Johns Hopkins University

Post-Quantum *age* with Kyber768

Madeline Estey, Sofia Rest, and Jamie Stelnik

EN.601.445: Practical Cryptographic Systems

Taught by Professor Matthew Green in Spring 2024

Abstract: *age* is a popular UNIX-style file encryption tool designed to be a simpler, more compact alternative to certain GnuPG use cases. Its recipient-identity scheme relies on the X25519 Key Encapsulation Mechanism (KEM) and provides tamper resistance and conditional authentication. We propose a robust, quantum-safe improvement to *age*'s existing KEM that combines X25519 with Kyber768, an asymmetric post-quantum KEM based on a variant of the module learning with errors (M-LWE) lattice problem. Our hybrid implementation of *age* maintains Indistinguishability Under Adaptive Chosen Ciphertext Attack (IND-CCA2) security, meets the National Institute of Standards and Technology's (NIST) security level 3 (i.e., roughly equivalent to AES 192), and ensures both classical and post-quantum cryptographic security (Schwabe and Mann, 2020). Our implementation is publicly available at https://github.com/srest2021/practical-crypto-project.

May 10, 2024

Contents

Chapter 1

Introduction

age is a popular UNIX-style file encryption tool designed to be a simpler, more compact alternative to certain GnuPG use cases. Its recipient-identity scheme relies on the CCA-secure X25519 KEM and provides tamper resistance and conditional authentication. Recipient-identity schemes are those where a public key is generated by some unique information about the identity of a user (e.g. a user's phone number). Additionally, a scheme being "CCA-secure" means that it is resistant to chosen-ciphertext attacks.

Our goal is to improve the existing **Key Encapsulation Mechanism (KEM)** in *age* by making it secure against post-quantum attacks. We considered the Kyber768 and X-Wing protocols when deciding how to accomplish this. Both Kyber768 and X-Wing are post-quantum KEMs (although X-Wing is specifically a hybrid KEM) that rely on the hardness of lattice problems.

Our research has led us to propose a robust, quantum-safe improvement to *age*'s KEM that combines X25519 with Kyber768. Our hybrid implementation of *age* maintains **Indistinguishability under Adaptive Chosen Ciphertext Attack (IND-CCA2)** security, meets NIST category 1 requirements (i.e., roughly equivalent to AES128), and ensures both classical and post-quantum cryptographic security (Schwabe and Mann, 2020).

In this report we will begin with Chapter 2: Background of *age*, Kyber768, and X-Wing. We will then describe our design and implementation (Chapter 3: Design) combining X25519 with Kyber768. To further understand the process, Chapter 4: Flow will then run through an example scenario where the improved *age* implementation is used. We will explain our choice of Kyber768 over X-Wing and analyze any existing flaws or limitations in our implementation in Chapter 5: Analysis and Limitations. To conclude, we will discuss further research in post-quantum KEM hybrid implementations in Chapter 6: Future Research. We will conclude our proposal with Chapter 7: Conclusion.

Chapter 2

Background

2.1 About *age*

2.1.1 Conceptual Summary of *age*

The file encryption tool and format *age*, written by Filippo Valsorda and publicly available on GitHub, is a simple and secure UNIX tool designed to fill the gap left by the hybrid-encryption software program GnuPG (Valsorda, 2022b). The base version of *age* allows the user to take charge of managing small, explicit keys, generate and encrypt with a secure passphrase, encrypt to one or more recipients, and even encrypt using *ssh-rsa* and *ssh-ed25519 SSH* public keys. *age* also offers the option of using plugins to support different encryption schemes. For example, *age-plugin-yubikey* supports hardware PIV tokens such as YubiKeys, and *age-plugin-sss* provides key splitting using Shamir's Secret Sharing.

At its core, *age* uses **recipients** to denote "public values... that a file can be encrypted to" and **identities** to denote "private values... that allow decrypting a file encrypted to the corresponding recipient" (Valsorda, 2023). *age-keygen(1)* generates an identity I, which contains (a) a random Curve25519 scalar e, and (b) $e * g$, where g is a fixed base point on the elliptic curve Curve25519 and $*$ denotes scalar multiplication. The scalar e is the secret key $\mathrm{I_{SK}}$ and $e * g$ is the public key $\mathrm{I_{PK}}$. The identity I may be "converted" to a recipient simply by setting the recipient's public key $\mathrm{R_{PK}}$ to the identity's public key $\mathrm{I_{PK}}$. At a high level, an identity generated by *age-keygen(1)* is essentially a public-private key pair, with a recipient being the public key in that pair. These keys are then used to define a KEM by which the symmetric key material used to encrypt a file can be securely transmitted.

2.1.2 *age* Cryptographic Scheme

age uses a KEM to securely exchange symmetric keys between parties. A KEM consists of three algorithms:

1. KeyGen(): Generates a public key PK and the corresponding private key SK.
2. Encapsulate(PK): Generates a shared secret s. Outputs a shared key h and a peer share c.
3. Decapsulate(SK, c): Recovers the shared secret s and shared key h.

In *age*, KeyGen() is used to generate an identity I (which may also be converted to the corresponding recipient R), while Encapsulate($\mathrm{R_{PK}}$) and Decapsulate($\mathrm{I_{SK}}$, c) are used to secure transmission of the shared secret s and, consequently, recovery of the shared key h. The one-time random file key used to encrypt the file can be "wrapped" and "unwrapped" (i.e., encrypted and decrypted) by the shared key.

The KEM for native *age* key pairs is **X25519**, which is an elliptic-curve Diffie-Hellman (ECDH) key exchange using Curve25519. The X25519 KEM provides IND-CCA2 security, which guarantees "ciphertext indistinguishability against adaptive chosen ciphertext attacks." It also allows users to create an "offline KEM-DEM public key encryption scheme" via encapsulation and decapsulation (Valsorda, 2022a).

Suppose that Alice would like to encrypt a file so that only Bob can decrypt it. While Bob knows his own identity $\mathrm{I_{PK}}$ and $\mathrm{I_{SK}}$, Alice only knows Bob as a recipient $\mathrm{R_{PK}}$. The cryptographic scheme that Alice and Bob would follow is described below.

