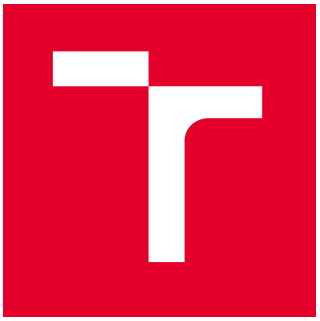

BRNO UNIVERSITY OF TECHNOLOGY

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

FAKULTA ELEKTROTECHNIKY
A KOMUNIKAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF TELECOMMUNICATIONS

ÚSTAV TELEKOMUNIKACÍ

LONG-TERM CLOUD STORAGE SECURITY

DLOUHODOBÁ BEZPEČNOST CLOUDOVÉHO ULOŽIŠTĚ

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR — **Bc. Filip Wojnar**
AUTOR PRÁCE

SUPERVISOR — **doc. Ing. Lukáš Malina, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2024

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

Master's Thesis

Master's study program **Information Security**

Department of Telecommunications

Student: Bc. Filip Wojnar ***ID:*** 203720

Year of study: 2 ***Academic year:*** 2023/24

TITLE OF THESIS:

Long-term Cloud Storage Security

INSTRUCTION:

The work is focused on the design and implementation of secure key management and the selection and implementation of a set of quantum-resistant cryptographic protections to ensure long-term security in cloud storage. The student can use existing results, i.e. libraries and source codes from previous works. The output of the thesis is a functional implementation of key management and a set of post-quantum cryptographic methods for long-term data security for a selected instance of Nextcloud.

RECOMMENDED LITERATURE:

[1] Menezes, Alfred, Van Oorschot, Paul C. a VANSTONE, Scott A.. Handbook of applied cryptography. Boca Raton: CRC Press, c1997. Discrete mathematics and its applications. ISBN 0-8493-8523-7.

[2] Web eID: electronic ID smart cards on the Web [online]. [cit. 2022-09-06]. Dostupné z: https://web-eid.eu.

Date of project specification: 5.2.2024 ***Deadline for submission:*** 21.5.2024

Supervisor: doc. Ing. Lukáš Malina, Ph.D.

doc. Ing. Jan Hajný, Ph.D.
Chair of study program board

ABSTRACT

The primary focus of the thesis is the topic of the long-term security of cloud-based archive storage. It delves into the key management lifecycle terms and principles outlined by NIST to establish a robust key management system. The concluding section of the initial chapter examines state of art options for Key Management Systems, covering industry giant like Amazon Web Services Key Management Services to open-source alternative called Hashicorp Vault.

The second chapter of the thesis is dedicated to Post-Quantum Safe Cryptography, exploring the NIST standardization process for Post-Quantum Cryptography (PQC). Within this realm, the thesis compares seven algorithms that successfully reached the final round of consideration, ultimately selecting Crystals-Dilithium for Digital Signature Algorithm (DSA) and Crystals-Kyber for Key Encapsulation Mechanism (KEM) as the most effective performers based on CPU time they require for their cryptographic operations.

The third chapter outlines the existing cloud-based archive system at BUT (Brno University of Technology) and proposes an initial cryptographic scheme for key management within this virtualized environment. This includes master key pair generation, data key pair generation, data at rest encryption, data at rest decryption, resource sharing, key rotation, key recovery and key revocation.

Finally the fourth chapter gets into nuances of implementation realized in this thesis and reviews the effectiveness of the solution with benchmarks focused on newly added cryptographic functionality of realized post-quantum safe key management.

KEYWORDS

Cloud storage, Key Management Lifecycle, Key Management System, Master key, Post-quantum safe cryptography, Crystals-Kyber, Crystals-Dilithium, Advanced Encryption Standard, liboqs

WOJNAR, Filip. *Long-term cloud storage safety*. Semestral Project. Brno: Brno University of Technology, Faculty of Electrical Engineering and Communication, Department of Telecommunications, 2024. Advised by doc. Ing. Lukáš Malina, Ph.D.

Author's Declaration

Author:	Bc. Filip Wojnar
Author's ID:	203720
Paper type:	Semestral Project
Academic year:	2023/24
Topic:	Long-term cloud storage safety

I declare that I have written this paper independently, under the guidance of the advisor and using exclusively the technical references and other sources of information cited in the paper and listed in the comprehensive bibliography at the end of the paper.

As the author, I furthermore declare that, with respect to the creation of this paper, I have not infringed any copyright or violated anyone's personal and/or ownership rights. In this context, I am fully aware of the consequences of breaking Regulation § 11 of the Copyright Act No. 121/2000 Coll. of the Czech Republic, as amended, and of any breach of rights related to intellectual property or introduced within amendments to relevant Acts such as the Intellectual Property Act or the Criminal Code, Act No. 40/2009 Coll. of the Czech Republic, Section 2, Head VI, Part 4.

Brno ....

author's signature*

*The author signs only in the printed version.

ACKNOWLEDGEMENT

I would like to thank the advisor of my thesis, doc. Ing. Lukáš Malina, Ph.D. for his valuable comments and suggestions while I was working on this thesis, providing me with guidance and mentoring on standards of master thesis and helping me navigate the world of post-quantum cryptography. I am also very grateful to Ing. Petr Muzikant who was helped me understand current implementation of the project, found time to have many deep consultations with me throughout the year and helped me with the debugging and integration of my solution into the project. Lastly on a personal note I would like to thank my beloved girlfriend Antonie for her mental support and encouragement over the last few years in the pursuit of my academic goals.

Contents

Introduction **19**

1 Key management and Cloud solutions **21**
1.1 Basic concepts . . . 21
1.2 Key Management Lifecycle . . . 23
1.3 Key Management Systems in Cloud solutions . . . 27
1.3.1 Key Management Methods in Cloud storages . . . 27
1.3.2 State of art Key Management Systems in Cloud storages . . . 28
1.3.3 Hashicorp Vault as Opensource Key Management System . . . 29
1.4 Master Key and It's Security . . . 30
1.5 Hardware Secure Modules (HSM) . . . 31
1.5.1 Hardware Security Module principles . . . 32
1.5.2 HSM and Master Key Security . . . 32

2 Post–Quantum Safe Cryptography **35**
2.1 NIST PQC Standardization Process . . . 35
2.1.1 NIST PQC Candidates . . . 35
2.1.2 NIST PQC Candidates Comparison . . . 36
2.2 Crystals-Kyber . . . 37
2.2.1 Parameter sets . . . 37
2.3 Crystals-Dilithium . . . 38
2.3.1 Parameter Sets . . . 38
2.4 AES-256-GCM . . . 39
2.5 Kyber1024 + AES256-GCM Use Case . . . 40

3 Proposed Solution for BUT Cloud Based Archive System **43**
3.1 Current Architecture of BUT Archive System . . . 43
3.1.1 Deploying and Running The System . . . 45
3.2 Chosen Libraries . . . 46
3.2.1 LibOQS . . . 46
3.2.2 Python Cryptography Module . . . 47
3.3 Proposed Cryptographic Scheme . . . 48
3.3.1 Encryption in Transit from NextCloud to Archive System . . . 48
3.3.2 Master Key Pair . . . 48
3.3.3 User Key Pairs . . . 50
3.3.4 Key Management Hierarchy . . . 50
3.3.5 Data at Rest Encryption . . . 51

3.3.6 Data at Rest Decryption 52
3.3.7 Key Recovery 53
3.3.8 Resource Sharing 53
3.3.9 Key Rotation & Re-Encrypting Mechanism 54

4 Solution Implementation & Performance Benchmarking 55
4.1 Implementation Details & Block Schemes 55
4.1.1 Developed Cryptography Module 55
4.1.2 Block Schemes of Processes 63
4.2 Performance Benchmarking 70
4.3 Future work 73

Conclusion 75

Bibliography 77

5 Content of the electronic attachment 81

List of Figures

1.1 Scheme of HSM API usage extracted from [24] 33
2.1 Digital signature algorithms comparison extracted from [7] 36
2.2 KEM algorithm comparison extracted from [7] 37
2.3 Illustration of KEM process to establish AES key extracted from [31]. 41
3.1 Current architecture of archiving system extracted from Martin Nohava's thesis [23] 44
3.2 Overview of structure of liboqs library extracted from [27] 46
3.3 Modified Python code for decapsulating secret, orifinal code can be found at `https://github.com/open-quantum-safe` 47
3.4 Diagram of master key generation 49
3.5 Hierarchy of Master and Data key pairs 51
3.6 Process of encrypting data at rest 52
3.7 Process of data at rest decryption 53
4.1 Structure of Cryptography Module Extending Archiving System . . . 56
4.2 PBKDF2Utils function for generation of the kdf secret 57
4.3 PBKDF2 encryption function used to encrypt Kyber Master Key . . 58
4.4 PBKDF2 decryption function used to decrypt Kyber Master Key . . 58
4.5 KyberUserUtils key pair generation 59
4.6 KyberUserUtils encryption function 60
4.7 KyberUserUtils decryption 61
4.8 Block scheme of file encryption process 64
4.9 Block scheme of file decryption process 64
4.10 Block scheme of entire encryption process in the system 66
4.11 Block scheme of entire decryption process in the system 67
4.12 Extended version of archiving system architecture previously mentioned in chapter 3.1 from [23] 69
4.13 Bash script used to benchmark cryptographic operations 71
4.14 Graph of encryption and decryption times for files of various sized . . 72

List of Tables

2.1 Parameter sets for Kyber. Table extracted from [8] 38
2.2 Parameter sets for Dilithium. Table extracted from [10] 39
4.1 Lenovo ThinkPad X1 Carbon Gen11th Specifications 70
4.2 Encryption and Decryption times for different file sizes 72

Introduction

We live in a world where the storing of digital data is becoming increasingly more popular and often necessary by governments, international and global corporations as well as individuals. There is a wide plethora of ways to store digital data, but especially Cloud storage is becoming exponentially more popular due to the multiple benefits it offers, such as accessibility, flexibility, and scalability.

