University of Sheffield

An Analysis of Post-Quantum Cryptographic Schemes for Real World Applications

Jude Gibson

Supervisor: Bhagya Wimalasiri

A report submitted in partial fulfilment of the requirements for the degree of BSc in Computer Science by Jude Gibson

in the

Department of Computer Science

May 29, 2024

Declaration

All sentences or passages quoted in this report from other people's work have been specifically acknowledged by clear cross-referencing to author, work and page(s). Any illustrations that are not the work of the author of this report have been used with the explicit permission of the originator and are specifically acknowledged. I understand that failure to do this amounts to plagiarism and will be considered grounds for failure in this project and the degree examination as a whole.

Name: Jude Gibson

Signature: 

Date: 29/05/2024

Abstract

The impending advent of a cryptographically relevant quantum computer presents risks to sensitive communications as well as stored data. The ability to efficiently break modern encryption schemes such as RSA or Diffie-Hellman means that a new standard of security needs to be implemented as soon as possible. Since the 1990s, researchers have been developing and analysing many different post-alternatives to modern security primitives.

The aims of this project have been to implement some of the most promising post-quantum primitives and analyse their performance for use in a post-quantum alternative to the Controller-Pilot Data Link Communications (CPDLC) application used in aviation. The main measurements used in this analysis included public key size, handshake runtime and memory consumption.

During my research, I found that the field of Lattice-based cryptography presented some of the most promising primitives for post-quantum cryptography, however due to their relatively young age, other methods with stronger security guarantees should also be explored as options.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The field of quantum computing first emerged in the early 1980s, exploring how the unique characteristics of quantum mechanics could be utilised in information processing. While the implications of such a computer are exciting, there is also concern to be had with the impending jump in computation power.

In 1994, Peter Shor published his eponymously named Shor's algorithm. This quantum algorithm is capable of computing the integer factorisation problem and the discrete logarithm problem in polynomial time[3]. The difficulty of both of these problems are exploited by many modern cryptographic primitives such as Diffie-Hellman, RSA and DSA, so efficiently computing these problems leads to efficiently breaking these security primitives as well.

The two main types of primitives that are under threat are encryption schemes such as Key Encapsulation Mechanisms (KEM) or Key Exchange protocols (KEX); and Digital Signature schemes (DS). KEMs and KEXs are used in generating a key that is shared between both parties and ensuring that this key remains secure from potential attackers. This provides confidentiality to the transmission. DSs are used to ensure that the other party is who they claim to be, that the message has not been tampered with and they they cannot deny that they have sent any amount of data. In other words, they provide authentication, integrity and non-repudiation guarantees to the transmission. Many different post-quantum alternatives have been developed for both primitive types, and they have been analysed and scrutinised to determine how their security would hold up against an attacker. In 2016, the National Institute of Standards and Technology (NIST) started a global competition to encourage researchers and developers to publish their own post-quantum primitives. By the end of 2017, 69 submissions had been made[4]. Only 7 of those still remain in consideration, 4 having already been selected for standardisation.

Even though a powerful enough quantum computer doesn't yet exist, it is still of utmost importance to update global standards for cybersecurity. This is because modern attackers could be stealing data that they are currently unable to decrypt, with the hopes of doing so once a commercial quantum computer is developed. This premise is referred to as an "harvest now, decrypt later" attack or retrospective decryption[5]. To reduce this risk, standards need to updated now so that even if sensitive data continues to be stolen, it will remain

secure. In addition, the time it would take to transition to a new post-quantum standard is likely to be long and uncertain enough that the risk is simply to high to wait. In a 2019 review, the US National Academy of Sciences stated that *"prioritisation of the developments, standardisation, and deployment of post-quantum cryptography is critical for minimising the chance of a potential security and privacy disaster"* [6].

1.1 Aims and Objectives

Due to the implications of Shor's algorithm, extensive research has been conducted into finding other problems that quantum computers cannot effectively compute, so that new primitives can be built from them. These post-quantum primitives can be divided into 5 distinct families, each based on a different problem or group of problems. These are: lattice-based cryptography, code-based cryptography, hash-based cryptography, supersingular elliptic curve isogeny and multivariate cryptography. The work that I have completed has involved implementing some of the most promising primitives that have been published and analysing them to determine which would be the most appropriate for use in communications within the aviation sector.

In the past, aviation has been renowned for being averse to technological upgrades; only recently has there been a serious effort to move from analogue to digital communication [**?**]. The current standard for two way communication between a pilot and a ground controller is the Controller-Pilot Data Link Communication (CPDLC) application. This protocol provides absolutely no confidentiality or authentication when transmitting data from one side to the other[**?**]. This means that somebody with enough knowledge and malicious intent could potentially hijack these transmissions, using the sensitive data or feeding false data to a pilot without their knowledge. For this reason, it seems to me that CPDLC is in need of a successor with the security properties guaranteed when communicating over the internet using TLS. Due to the upgrade averse nature of this industry, it is important that such a protocol remains secure for years to come. This is the motivation to provide a protocol with primitives founded in post-quantum cryptography. Should the advent of a relevant quantum computer occur, it should not have any effect on my CPDLC alternative.

Once I created a framework to analyse my selected primitives, I began testing them do learn more about their performance and where their strengths and weaknesses lie. When testing the performance of the KEMs, I measured the public key sizes, the runtime of the keypair generation and the memory consumption of the keypair generation. The measurements for the DSs were similar but I also included the size of the digital signature itself, as this is transmitted alongside the public encryption keys so its size has a large impact in the overall performance of the protocol.

1.2 Overview of the Report

The rest of this report will outline the research I conducted into post-quantum cryptography, the work I did in analysing my selected primitives and the results and conclusions that I gathered from my work.

Chapter 2 describes much of the relevant literature and existing work in the field. Chapter 3 describes my analysis of the project requirements and how I planned my project. Chapter 4 details the results that I collected. Chapter 5 details the analysis that I gathered from my research and my final conclusions.

Chapter 2

Literature Survey

2.1 Background

2.1.1 Power of the Quantum Computer

Much of this chapter outlines the risk of quantum algorithms and the countermeasures that can be put into place to resist them. However, I believe it is important to understand the power of the quantum computer before talking about any of these topics. A classical computer can be thought of as operating on a singular bitstring according to some program. This bitstring is simply some arbitrary length string of binary digits, each with a definitive state[7]. The registers of a classical computer are therefore 1-dimensional. The value of a single bit, or qubit, in a quantum computer is at each given time in a 'superposition', meaning its state is uncertain. This idea holds for strings of bits, giving rise to many states with a certain probability. A quantum algorithm is able to perform computation on all of these states at once. A quantum register must therefore be two-dimensional, as it must account of each of the possible states. A simple model to describe the register of a quantum computer is 2-dimensional Hilbert space[7].

$$H = H_1 = C \oplus C$$

Once the computation of the quantum algorithm is complete, the complex state of the system must be transformed back into a bitstring that represents the output of the algorithm. This process is called 'measurement' and is based on the quantum principle that the behaviour of a particle is uncertain until it has been observed. The final state of the algorithm is[7]

$$v = \sum_{I \in I_n} \alpha_I |I\rangle$$

where I is some output and α_I is the amplitude of said output.

This process is non-deterministic in nature and therefore yields bitstrings according to some probability distribution P_v. If I is the desired output, then the probability of the

measurement returning I is [7]

$$P_v(I) = |\alpha_I|^2 / \sum_{J \in I_n} |\alpha_j|^2$$

This means that the output of a given quantum algorithm converges to the output with the highest amplitude. If the 'non-desired' outputs have amplitudes higher than 0, then the correctness of the output is uncertain with some probability. This is currently the largest issue holding back the development of larger scale quantum computers capable of computing problems that require much more complex state space and more qubits. However research into quantum computing is gaining more attention than ever. In 2019 Google announced their Sycamore chip which contained 53-qubits[8] and in December 2023, Harvard researchers successfully created 'quantum circuits' which are more efficient in correcting the errors caused by the measurement process[9].

2.1.2 Scope of the Problem

When talking about post-quantum cryptography, it is vital to understand which aspects of cybersecurity are at risk so we can effectively provide alternatives to them. Shor's algorithm is important to consider in this discussion because it is able to reduce the computation of both the integer factorisation problem and the discrete logarithm problem to polynomial time[3]. The efficient computation of these problems means that an attacker with a quantum computer would be able to break any cryptographic primitives that work by exploiting these problems. This includes Rivest-Shamir-Ableman (RSA), the Digital Signature Algorithm (DSA) and Diffie-Hellman. This means the main primitives that are under threat are encryption schemes and digital signature schemes.

Interestingly, primitives based on hashes are believed to be quantum resistant. Due to their lossy nature, breaking a hash function would require a pre-image attack, where the input of the has function is somehow retrievable. This is not believed to be within the grasp of quantum computers, or any computer that we know of for that matter, therefore hash-based primitives are secure and actually are the basis for some post-quantum digital signature schemes.