1. Alice does **encapsulation** using Bob's public key $\mathrm{R_{PK}}$:
   (a) Alice generates an ephemeral as a random scalar e, and the peer share as $c = e * g$.
   (b) The shared secret s is computed.
   $$s = e * \mathrm{R_{PK}} \tag{2.1}$$
   (c) The shared wrapping key h is computed.
   $$h = \text{HKDF-SHA-256}(\text{input key material} = s, \text{salt} = c||\mathrm{R_{PK}}) \tag{2.2}$$
2. Alice encrypts the file M so that only Bob can decrypt it:
   (a) Alice generates a random one-time file key r.
   (b) The plaintext file M is encrypted with the file key r and a fixed nonce N to get ciphertext C.
   $$C = \text{ChaCha20Poly1305.Encrypt}(M, \text{key} = r + N) \tag{2.3}$$
   (c) The file key r is wrapped (i.e., encrypted) using the shared wrapping key h.
   $$w = \text{ChaCha20Poly1305.Encrypt}(r, \text{key} = h) \tag{2.4}$$
3. Alice constructs the output file, which consists of the ciphertext C and a header containing a MAC, the peer share c, and the wrapped file key w.
4. Bob does **decapsulation** using his own identity I and the data contained in the output file header:
   (a) The shared secret is recovered.
   $$s = c * \mathrm{I_{SK}} \tag{2.5}$$
   (b) The shared wrapping key h is recovered.
   $$h = \text{HKDF-SHA-256}(\text{input key material} = s, \text{salt} = c||\mathrm{I_{PK}}) \tag{2.6}$$

5. Bob decrypts the ciphertext C to recover the original plaintext M:

   (a) The file key r is unwrapped (i.e., decrypted) using the shared wrapping key h.

   $$r = \text{ChaCha20Poly1305.Decrypt}(w, \text{key} = h) \tag{2.7}$$

   (b) The ciphertext C is decrypted to plaintext M using the file key r.

   $$M = \text{ChaCha20Poly1305.Decrypt}(C, \text{key} = r + N) \tag{2.8}$$

2.1.3 AEAD mode in *age*

To additionally provide tamper resistance, *age* encrypts files using secure AEAD mode. A file is split into 64 KiB chunks, which are each encrypted with an AEAD. During decryption, *age* verifies each chunk against a MAC before releasing the plaintext. This verification use done using a scheme called STREAM, which places a counter in the AEAD nonce (Hoang et al., 2015). The scheme ensures that the message was not partially tampered with, and prevents any unauthenticated plaintext from being fed to tar or the shell.

2.1.4 *age* and Authentication

The significance of *age* may be partly derived from its conditional authentication properties. *age* provides a guarantee that an attacker cannot generate a file that is decryptable without knowledge of the recipient (Valsorda, 2022a). This is due to the fact that R_{PK} is necessary to derive the shared key and, consequently, the wrapping key for the file. Ensuring the secrecy of a recipient string thus ensures that no attacker can forge *age*-encrypted files under this recipient. Assuming recipient secrecy, authentication is guaranteed not only for all X25519 recipients, but also for all X25519+Kyber768 recipients (Valsorda, 2022a). We note, however, that encrypting a file to multiple recipients exposes knowledge of the recipients to each other, enabling them to encrypt files to each other (Valsorda, 2022a).

age covers a multitude of use cases, not all of which require authentication. Secrets stored next to open-source source code do not require authentication, nor does a local password store or self-authenticating data that starts with, for example, a secret token (Valsorda, 2022a). *age* may also be used to safely delete files by encrypting them and ensuring that the key is kept away from permanent storage. However, *age's* authentication properties assure that *age*-encrypted data backed up through cloud storage is safe from inspection or tampering by the cloud provider (although the backup may be entirely replaced). *age*'s versatile, simple design allows users to apply it to many different use cases, although care must be taken not to assume authentication properties for every case.

2.2 About Kyber768

2.2.1 High-Level Description of Kyber768

Kyber768 is an IND-CCA2-secure post-quantum public key encryption system and KEM that placed as a finalist in NIST's 2022 PQC competition (Schwabe and Mann, 2020). Kyber allows communicating parties to establish a shared secret between them. The algorithm

can be used at three different security levels, with the Kyber512, Kyber768, and Kyber1024 variants. Kyber768, the variant we are focusing on in our project, has a NIST security level of 3 and is roughly equal to AES 192 security-wise. For this variant, the secret keys are 2400 bytes in size, the public keys are 1184 bytes, and the ciphertexts are 1088 bytes long (Peikert, 2016).

2.2.2 Kyber768 and M-LWE

The Kyber process is based on the hardness of the **Module Learning With Errors (M-LWE)** problem, a variant of the learning with errors lattice problem, with respect to "classical and quantum random oracle models" (Bos et al., 2017). The Plain **Learning With Errors (LWE)** problem uses lattices as its mathematical foundation. An n-dimensional lattice L is any subset of R^n (where R is the set of real numbers) that is both:

1. an *additive subgroup*: $0 \in L$, and $-x, x + y \in L$ for every $x, y \in L$
2. *discrete*: every $x \in L$ has a neighborhood in R^n in which x is the only lattice point

2.2.3 Description of Lattice-Based Cryptography

Lattices are used in cryptography because problems related to them are difficult to solve, which then ensures the security of the scheme that is based on them. One of the most utilized, hard-to-solve lattice problems, for example, is the **Shortest Vector Problem (SVP)**. The goal of this problem is to find the shortest vector of a lattice given a basis of the lattice. In the Plain LWE problem, the goal is to recover a secret $s \in \mathbb{Z}_a^n$ given a group of linear equations on s, where each equation is correct up to a small additive error. Each equation follows the form of: $a_i * s + e_i$, where e is the error and a is a random vector from $\mathbb{Z}_q^n$ (q is some positive integer modulus). The additive error is what makes this problem difficult. Without the error, s could be found by constructing systems of equations and using traditional Gaussian elimination in polynomial time (i.e., time required is a simple polynomial function of the size of the input).