It is not only important to ensure the safety of cloud storage right now, but also in the future. There is potentially the creeping threat of new technology of quantum computers and the algorithms, like Grover's and Shor's, they are capable of executing. These algorithms could break the encryption of data we currently consider unbreakable and expose, for example, government secrets. This is specifically problematic for cloud storage due to the nature of shared resources and remote accessibility.

The focus of this thesis is to explore the current state of Cloud storage safety, including key management in cloud solutions, and explore the emerging threats of quantum computers on current state of art cryptography. The next phase will be dedicated to the exploration and comparison of developing quantum-safe cryptography protocols that are expected to withstand the threats mentioned above. Lastly, there will be introduced a cryptographic protocol proposal and its implementation on the NextCloud instance.

By addressing these potentially devastating issues, this thesis hopes to contribute to the developing field of cloud storage, post-quantum cryptography and quantum computing threats. Ultimately, this thesis is supposed to help the reader understand current problems of long-term cloud storage safety and offer a comprehensive overview of already available and developing remedies to avoid possible pitfalls in the future.

The last chapters of the thesis then use these concepts to architect and implement solution for key management and data confidentiality solution for archiving system developed at BUT (Brno University of Technology). They also describe selected parts of code and provide benchmarking data on how effective the solution is.

1 Key management and Cloud solutions

1.1 Basic concepts

This chapter will be focused on explaining the terms necessary to understand key management and cryptographic algorithms. It will also explain the basic aspects and operations of key management lifecycle. And lastly it will introduce Key management systems used in cloud based solutions.

Successful key management is the key ingredient to the security of a cryptosystem. Key management is focused on manipulation with public and private cryptographic keys in the system at the user level. One of the most important requirements for well designed key management is to ensure the following security services [3]:

1. **Confidentiality** - service that prevents the disclosure of information to unauthorized entities. The information is encrypted and can only become intelligible again by using decryption which only the authorized entity is able to perform.
2. **Data integrity** - prevents unauthorized alteration of data. This includes operations like inserting additional data, modifying existing data or deleting certain parts of it. Instead of ensuring the data hasn't been altered at all, it is better to come up with a mechanism that can verify with certainty if data has been altered or not.
3. **Authentication** - this service is focused on establishing the identity of entity that has created the data to verify it's origin. Most commonly, authentication is provided by digital signatures or message authentication codes.
4. **Non-repudiation** - this is combination of data integrity and authentication. Meaning this service is used to provide the proof of integrity and origin of the data. Additionally, there is a requirement that both integrity and origin can be verified by a third party. This service prevents entities to deny involvement in actions they have previously done.

In order to satisfy these requirements there is wide range of decisions that has to be made. One of the significant decisions is to pick the right cryptographic algorithms based on their cryptographic functionality depending on the situation and requirements of the application for which the key management system is being designed for. In case of this thesis these algorithms additionally have to be post quantum safe or in other words be unsolvable by Shor's or Grover's algorithm in polynomial time. Cryptographic algorithm functionality can be divided into 6 groups [2]:

1. **Hash Function** - it plays a role in verifying data integrity. When the hash value (digest) is calculated from the data it generates smaller fixed size value. Any changes to the data will lead to drastic changes in the hash value. This allows users to detect if the data has been tampered with. Hash functions also play a huge role in digital signatures. A widely used hash function is for example SHA-512.
2. **Encryption/Decryption Algorithms** - encryption is used to provide confidentiality for data. When data are encrypted they transform from plaintext into ciphertext. Ciphertext can be later decrypted back into plaintext. This is usually achieved with asymmetric cryptography. Public key is used for encryption and private key is used for decryption. Currently one of the most popular encryption algorithms is the symmetric AES. For PQC (Post quantum safe cryptography) for example the asymmetric lattice-based Kyber can be used.
3. **Message Authentication Codes** - used to confirm data authentication and integrity. It requires secret key known only to the third-party entity. Utilizing this key, it is possible to calculate cryptopgrahic checksum which can be used as a check to ensure data was not changed and the checksum was calculated by expected user or system.
4. **Digital Signature Algorithms** - can provide authentication, integrity and non-repudiation at the same time. Digital signatures are used in combination with hash algorithms and are not limited by the length of the data. Digital signatures play a huge role in creation of X509 certificates, which are the foundational stone for trust in the establishment of HTTP connections over TLS. Currently, the most popular algorithms are RSA and ECDSA. For PQC algorithms, there is for example the option to use lattice-based Dilithium.
5. **Key Establishment Algorithms** - serve as a tool to safely set up keys between communicating entities. There are 2 established methods to do so. The first is called key transportation and the second one is key agreement. Key transportation distributes an already generated key in encrypted form from one entity to another, which then decrypts the key after it is received. Whereas key agreement work on the principle where both entities are actively participating in calculation of agreed key. This is accomplished with public keys. Each entity either generates their own key pair or both use shared ephemeral key. Example of such algorithm is ECDH.
6. **Random Number Generation** - PRNG (Pseudo random number generators), or more specifically the subset of those called CSPRNG (cryptographically secure pseudorandom number generator) are a requirement for generation of key pairs. Only CSPRNG should be used for cryptography. CSPRNG algorithms often use either hash functions or AES to generate random numbers

suitable for cryptographic applications.

1.2 Key Management Lifecycle

First of all lets briefly focus on Key Management Lifecycle guidelines introduced by NIST(National Institute of Standards and Technology). It is considered to be the basic guideline on how the cryptographic keys should be handled from their creation to their end of life. NIST divides Key Management Lifecycle into following 11 categories [1]:

1. User Registration
2. System and User Initialization
3. Keying Material Installation
4. Key Establishment
5. Key Registration
6. Operational Use
7. Storage of Keying Material
8. Key Update
9. Key Recovery
10. Key De-registration and Destruction
11. Key Revocation

1. User Registration

During registration, an entity becomes an authorized member of a security domain. This includes the acquisition or creation and exchange of the initial keying material [1].

2. System and User Initialization

System initialization involves the set up or configuration of the system for secure operation. User initialization includes the installation or use of key material user obtained during user registration step. It is this step where IDs, passwords and user validation occurs. This step usually also includes the negotiation for algorithm preferences and installation of key at certificate authority.

3. Keying Material Installation

The security of the whole system is highly dependent on the security of keying material installation. During this stage, keying material is installed for operational

use within an entity's software, hardware, system, application, crypto module, or device using a variety of techniques [3]. Keying material installation takes place in the initial set-up phase, when new keying material is installed and when existing keying material is being replaced.

4. Key Establishment

As was already mentioned in Key Management section, this step focuses on the sharing of cryptographic keys or key material between involved entities. This is probably the most customizable step in the key management lifecycle and according to NIST can be further divided into following groups [2].

1. **Generation and Distribution of Public/Private Key Pairs**
2. **Generation and Distribution of Symmetric Keys**
3. **Generation and Distribution of Other Keying Material**

4.1 Generation and Distribution of Public/Private Key Pairs

When digital signatures or key establishment algorithms are used, there are public and private keys involved. These public and privates keys have to be generated in accordance with the mathematical specification of an approved standard. One such standard, that is generally recommended, is FIPS 140-2. Private keys should never be distributed to other entities. In regards to distribution of public keys, there is a distinction depending whether the public key is of static type, or ephemeral type.

4.1.1 Distribution of Static Public Keys

Static public keys are typically relatively long-lived, as their intention is to be used for multiple executions of the encryption algorithm over a longer period of time. The receiver of the public key should always have assurance of the following information:

1. The owner of the key pair (received public key and unknown private key) is known.
2. The receiver knows the purpose of the public key (for example signature, key agreement).
3. Parameters associated with the public key are known.
4. Proof that the public key has been properly generated and the owner of the public key has the associated private key.

A good and relevant example of this is the public key signed by trusted certificate authority. It is also worth mentioning that the confirmation of information

mentioned in the list above should be performed by the receiver before it is used, not by the certificate authority.

The keys falling into this category are following [3]:

- The signature verification key
- The public authentication key
- The key transport public key
- The static key agreement public key
- The public authorization key

4.1.2 Distribution of Ephemeral Public Keys

As naming suggests, ephemeral public keys have short lifetime and therefore they are unique to each key execution of key establishment process. As is common in the industry, it also in this case that the ephemeral public key and associated private key, should be generated in accordance with the approved standard, for example the FIPSXXX standard. The receiver of the ephemeral key should have the assurance of validity of identical information, as was already mentioned in the list from section 4.1.1.

5. Key Registration

During key registration, keying material is bound to information or attributes associated with a particular entity. This information typically includes the identity of the entity associated with the key material, but may also include authorization information or specify the level of trust. This step is typically performed when the entity is a participant in a key management infrastructure, such as a public key infrastructure (PKI) or Kerberos realm. The binding is performed by a trusted third party, such as a PKI certification authority or a Kerberos realm server[1].

6. Operational Use

Specifies operational availability of the keying material. Under normal circumstances, keying material is operational from its creation until the end of its life, which is usually defined by its expiration date.

7. Storage of Keying Material

Keying material should be stored depending on its type, protection requirements and lifecycle stage. When the keying material is required for operational use, the keying material is acquired from operational storage when not present in active

memory. If the keying material in active memory or operational storage is lost or corrupted, the keying material may be recovered from a backup storage, providing that the keying material has been backed up [2]

8. Key Update

Once the key gets to the end of its lifecycle or before the end of the lifecycle it is required to replace the old key with a new one. There are a few ways to achieve this. The first is rekeying, which is available if the new key doesn't mathematically depend on the old one. The second option is the key establishment method, which was discussed earlier in the fourth section. The third option is replacement by a key update function. This method transforms the old key or master key by modifying it into a new key. In other words, the new key is derived from the old key or master key.

9. Key Recovery

Key recovery is a wider term with multiple available techniques. All of the techniques result in the recovery of the cryptographic key and information associated with this key. The process of key recovery involves retrieving keying material from backup or archive storage, which allows to recover the key alongside the identification of its owner, the date the key was originally created, the organization or application that created the key, and the identification of who owns the data protected by the key.