2.1.3 NIST Post-Quantum Cryptography Competition

In 2016, the National Instituate for Standards and Technology (NIST) announced that they were holding a global competition intended to encourage the development and subsequent standardisation of several quantum resistant primitives[10]. By the end of 2017, 69 submissions had been made. While the competition is still ongoing at the time of writing, there have been 3 rounds of eliminations (the 4th round is still ongoing). During the 3rd round, the first official standardisations were made[4]. The 4 primitives that were standardised were: CRYSTALS-KYBER, CRYSTALS-DILITHIUM, Falcon and SPHINCS+. KYBER is the only key encapsulation mechanism in this list; the rest are digital signature schemes. It

is also important to understand that the only one of these primitives not based in lattice problems is SPHINCS+, which is a hash-based primitive. The aim of the current 4th round is to introduce more diversity into the list of standardised primitives should the security of lattice-based cryptography be found to be insecure at a later date. As such, the 4 algorithms that were part of the 4th round at its beginning were 3 code based KEMs (McEliece, HQC and BIKE) and another KEM based in supersingular elliptic curve isogeny (SIKE). However due to an attack published in August 2022, SIKE was found not to be secure and was subsequently removed from the competition[11].

2.2 Families of Post-Quantum Cryptography

Research into post-quantum cryptography has been ongoing since the 1990s and as such, several different families of primitives exist. There are 5 in total and I will outline them all briefly, explaining the most promising families in more detail.

2.2.1 Lattice-Based Cryptography

Some of the most promising post-quantum primitives are those based in lattices. In brief, a lattice is a set of points in multi-dimensional space defined by some other set of vectors. The sums of these vectors is how the points within the lattice is defined. Lattices defined in 2 dimensions and small vectors are relatively simple, but when scaled to several hundred dimensions with much larger vectors, the complexity scales quickly.

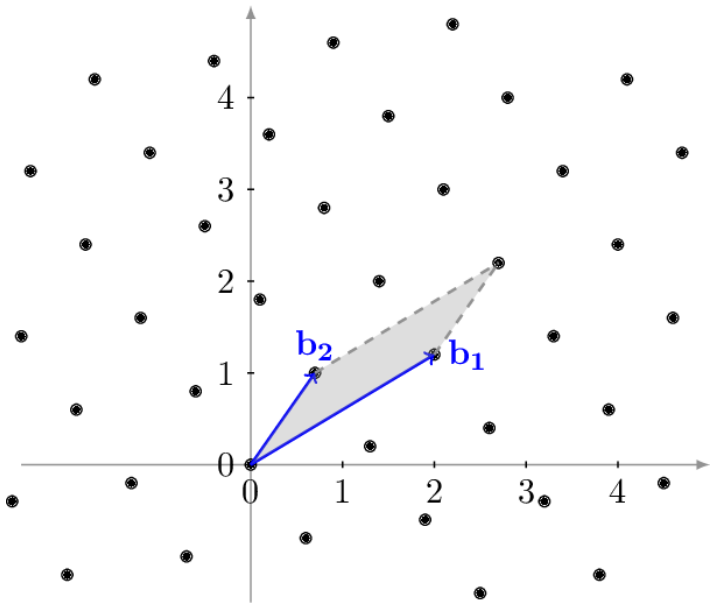

Figure 2.1: An example of a simple lattice construction[1]

Out of the 4 post-quantum primitives standardised by NIST, 3 of them were based in lattices. This is largely due to their impressive efficiency and scalability. CRYSTALS-KYBER

and CRYSTALS-DILITHIUM are two of these algorithms. They are both based on the shortest vector problem (SVP) where given a point on the lattice, you must find the point closest to it. Since the points are not explicitly defined, in order to find this you must search the space by summing the vectors that define the lattice. Using vectors that are close but not equal to define the lattice makes this problem even harder.

However, since security guarantees are often presumed based on how well a system fairs against attacks over some amount of time, the security guarantees of lattice-based cryptosystems are fairly weak. The first cryptosystem based on lattices was proposed in 1996 by Miklós Ajtai[12], however the first cryptosystem based on lattices with proven security guarantees based on worst-case hardness assumptions wasn't proposed until 2005 by Oded Regev[13]. This makes it a relatively young family of cryptography and therefore the security guarantees are not yet well understood. It is worthwhile noting that lattice-based cryptosystems with strong, provable security guarantees do exist[14]. However they are almost all too inefficient to be used in any kind of practical setting.

2.2.2 Code-Based Cryptography

Code-based cryptography is the oldest family within post-quantum cryptography. The idea was presented by Robert McEliece in 1978 as a part of his eponymously named KEM[15], however at the time his work didn't get much attention. When research into post-quantum cryptography became more common in the 1990s, researchers began looking at McEliece's KEM as a possible candidate for a post-quantum encryption scheme.

McEliece's algorithm is based on introducing errors into a plaintext according to the private key, which is some binary Goppa code. The public key is a random permutation of this Goppa code and is used to introduce the errors into the plaintext. Only the owner of the private key is able to remove these errors.

This family of post-quantum cryptography has been studied and tested since the late 1970s and as such, its security guarantees are well understood. However there are other limitations that make code-based systems impractical. As shown in section 4.1.2, the public keys generated by these systems are far too large for most applications, ranging from around 100 kilobytes to 1 megabyte. For comparison, a public key used by the Diffie-Hellman key exchange range from 1024 bits to 2048 bits. Not only this, but code-based systems are known to be among the most memory intensive systems within post-quantum cryptography.

2.2.3 Hash-Based Cryptography

As stated in section 2.1.1, hash functions are widely believed to be quantum secure. Hash functions need to be resistant to attacks on their pre-image, that is to say that given a hashed output y, an attacker should not be able to find an input x such that $h(x) = y$. The best known algorithms for computing the pre-image of a hash function run in $O(2^n)$ time. A quantum pre-image attack was published in 2020 that reduced this time to $O(2^{n/3})$ [16]. While this is faster, it is still an exponential time attack so the quantum security of hash

functions appears to remain in tact.

When creating a digital signature scheme, it is important to ensure that information cannot be leaked about the private key when using the same key more than once. If this is the case, the digital signature scheme must generate a new private key each time the signing function is called. Hash functions that are also collision resistant, which is to say that finding two distinct inputs x and x' that map to the same output $h(x) = h(x')$ is infeasible, can be used to create digital signature schemes that are able to sign multiple messages using a single private key. This makes them very effective as signature schemes, since computing a private key for a signature scheme is expensive.

It is worth noting that hash-based cryptography does not extend to encryption schemes due to the lossy nature of hash functions. Much of the input is lost, so no message can be recovered from the hashed output.

2.2.4 Supersingular Elliptic Curve Isogeny

Elliptic curve cryptography was introduced in the 1980s as a way to reduce the key size while maintaining the security of primitives that take advantage of the discrete logarithm problem. This was applied to primitives such as Diffie-Hellman and DSA following the 2005 RSA conference. Elliptic curve cryptography is in widespread use in modern cryptosystems due to its advantages over other primitives based on algorithms that use integer factorisation. However they are no less vulnerable to quantum algorithms such as Shor's algorithm.

Isogeny based cryptography expands on the idea of elliptic curves by taking advantage of the properties of isomorphic curves[17]. An elliptic curve over a finite field is given by the formula:

$$y^2 = x^3 + ax + b$$

Two elliptic curves are isomorphic if they have the same j-invariant. A curve's j-invariant is given by the formula:

$$j(E) = 1728\frac{4a^3}{4a^3 + 27b^2}$$

An isogeny ϕ between two elliptic curves is a rational map $\phi : E \to E'$[1]. An isogeny graph can then be constructed from these isogenies, showing how curves can be mapped to eachother. A cryptographic system can then be construsted. Alice and Bob both start from a known curve E_0 and take secret, random walks within the isogeny graph to some curves E_a and E_b. They share these secret walks with one another and take their peer's walk so that arrive at the same secret curve E_s only known to them.

This class of post-quantum cryptography was extremely promising for a time as they boasted very small public keys and ciphertexts. Supersingular Isogeny Key Exchange (SIKE) was submitted to the NIST competition in 2016 and survived for 6 years before being broken in 2022. The attack was able to retrieve the secret key in only an hour using a single core [11]. Due to this attack, there are currently no primitives of this class in the NIST competition

[1] An isogeny class can only contain either ordinary or supersingular elliptic curves, not both.

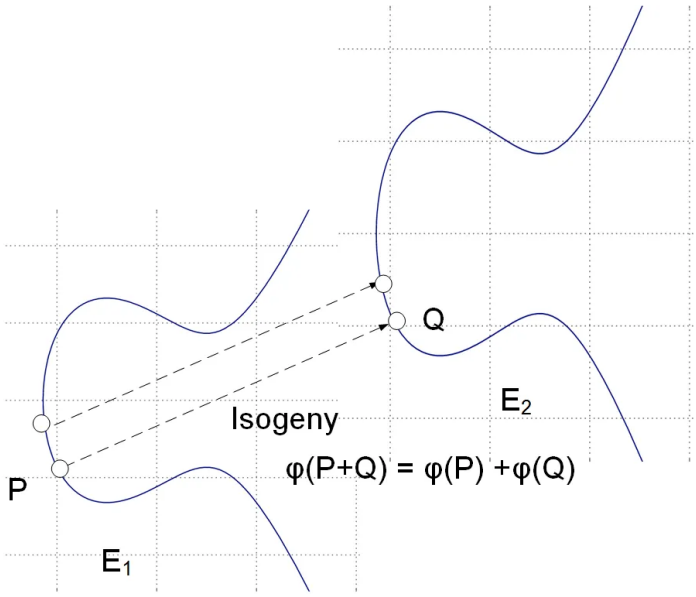

Figure 2.2: An example of an elliptic curve isogeny[2]

and as such, I will be disregarding isogeny-based cryptography in my implementation.

