optimisation. It is a small algorithm and due to this has small key sizes, so it prioritising performance over security.

Benchmark	Score	
Pre_Quantum.TwoFish.keyGeneration	36799950	
Pre_Quantum.TwoFish.encrypt	176052.4	
Pre_Quantum.TwoFish.decrypt	177121	
Pre_Quantum.TwoFish.generateMAC	121225.6	Total Score
		<b>37274349</b>

## 5.4. Was My Project a Success?

I believe my project was a success. I've implemented everything I hoped to include, even introducing an extra two post-quantum algorithms I initially didn't consider. I haven't encountered any final issues with my project. The only other feature I would have loved is to run the program through an executable JAR file, unfortunately, this was too troublesome. I believe I made the correct decisions, albeit not obvious immediately at times.

The main goal of my project was to answer whether we should switch to quantum algorithms sooner rather than later. I believe if my project provided a solid foundation towards an answer then it was a success. So, do I believe we should make the jump to post-quantum cryptography now? Yes, and no. Whilst I am happy to sacrifice some performance over security, it's not always feasible in every situation. I believe we should make the jump regarding some aspects of our lives, such as secure communications like texting or passwords



and file storage. These don't need to be necessarily quick; we would rather have the extra security so it would make sense to implement them in these fields. About Wi-Fi communication, phone calls, or anything that needs to be in real-time, I believe that making that jump isn't worth the performance drop-off. There is a large performance difference between the eras of cryptography, one that a modern computer yet cannot catch up to be on par with the performance levels we have with classic algorithms now. I think in couple more years given more advancements in computers on the algorithms themselves, we can fully become quantum resistant.

## **6. Acknowledgements**

I would like to thank my project supervisor Paul Barry for his help throughout the project and the friendly chats we had every week.

I would also like to thank Conor McKenna, for reminding me that no matter how hard my project gets, and how much I may struggle, I will never feel the pain of being a Tottenham fan.

I declare that all material in this submission, e.g. thesis/essay/project/assignment, is entirely my own work except where duly acknowledged. I have cited the sources of all quotations, paraphrases, summaries of information, tables, diagrams, or other material, including software and other electronic media in which intellectual property rights may reside. I have provided a complete bibliography of all works and sources used in the preparation of this submission I understand that failure to comply with the Institute's regulations governing plagiarism constitutes a serious offence.

Student Name : Cavan Phelan

Student Number: C00249198

Student Signature: Cavan Phelan