10. Key De-registration and Destruction

When the existence of the key becomes irrelevant to the system or application, it is required to have the means of removing this key and all associations with it. This can be achieved by de-registration of the key and furthermore with the destruction of the key. This means erasing the key and all traces of it on every media where the keying material was available. It is also worth noting, that it is impossible to prove this was achieved and that retention of the public key isn't considered a security problem.

11. Key Revocation

It is possible that for reasons like key compromise, removal of the entity from an organization, and so on, there will be a need to remove keying material from use before its end of the lifecycle. This can be done by notifying all entities that may be using this keying material to remove its validity as it should no longer be used. The notification should unmistakably identify the keying material, the date it was

revoked, and also the reason for the revocation. Additionally based on the reason for revocation entities that were using the keying material should decide if the entities should for example honor all signatures created by the keying material before the revocation happened.

1.3 Key Management Systems in Cloud solutions

First of all, it is important to mention the different types of cloud service models and cloud deployment modes. According to NIST SP 800-145[4], there exist 3 cloud service models. Infrastructure-as-a-service(IaaS), Platform-as-a-service(PaaS) and Software-as-a-service(SaaS). Furthermore, the 4 different cloud deployment modes are Public, Private, Community, and Hybrid.

As was mentioned in section 1.2, secure key management has to provide confidentiality, data integrity, authentication, and non-repudiation.

The cloud solution, which will be introduced in Chapter 3. and on, which the implementational part of this thesis will be deployed on, is of IaaS service model and Hybrid deployment mode.

1.3.1 Key Management Methods in Cloud storages

There are six major key management methods for cloud storages, based on how the key is stored and distributed [12].

1. **Management of Key At Client Side** - data are stored at the cloud service provider side in an encrypted form. Keys are stored and maintained on the client side. This usually requires homomorphic encryption and the cryptographic operations are done on encrypted data on the server side.
2. **Key management at Cloud Service Provider Side** - keys are managed at the Cloud service provider side. This means that if the keys are lost, the client is forever unable to read their data present on the cloud.
3. **Management of Key at Both Sides** - in this approach, the key is divided into two parts. One part is stored on the client side and the other on the server side. Data can be decrypted only if both parts are combined together. It is a great compromise between the previous two approaches and is also relatively scalable. The downside is that if any part of the key is lost, the data becomes impossible to decrypt.
4. **Key Splitting Technique** - key is split into multiple parts and distributed among users. If a single user wants to decrypt data, they need key parts from

the rest of the users. This can also work on the principle of threshold where only k of n keys are required to decrypt data.

5. **Key Management At Centralized Server** - approach based on public key cryptography. Data is stored in an encrypted form on the server side and encrypted by a public key, which is stored on the server side. When a user accesses data, it is decrypted by a private key stored on the device of each user. The disadvantage of this system is the classic problem of a single point of failure in case the server crashes or encounters other issues.
6. **Group Key Management For Cloud Data Storage** - data is stored in the cloud by a chosen group. The group key is established for the encryption of data on the server side. Group key is formed by multiple partial keys maintained by users in a chosen group. If the chosen group changes (for example somebody leaves or joins the company) new group key is formed.

1.3.2 State of art Key Management Systems in Cloud storages

When it comes to state of art key management systems (KMS) in cloud storage it requires taking into consideration more than just NIST key management lifecycle requirements. Even though the cloud provides tremendous business and technical advantages, security concerns surrounding it are preventing cloud from getting widespread [17].

Amazon Web Services

Amazon Web Services are currently the biggest cloud provider on the planet and their product is considered state of art by many academics. Customers are encouraged to adopt AWS security practices for a secure application environment, prioritizing data confidentiality, integrity, and availability. AWS employs certifications like SOC 1, ISO 27001, and PCI DSS, ensuring compliance with industry standards [17]. Physical security measures safeguard Amazon controlled data centers globally. AWS services are designed with built-in security features, balancing flexibility with access restrictions. Users can encrypt data in the cloud, implement backup and redundancy measures, and follow AWS guidelines to ensure data protection and application continuity.

1. **AWS Infrastructure Security** - when transitioning IT infrastructure to AWS, a shared responsibility model emerges. AWS handles operations, management, and control from the host OS to physical security. Users manage the guest OS, application software, and AWS security group firewall configuration.

2. **Security Best Practices** - security is a top priority in a multi-tenant environment, requiring implementation across all layers of cloud application architecture. While the service provider manages physical security, users are responsible for network and application-level security.
3. **Protect data in transit** - for sensitive or confidential data, enable SSL on the server instance. Obtain a certificate from an external certification authority such as VeriSign or Entrust. The certificate's public key authenticates the server to the browser and establishes the shared session key for encrypting data in both directions [17].
4. **Protect stored data** - to enhance security when storing sensitive data in the cloud, users can encrypt individual files before uploading them. Decrypting is necessary after download. Regardless of the chosen system or technology, encrypting data at rest poses challenges, such as the risk of data loss if keys are misplaced or leaked. It is crucial to assess the key management capabilities of offered products to minimize such risks.
5. **Protect AWS credentials** - AWS provides two types of security credentials: AWS access keys and X.509 certificates. The AWS access key consists of an access key ID and a secret access key. When utilizing the REST or Query API, users must employ the secret access key to compute a signature for authentication. For enhanced security, it is recommended to transmit all requests over HTTPS to prevent modifications during transit.
6. **Manage multiple users with IAM** - Identity and Access Management (IAM) eliminates the necessity of sharing passwords or access keys and facilitates easy enabling or disabling of access for Users. It supports security best practices, like least privilege, by assigning unique credentials to each User and granting permission only for the necessary AWS Services and resources essential for their tasks [17].
7. **Secure Applications** - Amazon EC2 instances are safeguarded by security groups, which are rule sets defining allowed inbound network traffic. Users can specify TCP, UDP ports, ICMP types/codes, and source addresses. Security groups act as basic firewalls for instances.

1.3.3 Hashicorp Vault as Opensource Key Management System

HashiCorp Vault is a tool designed for managing secrets and protecting sensitive data in a distributed infrastructure. It provides a secure and centralized way to store, access, and distribute secrets such as API keys, passwords, certificates, and encryption keys. Vault is particularly useful in dynamic cloud or containerized environments where applications and services need access to secrets securely. Hashicorp

vault will be used in the implementational part of the thesis as SSM (Software secure module). It has been evaluated by [20] as the best free KMS solution for small businesses based on SWOT(Strength, Weakness, Opportunities, and Threats) analysis. SWOT analysis was done on 4 metrics including systems, infrastructure, applications, and human resources.

Hashicorp vault was chosen for this thesis as a potential candidate due to its performance, ability for access control, the option to use the vault server through REST API, and its business potential due to the option for the enterprise version. The only potential downside Hashicorp vault has, is its inability to start a server without first unlocking it via key [20]. Other opensource options that were explored are OpenStack Barbican and Knox. For readers interested in a detailed comparison of KMS market options thesis suggests readers go through [19]. For more details on HashiCorpVault, OpenStack Barbican, and Knox readers might be interested in the thesis from Jacob Gustaffson [21].

However, as with any type of vault solution, there is also an associated access management layer. This means that the product still relies on the authentication of the chosen method, e.g. AWS or Azure authentication, to secure the keys inside Vault. Even though the protection against extraction is high, it relies on authentication and if this authentication scheme is broken, the attacked can still access the operations that are performed by the secret key [25]. Therefore, this solution is mostly suitable in combination with paid and well-designed authentication schemes, like the ones from AWS or Azure mentioned above, and might not be the best choice for our system, which will be introduced in Chapter 3. Another problem with Hashicorp vault is, that it does not support post-quantum algorithms like Crystals-Kyber at this time, which would be a preferable solution for the long-term security in this thesis.

1.4 Master Key and It's Security

The Master Key is a cryptographic key that is used to encrypt and decrypt other keys, such as data encryption keys (DEKs). It is a critical component of cloud security, as it is used to protect sensitive data in cloud storage and applications. The security of the Master Key is paramount, as a leaked key can lead to a data breach.

To ensure the security of the Master Key, it is essential to follow best practices for key management. Microsoft recommends using a Zero Trust security strategy, which comprises of three principles: "Verify explicitly," "Use least privilege access," and "Assume breach" [13]. Data protection, including key management, supports the "use least privilege access" principle. Other best practices include using strong

passwords, implementing multi-factor authentication, and regularly rotating keys [13].

Traditional key management solutions based on HSM (Hardware Security Module) or TPM (Trusted Platform Module) provide strong security, but they are expensive and do not scale well. Therefore, it is necessary to find a solution to meet both security and scalability [12].

Protecting Master Key

The key protection is an important guarantee for the security of key cryptography systems. The basic principle to guarantee key security, is that the key cannot appear in plain text except for the generation, distribution, loading and storage of the key in the security cabinet [12].

Generation of Master Key

The process of master key generation is foundational to the security of cryptographic systems. This section examines the methodologies and algorithms employed for the creation of master keys in cloud computing. It addresses the importance of randomness, entropy sources, and key length in generating secure master keys along with consideration for cryptographic strength.

Secure Storage of Master Key

The secure storage of the master key is paramount to preventing unauthorized access and potential compromise. This section explores encryption techniques, hardware security modules (HSMs), and secure key storage protocols utilized to safeguard master keys. Additionally, it evaluates resilience against various attack vectors, including physical and cyber threats.

1.5 Hardware Secure Modules (HSM)

In this section, we explore the vital role of Hardware Security Modules (HSMs) in modern cryptographic systems, focusing on their definition, functionality, and their importance in the security of master key in key management. The section is structured into two main subsections, the first focuses on HSM principles in general, and the second focuses on the role of HSM in key management and operations they provide.

1.5.1 Hardware Security Module principles

A Hardware Security Module (HSM) is a physical or virtual device that manages digital keys for strong authentication and provides cryptoprocessing. These devices are designed to securely generate, store, and manage cryptographic keys used in encryption and decryption operations. There exist multiple standards to evaluate the security of HSMs but most often it is evaluated in accordance to FIPS 140-2.

Another important attribute of HSMs is that they are tamper-responsive. This means, that if there is an attempt to penetrate the HSM, it will trigger an automatic erasure of all secret information stored in the memory of the device. The HSM is either a peripheral device or a bus-connected device communicating with the host machine via Ethernet cable or fiber cable.