2.2.5 Multivariate Cryptography

Multivariate cryptography is the study of using multivariate quadratic polynomials over a finite field. A set of quadratic variables are used as the public key as such[18]:

$$P = (p1(w1, ..., wn), ..., pm(w1, ..., wn))$$

where each P_i is a non-linear polynomial in $\mathbf{w} = (w_1, ..., w_n)$:

$$zk = pk(\mathbf{w}) := \sum_i P_{ik} w_i + \sum_i Q_{ik} w_i^2 + \sum_{i>j} R_{ijk} w_i w_j$$

The private key of the system consists of data that has been collected during the generation of the public key. This allows the encryption function for the cryptosystem to be easily inverted. Inverting such a polynomial map without the private key is equivalent to solving a set of quadratic polynomials over a finite field, which is believed to be an NP-complete problem.

The first example of a multivariate cryposystem was the C* system proposed in 1988 by Tsutomu Matsumoto and Hideki Imai[19]. It was broken in 1995, but laid the groundwork for many subsequent systems that improved the security of the idea. One of these was the Rainbow signature scheme which was proposed in 2004 by Jintai Ding and Dieter Schmidt[20]. It was a finalist in the NIST PQC competition before being unexpectedly broken in October 2022 by an attack outlined in the now infamous paper "Breaking Rainbow Takes a Weekend

on a Laptop"[21]. As a result, there are currently no primitives based on multivariate cryptography in the NIST PQC competition.

2.3 Symmetric and Asymmetric Cryptography

Modern cryptography schemes can be divided into two distinct categories: symmetric and asymmetric cryptography. In asymmetric systems, each party involved with the communication generates their own private key and public key. When sending a message, the recipient's public key is used to encrypt the message. Since this encryption is one-way, only the recipient's private key can be used to decrypt the ciphertext and reveal the plaintext. Symmetric systems use a private key x and a peer public key y to compute a shared key $k = y^x$. This shared key is always equal to the other party due to the way it is generated, so it can be used with a symmetric cipher such as the Advanced Encryption Scheme (AES). Diffie-Hellman is an example of an asymmetric key exchange system that can be used to derive a symmetric key.

This project will be entirely focused on symmetric encryption, due to the fact that in the years since Diffie and Hellman's paper, asymmetric encryption has been phased out of use. This is largely due to the smaller key sizes and the resulting speed up of execution time. All of the post-quantum encryption schemes that I will be evaluating, such as CYSTALS-KYBER and McEliece, are examples of symmetric encryption schemes. This also applies to certificate schemes, such as the HMAC scheme I will be utilising.

Digital signature schemes on the other hand make use of asymmetric cryptography. The lack of efficiency of asymmetric cryptography has little effect on digital signature schemes as they are only used a handful of times; in my protocol they are only used during the key exchange. This is in direct contrast to encryption schemes in which the derived shared key is used as long as the communication endures.

2.4 Key Encapsulation Mechanisms

Key Encapsulation Mechanisms are a type of symmetric encryption scheme much like Key Exchange schemes. The main difference between a KEM and a KEX is how the shared key is generated. In a KEX, both parties use their own private key alongside their peer's public key to generate a shared key individually. In a KEM a single party generates a shared key using its own private key and some other public key. This shared key is generated as a ciphertext, meaning that it cannot be used in this form. It must first be decapsulated by the peers private key before it can be used in any meaningful way. This makes KEMs a symmetric key encryption scheme where the symmetric key is transmitted using an asymmetric key encryption scheme.

Key encapsulation mechanisms are entirely symmetric systems and as such are preferred to key exchange schemes which are asymmetric systems that are used to derive a symmetric key. Since symmetric key schemes are preferred, post-quantum encryption schemes are almost

all symmetric. SIKE is an example of a post-quantum system that is asymmetric, however a symmetric version does also exist[22].

2.5 Hybrid Cryptography

The security guarantee of a given cryptographic primitive or class of primitive is often assumed based on the primitive's age and how much research has gone into trying to break it. If a primitive has withstood decades of scrutiny without being broken, then it can be assumed with high certainty that the primitive is secure. This is the case for countless classical primitives such as Diffie-Hellman, RSA and DSA. If current assumptions that a viable quantum computer can be created are proven wrong, then these systems would likely remain secure for a very long time. This security guarantee unfortunately cannot be assumed for many post-quantum primitives since they are relatively young. As mentioned in section 2.2.4, SIKE is a good example for this. It was believed to be secure for many years before its security was broken by a devastating key recovery attack[11]. A similar story can be found in the multivariate digital signature scheme Rainbow[21].

This is the main motivation for a hybrid encryption scheme, using a classical primitive as well as a post-quantum primitive. This can be done using a Key Derivation Function (KDF), which will be discussed in greater detail in section 2.6. Finding a hybrid key would require the attacker to find both the post-quantum and the classical keys first, before using them with the same KDF to generate the final hybrid key. This ensures that a shared key remains resistant to both classical and quantum computers.

KEMTLS is an example of a propsed hybrid cryptographic protocol that relies on the KEM to authenticate the communication rather than a digital signature[23]. This solves a common problem with hybrid protocols which is that the signature size can often be large and unwieldy. Removing it altogether reduces the size of the overall data package significantly. While worth mentioning, this is not a path I will be taking with my protocol, as my intention is to evaluate the performance of both post-quantum encryption schemes and digital signature schemes.[2]

2.6 Key Derivation Functions

A key derivation function is a deterministic function that takes a key and a salt and derives a new key. Typically, this is done using some master key with the intention of producing several sub-keys, each with a different salt. This can be used for key formatting or key lengthening among other uses. For our purposes however, it is important to note that the salt to this algorithm can be any psuedorandom string, including a second input key. This makes it an incredibly useful function for creating a hybrid key. A KDF is formalised as[24]:

$$KDF(\sigma, r, c, \ell) = K(1)||K(2)||...||K(t),$$

[2]KEMTLS still requires the use of digital signatures for storing long term keys.

where σ represents the source key to be input, r represents the salt, ℓ represents how many key bits should be generated and c represents context information, which may contain data related to the application calling the KDF.

For my implementation, I will be using HKDF which is a KDF implemented using the hash function HMAC[24]. In HKDF, the keys $K(1)$ to $K(t)$ are derived as:

$$PRK = HMAC(r, \sigma),$$

$$K(1) = HMAC(PRK, c||0),$$

$$K(i+1) = HMAC(PRK, K(i)||c||i), 1 \leq i < t$$

2.7 Digital Signatures

The rest of this chapter will be focused on different methods of authentication and integrity. Digital signatures provide both of these properties as well as non-repudiation. In cryptographic protocols, authentication guarantees that the other party a device communicates with is who they claim to be. In other words, this is how we guarantee that we are not communicating with some malicious third-party that has hijacked the transmission. This is vital to ensure that sensitive information is only transmitted to parties that have permission to access said information. Integrity is the assurance that any data received from another party has not been tampered with. If a malicious user was able to tamper the data undetected it could have disastrous consequences. For example, if an attacker were able to alter the shared key transmitted during a KEM handshake, the recipient would only be able to encrypt and decrypt messages to and from the attacker. In the context of aviation, this could mean that an attacker could issue instructions to a pilot to alter its course or even induce a collision. If the authentication was also compromised, the user would likely consider the attacker to be the correct party instead of a malicious third party. Non-repudiation simply ensures that a party cannot make false claims about whether they were or were not involved in communication at a certain point in time. This is heavily linked with authentication and so once established, can be implied by continued authentication.

A digital signature is a string constructed by the sender's private signature key and the data that they are sending. This signature should be unique to the individual message and the private key of the sender. Much like hash-functions, a digital signature scheme should aim for collision resistance, meaning that finding two signatures s and s' such that $s = s'$ should be infeasible. Since a signature is derived from the transmitted data itself, verifying both integrity and authentication is trivial. The recipient can simply verify the signature using the received data and the sender's public key. The most common classical digital signature schemes are RSA and DSA. Both are efficient and produce small signature sizes, however both are susceptible to attacks using Shor's algorithm and thus need to be updated for the post-quantum era.

CRYSTALS-DILITHIUM is a promising candidate for a post-quantum digital signature

scheme[25]. It is based on lattices which as we already established contains several very difficult problems for quantum computers. DILITHIUM itself is based on the module learning with errors (MLWE) problem, where a vector must be recovered from noisy lattice data. The algorithm is balanced across the board with moderate key/certificate sizes and moderate generation speed for both. When compared to FALCON[26], another lattice-based digital signature scheme, DILITHIUM is a relatively simple algorithm to implement since it doesn't require complex tree structures and samples from a uniform distribution instead of a gaussian. The main advantage of FALCON however is that it boasts much smaller key/certificate sizes.

2.8 Message Authentication Codes

A large drawback of using a digital signature is that they are computationally expensive. They can consume large amounts of memory and, depending on the specific signature scheme, can have long runtimes often upwards of half a second. To counter this issue, we use digital signature schemes only during the initial handshake when the protocol is first called. After this, we switch to message authentication codes (MAC). A MAC is a computationally cheaper alternative to a digital signature, which uses a symmetric key to generate a tag from the transmitted data. However, it does not provide the non-repudiation guarantees that we need our protocol to have. As previously mentioned, since non-repudiation has already been established by the digital signature, continued authentication from the MAC is enough to provide implied non-repudiation.

MAC schemes are based on hash functions like SHA-256 which we established are quantum resistant in section 2.2.3. Therefore, when implementing my protocol, I am able to use established classical MAC schemes. HMAC is the scheme that I have elected to use, due to its common use, simple implementation and its strong security guarantees.