M-LWE on the other hand has a similar overall system structure as LWE, but some of the underlying algebra is different. In M-LWE, the scalars in LWE are replaced by ring elements. A polynomial ring is $R_q := \mathbb{Z}_q[x]/f(x)$ for some polynomial f(x). Instead of working with vectors where the components are integers modulo some prime, we work with vectors whose components are elements of a ring.

2.3 About X-Wing

2.3.1 High-Level Description of X-Wing

X-Wing is a hybrid KEM based on X25519 and ML-KEM-768 designed by Manuel Barbosa et al. (Barbosa et al., 2024). X-Wing has improved efficiency compared to a generic KEM combiner. X-Wing is contained in a class of KEMs where a simple, more efficient combiner yields an IND-CCA secure hybrid KEM despite not using the KEM ciphertext during key derivation.

The paper strives to prove two main assumptions:

1. X-Wing is a classically IND-CCA secure KEM if the strong Diffie-Hellman assumption holds in the X25519 nominal group.
2. X-Wing is a post-quantum IND-CCA secure KEM if ML-KEM-768 is itself an IND-CCA secure KEM and SHA3-256 is secure when used as a pseudorandom function.

If the previous two points hold, X-Wing is secure if either X25519 or ML-KEM-768 is secure. However, it is important to note that the NIST standard ML-KEM has not been finalized yet, so X-Wing is similarly unfinalized (Barbosa et al., 2024).

2.3.2 Design of X-Wing

The goal of X-Wing is to be usable and the go-to solution in most applications (Barbosa et al., 2024). The two most important aspects to a usable hybrid KEM are (a) its security guarantees and performance, and (b) its implementation simplicity. Regarding security guarantees, X-Wing targets IND-CCA2 security at 128 bits with extra margin for ML-KEM. Through this, there is a security guarantee with comfortable margin, while retaining performance. Now, when it comes to a simple implementation, X-Wing is straightforward to implement with X25519 and ML-KEM-768 as black boxes. As the go-to standard traditional key agreement with excellent performance, X25519 provides strong classical security, while ML-KEM is currently NIST's only choice for a post-quantum KEM.

The final-key derivation of X-Wing includes an X25519 public key and a ciphertext. This ciphertext includes both the Diffie-Hellman long-term and ephemeral public keys. The X25519 public key is added as a measure of security against multi-target attacks. The X25519 ciphertext is added because X25519, seen as a KEM, is not ciphertext second preimage resistant. Removing either or both tokens would not change the performance by much. It is important to note that the ML-KEM-768 ciphertext is not included in the final-key derivation as it is in generic KEM combiners. The X-Wing KEM private key, public key, ciphertext, and shared key, which make up the X-Wing design summary, can be seen in Figure 2.1 (Barbosa et al., 2024).

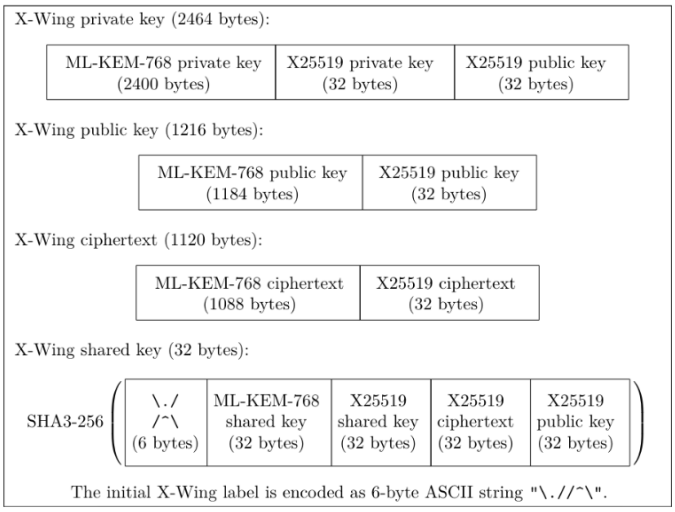


Figure 2.1: X-Wing Design Summary

2.3.3 Preliminaries of X-Wing

Many terms are defined and proven correct in order to prove the security and correctness in the implementation of X-Wing. We will briefly define these terms, while assuming the given proofs are successful.

1. A **Key Encapsulation Mechanism (KEM)** is a triple of algorithms KEM = {KeyGen, Enc, Dec} with a public keyspace PK, private keyspace SK, ciphertext space C, and shared keyspace K (Barbosa et al., 2024).

2. **IND-CCA advantage for KEMs**: The advantage of A in breaking the IND-CCA security of KEM is defined as

$$\mathrm{Adv}^{\mathrm{KEM}}_{\mathrm{IND\text{-}CCA,\ A}} = |\Pr[\mathrm{IND\text{-}CCA}^{0}_{\mathrm{KEM,\ A}} \Rightarrow 1] - \Pr[\mathrm{IND\text{-}CCA}^{1}_{\mathrm{KEM,\ A}} \Rightarrow 1]|.$$

3. **Nominal Group**: A nominal group $\mathrm{N} = (\mathrm{G}, \mathrm{g}, \mathrm{p}, \mathrm{e_h}, \mathrm{e_u}, \exp)$ consists of an efficiently recognizable finite set of elements G, a base element $\mathrm{g} \in \mathrm{G}$, a prime p, a finite set of honest exponents $\mathrm{e_h} \in \mathbb{Z}$, a finite set of exponents $\mathrm{e_u} \subset \mathbb{Z}/p\mathbb{Z}$, and an efficiently computable exponentiation function exp (Barbosa et al., 2024).

4. **Quantum Superiority Fighter (QSF)**: A hybrid KEM based on two building blocks, a nominal group and another KEM, that is secure even if one of the building blocks loses its security properties. X-Wing is a concrete instantiation of the QSF framework, using X25519, ML-KEM-768, and SHA3-256.

2.3.4 Definition of X-Wing

The X-Wing KEM is defined as a tuple of algorithms, {KeyGen, Enc, Dec}, created using ML-KEM-768, X25519 Diffie-Hellman key exchange, and the SHA3-256 hash function. The steps of the three algorithms are defined in Figure 2.2 below (Barbosa et al. (2024)).