1.5.2 HSM and Master Key Security

Master keys are the apex of the key hierarchy in cryptographic systems used to encrypt or generate other keys, such as session keys or transaction keys. Managing these master keys securely is critical, as they can decrypt large volumes of data or create additional keys.

HSM API

Communication with HSM is usually done via HSM API. The vendor of HSM usually provides API modifiable by the customer or the vendor to make HSM better suited for the specific scenario it will be used for. Three-layer Architecture of HSM API can be seen in figure 1.1. Those layers are Hardware Layer, Firmware Layer, and Software/API Layer. The API is then used as any other API in a manner of requests and responses. These requests specify which cryptographic operation should be performed with the master key. This includes operations like encryption/decryption, generating a key, or signing [24]. This way the sensitive master key can rest in the tamper-resistant and temper-responsive HSM which provides a high level of security, while still being used for its purpose with relative ease.

AWS CloudHSM

The specific HSM, which might be a good solution for the system we will introduce in Chapter 3, is AWS CloudHSM. This HSM is deployed in the data center of AWS and managed by AWS. AWS CloudHSM provides hardware security modules in the AWS Cloud with which the customer can generate, store, import, export, and manage cryptographic keys. Encrypt and decrypt data with symmetric or asymmetric algorithms. Compute message digests and hash-base message authentication codes

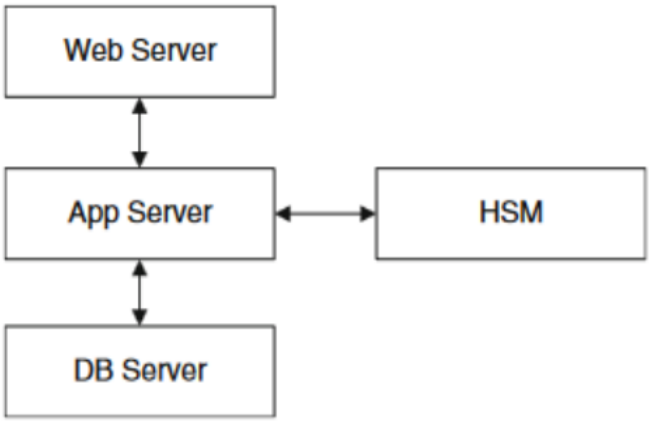

Fig. 1.1: Scheme of HSM API usage extracted from [24]

by cryptographic hash functions. Sign data cryptographically and verify signatures and generate secure random cryptographically [26].

AWS CloudHSM runs on Amazon EC2 instances and users can access it with Amazon Virtual Private Cloud(VPC) security controls. Additionally, the CloudHSM client establishes mutually authenticated SSL channels with the user, which are used to connect to the application. Since the instances are located on Amazon VPC they are managed by AWS Admin, however, admin does not have access to the master or user keys stored on the HSM, only the user can access his own keys.

2 Post–Quantum Safe Cryptography

With the emergence of quantum computing came also the birth of new algorithms threatening the security of encryption algorithms previously considered mathematically impossible to break. This specifically includes key-establishment schemes and digital signatures that are based on integer factorization problem and discrete logarithm problem like RSA, ECDSA and ECDH. Although the timeline for when quantum computers will have enough Qubits to break these cryptographic algorithms is not certain. This threat is something that requires the focus of the whole industry ahead of time as adoption of new cryptographic practices can take ten or more years.

Another algorithm threatening the current cryptography that can be executed on quantum computers is Grover's search algorithm. The Grover algorithm can be used for brute force attacks as it pushes the boundaries of secure key lengths. For example, AES with a 128-bit security level will drop to the 64-bit security level [14].

2.1 NIST PQC Standardization Process

The National Institute of Standards and Technology (NIST) is hosting a yearly public competition-like process to come up with potential candidates for standardized Post-Quantum safe public key cryptography. The chosen new standards will be compliant with FIPS 186-4 and Digital Signature Standard(DSS). This competition-like process is referred to as NIST PQC (post-quantum cryptography) Standardization Process.

This thesis focuses on the third round of NIST PQC which took place in 2022. While considering candidates NIST puts the biggest emphasis on the security of the proposed PQC algorithm. The secondary metric is performance. For performance evaluation, NIST considers both key and ciphertext byte size as well as computational complexity.

2.1.1 NIST PQC Candidates

The following table contains all the algorithms that made it into the finals of the third round of NIST PQC standardization process [6]. Out of 15 algorithms only 7 have been nominated as finalists. Four of those finalists are key encapsulation mechanisms(KEMs) and three of them are digital signatures.

Public-Key Encryption/KEMs	Digital Signatures
Classic McEliece	CRYSTALS–Dilithium
CRYSTALS–KYBER	FALCON
NTRU	Rainbow
Saber	

2.1.2 NIST PQC Candidates Comparison

The following chapter will have a look at different candidates from NIST PQC competition and compare their performance costs on Android, IOS, and Cortex platforms. Their empirical results showed that PQC algorithms cause affordable latency in TLS. The performance is similar to ECDH and ECDSA. PQC schemes just cause higher packet overhead than classic schemes [7]

From the figures below 2.1 and 2.2, we can easily see that Kyber is the best performing key encapsulation mechanism and Dilithium does best from Digital signature algorithms. It should also be mentioned that Falcon shows great potential for systems that are significantly more reliant on signature verification as opposed to key generation and signature creation.

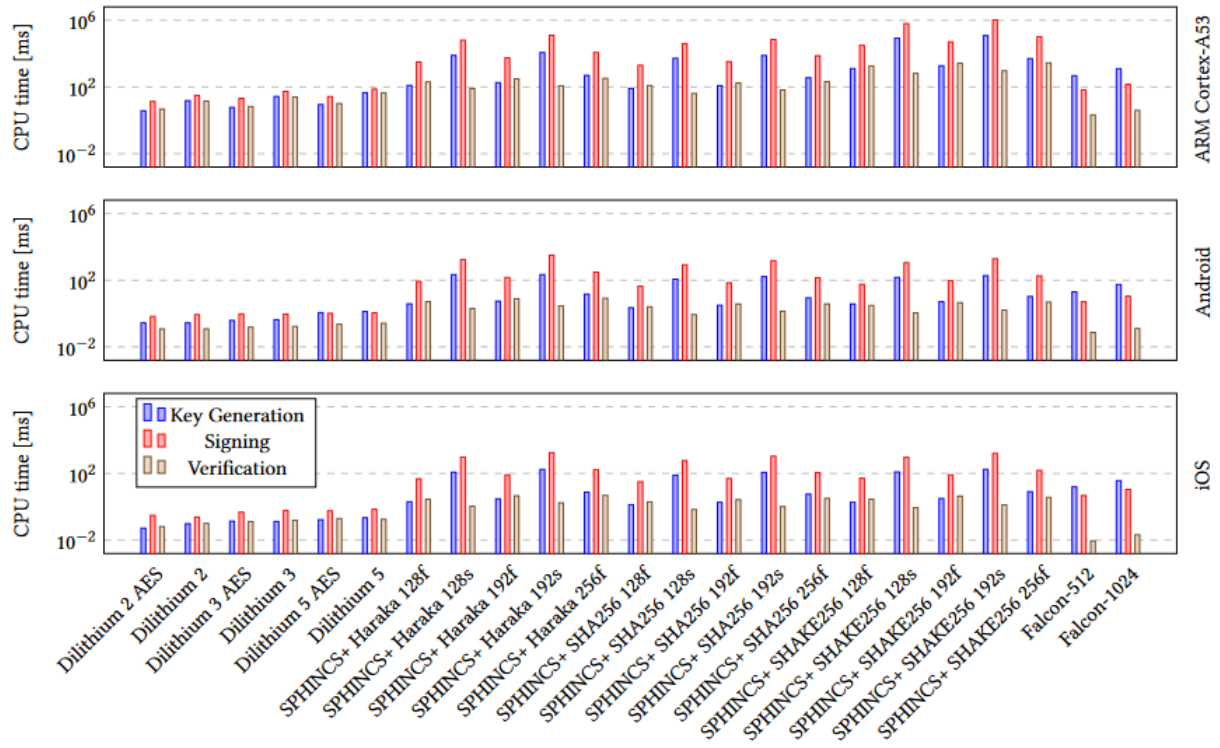

Fig. 2.1: Digital signature algorithms comparison extracted from [7]

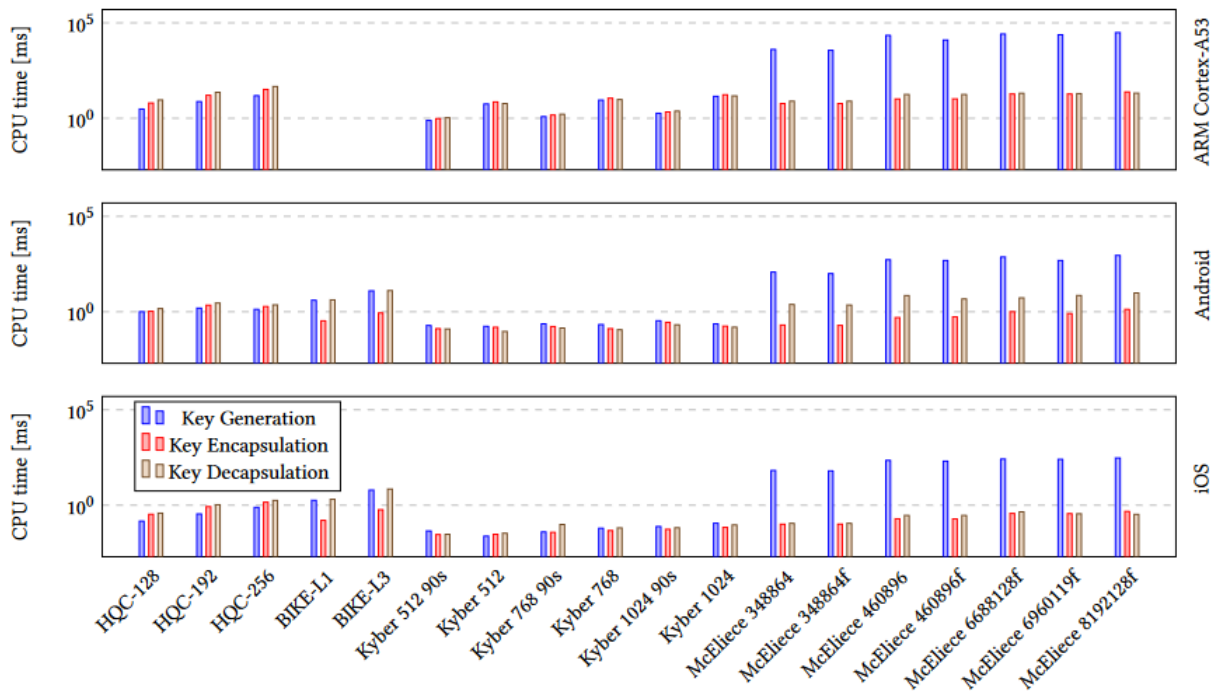

Fig. 2.2: KEM algorithm comparison extracted from [7]

2.2 Crystals-Kyber

Crystals-Kyber is the version of Kyber with a parameter set for the highest security. Kyber is an asymmetric/public key algorithm. It is used as an IND-CCA2-secure key-encapsulation mechanism (KEM). It is considered a lattice-based algorithm which bases its security on the hardness of solving the learning with errors problem in module lattices.