Chapter 3

Planning, Design and Implementation

3.1 Project Requirements

When designing a cryptographic protocol, there are several properties that one should be aiming to guarantee. In my protocol I aim to provide confidentiality, authentication, integrity and non-repudiation. However for a resource constrained environment such as secure aircraft communication, one must also consider ways to create a more efficient pipeline. This is particularly true when it comes to the bottlenecks that will be found in data transmission. CPDLC uses VHF data link mode 2, which has a transmission data rate of 31.5Kbps[27], roughly 4KBps. This means that any data we send via the protocol must be as small as possible, whether that be the ciphertext or the signature.

Confidentiality is provided by the encryption primitives included in the protocol. As stated in section 2.5, a post-quantum primitive should aim to remain secure against both classical and quantum adversaries, so a hybrid encryption key is required.

The digital signature scheme included in the protocol will provide the authentication, integrity and non-repudiation guarantees for the communication. However as stated in section 2.8, digital signature schemes tend to be both large and computationally expensive. In a resource constrained environment like this, these can become serious problems. To circumvent this issue, our digital signature scheme will be used only in the initial handshake and any subsequent communications will be authenticated by message authentication codes.

Finally, this project needs to simulate the data transmission that would be taking place between an aircraft and a ground controller. Therefore, the program will be run across two distinct terminals, running either the server code or the client code. This way, an initial handshake can take place followed by some arbitrary number of messages, all of which will be secure against attacks.

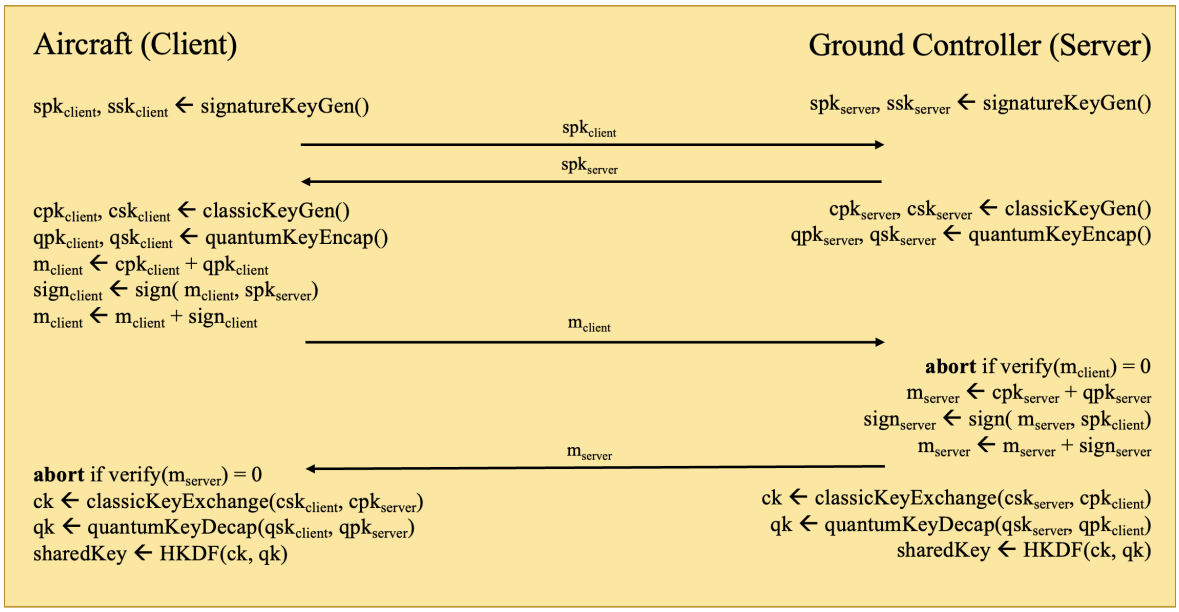

Figure 3.1: An overview of the handshake structure of the protocol

3.2 Design

Figure 3.1 demonstrates the overall structure that the handshake will use. Firstly both parties generate an asymmetric signature key pair. This is done with a post-quantum signature scheme such as DILITHIUM or SPHINCS+. In practice, both parties are likely to already know their peer's public signature key. The key transmission is shown in figure 3.1 for completeness.

Once the signature public keys have been transmitted, both parties generate both classical and quantum encryption key pairs. These will be used to create the final symmetric hybrid key. If certificate verification fails, the communication will be aborted at which point the pilot is free to reattempt the connection. The public keys will be concatenated to create a public key package. The aircraft then creates a signature from the key package using the servers public signature key. This is further concatenated with the key package and everything is sent to the server.

If signature verification fails, the communication will be aborted at which point the pilot is free to reattempt the connection. If the verification succeeds however, then the same process is repeated by the ground controller to transmit the public encryption keys to the aircraft. At this point, both parties are now in possession of each others public encryption keys, so classical key exchange and quantum key decapsulation can occur. This results in both classical and quantum symmetric keys. Both of these will then be input into a key derivation function to produce the final hybrid shared key.

Beyond the initial handshake, the protocol remains relatively straightforward. The hybrid key can be used for both message encryption and decryption using a standard block cipher

and since non-repudiation has been provided by the digital signature scheme, the protocol can switch to the use of message authentication codes to provide authentication and integrity. Once the communication is over, the client will send a specified termination command to the server causing the connection to be severed.

3.3 Implementation

3.3.1 Language and libraries

The protocol itself is written in python using a number of libraries to streamline the implementation. For the cryptographic primitives, I used the PQCrypto library[28] and the cryptography library[29] and for the data transmission, I used the socket library[30]. PQCrypto was developed by several researchers within the field of post-quantum cryptography and aims to collate as many primitives as possible. In particular, there are implementations for all of the NIST standardised primitives as well several other prominent primitives that have either been removed from the competition or that are still in the final round. The cryptography python library contains implementations for many classical primitives including key exchange schemes like Diffie-Hellman and block ciphers such as AES. The socket library provides implementations useful for communicating across networks. For this network communication, I used port 8282 for the handshake and port 8000 for the secure communication. I used 2 different ports because during implementation and testing, I was having issues with data collisions between the end of the handshake and the beginning of the secure communication.

When testing the performance of the different primitives, I used the tracemalloc and linecache libraries contained within the standard python package for finding the memory consumption of the various functions within the project. Finally I used the standard time library when determining the runtime of the functions.

3.3.2 Classes and Methods

When implementing the protocol, I used 5 classes: *Node, Server, Client, SignatureKeys and Protocol*. Server and Client are subclasses that inherit from the Node superclass and aim to simulate the behaviour of and aircraft and ground controller. The Node superclass contains methods for generating all of the keys that will be needed for encryption and digital signing, as well as utility methods for setting and getting all of the different keys that are stored within the class. While the Node class does include a method for generating and setting the signature keys, the signature keys used are actually stored in a seperate python class called SignatureKeys. This is because, as stated in section 3.2, each party is likely to know the other party's public signature prior to communication. For this reason, the public keys and their respective private have been hard coded into the SignatureKeys class.

The server and client classes extend the Node class by introducing methods that encode the behaviour during the handshake and normal runtime. The methods $Server.listen\_for\_key()$ and $Client.send\_key()$ are the methods used during the handshake. The behaviour of these

is shown in figure 3.1. The methods $Server.run\_server()$ and $Client.run\_client()$ are the methods used once the symmetric key has been established. The logic for them is fairly trivial: the server listens for activity on port 8000 and the client sends some message encrypted using the method $Protocol.encrypt()$ and signed using the signing method of the selected primitive. The message is verified using the signature primitive's verify method and decrypted using $Protocol.decrypt()$. Finally a response is sent back to the client using the exact process. If the message sent to the server reads $'close'$, the server sends the same message in response, causing both nodes to terminate.

The *protocol* class contains methods for encryption, decryption, MAC generation and verification, serialisation and deserialisation. The $encrypt()$ method first pads some message to a multiple of 128-bits so that it can be used with AES. A random initial vector (IV) is then generated so that AES can be used with the chain block cipher (CBC) mode. CBC is a secure method of encryption with AES because each plaintext block encrypted by the cipher is first combined the previous ciphertext block by a bitwise XOR operation. The $generate_MAC()$ method is then called to create an authentication certificate for the message. The message is then encrypted with the block cipher before being concatenated with the IV and the certificate. The $decrypt()$ method is the inverse of of this function, first stripping the MAC and IV before verifying the certificate and decrypting the message using AES once again.

Some message serialisation was needed during the handshake because the socket library only sends bit-like values and the keys for Diffie-Hellman and ECDSA did not adhere to this. The $serialize()$ method simply takes a key and encodes it to privacy enhanced mail (PEM) format. $deserialize()$ takes a PEM encoded key and returns the original key.

Chapter 4

Results

4.1 Encryption Schemes

4.1.1 Primitives and Measurements

Once the protocol was fully implemented, I began testing 3 different encryption schemes. I tested the classical KEX Elliptic Curve Diffie-Hellman as a control so that I could get an understanding of modern encryption performance. Then I tested CRYSTALS-KYBER, a lattice-based KEM, and McEliece, a code-based KEM, as post-quantum alternatives. KYBER is the only currently standardised post-quantum encryption primitive and McEliece is currently in the 4th round of the NIST PQC competition. The other 2 encryption primitives in the competition are SIKE, which has been proven to be insecure, and HQC, which the PQCrypto library does not currently support. For these reason, my focus was on KYBER and McEliece.