Algorithm KeyGen()
$sk_1, pk_1 \leftarrow\$$ ML-KEM-768.KeyGen()
$sk_2 \leftarrow\$$ random(32)
$pk_2 \leftarrow$ X25519.DH$(sk_2, g_{\text{X25519}})$
$sk \leftarrow (sk_1, sk_2, pk_2)$
$pk \leftarrow (pk_1, pk_2)$
return (sk, pk)

Algorithm Enc(pk)
$(pk_1, pk_2) \leftarrow pk$
$sk_e \leftarrow\$$ random(32)
$c_2 \leftarrow$ X25519.DH$(sk_e, g_{\text{X25519}})$
$k_1, c_1 \leftarrow\$$ ML-KEM-768.Enc(pk)
$k_2 \leftarrow$ X25519.DH(sk_e, pk_2)
$s \leftarrow$ "\./⌃\"$\|k_1\|k_2\|c_2\|pk_2$
$k \leftarrow$ SHA3-256(s)
$c \leftarrow (c_1, c_2)$
return (k, c)

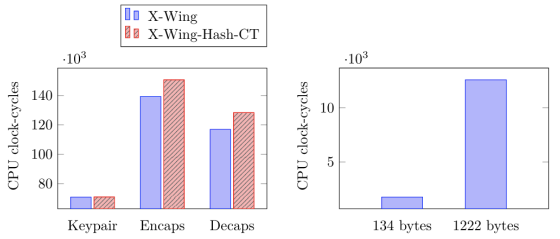
Algorithm Dec(c, sk)
$(sk_1, sk_2, pk_2) \leftarrow sk$
$(c_1, c_2) \leftarrow c$
$k_1 \leftarrow$ ML-KEM-768.Dec(c_1, sk_1)
$k_2 \leftarrow$ X25519.DH(sk_2, c_2)
$s \leftarrow$ "\./⌃\"$\|k_1\|k_2\|c_2\|pk_2$
$k \leftarrow$ SHA3-256(s)
return k

Figure 2.2: X-Wing Algorithms

2.3.5 Performance Results of X-Wing

Through the data analysis seen in (Oliveira et al., 2024), an 8% and 9% performance gain was recorded for encapsulation and decapsulation, respectively, by omitting the ML-KEM-768 ciphertext in the input to SHA3-256. This was recorded by counting the CPU clock-cycles used by the key generation, encapsulation, and decapsulation algorithms for X-Wing.

As seen in the following Figure 2.3, the clock cycles of X-Wing and X-Wing-Hash-CT are compared. X-Wing is the optimized implementation discussed in this paper. X-Wing-Hash-CT is the variant of X-Wing that includes the ML-KEM-768 ciphertext in the input to SHA3-256. Hence, X-Wing-Hash-CT has 1222 bytes for input while X-Wing only has 134 bytes for input. It takes more clock cycles for the X-Wing-Hash-CT to complete all operations of keypair generation, encapsulation, and decapsulation (Barbosa et al., 2024).



(a) Benchmarks for X-Wing and X-Wing-Hash-CT AVX2 implementations.

(b) Benchmarks for SHA3-256 implementation with different input sizes.

Figure 2.3: Benchmark Plots

2.4 Kyber768 vs. X-Wing

In order to decide how to design our post-quantum implementation of *age*, we ask whether the optimal solution involves (a) integrating Kyber768 into *age*'s existing architecture, or (b) revising and molding *age*'s architecture around X-Wing. While Kyber768 is a post-quantum KEM, X-Wing is a hybrid KEM of the classical X25519 key exchange and the post-quantum ML-KEM-768. If we choose to integrate Kyber768 into *age*, we will create our own hybrid

KEM using X25519 and Kyber768. Therefore, the only difference in our implementations will be in the usage of Kyber768 or of ML-KEM-768 as our post-quantum KEM.

The similarities and differences between ML-KEM-768 and Kyber768 are discussed in the FIPS 2023 Draft of ML-KEM (NIST, 2023). We will use this discussion to decide whether Kyber768 or X-Wing should be used to implement post-quantum security in *age*.

We further note that differing implementations of the tuple of KEM algorithms for ML-KEM (ML-KEM.KeyGen, ML-KEM.Encaps, and ML-KEM.Decaps) are permitted if these different procedures always produce the correct output for every input. Thus, we will only discuss the differences between Kyber768 as outlined in Bos et al. (2017) and ML-KEM where there are differences in input-output as well (NIST, 2023).

The ML-KEM scheme is derived from the round-three version of the CRYSTALS-Kyber KEM. The differences between CRYSTALS-Kyber and the ML-KEM scheme noted in NIST (2023) are as follows:

1. The shared secret key length in CRYSTALS-Kyber depends on how the key is used in the relevant application. In ML-KEM, the shared secret key length is fixed to 256 bits.

2. While CRYSTALS-Kyber and ML-KEM both use the Fujisaki-Okamoto transform, ML-KEM uses a different variant of the transform. Thus, ML-KEM.Encaps no longer includes a hash of the ciphertext in the derivation of the shared secret. ML-KEM.Decaps is adjusted to work with this change as well.

3. In CRYSTALS-Kyber, the initial randomness m in the encapsulation algorithm was hashed before being used. This step is unnecessary and not performed in ML-KEM.

4. ML-KEM has more input validation steps than CRYSTALS-Kyber.

Based on the comparison provided, it is clear that Kyber768 and ML-KEM-768 (used in X-Wing) have distinct differences in their implementations, particularly regarding the fixed key length, variant of the Fujisaki-Okamoto transform, and input validation steps. However, leveraging *age*'s existing code for X25519 provides classic cryptographic security and avoids the need to remove functional code from *age*. Additionally, by implementing a custom hybrid KEM, we have more control over the integration process and can tailor it to meet the specific requirements and constraints of the *age* system. Given these considerations, we decided to implement a hybrid KEM using Kyber768 and X25519 within *age*, rather than adopting the preexisting X-Wing implementation.