2.2.1 Parameter sets

There are three specified parameter sets, see table 2.1 Kyber512, Kyber768, and Kyber1024 [8]. When Crystals-Kyber is mentioned, it refers to the parameter set of Kyber1024 which provides the highest security of NIST security level 5 which is equivalent to AES 256.

Table 2.1: Parameter sets for Kyber. Table extracted from [8]

	n	k	q	n_1	n_2	(d_u, d_v)	δ
Kyber512	256	2	3329	3	2	(10, 4)	2^{-139}
Kyber768	256	3	3329	2	2	(10, 4)	2^{-164}
Kyber1024	256	4	3329	2	2	(11, 5)	2^{-174}

- **n** is always set to 256 and specifies the entropy to encapsulate the key with 256 bits.
- **k** is the dimension of the lattice as a k multiplied by n, if a user wants to increase Kyber's security changing the value of k is the main way to do so.
- **q** is a small prime number which satisfies $n|(q-1)$. It is a requirement for NTT-based multiplications. There are two smaller prime numbers that satisfy this, however, using these would result in a higher failure probability and thus make Kyber, not CCA (Chosen ciphertext attack) secure.
- δ is the failure probability.
- Rest of the parameters n_1, n_2, d_u and d_v are chosen based on requirements for security, ciphertext size and failure probability.

2.3 Crystals-Dilithium

Similarly, Crystals-Dilithium is named for the Dilithium algorithm set with parameters for maximum security, see table 2.2. Crystals-Dilithium is also sometimes called Dilithium 5. Dilithium is an asymmetric/public key algorithm used for Post Quantum digital signatures. Similarly to Kyber also Dilithium is based on the hardness of solving the learning with errors problem in module lattices.

2.3.1 Parameter Sets

- **q** used to denote a prime number, which is a key parameter in many cryptographic algorithms. In Dilithium, it represents the size of a finite field or a group.
- **d** associated with the difficulty of the underlying mathematical problem that the cryptographic scheme relies on. Related to the hardness of solving certain mathematical equations.
- τ used in mathematics to represent various quantities. In Dilithium, it is related to the security and efficiency of the digital signature scheme.

- γ_1 and γ_2 additional security parameters in the Dilithium signature scheme. They are used to adjust the scheme's security level or performance characteristics.

Table 2.2: Parameter sets for Dilithium. Table extracted from [10]

NIST Security Level	q	d	τ	γ_1	γ_2	(k, l)	η	β	ω
2	8380417	13	39	2^{17}	$(q-1)/88$	(4, 4)	2	78	80
3	8380417	13	49	2^{19}	$(q-1)/32$	(6, 5)	4	196	55
5	8380417	13	60	2^{19}	$(q-1)/32$	(8, 7)	2	120	75

Main Operation

From the perspective of computational complexity, the main operation in the entire scheme is polynomial multiplication over the ring R_q via the NTT. To be more precise, the multiplication operands in this scheme are vectors or matrices whose coefficients are polynomials in R_q, so there are many continuous polynomial multiplications in the scheme [10].

Signature Generation

The Sign algorithm first generates a seed $\rho^{'}$ and then performs a loop to generate a signature until it meets a series of security conditions. In the loop, the main operations are hashing and four multiplications, i.e.,A_y,cs_1,cs_2, and ct_0 [10].

2.4 AES-256-GCM

This section will shortly describe AES (Advanced Encryption System). It is a block cipher-based symmetric key cryptography algorithm used for (Encryption/Decryption) of information to provide the confidentiality and integrity to the data. The key sizes of AES are 128 bits, 192 bits and 256 bits when using AES we also have to take into consideration that Grover's algorithm makes its security half, meaning AES-256 can be broken with 2^{128} iterations, therefore in scenarios where post-quantum safety is required only AES-256 should be used.

The input size to the (encryption/decryption) is a single 128-bit block. This block is represented as a 4×4 square matrix of bytes. This block is copied to the state array and it is modified at each stage of (encryption/decryption). After the last stage, the state is copied to the output matrix [29].

The GCM part in the AES-256-GCM refers to Galois/Counter Mode, this version of AES is a very popular encryption use case. AES-GCM has good performance, is

highly optimized, FIPS-approved, and well-trusted. In order to reuse a single AES key for multiple encryptions there exists the concept of initialization vector (IV) which is a random value often also referred to as nonce (Number Used Once). This value is typically 12 bits long. It is also worth mentioning NIST limits the number of messages that can be encrypted with a single key to 2^{32} [28]. This is a pretty high number and in our use case, it will not be limiting us as the re-encryption will be happening once every few months or weeks.

2.5 Kyber1024 + AES256-GCM Use Case

The KEM (Key Encapsulation Mechanism) in Kyber1024 involves generating a pair of keys, a public key and a private key. The public key can be shared openly, while the private key is kept secret. In practice, when a sender wishes to securely transmit/store data, they use the public key to encapsulate a randomly generated secret key. The encapsulation process produces a ciphertext, often referred to as the KEM ciphertext. This ciphertext can be safely transmitted/stored in the insecure space and can be only accessible by the holder of the corresponding private key.

Upon time of decryption, the owner uses their private key to decapsulate the stored KEM ciphertext. This decapsulation process retrieves the original secret key, which was randomly generated and encapsulated in the storing process. It is crucial that this decapsulation mechanism is secure against both classical and quantum attacks, as the integrity and confidentiality of the subsequent symmetric encryption rely on it. The whole process is very nicely illustrated in the figure 2.3.

Once the shared secret is securely established via the KEM process, it is utilized to initiate an AES256-GCM symmetric encryption key. AES256-GCM (Advanced Encryption Standard 256-bit Galois/Counter Mode) is a widely trusted symmetric key cipher that provides both confidentiality and data integrity. The shared secret generated and established through Kyber1024 serves as the key for AES256-GCM encryption. Before the AES encryption process begins, a nonce (number used once) is generated—this is crucial as it ensures the uniqueness of each encryption operation, preventing various types of cryptographic attacks such as replay attacks. This nonce has to be same in the encryption and decryption process to reverse the encryption function, therefore it needs to be saved alongside with ciphertext.

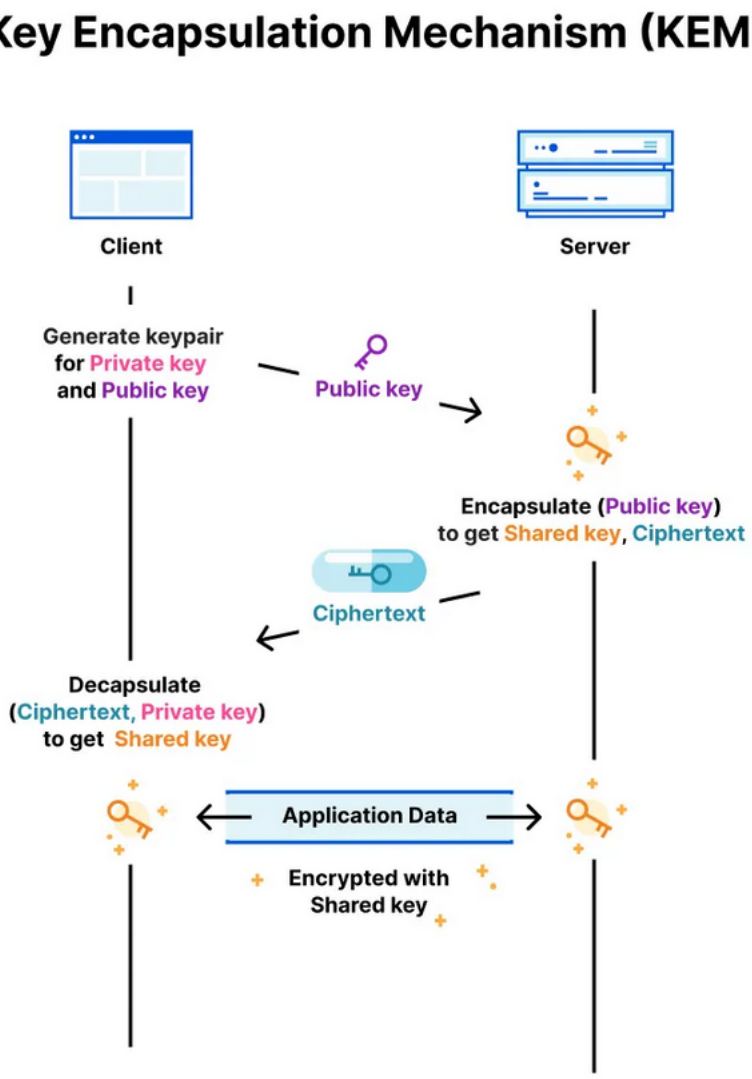

Fig. 2.3: Illustration of KEM process to establish AES key extracted from [31].