There were 3 measurements that I decided to use to test the performance of these primitives. Firstly, I tested the public key sizes because, as stated in section 3.1, aircraft communications is a constrained environment with a slow transmission rate. If the public keys of the parties are too large, then the handshake would take far longer than is acceptable. Next, I tested the memory consumption of the primitives. Many aircraft in use today have been in service for many years and as a result, the on board computers may have sub-optimal memory capacity. Therefore, selecting a primitive have uses as little memory as possible is vital. Finally, I tested the overall runtime of the key establishment methods provided by the primitives. The runtime of an algorithm is a good way to gain some understanding of its overall efficiency and performance and while key generation runtime is never longer than reasonable, selecting a primitive that runs quickly is always going to be useful.

4.1.2 Public Key Size

The public key size of each of these primitives is always the same, so I only had to get a single result for each primitive. Table 4.1 shows the results.

As you can see, the sizes vary massively. Using Diffie-Hellman as a baseline, we can observe that both post-quantum alternatives generate public keys far larger than classical

Table 4.1: A table showing the public key sizes of Diffie-Hellman, KYBER and McEliece

	Public Key Size (bytes)
Diffie-Hellman	384
KYBER	1568
McEliece	261120

primitives. Despite being less than ideal, KYBER does outperform McEliece vastly in this test with a public key around 166x smaller. McEliece's massive public key means that for our data rate constrained environment, it is likely to be extremely impractical. In fact to transmit as public key of this size at a rate of 4KBps, it would take ∼65 seconds, which is clearly far longer than is acceptable for our purposes.

4.1.3 Memory Consumption

Next I tested the memory consumption of the 3 primitives. I tried this multiple times and each time I got the same result so once again, I only needed to get a single result for each primitive. Figures 4.1, 4.2 and 4.3 show the most memory consuming lines of code within each of their key generation functions.[1]

Unfortunately, due to the implementation of the cryptography library, the key generation for Diffie-Hellman uses an unexpectedly large amount of memory at 913.6KiB. This library was written entirely in python whereas PQCrypto was written in C and then wrapped in python. C allows the developer to specifically state how memory is allocated so in general, the implementations are more efficient. That being said, the focus of this project is analysing post-quantum alternatives. However, I felt it was important to include this in the results section for completeness and also to highlight that the individual implementation of the primitive does play a part in its performance.

As shown in figures 4.2 and 4.3, the memory cost of McEliece is around 46x larger than KYBER. Breaking the process down, we see that KYBER allocated only 1.6KiB to generating

[1]Note that any lines from the file *tracemalloc.py* were used for collecting these results and thus will be excluded from analysis. These lines always use 0.6KiB of memory.

```
Memory allocation for ECDF keygen:
Top 5 lines
#1: openssl/binding.py:65: 208.0 KiB
    setattr(conditional_lib, attr, getattr(lib, attr))
#2: openssl/binding.py:63: 55.9 KiB
    for attr in dir(lib):
#3: python3.11/enum.py:560: 16.2 KiB
    enum_class = super().__new__(metacls, cls, bases, classdict, **kwds)
#4: email/quoprimime.py:55: 15.1 KiB
    _QUOPRI_MAP = ['=%02X' % c for c in range(256)]
#5: python3.11/locale.py:900: 12.8 KiB
    locale_alias = {
1897 other: 606.3 KiB
Total allocated size: 914.2 KiB
```

Figure 4.1: The most memory consuming lines in the Diffie-Hellman key generation function

```
Memory data for asymmetric key gen:
Top 5 lines
#1: kem/common.py:13: 3.1 KiB
    secret_key = bytes(ffi.buffer(secret_key_buf, lib.CRYPTO_SECRETKEYBYTES))
#2: kem/common.py:12: 1.6 KiB
    public_key = bytes(ffi.buffer(public_key_buf, lib.CRYPTO_PUBLICKEYBYTES))
#3: kem/common.py:6: 0.5 KiB
    public_key_buf = ffi.new("uint8_t [{}]".format(lib.CRYPTO_PUBLICKEYBYTES))
#4: python3.11/tracemalloc.py:560: 0.3 KiB
    return Snapshot(traces, traceback_limit)
#5: python3.11/tracemalloc.py:423: 0.3 KiB
    self.traces = _Traces(traces)
2 other: 0.4 KiB
Total allocated size: 6.3 KiB
```

Figure 4.2: The most memory consuming lines in the KYBER key generation function

```
Memory data for asymmetric key gen:
Top 5 lines
#1: kem/common.py:12: 255.0 KiB
    public_key = bytes(ffi.buffer(public_key_buf, lib.CRYPTO_PUBLICKEYBYTES))
#2: kem/common.py:13: 6.3 KiB
    secret_key = bytes(ffi.buffer(secret_key_buf, lib.CRYPTO_SECRETKEYBYTES))
#3: python3.11/queue.py:258: 3.4 KiB
    class _PySimpleQueue:
#4: python3.11/queue.py:28: 2.6 KiB
    class Queue:
#5: python3.11/queue.py:223: 2.5 KiB
    class PriorityQueue(Queue):
81 other: 25.1 KiB
Total allocated size: 294.9 KiB
```

Figure 4.3: The most memory consuming lines in the McEliece key generation function

its public key and a further 3.1KiB for its secret key. McEliece on the other hand used 255KiB for generating its public key alone and another 6.3KiB went towards generating its secret key. This is clearly not ideal for the environment that I am designing this protocol for, so once again KYBER appears to be the more optimal solution

4.1.4 Runtime

The final measurement that I collected was the average runtime of each of the primitives. Runtime has a tendency to deviate from the average, so I had to collect many different results for these. For Diffie-Hellman I was able to collect 384 measurements of runtime. For KYBER and McEliece respectively, I collected 100 and 120 different times. Table 4.2 shows the mean and variance for each primitive. Figures 4.4, 4.5 and 4.6 show how all of the times compared.

Looking at table 4.2, we can see that KYBER once again proves itself to be a highly

Table 4.2: A table showing the mean and variance of the runtimes of Diffie-Hellman, KYBER and McEliece

	Mean Runtime (s)	**Variance**
Diffie-Hellman	$\sim 1.4 \times 10^{-2}$	$\sim 7.382 \times 10^{-5}$
KYBER	$\sim 8.97 \times 10^{-5}$	$\sim 1.058 \times 10^{-10}$
McEliece	$\sim 6.8 \times 10^{-2}$	$\sim 6.867 \times 10^{-4}$

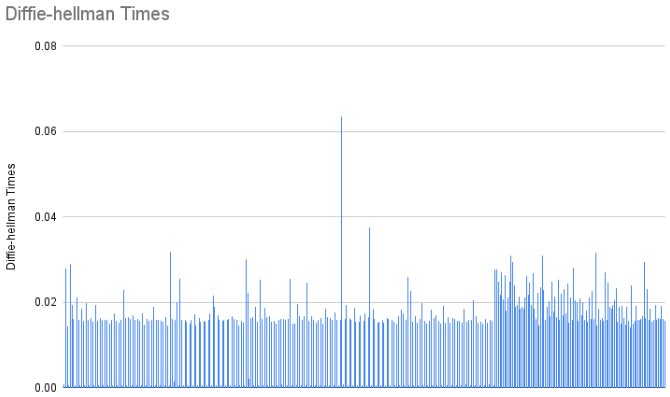

Figure 4.4: A graph showing the runtimes for Diffie-Hellman key generation

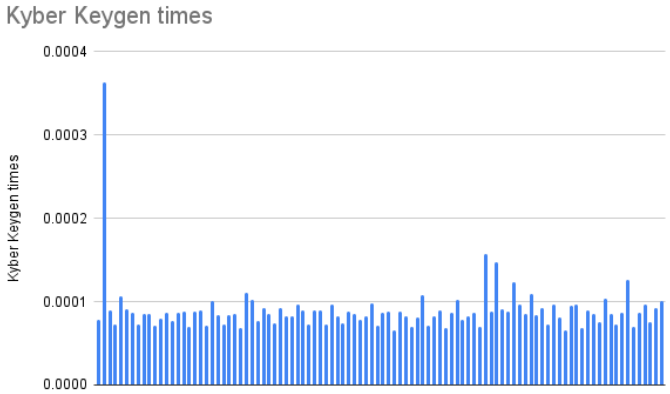

Figure 4.5: A graph showing the runtimes for KYBER key generation

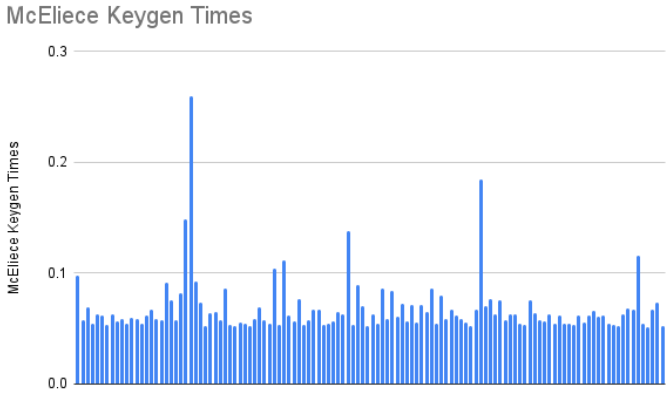

Figure 4.6: A graph showing the runtimes for McEliece key generation

efficient algorithm with an average runtime of only $\sim 8.97 \times 10^{-5}$. Both Diffie-Hellman and McEliece are orders of magnitude slower than this with Diffie-Hellman running at $\sim 1.4 \times 10^{-2}$ and McEliece running at $\sim 6.8 \times 10^{-2}$. While the latter two are significantly slower than KYBER, neither is slow enough to be completely discounted based on runtime. In fact that slowest time that I collected overall was 0.2596 seconds by McEliece, still only a fraction of a second.