Chapter 3

Design

3.1 Introduction to Our Hybrid KEM

In our hybrid implementation of *age*, Kyber768+X25519, we propose a new encryption scheme which takes into account not only the native X25519 keypair but also a Kyber768 keypair, both of which will be incorporated into a hybrid shared key. This scheme will ensure that encrypted files are secure against both classical and post-quantum attacks. Specifically, we assert that our implementation meets NIST's category 1 requirements, which will be further discussed in Chapter 5: Analysis and Limitations.

3.2 Cryptographic Scheme

Our hybrid implementation requires two sets of public and secret keys for X25519 and Kyber768. While *age-keygen(1)* outputs the native X25519 keypair $\mathrm{I}^{\mathrm{X}}_{\mathrm{SK}}$ and $\mathrm{I}^{\mathrm{X}}_{\mathrm{PK}}$=$\mathrm{R}^{\mathrm{X}}_{\mathrm{PK}}$, our implementation additionally utilizes the Kyber768 keypair $\mathrm{I}^{\mathrm{K}}_{\mathrm{SK}}$ and $\mathrm{I}^{\mathrm{K}}_{\mathrm{PK}}$=$\mathrm{R}^{\mathrm{K}}_{\mathrm{PK}}$.

The hybrid cryptographic scheme between Alice and Bob is described as follows:

1. Alice does **encapsulation** using Bob's public keys $\mathrm{R}^{\mathrm{X}}_{\mathrm{PK}}$ and $\mathrm{R}^{\mathrm{K}}_{\mathrm{PK}}$:
   (a) Alice generates an ephemeral as a random scalar e and the corresponding ephemeral share, or X25519 peer share, as $c_1 = e * g$.
   (b) The X25519 shared secret s is computed.
   $$s = e * \mathrm{R}^{\mathrm{X}}_{\mathrm{PK}} \tag{3.1}$$
   (c) Alice generates the Kyber768 peer share c_2 and Kyber768 shared key k.
   $$(c_2, k) = \text{Kyber.Encapsulate}(\mathrm{R}^{\mathrm{K}}_{\mathrm{PK}}) \tag{3.2}$$
   (d) The hybrid shared wrapping key h is computed. Here, in addition to including the Kyber768 shared key k in the input key material, we move the X25519 peer share c_1 and Bob's public key $\mathrm{R}^{\mathrm{X}}_{\mathrm{PK}}$ from the salt to the input key material.
   $$\begin{aligned} h = \text{HKDF-SHA-256}(&\text{input key material} = s||c_1||\mathrm{R}^{\mathrm{X}}_{\mathrm{PK}}||k, \\ &\text{salt} = \text{“age-encryption.org/Kyber768+X25519”}) \end{aligned}$$

2. Alice encrypts the file M so that only Bob can decrypt it:
   (a) Alice generates a random one-time file key r.
   (b) The plaintext file M is encrypted with the file key r and a fixed nonce N to get ciphertext C.
   $$C = \text{ChaCha20Poly1305.Encrypt}(M, \text{key} = r + N) \tag{3.3}$$
   (c) The file key r is wrapped (i.e., encrypted) using the hybrid shared wrapping key h.
   $$w = \text{ChaCha20Poly1305.Encrypt}(r, \text{key} = h) \tag{3.4}$$
3. Alice constructs the output file, which consists of the ciphertext C and a header containing a MAC, the peer shares c_1 and c_2, and the wrapped file key w.
4. Bob does **decapsulation** using his own identity I and the data contained in the output file header:
   (a) The X25519 shared secret s is recovered.
   $$s = c_1 * \mathrm{I}^{\mathrm{X}}_{\mathrm{SK}} \tag{3.5}$$
   (b) The Kyber768 shared key k is recovered.
   $$k = \text{Kyber.Decapsulate}(\mathrm{I}^{\mathrm{K}}_{\mathrm{SK}}, c_2) \tag{3.6}$$
   (c) The hybrid shared wrapping key h is recovered.
   $$\begin{aligned} h = \text{HKDF-SHA-256}(&\text{input key material} = s||c_1||\mathrm{I}^{\mathrm{X}}_{\mathrm{PK}}||k, \\ &\text{salt} = \text{“age-encryption.org/Kyber768+X25519”}) \end{aligned}$$
5. Bob decrypts the ciphertext C to recover the original plaintext M:
   (a) The file key r is unwrapped (i.e., decrypted) using the hybrid shared wrapping key h.
   $$r = \text{ChaCha20Poly1305.Decrypt}(w, \text{key} = h) \tag{3.7}$$
   (b) The ciphertext C is decrypted to plaintext M using the file key r.
   $$M = \text{ChaCha20Poly1305.Decrypt}(C, \text{key} = r + N) \tag{3.8}$$

3.3 Changes Seen in Hybrid KEM Design

In our hybrid design, we pass the Kyber768 shared key k to HKDF-SHA-256 when generating the hybrid shared wrapping key h. Moreover, we make an improvement to the previous X25519 scheme, which passed the ephemeral share and recipient to the salt. As the salt should be “random or fixed for domain separation,” we instead move them to the input key material along with k and replace the salt with a fixed string (Valsorda, 2022b).

Variable	Size (bytes)
X25519 public key I^{X}_{PK}	32
X25519 secret key I^{X}_{SK}	32
X25519 ephemeral e	32
X25519 peer share c_1	32
Kyber768 public key I^{K}_{PK}	1184
Kyber768 secret key I^{K}_{SK}	1152
Kyber768 shared key k	32
Kyber768 peer share c_2	1088
Hybrid shared wrapping key h	32
File key r	16

Table 3.1: Sizes in bytes of variables in our hybrid design.

In order to handle the logistical challenges of combining two key pairs into one identity, we introduce a new prefix scheme to *age* and *age-keygen(1)* that indicates the presence of a hybrid identity. We change the existing public key prefix from `age1` to `agex1` for X25519 and `agek1` for Kyber768, and the existing private key prefix from `AGE-SECRET-KEY-1` to `AGE-X-SECRET-KEY-1` for X25519 and `AGE-K-SECRET-KEY-1` for Kyber768. On detection of any hybrid prefix, our implementation will also search for the closest matching X25519 or Kyber768 keypair in the file and hand off both pairs to the encapsulation and decapsulation functions.