3 Proposed Solution for BUT Cloud Based Archive System

Currently, the BUT cloud-based archive system communicates with the Nextcloud instance via REST API and gets the records from the police investigators this way. It is necessary to ensure that the system can provide long-term data integrity and confidentiality that would also be considered post-quantum safe. The project is mostly written in Python with Nextcloud parts of it being written in PHP.

It is also important to reasonably specify who will be eligible to retrieve the information back from the archive and thoroughly consider how the processes of key revocation and key recovery will work. The thesis is based on the expectation that there exists a trust worthy admin of the archive system, who can safely keep a twelve-word mnemonic password used to protect the master key of the archive system.

The individual parts of the whole system, like Archive System, Nextcloud instance, MPC instance, and WebID instance, are running as docker containers and the whole network can be simulated with docker-compose. One of the requirements for key management is that it should exist like a standalone system on the archiving system, only meaning it is not dependent on other applications like NextCloud instance or WebID.

3.1 Current Architecture of BUT Archive System

In this section, we will explore the current archiving and timestamping process in the BUT archive system, and we will propose how the service responsible for this process could be extended by an encryption process. For further information read the following two theses are recommended as they write about design decisions and the current state of the system [23] and [22]. The scheme of the architecture can be seen in the figure 3.1.

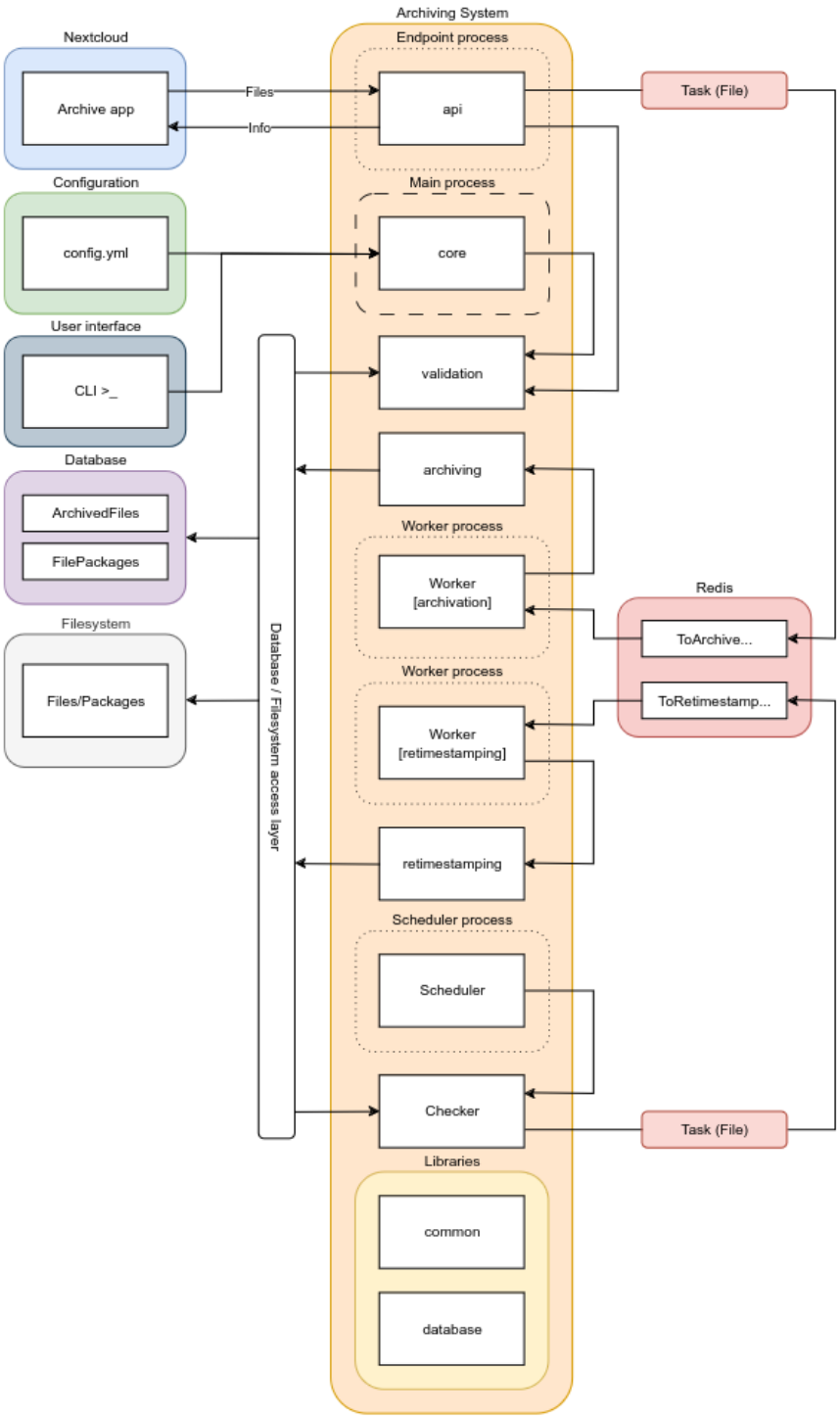

Fig. 3.1: Current architecture of archiving system extracted from Martin Nohava's thesis [23]

This thesis will not describe the whole architecture made in previous theses, but there are 3 important modules worth briefly describing to get a better understanding of cryptographic solutions further introduced in this chapter.

- **api:** This module is responsible for communication through REST API between the archivation system and nextcloud instance, which are running in separate docker containers. Through post requests, it is possible to send files from the nextcloud instance into the archivation system.
- **archivation:** This module is responsible for signing of data transfered through REST API. In this module, the data is signed with the dilithium private key, timestamped, and archived on the archiving system. The process is automated through the archivation worker subprocess awaiting incoming requests. This module is greatly suited to be extended by encryption functionality by the cryptographic module developed in this thesis.
- **retimstamping:** Module responsible for re-timestamping of the archived files. Similarly to the archivation module, there also exists a worker subprocess controlling the expiration of timestamps and re-timestamps files, that have a timestamp at the end of the life. This makes this module a great candidate to add a re-encryption mechanism by cryptographic module.

3.1.1 Deploying and Running The System

The separate applications and databases are running as docker containers. Therefore in order to run the whole system, docker and docker compose are required. All of the containers are based on images from Docker Hub. The archivation system specifically uses an Ubuntu-based image with Python called `python:3.11-bullseye`.

All of the properties for applications like the port they run on, the credentials for administrators, the volumes and environment variables, are specified in the `docker-compose.yml` file. In order to build containers from this file, one can simply run `docker compose build`, once the images are built they can be started by command `docker compose up -d`. The application is the nextcloud application from which the archivation system is accessed through UI, it is then running on `https://localhost:8443`. Additionally, configuration more specific for the standalone archiving system instead of the whole ecosystem can be found in `config.yml` file. This file is loaded when the archivation system is started and can be easily extended for new environment variables or change the existing ones.

3.2 Chosen Libraries

The following subsections introduce individual libraries used for key management solution of cloud-based archive system. This section should provide the reader with an explanation of why the libraries were chosen and also describe the basic understanding of their structure and capabilities.

3.2.1 LibOQS

LibOQS is an opensource library developed as part of the post-quantum safe initiative with website `https://openquantumsafe.org/`. LibOQS is focused on NIST PQC cryptography available in multiple languages including Java, Javascript, GO, C, C++ and Python. The thesis intends to use the Python version of LibOQS library. All of the development for the library is happening in this public GitHub repository `https://github.com/open-quantum-safe`. For the purposes of the project, the most important part of the library is its implementation for Kyber KEM (Key Encapsulation Mechanism). Liboqs supports Kyber512, Kyber764 and Kyber1024, we will use the most secure version Kyber1024 (Crystals-Kyber). It has a public key length of 1568 bytes and a private key length of 3168 bytes. An overview of different parts of liboqs library can be seen in figure 3.2

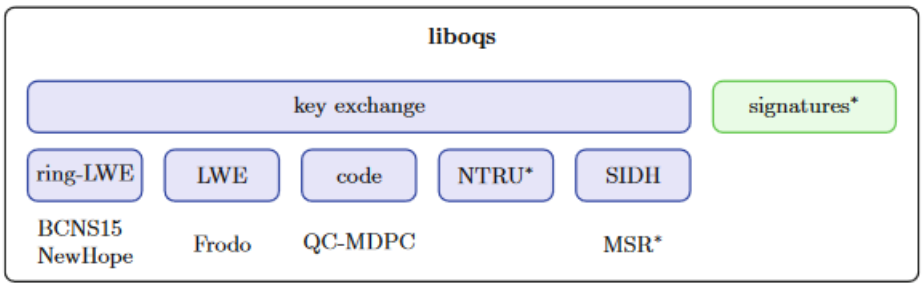

Fig. 3.2: Overview of structure of liboqs library extracted from [27]

In order to make liboqs usable for purposes of the thesis and make libraries's operation viable for long-term confidentiality of the files archived on the system, it is required that liboqs provides functionality to load public and private keys from the database or disk and makes them available for encapsulation or decapsulation of the shared secret. This is not, unfortunately, the case with libraries decapsulate_secret function. It is however easily fixable just by slightly tweaking the function and overloading it in our project. The modified function with parameterized private_key loading can be seen below in figure 3.3.

```python
def decapsulate_secret(self, ciphertext, private_key):
    """
    :param ciphertext: The ciphertext to decapsulate.
    :param private_key: The private key to use for decapsulation.
    """
    if len(private_key) != self.kem._kem.contents.length_secret_key:
        raise ValueError("Invalid private key length.")

    # Create ctypes buffers for the operation
    ct_ciphertext = ct.create_string_buffer(ciphertext,
        self.kem._kem.contents.length_ciphertext)
    ct_private_key = ct.create_string_buffer(private_key,
        self.kem._kem.contents.length_secret_key)
    ct_shared_secret = ct.create_string_buffer(
        self.kem._kem.contents.length_shared_secret)

    # Perform the decapsulation using the provided private key
    result = oqs.native().OQS_KEM_decaps(
        self.kem._kem, ct.byref(ct_shared_secret),
        ct_ciphertext, ct_private_key )
    if result != 0:
        raise Exception("Decapsulation failed")

    return bytes(ct_shared_secret)
```

Fig. 3.3: Modified Python code for decapsulating secret, orifinal code can be found at `https://github.com/open-quantum-safe`

3.2.2 Python Cryptography Module

Another library used in the solution of the project is the official Python cryptography module. It is one of the most widely used and tested Python libraries for already widely adopted cryptography. It was intended to be the standard library for cryptographic operations in Python. In the project, this library is mainly used to provide functions for the creation of AES256-GCM and operations with these keys. It is also used to generate AES256-CBC from PBKDF12 and 12 mnemonic password. For more information on the library see `https://cryptography.io/en/latest/`

3.3 Proposed Cryptographic Scheme

A lot of processes in this section would highly benefit from the assurance of communicating with authenticated users. In the existing system, there is a concept of file owner inside of Nextcloud instance, which will be used to identify which keys should be used for relevant operations in the system. However, the system at this point does not guarantee strong authentication of the file owner with for example OAuth2 outside of Nextcloud. The proposed cryptographical system will be utilizing Kyber1024 asymmetrical keys from which there can be created shared secret that can be further used to generate AES256-GCM, which can finally be used to encrypt archived files.