It is also worth noting the variances of the different primitives. KYBER has an extremely small variance, showing that it is not only fast, but consistent as well. Diffie-Hellman and McEliece both have higher variances, meaning that they are much more inconsistent than KYBER. Due to the high required margin of safety exhibited by aircraft communications, it is extremely important that the communication protocol deviates as little of possible.

4.2 Digital Signature Schemes

4.2.1 Primitives and Measurements

The other type of primitive that I tested was the post-quantum signature schemes. Again I selected 3 different signature schemes for testing. These were the Elliptic Curve Digital Signature Algorithm (ECDSA), CRYSTALS-DILITHIUM, a lattice based signature scheme, and SPHINCS+, a hash-based signature scheme. One again these consist of 1 classical primitive and 2 post-quantum alternatives respectively.

The measurements for the signature schemes were largely identical, except I measured the signature size as well as the public key size. I still believe that it is important to understand the public key sizes of these algorithms since they will have to stored in a database somewhere, however it is the signature that is actually getting transmitted across the channel. Therefore the size of this signature will have a large impact on the performance of the overall protocol.

4.2.2 Public Key Size and Signature Size

As stated in section 4.1.2, transmitting a large piece of datum across a constrained channel can take longer than is acceptable. Therefore when selecting an appropriate signature scheme, it is important to consider the size of the signature that if being transmitted along with the encryption public keys. As well as this, the signature public keys need to be stored in a database of some kind, so that each party is quickly able to access them to sign the encryption keys. Since an aircraft will need to store all of the public keys for the ATCs along its route, and an ATC needs to store the public keys for all of the aircraft that may fly past it, this database may end up becoming very large. Therefore the size of the public keys of the signature scheme is also an important fact to consider.

Table 4.3 shows the size of the public keys and the signatures of the 3 primitives. We see that ECDSA has a very small public key size and a very small signature size.[2] Using 215 and

[2]The signature size of ECDSA appeared to be non-deterministic, returning sizes between 102bytes and 108bytes with 105 being the mode, suggesting a normal distribution of signature sizes.

Table 4.3: A table showing the public key and signature sizes of ECDSA, DILITHIUM and SPHINCS+

	Public Key Size (bytes)	**Signature Size (bytes)**
ECDSA	215	102-108
DILITHIUM	1760	3366
SPHINCS+	64	49216

105 as the baseline for public key size and siganture size, we see that SPHINCS+ excels with its public key at only 64 bytes, but fails with its signature size at 49,216bytes. A signature of this size would take ∼10 seconds to be transmitted along a 4KB bit rate channel. While 10 seconds isn't a huge amount of time, it is still less than ideal. When an aircraft crosses into a new ATC sector, the amount of time taken to establish connection with the new ground controller should be minimal to prevent the risk of mid-air collisions. DILITHIUM on the other hand generates much smaller signature sizes, allowing the transmission to take place in under a second. This greatly reduces any risk arising from an aircraft flying without direct communication with an ATC.

However, if the public key size is large, storing every public key for a given region may become infeasible. There are currently over 648 ATC towers operating in the United States alone[31]. Storing a DILITHIUM key for every single one of these results in a database of 1,140,480bytes, just over a Megabyte. When you take international flights or metadata about each ATC into account, this database grows even larger very quickly.

4.2.3 Memory Consumption

Memory consumption for digital signature schemes can be broken down into two parts, each regarding a different process. First, I wanted to observe the memory allocation during the key generation process. Then I also tested the memory allocation of the signature generation process.

Figures 4.7, 4.8 and 4.9 show the memory allocation for the signing process. It is unsurprising that the memory allocation is somewhat linked to the public key sizes. SPHINCS+ once again achieves very highly in this regard, only using 0.9KiB of memory if you exclude the data collection code. The public and secret key generation only use 0.2KiB each. The classical primitive ECDSA has the next lowest memory cost at only 2.5KiB, 0.3Kib of which is allocated to both the public and secret keys. Finally, DILITHIUM uses 6.2KiB of memory for the process of key generation, 1.8Kib of which is allocated to the public key and another 3.8Kib of which is allocated to the secret key. While DILITHIUM shouldn't be disqualified based on this larger memory consumption, the extremely low memory costs of SPHINCS+ key generation do make it an attractive option.

Turning to the memory consumption of the signature generation, figures 4.10, 4.11 and 4.12 show how this function performed for each primitive. This time, both post-quantum primitives are out-performed by ECDSA, which only uses a modest 1.7KiB to generate a signature. DILITHIUM uses just over twice this amount at 3.9KiB and SPHINCS+ uses

```
Memory data for signature keypair gen:
Top 5 lines
#1: Project/Node.py:249: 0.9 KiB
    file = open("ECDSA-keygen-time.txt","a")
#2: openssl/ec.py:90: 0.5 KiB
    return ec._CURVE_TYPES[sn]()
#3: openssl/ec.py:181: 0.3 KiB
    return _EllipticCurvePublicKey(self._backend, public_ec_key, evp_pkey)
#4: openssl/backend.py:1034: 0.3 KiB
    return _EllipticCurvePrivateKey(self, ec_cdata, evp_pkey)
#5: python3.11/tracemalloc.py:560: 0.3 KiB
    return Snapshot(traces, traceback_limit)
5 other: 0.8 KiB
Total allocated size: 3.1 KiB
```

Figure 4.7: The most memory consuming lines in the ECDSA key generation function

```
Memory data for signature keypair gen:
Top 5 lines
#1: sign/common.py:14: 3.8 KiB
    secret_key = bytes(ffi.buffer(secret_key_buf, lib.CRYPTO_SECRETKEYBYTES))
#2: sign/common.py:13: 1.8 KiB
    public_key = bytes(ffi.buffer(public_key_buf, lib.CRYPTO_PUBLICKEYBYTES))
#3: python3.11/tracemalloc.py:560: 0.3 KiB
    return Snapshot(traces, traceback_limit)
#4: python3.11/tracemalloc.py:423: 0.3 KiB
    self.traces = _Traces(traces)
#5: sign/common.py:8: 0.2 KiB
    secret_key_buf = ffi.new("uint8_t [{}]".format(lib.CRYPTO_SECRETKEYBYTES))
2 other: 0.4 KiB
Total allocated size: 6.8 KiB
```

Figure 4.8: The most memory consuming lines in the DILITHIUM key generation function

```
Memory data for signature keypair gen:
Top 5 lines
#1: python3.11/tracemalloc.py:560: 0.3 KiB
    return Snapshot(traces, traceback_limit)
#2: python3.11/tracemalloc.py:423: 0.3 KiB
    self.traces = _Traces(traces)
#3: sign/common.py:10: 0.2 KiB
    if 0 != lib.crypto_sign_keypair(public_key_buf, secret_key_buf):
#4: sign/common.py:8: 0.2 KiB
    secret_key_buf = ffi.new("uint8_t [{}]".format(lib.CRYPTO_SECRETKEYBYTES))
#5: sign/common.py:7: 0.2 KiB
    public_key_buf = ffi.new("uint8_t [{}]".format(lib.CRYPTO_PUBLICKEYBYTES))
2 other: 0.3 KiB
Total allocated size: 1.5 KiB
```

Figure 4.9: The most memory consuming lines in the SPHINCS+ key generation function

48.8KiB of memory to generate its signature. Just as before, this roughly falls in line with the overall size of the signature: the larger the signature, the more memory is required to create it.

Overall for this constrained environment, the lower the memory consumption the better. Since the key pair has already been generated, the signature generation is the more important aspect of memory allocation. SPHINCS+ is outperformed by DILITHIUM in this regard by just over 10x. While memory consumption is not the most vital of the measurements I am using, it is still worth taking into consideration, especially given how many different types of aircraft this protocol could apply to.

```
Memory data for signature gen:
Top 5 lines
#1: Project/Client.py:78: 1.0 KiB
    file = open("ECDSA-sign-time.txt","a")
#2: python3.11/tracemalloc.py:560: 0.3 KiB
    return Snapshot(traces, traceback_limit)
#3: python3.11/tracemalloc.py:423: 0.3 KiB
    self.traces = _Traces(traces)
#4: openssl/ec.py:105: 0.2 KiB
    siglen_ptr = backend._ffi.new("unsigned int[]", 1)
#5: openssl/ec.py:104: 0.2 KiB
    sigbuf = backend._ffi.new("unsigned char[]", max_size)
4 other: 0.4 KiB
Total allocated size: 2.3 KiB
```

Figure 4.10: The most memory consuming lines in the ECDSA signature generation function

```
Memory data for signature gen:
Top 5 lines
#1: sign/common.py:39: 3.3 KiB
    return bytes(ffi.buffer(signature_buf, signature_len))
#2: python3.11/tracemalloc.py:560: 0.3 KiB
    return Snapshot(traces, traceback_limit)
#3: python3.11/tracemalloc.py:423: 0.3 KiB
    self.traces = _Traces(traces)
#4: sign/common.py:35: 0.2 KiB
    if 0 != lib.crypto_sign_signature(signature_buf, signature_len, message, len(message), secret_key):
#5: sign/common.py:32: 0.2 KiB
    signature_buf = ffi.new("uint8_t [{}]".format(lib.CRYPTO_BYTES))
1 other: 0.1 KiB
Total allocated size: 4.5 KiB
```

Figure 4.11: The most memory consuming lines in the DILITHIUM signature generation function

```
Memory data for signature gen:
Top 5 lines
#1: sign/common.py:39: 48.1 KiB
    return bytes(ffi.buffer(signature_buf, signature_len))
#2: python3.11/tracemalloc.py:560: 0.3 KiB
    return Snapshot(traces, traceback_limit)
#3: python3.11/tracemalloc.py:423: 0.3 KiB
    self.traces = _Traces(traces)
#4: sign/common.py:35: 0.3 KiB
    if 0 != lib.crypto_sign_signature(signature_buf, signature_len, message, len(message), secret_key):
#5: sign/common.py:38: 0.2 KiB
    signature_len = struct.unpack("Q", ffi.buffer(signature_len, 8))[0]
1 other: 0.2 KiB
Total allocated size: 49.4 KiB
```

Figure 4.12: The most memory consuming lines in the SPHINCS+ signature generation function

4.2.4 Runtime

Similarly to section 4.2.3, I measured the average runtimes for both the key pair generation function and the signature generation function. When measuring the runtimes for key generation, I was able to measure 104 times, 156 times and 121 times for ECDSA, DILITHIUM and SPHINCS+ respectively. When measuring signature generation I measured 34, 50 and 40 times respectively. Table 4.4 shows the mean runtimes and variances for the key pair generation functions.