We additionally address the fact that the size of a hybrid recipient increases significantly from 32 bytes to 1216 bytes (or 1960 characters after bech32-encoding), making it difficult for users to easily copy and paste their hybrid keys into the command line (Valsorda, 2022b). We assert the use of an alternative option already offered by *age*, that is, a user might output the generated identity to a file. Then, by using the `--recipients-file` or `--identity` arguments, users can easily encrypt their files by referencing an existing file that contains the desired keys. In Table 3.1, we specify the sizes of variables in our hybrid cryptographic scheme, including the Kyber768 key sizes.

3.4 Result from Hybrid KEM

By implementing a secure hybrid file key and shared key, a hybrid KEM, and new prefixes for reading and writing hybrid recipients and identities, our proposal for a post-quantum version of *age* maintains the core pillars of tamper resistance and authentication, assuming recipient secrecy. We sacrifice the copy-pastability of small keys for security against both classical and post-quantum attacks. Our X25519+Kyber768 implementation marks an important advancement in the post-quantum direction for secure, UNIX-style file encryption.

Chapter 4

Flow

In the following section, we demonstrate the flow and utility of our scheme through various use cases, along with the corresponding commands and example file contents. As specified in Chapter 3: Design, we recommend using files instead of directly pasting keys into the terminal, due to the larger size of the Kyber768 keypair. We denote truncation of a file's contents as `[...]` for space and clarity.

4.1 Encrypting to a Single Recipient

For this use case, say Alice wants to encrypt her file `employee_info.txt` with sensitive information to Bob.

1. Generate Bob's identity:

   ```
   age-keygen -o identity_bob.txt
   ```

   The generated file `identity_bob.txt` will contain Bob's X25519 and Kyber768 secrets. Note that when the identity file is parsed, only the secrets are read. In order to stay true to *age*'s native design, instead of storing the raw Kyber768 secret key, we store the seed used to deterministically generate the Kyber768 keypair, so that we can recover both the public and secret keys from an identity file. We expand more on this decision in Chapter 5: Analysis and Limitations.

   ```
   # created: 2024-05-04T12:42:35-04:00
   # X25519 public key: agex165k8pnp7g8hlumr2mf5hply2s2arkg7zkddyxejss[...]
   AGE-X-SECRET-KEY-1AGX9PTE3TZD960QF9XA0Y509Z5DNC9EW00CERW45PRL8HJZ6S[...]
   # Kyber768 public key: agek1dz9e3chzenr029znyy9t9u2cug5xmgw25huun8k[...]
   AGE-K-SECRET-KEY-182WYW9E3PJGUKTWH4L7ETJSE82MDFKE4G974Z6EJ4R2R4H7GX[...]
   ```

2. Convert Bob's identity file to a recipient file:

   ```
   age-keygen -y -o recipient_bob.txt < identity_bob.txt
   ```

The generated file `recipient_bob.txt` will contain Bob's X25519 and Kyber768 public keys, which are accessible to Alice.

```
agex165k8pnp7g8hlumr2mf5hply2s2arkg7zkddyxejssph549lg93sskttkps
agek1dz9e3chzenr029znyy9t9u2cug5xmgw25huun8ktp28e2kwguvzgrsykk4pfz7[...]
```

3. Encrypt the file `employee_info.txt` to the recipient Bob and write the ciphertext to `employee_info.txt.age`:

```
age -R recipient_bob.txt employee_info.txt > employee_info.txt.age
```

4. Decrypt the file `employee_info.txt.age` with Bob's identity and write the recovered plaintext to `employee_info_decrypted.txt`:

```
age -d -o employee_info_decrypted.txt
    -i identity_bob.txt employee_info.txt.age
```

4.2 Encrypting to Multiple Recipients

Say Alice wants to encrypt her file `employee_info.txt` so that both Bob and Charlie can decrypt it. In order to do so, only one extra step is required to build a recipients file containing Bob and Charlie's keys, described as follows.

1. Generate Bob and Charlie's identities as usual:

```
age-keygen -o identity_bob.txt

age-keygen -o identity_charlie.txt
```

2. Convert Bob and Charlie's identity files to recipient files as usual:

```
age-keygen -y -o recipient_bob.txt < identity_bob.txt

age-keygen -y -o recipient_charlie.txt < identity_charlie.txt
```

3. Paste Bob and Charlie's recipient information into one recipient file `recipients.txt`:

```
# Bob's public keys
agex165k8pnp7g8hlumr2mf5hply2s2arkg7zkddyxejssph549lg93sskttkps
agek1dz9e3chzenr029znyy9t9u2cug5xmgw25huun8ktp28e2kwguvzgrsykk4pfz7[...]

# Charlie's public keys
agex1rsujzdh2mfxe5v3zmgfdwvg54f4rkn33rasu8ujc83a82r3ns3ysjltxaf
agek1fdzzzf2c0nmsge07mwljdkt59dczv3rnfm3tpj9nzathnyw8d84qv6xgrflzz6[...]
```

Note that the X25519 and Kyber768 keypairs corresponding to a single recipient must be adjacent to each other to be parsed correctly.

4. Encrypt the file `employee_info.txt` to the recipients in the file `recipients.txt` (Bob and Charlie) and write the ciphertext to `employee_info.txt.age`:

```
age -R recipients.txt employee_info.txt > employee_info.txt.age
```

5. Bob and Charlie can now separately decrypt the file `employee_info.txt.age` using their own identities:

```
age -d -o employee_info_decrypted_bob.txt
    -i identity_bob.txt employee_info.txt.age

age -d -o employee_info_decrypted_charlie.txt
    -i identity_charlie.txt employee_info.txt.age
```

Chapter 5

Analysis and Limitations

5.1 Security Analysis

With our hybrid implementation of *age*, Kyber768+X25519, we believe we have met NIST's category 1 requirements, i.e., any attack that breaks our scheme must "require computational resources comparable to or greater than those required for key search on a block cipher with a 128-bit key (e.g. AES128)" (NIST, 2017). Because the native *age* implementation utilizes a 128-bit symmetric file key, we ask whether the existing 128-bit key space is secure enough to be paired with Kyber768.