There exists a hierarchy of keys, where every user or group of users will have its Kyber1024 key pair used to encrypt the files of this owner. Additionally, the user/owner private keys stored on the archiving system will be protected by Kyber1024 Master Kyber Public Key and Master Private Key will be protected by PBKDF2 key derived from 12-word mnemonic.

3.3.1 Encryption in Transit from NextCloud to Archive System

When data are in transition from NextCloud instance to Archive storage, they are transferred over TLS. However, as of now, once they are saved on the archiving system the data at rest are not protected yet. It is also expected, that there is proven authenticity of police investigators sending data through NextCloud instance. Once data arrive at archive storage, they are paired with metadata which are used to create a link between the file that is about to be stored on the archive system and police officers identity.

3.3.2 Master Key Pair

The master key pair will be managed on the archiving system by archive administrator. It will be an asymmetric cryptographic algorithm for key encapsulation mechanism (KEM) like Crystals-Kyber and the private key will be saved in the system and will be encrypted by twelve-word mnemonic and PBKDF2 (Password-Based Key Derivation Function 2). The public key will be used to encapsulate a secret from which a more effective AES256-GCM key will be created and this key will later be used to encrypt user private keys. User key pairs are Kyber keys created with the intent to protect sensitive information of specific police investigators, police investigator groups, or system roles that interact with the archive system. The master key plays the role of creating the root of trust in the system and ensuring it is not leaked, it is essential to the security of the system.

Master Key Generation

The master key pair will be generated with a specialized script run on the archive system setup. This script will immediately, after generation, encrypt the master private key with PBKDF2 and twelve-word mnemonic to simulate an encrypted secure container. This should be performed only one time, when the archive system is initially deployed and twelve-word mnemonic will be shown only one time to the archive administrator, who should note it and safely store it somewhere. After this master private key will never exist as plaintext outside of memory and its plaintext form in memory will be as limited as possible. It is expected that the archive admin can be trusted and the twelve-word mnemonic will not be lost. It would be preferable, to create this safe container with HSM (hardware secure module). However, due to the nature of the project being in the proof of concept stage and being fully virtualized, this is not possible. The process is visualized in the diagram below in figure 3.4.

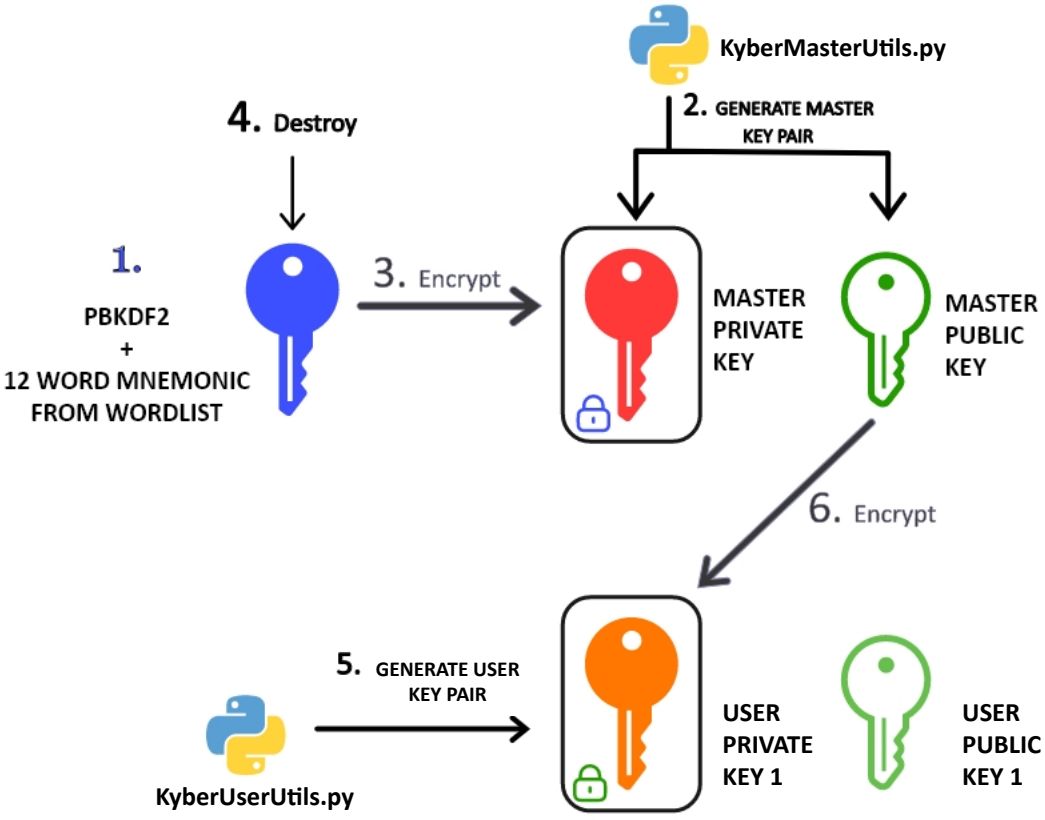

Fig. 3.4: Diagram of master key generation

3.3.3 User Key Pairs

User key pairs will be, similarly to master key Kyber1024 key pairs, used to establish symmetric and more effective AES256-GCM keys, which can be finally used to encrypt data saved on the archiving system. Private keys will be encrypted by master public key and public keys will be available in plain text to the services/workers in the archiving system. There will be one user key pair per police investigator, police investigator group, or system role. If user key pairs do not exist for the relevant owner or group, they are generated during the archivation process.

3.3.4 Key Management Hierarchy

The figure below 3.5 illustrates the hierarchy of master and user key pairs. The figure clearly illustrates encryption dependencies and the order, in which the trust is established in the key management system. The main point of creating multiple user key pairs for officers and groups of officers is to minimize the blast radius in case a key is exposed. The whole chain of trust is dependent on the password-based key derivation function and twelve-word mnemonic, known only to the archive admin and also managed by the archive admin. As was said previously, this key should never exist outside of memory and also should be kept in memory only for a brief moment. All of the private keys, both from master key pair and data key pairs, also never exist in plaintext outside of memory, and their plain text form in plain text memory is as limited as possible. Note that all of the private keys, both master and user private keys, never exist in plaintext outside of the memory. In the database or filesystem private keys will always be encrypted.

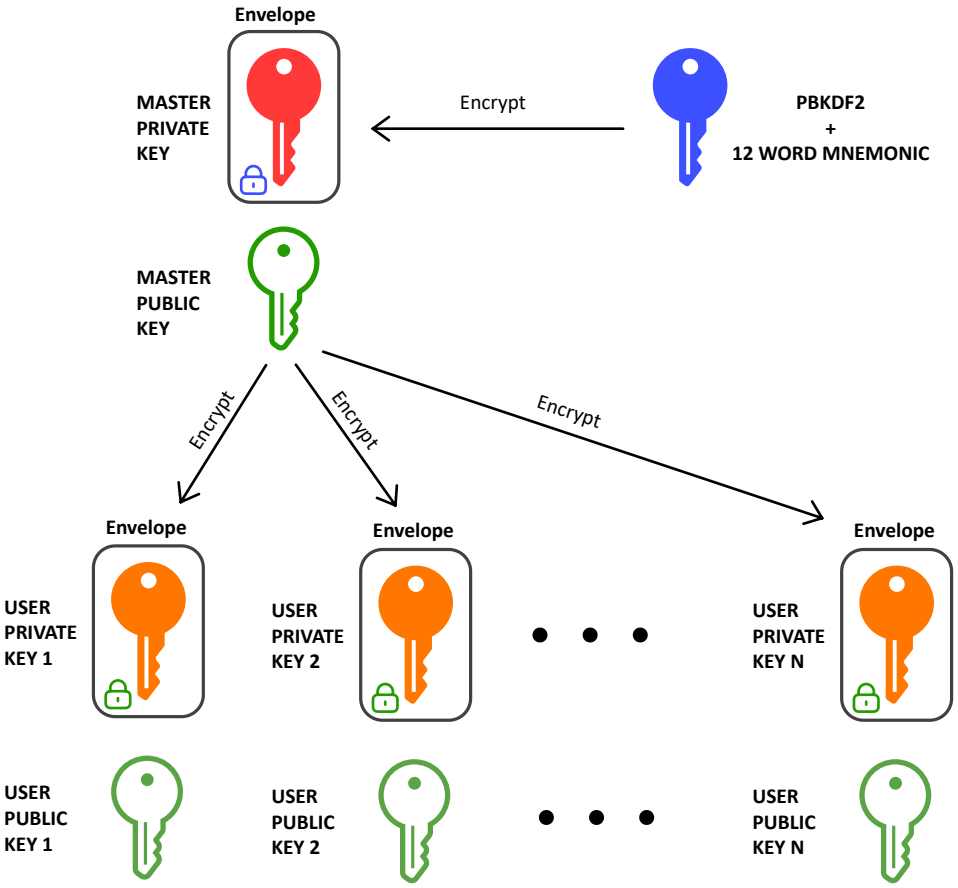

Fig. 3.5: Hierarchy of Master and Data key pairs

3.3.5 Data at Rest Encryption

When data arrive from the NextCloud instance, they are placed in the ARCHIVE_QUEUE, which marks them to the archivation worker to start with all of the processes required for archivation like timestamping, creation of the hash, and signing the package with Dilithium private key. This thesis will extend this functionality by encrypting data with an AES symmetric key created from the user public key. This has to be done as the first thing before creating the hash of the file so the file can later be validated before decryption. Data will also need to have some type of metadata with information about the file owner, nonce used to create the AES key, and ciphertext from which the shared secret can be recovered with Kyber private key later at the time of decryption. An illustration of the process can be seen in figure 3.6.