Table 4.4: A table showing the mean runtimes and the variance for the key generation of ECDSA, DILITHIUM and SPHINCS+

	Mean Runtime (s)	**Variance**
ECDSA	0.0007116	8.1266×10^{-8}
DILITHIUM	0.0001226	8.5887×10^{-9}
SPHINCS+	0.0124527	3.6164×10^{-6}

As you can see, DILITHIUM is easily the fastest out of any of the algorithms. Not only this, but it produced the lowest variance, making it an incredibly consistent algorithm. ECDSA is the second fastest, around 7x slower than DILITHIUM and producing a variance around 10x larger than DILITHIUM. Finally, SPHINCS+ took just over 1/100th of a second to generate a key pair, around 1000x slower than DILITHIUM, while also producing a variance around 4000x larger. While this is easily the slowest of the 3 key generation functions, ~0.01 seconds is hardly noticeable by most people and therefore, SPHINCS+ should not be judged harshly based on this. Figures 4.13, 4.14 and 4.15 show the overall times I collected.

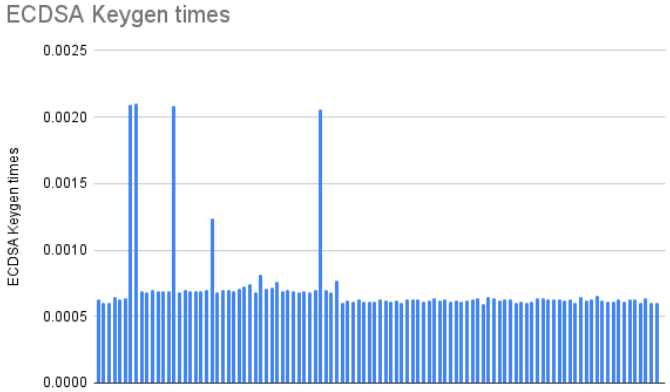

Figure 4.13: A graph showing the runtimes for ECDSA key generation

However, the more important aspect of a signature scheme for our purposes is the signature generation. This is the function that must be executed for every handshake between a pilot and a ground controller. Table 4.5 shows the mean runtimes and variance of the signature generation functions.

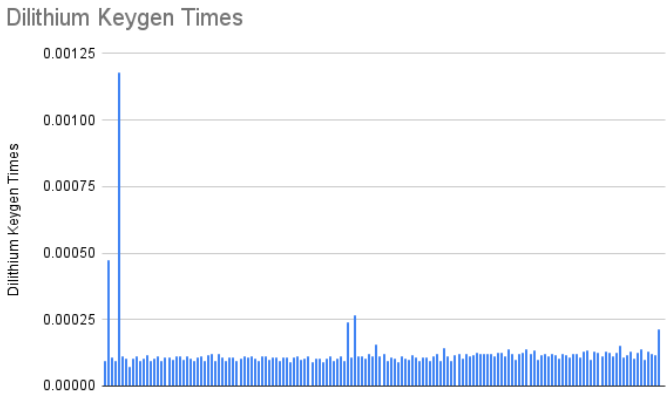

Figure 4.14: A graph showing the runtimes for DILITHIUM key generation

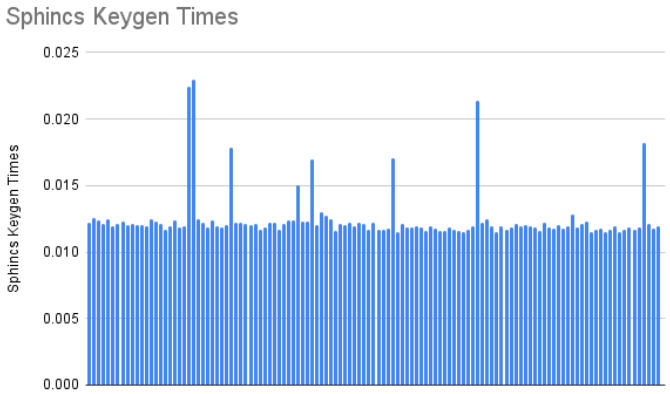

Figure 4.15: A graph showing the runtimes for SPHINCS+ key generation

As you can see, both ECDSA and DILITHIUM are extremely fast with DILITHIUM only being 2x as fast as ECDSA. Looking at the variances, DILITHIUM also appears to less consistent compared to ECDSA. SPHINCS+ on the other hand really struggles to compete with the other 2 primitives, taking on average over a quarter of a second to sign a message. This is another big issue with SPHINCS+ and definitely holds it back from being an effective alternative to ECDSA. Figures 4.16, 4.17 and 4.18 show the overall runtimes for the signature generation.

Table 4.5: A table showing the mean runtimes and the variance for ECDSA, DILITHIUM and SPHINCS+

	Mean Runtime (s)	**Variance**
ECDSA	0.00078374	1.0738×10^{-8}
DILITHIUM	0.00035104	5.38×10^{-8}
SPHINCS+	0.27083	2.0216×10^{-3}

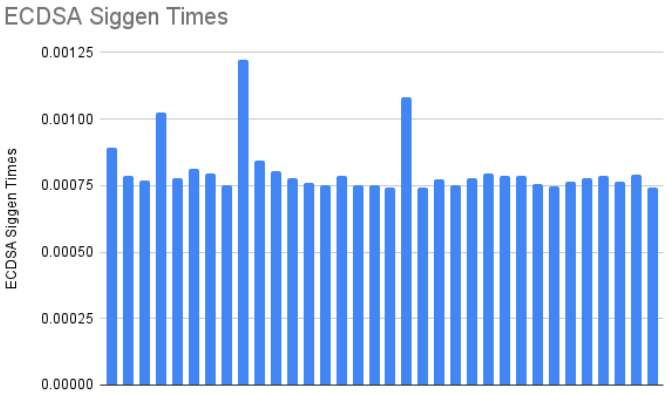

Figure 4.16: A graph showing the runtimes for ECDSA signature generation

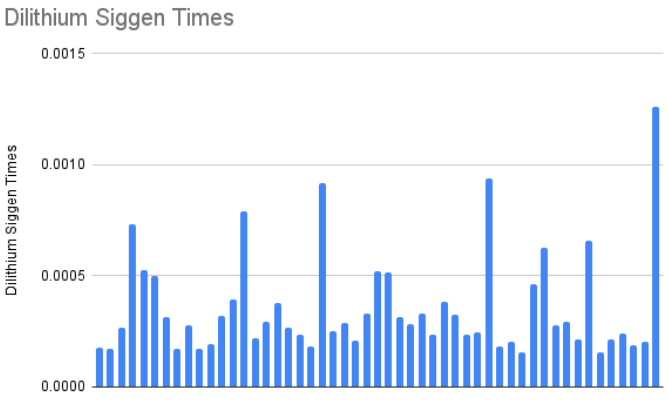

Figure 4.17: A graph showing the runtimes for DILITHIUM signature generation

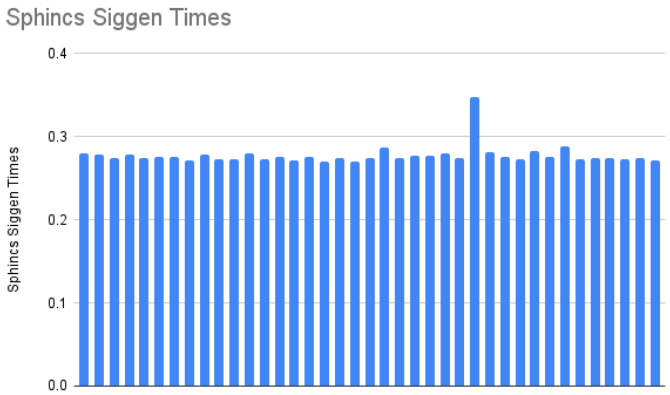

Figure 4.18: A graph showing the runtimes for SPHINCS+ signature generation

4.3 Protocol Runtime

Once I had tested each of the primitives independently, the final piece of data that I collected was the total runtime from the start of the handshake to the point where a final shared key was agreed upon. I conducted this test 20 times for each of the possible combinations of post-quantum primitives. Table 4.6 shows the average times and variances.