NIST's security evaluation criteria measure a scheme's security via the number of logical gates required to break AES, both for classical computers in terms of classical gates and for quantum computers with respect to `MAXDEPTH`, or the circuit depth to which a quantum algorithm is restricted (NIST, 2017). Possible `MAXDEPTH` values may range from 2^{40} to 2^{96} logical gates. NIST estimates that "the classical and quantum gate counts for the optimal key recovery" for AES128 are $2^{170}/\texttt{MAXDEPTH}$ quantum gates or 2^{143} classical gates, respectively (NIST, 2017); assuming a worst-case scenario of $\texttt{MAXDEPTH} = 2^{96}$, any number of quantum gates matching or exceeding 2^{74} meet the criteria.

While AES128 can be broken by Grover's algorithm with a quadratic quantum speedup, realistically implementing such long-running serial computations on quantum computers is challenging (NIST, 2017), and running it on a 128-bit key space would require 2^{106} logical gates (Valsorda, 2022b). We therefore conclude that *age*'s 128-bit file key size is sufficient to meet NIST's category 1 requirements.

5.2 Design Analysis

Our implementation is a hybrid KEM, meaning we are using both X25519 and Kyber768 protocols. The use of Kyber768 provides us with security against post-quantum algorithms, as previously discussed. X25519 itself is not post-quantum secure due to its use of elliptic-curve Diffie-Hellman (ECDH) whose security is based on the hardness of the discrete logarithm problem. While no post-quantum adversary exists today, security research suggests that quantum computers would be able to solve the discrete logarithm problem, rendering ECDH and similar protocols insecure. Shor's algorithm is one existing quantum algorithm that could potentially solve the discrete logarithm problem. However, there is currently no quantum

computer that contains enough qbits (basic unit of quantum information) to carry this out. We decided to use both the X25519 and Hybrid protocols to secure age against both classical and post-quantum attacks and ensure that an attacker would have to break both schemes to decrypt messages (more layers of security).

One important design decision we made for our hybrid KEM was to store the seed value used to deterministically generate our Kyber768 public and secret keys in the identity files, instead of the secret key itself. *age*'s native design, which stores only the X25519 secret key and derives its corresponding public key during parsing, requires that we similarly store only the Kyber768 secret. However, the Kyber768 Go library we used did not provide a method to derive the public key from the secret one (Go, 2024). Instead, we were able to use the NewKeyFromSeed(seed) function to deterministically recover a keypair from a given seed. Another Kyber768 Go library we found provided a method to derive the public key, but we were unable to proceed due to "Use of internal package not allowed" errors. For these reasons, we decided to store the random seed alongside the X25519 secret in each identity file.

5.3 Challenges

5.3.1 Choosing Between Kyber768 vs. X-Wing

Our two primary challenges when deciding whether to implement a hybrid KEM with Kyber768 or the existing X-Wing hybrid KEM included issues regarding:

1. **Performance Runtimes**: Ensuring comparable performance runtimes between Kyber768 and ML-KEM-768 in X-Wing posed a challenge due to differences in their cryptographic algorithms and inputs. However, this challenge was effectively addressed through thorough benchmarking. Leveraging existing benchmarking data from X-Wing and considering that Kyber768 has one less ciphertext in the SHA3-256 input, we believe that a Hybrid KEM with Kyber768 with have comparable or even more efficient runtime compared to X-Wing.

2. **Implementation Complexity**: Because X-Wing was already a completed Hybrid KEM, it was a challenge to decide whether to incorporate this functioning KEM into *age* or create our own Hybrid KEM using the X25519 classically secure code currently existing in *age* implementations with the existing, separate implementations of Kyber768. Our runtime analysis as well as caution to remove working code led us to the decision to create our own Hybrid KEM with Kyber768+X25519, leveraging the advantages of both cryptographic primitives while maintaining compatibility with existing codebases.

5.3.2 Building Our Implementation

Another challenge we encountered while building and testing our implementation was managing packages through `go.mod` and other tools. We were unable to get the `go build` command to correctly build our updated implementation, presumably due to an unidentified issue with package imports or definitions. However, we were able to successfully use

`go run . [ARGUMENTS]` to run and test our implementation, with the caveat that the command must be run from the correct directory (that is, the `practical-crypto-project` directory in order to run the `age` package and the `practical-crypto-project/cmd/age` directory in order to run the `age-keygen` package).

For testing purposes, below we provide some example successful commands that may be run from the `practical-crypto-project` directory.

- Instead of:

  ```
  age -R recipient_bob.txt employee_info.txt > employee_info.txt.age
  ```

  Use:

  ```
  go run . -R recipient_bob.txt employee_info.txt > employee_info.txt.age
  ```

- Instead of:

  ```
  age-keygen -o identity_bob.txt
  ```

  Use:

  ```
  cd cmd/age

  go run . -o identity_bob.txt
  ```

Chapter 6

Future Research

6.1 Future Implementations of Hybrid KEM

6.1.1 X-Wing

Our X25519+Kyber768 implementation is only one possibility of what a post-quantum *age* could look like. Future work might attempt to implement the X-Wing scheme described in Section 2.3 and further discussed in comparison to Kyber768 in Section 2.4. While our scheme displays a lot of similarities to the X-Wing scheme, we note that the X-Wing private key consists of not only the X25519 and ML-KEM-768 private keys, but also the X25519 public key, and such an implementation within *age* would need to make the necessary adjustments.

As previously discussed, benchmarking procedures have shown promising results for the X-Wing scheme. Counting CPU clock-cycles used by key generation, encapsulation, and decapsulation algorithms for X-Wing and X-Wing-Hash-CT revealed significant performance gains compared to other cryptographic schemes. Notably, by omitting the ML-KEM-768 ciphertext from input to SHA3-256, there is an 8% and 9% performance gain for encapsulation and decapsulation, respectively. These benchmarking results underscore the potential for improved efficiency and performance in post-quantum cryptographic schemes like the X-Wing scheme, further highlighting the importance of exploring alternative implementations within the *age* ecosystem.