Table 4.6: A table showing the mean runtimes and the variance for each of the primitive combinations

	Mean Runtime (s)	**Variance**
KYBER-DILITHIUM	0.022702	0.000071890
KYBER-SPHINCS+	0.57357	0.018255
McEliece-DILITHIUM	0.11087	0.0014425
McEliece-SPHINCS+	0.68574	0.0025660

Perhaps unsurprisingly, when the 2 lattice-based primitives are paired the handshake is incredibly efficient, only taking ∼0.02 seconds for both parties to agree on a shared key. The pairing of McEliece and DILITHIUM also runs fast, at just over 0.1 seconds. When SPHINCS+ is used however, the handshake becomes much slower at just under 0.6 seconds with KYBER and just under 0.7 seconds with McEliece.

Chapter 5

Analysis and Conclusions

5.1 Results Analysis

For an environment with constraints like aircraft communications, it is vital to consider the wide range of computers that may be found aboard aircraft and also the low data rate used in wireless communications. Computing cryptographic keys can be expensive and signatures even more so. This means whichever primitives that I select need to be efficient to compute, with relatively small public keys and signatures.

KYBER and DILITHIUM both performed extremely well across nearly every test I consucted. Both are lattice-based primitives, which would explain their efficiency and both have been standardised by NIST. Compared to classical primitives, both suffer from larger public keys. However when compared to other post-quantum alternatives, their public keys are actually among some of the smallest. DILITHIUM suffered from a larger memory consumption during the key generation when compared to both ECDSA and SPHINCS+, however it outperformed SPHINCS+ dramatically when I looked at the memory allocation during signature generation.

McEliece is a code-based KEM that was first outlined in 1978, meaning that its security has been studied and scrutinised for over 4 decades. The fact that it remains unbroken tells us that we can be fairly certain that its security will hold in the future. However, McEliece suffers from an extremely large public key. As stated in section 4.1.2, transmitting a McEliece public key across a 4KiB bit rate channel would take ∼65 seconds, far too long for the purposes of aircraft communication. For this reason alone, we can say that McEliece is not appropriate for my protocol. McEliece may instead be better suited in an environment where the parties are already aware of each others public encryption keys, or environments with wireless channels with large enough bandwidth to accommodate such a large key.

SPHINCS+ has some properties that make it extremely promising, however there are other aspects that let it down somewhat. Firstly, much like McEliece, its security is well understood by the cryptography community since it is a hash-based primitive. It benefits from a very small public key which is inexpensive and fairly quick to produce. However, it suffers from a large signature and a very slow signature generation time. While these

problems do make SPHINCS+ a less promising candidate than DILITHIUM, I do not think that SPHINCS+ would not work at all for this purpose. The largest issue facing it is that with its large signature size, it would take take ∼10 seconds to transmit a signed key. This brings the total time for the handshake to around 20 seconds considering that both parties must be verified in this way. However if the pilot was able to begin the handshake while still in range of the previous ATC, this issue may be circumvented somewhat.

5.2 Conclusion

Overall, I believe the most appropriate primitive pair for a post-quantum alternative to CPDLC is KYBER-DILITHIUM. Both primitives are incredibly efficient in terms of memory consumption and runtime, and both boast some of the smallest public keys in post-quantum cryptography. However, since both primitives are based in lattices, their security is still relatively young and untested. To circumvent this, I recommend a hybrid encryption key using a classical key establishment primitive such as elliptic curve Diffie-Hellman. Should the security of lattice-based cryptography be proven to be ineffective, then I recommend an update to the protocol, replacing DILITHIUM with SPHINCS+. Since it is based on hashes, its security has a much stronger guarantee and while it does have problems, they are not so devastating that a pilot cannot continue their work mostly uninterrupted.

Bibliography

[1] J. Freeman, "The progression of lattice-based cryptography," 2018. [Online]. Available: https://api.semanticscholar.org/CorpusID:210835133

[2] B. Buchanan, "Goodbye to elliptic curves? ... not, quite ... here come isogenies," 2022, [Online; accessed May 29, 2024]. [Online]. Available: https://medium.com/asecuritysite-when-bob-met-alice/goodbye-to-elliptic-curves-not-quite-here-come-isogenies-88881ad66e48

[3] P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM Rev.*, vol. 41, 1995. [Online]. Available: https://api.semanticscholar.org/CorpusID:2337707

[4] M. Yesina, Y. Ostrianska, and I. Gorbenko, "Status report on the third round of the nist post-quantum cryptography standardization process," *Radiotekhnika*, 2022.

[5] Olejnik and Riemann, "Quantum computing and cryptography," *TechDispatch*, p. 2, 2020.

[6] M. H. et al., "Quantum computing: Progress and prospects," *Washington, Dc: The National Academies Press,*, p. 9, 2019.

[7] S. Hallgren and U. Vollmer, "Quantum computing. in: Bernstein, d.j., buchmann, j., dahmen, e. (eds) post-quantum cryptography." *Springer*, pp. 19,20, 2009.

[8] A. K. B. R. e. a. Arute, F., "Quantum supremacy using a programmable superconducting processor," *Nature 574*, 2019.

[9] M. D. L. Daniel Bochen Tan, Dolev Bluvstein and J. Cong, "Compiling quantum circuits for dynamically field-programmable neutral atoms array processors," *Quantum 8, 1281 (2024).*, 2024.

[10] L. C. et al., "Report on post-quantum cryptography," *NIST IR 8105*, 2016.

[11] W. Castryck and T. Decru, "An efficient key recovery attack on sidh," in *International Conference on the Theory and Application of Cryptographic Techniques*, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:258240788

[12] M. Ajtai, "Generating hard instances of lattice problems (extended abstract)," *Electron. Colloquium Comput. Complex.*, vol. TR96, 1996. [Online]. Available: https://api.semanticscholar.org/CorpusID:6864824

[13] O. Regev, "On lattices, learning with errors, random linear codes, and cryptography," in *Symposium on the Theory of Computing*, 2005. [Online]. Available: https://api.semanticscholar.org/CorpusID:53223958

[14] D. Micciancio, "1 . 2 lattice-based cryptography," 2008, p. 3. [Online]. Available: https://api.semanticscholar.org/CorpusID:2148407

[15] R. J. McEliece, "A public key cryptosystem based on algebraic coding theory," 1978. [Online]. Available: https://api.semanticscholar.org/CorpusID:56502909

[16] P. Wang, S. Tian, Z. Sun, and N. Xie, "Quantum algorithms for hash preimage attacks," *Quantum Eng.*, vol. 2, 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:216266853

[17] L. D. Feo, "Mathematics of isogeny based cryptography," *ArXiv*, vol. abs/1711.04062, 2017. [Online]. Available: https://api.semanticscholar.org/CorpusID:9252863

[18] J. Ding and B.-Y. Yang, "Multivariate public key cryptography," 2009, pp. 1–3. [Online]. Available: https://api.semanticscholar.org/CorpusID:118059106

[19] T. Matsumoto and H. Imai, "Public quadratic polynominal-tuples for efficient signature-verification and message-encryption," in *International Conference on the Theory and Application of Cryptographic Techniques*, 1988. [Online]. Available: https://api.semanticscholar.org/CorpusID:45907222

[20] J. Ding and D. S. Schmidt, "Rainbow, a new multivariable polynomial signature scheme," in *International Conference on Applied Cryptography and Network Security*, 2005. [Online]. Available: https://api.semanticscholar.org/CorpusID:6571152

[21] W. Beullens, "Breaking rainbow takes a weekend on a laptop," *IACR Cryptol. ePrint Arch.*, vol. 2022, p. 214, 2022. [Online]. Available: https://api.semanticscholar.org/CorpusID:247525240

[22] "Sike – supersingular isogeny key encapsulation," https://sike.org/, [Online; accessed May 29, 2024].

[23] S. Celi, A. Faz-Hernández, N. Sullivan, G. Tamvada, L. Valenta, T. Wiggers, B. Westerbaan, and C. A. Wood, "Implementing and measuring kemtls," *IACR Cryptol. ePrint Arch.*, vol. 2021, p. 1019, 2021. [Online]. Available: https://api.semanticscholar.org/CorpusID:236963342

[24] H. Krawczyk, "Cryptographic extraction and key derivation: The hkdf scheme," in *IACR Cryptology ePrint Archive*, 2010, pp. 6,11. [Online]. Available: https://api.semanticscholar.org/CorpusID:16306886

[25] L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, P. Schwabe, G. Seiler, and D. Stehlé, "Crystals-dilithium: A lattice-based digital signature scheme," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2018, pp. 238–268, 2018. [Online]. Available: https://api.semanticscholar.org/CorpusID:3593118

[26] P.-A. Fouque, J. Hoffstein, P. Kirchner, V. Lyubashevsky, T. Pornin, T. Prest, T. Ricosset, G. Seiler, W. Whyte, and Z. Zhang, "Falcon: Fast-fourier lattice-based compact signatures over ntru," 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID:231637439

[27] A. V. Gurtov, T. Polishchuk, and M. Wernberg, "Controller–pilot data link communication security," *Sensors (Basel, Switzerland)*, vol. 18, p. 3, 2018. [Online]. Available: https://api.semanticscholar.org/CorpusID:29163226

[28] "Pqcrypto library," https://libpqcrypto.org/python.html, [Online; accessed May 29, 2024].

[29] "Cryptography library," https://cryptography.io/en/latest/, [Online; accessed May 29, 2024].

[30] "Socket library," https://docs.python.org/3/library/socket.html, [Online; accessed May 29, 2024].

[31] "National air traffic control system appendix e," https://www.jeffco.us/DocumentCenter/View/860/Airport-Master-Plan-Update-Appendix-E-National-Air-Traffic-Control-System-PDF?bidId=, [Online; accessed May 29, 2024].