6.1.2 Support for Both X25519 and X25519+Kyber768

Future endeavors may also focus on allowing *age* to support both our hybrid KEM and the native X25519 KEM. However, careful measures must be taken to ensure that hybrid and native recipients are never mixed, as doing so would negate the post-quantum security measures originally placed on the file (Valsorda, 2022b).

6.2 "Quantum Algorithms for Lattice Problems"

We further acknowledge a recent e-print by Yilei Chen, "Quantum Algorithms for Lattice Problems," which claimed to have devised an algorithm that efficiently solves the **shortest independent vector problem (SIVP)**. If correct, Chen's algorithm could threaten the "hardness" and, consequently, the cryptographic security of many if not all lattice-based

problems (Green, 2024). As our post-quantum algorithm of choice, Kyber768, is based off the M-LWE lattice-based problem, an efficient algorithm that can break the SIVP problem could negate the post-quantum security aspect of our hybrid cryptographic scheme.

Fortunately for lattice-based cryptography, a bug identified within Chen's algorithm represents a critical factor in evaluating the validity of the algorithm's security implications as well as the algorithm itself (Green, 2024). Moreover, the specific parameters targeted by the algorithm do not immediately apply to Kyber768 (Green, 2024). We must also consider the possibility that the algorithm could pose difficult or costly to run in practice, even after quantum computers powerful enough to run it become available. Taking into account the uncertainty and potential lack of feasibility surrounding Chen's proposed algorithm, we believe Kyber768 is still sufficient to ensure post-quantum security.

Future work could provide a comprehensive overview and analysis of the potential impact of Chen's algorithm on lattice-based cryptography, as well as its relevance to our hybrid KEM, X25519+Kyber768.

Chapter 7

Conclusion

7.1 Summary

Our research focused on the UNIX-style file encryption tool *age* and how it could be improved. Currently, *age*'s implementation uses the X25519 KEM, which only provides security against classical computers and adversaries. With research suggesting that quantum computers may be able to solve the math problems providing security for such protocols, looking into post-quantum secure KEMs is important. Kyber768 and X-Wing are two KEM protocols in development that may provide post-quantum security due to their reliance on the hardness of lattice problems. X-Wing is a hybrid KEM that involves X25519 and ML-KEM-768, while Kyber768 is a KEM whose design enables it to be implemented into existing encryption tools like *age*. Our improvement upon *age* allows users to encrypt files using the hybrid KEM X25519+Kyber768 that ensures both classical and post-quantum security.

Future research can be done not only to improve our hybrid KEM to work for many tools similar to *age*, but also can be done to improve its performance so that it is fast or faster than X-Wing. Beyond this, future research should be done to make sure that lattice-based cryptography is secure, or can be modified to stay secure.

7.2 Project Deliverables

After researching *age*, the Kyber-768 post-quantum cryptographic scheme, and the X-Wing hybrid KEM, we were able to create a hybrid KEM with X25519 traditional security and Kyber-768 post-quantum security. With this hybrid KEM, *age* is secure traditionally and post-quantum, and even if one security fails, the other will keep *age* safe. The KEM is available on GitHub at https://github.com/srest2021/practical-crypto-project and can be used via the command line as described in Chapter 4: Flow and Section 5.3.2: Building Our Implementation.

This paper is our other project deliverable. Before taking Practical Cryptographic Systems this Spring, our team members had very little background in cryptography. Specifically, this project allowed us to explore what "post-quantum" cryptography is, specific advancements in the field, the different components of cryptographic systems, and how to implement one. We felt that a paper would allow us to demonstrate our understanding of all the new topics we've learned.

References

Barbosa, M. et al. (2024). X-wing: The hybrid kem you've been looking for. Accessed April 14, 2024.

Bos, J. et al. (2017). Crystals – kyber: a cca-secure module-lattice-based kem. Accessed May 9, 2024.

Go (2024). kyber768. Accessed May 9, 2024.

Green, M. (2024). A quick post on chen's algorithm. Accessed May 1, 2024.

Hoang, V. T. et al. (2015). Online authenticated-encryption and its nonce-reuse misuse-resistance. Accessed March 22, 2024.

NIST (2017). Post-quantum cryptography. Accessed April 30, 2024.

NIST (2023). Module-lattice-based key encapsulation mechanism standard. fips 203 (initial public draft). Accessed April 22, 2024.

Oliveira, T. et al. (2024). X-wing benchmarks. Accessed April 14, 2024.

Peikert, C. (2016). A decade of lattice cryptography. Accessed May 1, 2024.

Schwabe, P. and Mann, J. (2020). Crystals. Accessed March 23, 2024.

Valsorda, F. (2022a). age and authenticated encryption. Accessed March 22, 2024.

Valsorda, F. (2022b). Kems and post-quantum age. Accessed March 22, 2024.

Valsorda, F. (2023). age(1). Accessed March 22, 2024.

List of Abbreviations

AEAD	Authenticated Encryption with Associated Data
AES	Advanced Encryption Standard
CCR	Ciphertext Collision Resistent
DEM	Data Encapsulation Mechanism
ECDH	Elliptic-Curve Diffie-Hellman
IND-CCA	Indistinguishability Under Chosen Ciphertext Attack
IND-CCA2	Indistinguishability Under Adaptive Chosen Ciphertext Attack
FIPS	Federal Information Processing Standards Publication
HKDF	HMAC-based Key Derivation Function
HMAC	Hashed Message Authentication Code
KEM	Key Encapsulation Mechanism
LWE	Learning With Errors
MAC	Message Authentication Code
ML-KEM	Module Lattice-based Key Encapsulation Mechanism
M-LWE	Module Learning With Errors
NIST	National Institute of Standards and Technology
QSF	Quantum Superiority Fighter
SIVP	Shortest Independent Vector Problem
SVP	Shortest Vector Problem