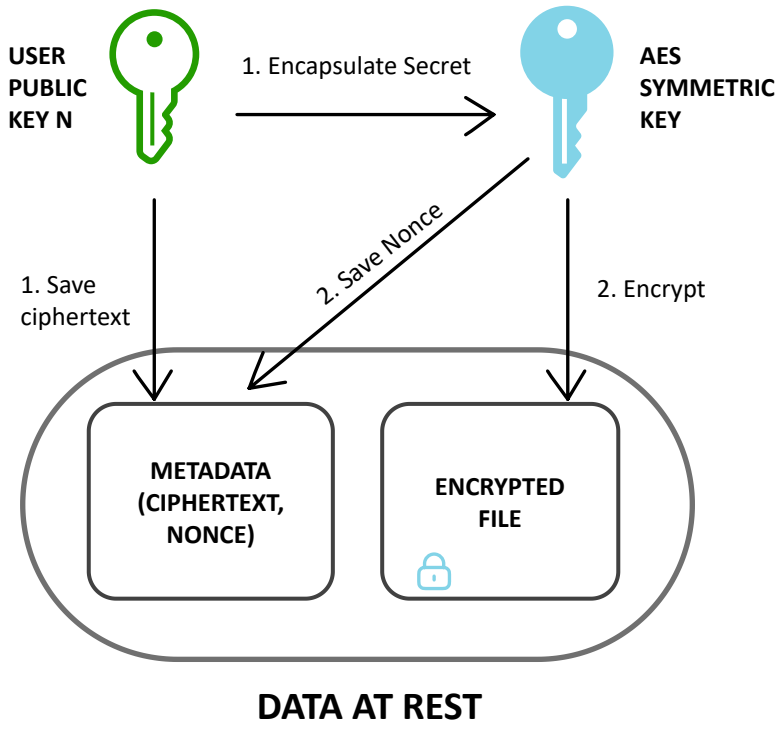

Fig. 3.6: Process of encrypting data at rest

3.3.6 Data at Rest Decryption

The decryption process is the more difficult part of the archiving system as this operation requires the decryption of multiple keys. For example, if there is a request to decrypt a file owned by the user ncadmin. First of all the twelve-word mnemonic has to be entered into the system. From this password with the use of PBKDF2, the master private key can be decrypted, then ncadmin's user private key has to be decrypted and only after that the file can be decrypted. In this process, all of the decrypted keys exist only in the memory and the ciphertext and nonce stored in metadata are required, see figure 3.7.

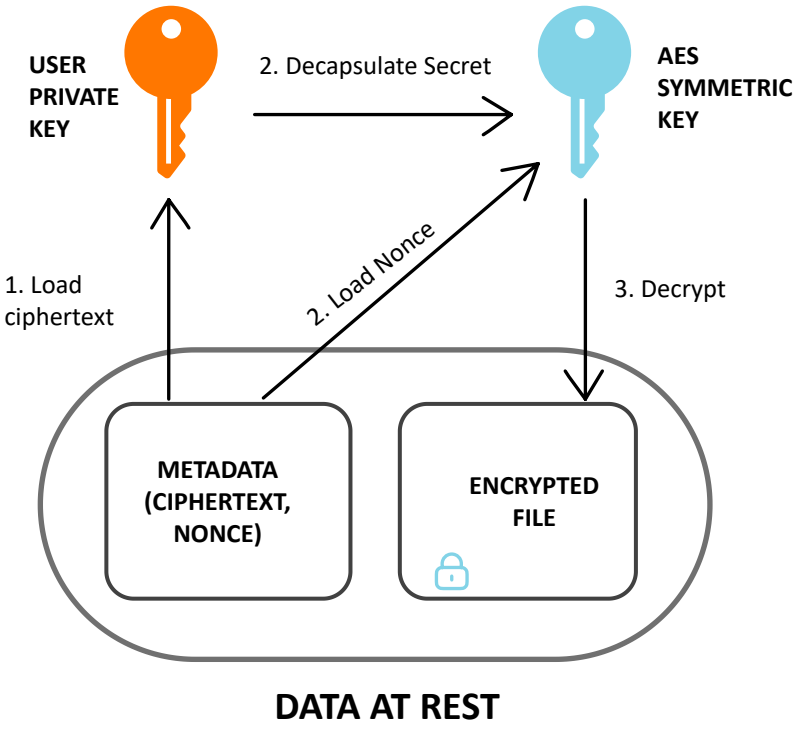

Fig. 3.7: Process of data at rest decryption

3.3.7 Key Recovery

The approach chosen in this thesis is to create a server-side key management system minimizing the likelihood of keys being lost as it removes the issue where the user has to manage his own key pair and can lose it while changing devices or losing data on his personal device. However, there still could be other problems with for example corrupted disk or some type of attack against the archiving system. Therefore, it is required to periodically backup the keys and store them preferably physically outside of the archiving system. This way there is minimal risk to lose the ability to decrypt any files, even over long periods of time.

3.3.8 Resource Sharing

Resource sharing should be done via the Identity and Access Management approach. If a police officer requests to share resources stored on the cloud with another officer, it will most likely be done by extending metadata associated with the file by adding the identity of the officer who should be granted access. Ideally, this will be done by OAuth2 or OIDC. However, this is not within the scope of this thesis.

3.3.9 Key Rotation & Re-Encrypting Mechanism

It is recommended practice in the Key Management Lifecycle to rotate keys. The timestamping service introduced by my predecessor [23] will be extended by a re-encrypting function designed to change the key protecting the stored file.

The Kyber encapsulate secret function generates different `shared_secret` every time it is called. We will use this behavior to create a new AES symmetric key every time the re-timestamping worker finds a file that needs to be re-timestamped. When this happens, the file gets decrypted in the memory and overwritten with the same file encrypted with the new key, and ciphertext and nonce in the file metadata will similarly be changed in order to still have the ability to decrypt the file later.

4 Solution Implementation & Performance Benchmarking

This chapter will finally get into the specifics of the implementation of the project. It will introduce processes within the project, describe python code snippets written in the project, and provide a better understanding of the cryptographic module of the archiving system by introducing and describing block schemes. Later, sections will be then focused on benchmarking that has been done on the cryptographic module and lastly provide some ideas for future work in the archiving system.

4.1 Implementation Details & Block Schemes

This section focuses on the explanation of cryptographic operations in the system. It describes those processes using block schemes and also showcases and describes the most important parts of Python code responsible for those operations in the archiving system.

4.1.1 Developed Cryptography Module

The cryptography module implemented to extend the archiving system with the ability to provide long-term confidentiality of data at rest is divided into 3 main classes. A visual representation of the cryptography module can be seen in the figure below 4.1. The classes are used to provide confidentiality of keys from the top down, meaning PBKDF2Utils are used to protect keys from KyberMasterUtils and KyberMasterUtils keys are used to protect KyberUserUtils key pairs.

- **KyberUserUtils:** Class provides basic operations for operations and manipulation with Kyber1024 user key pair and AES256-GCM constructed from their shared secret.
- **KyberMasterUtils:** Class provides basic operations for operations and manipulation with Kyber1024 master key par and AES256-GCM constructed from their shared secret.
- **PBKDF2Utils:** Class used to construct AES256-CBC key from twelve-word mnemonic and operations with this key.

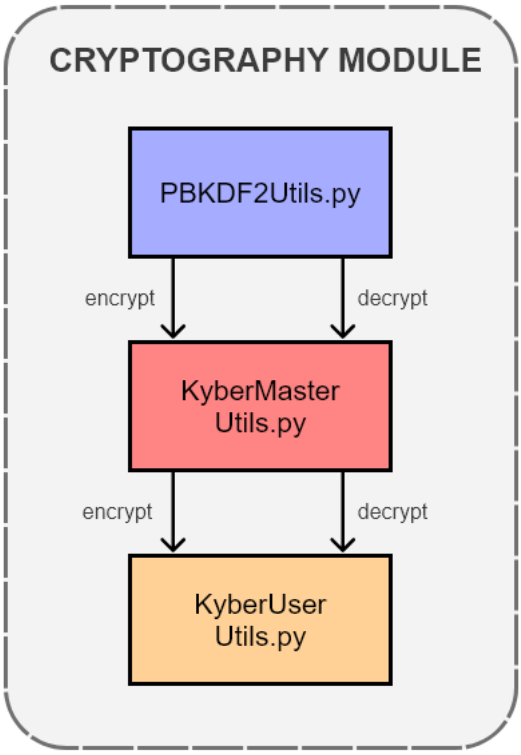

Fig. 4.1: Structure of Cryptography Module Extending Archiving System

PBKDF2Utils

This class is used to construct the AES256-CBC key from a twelve-word mnemonic and execute operations (encrypt, decrypt) with this key. The purpose of the class is to provide functions required to add the ability to protect and use the Kyber1024 master key. Three main functions found in the PBKDF2Utils are generate_key(password), encrypt(plaintext), and decrypt(ciphertext).

The generate_key() function requires in total of 5 parameters. The first parameter algorithm specifies a hash function that will be used during the key derivation, in our case `SHA256`. The second algorithm specifies the length of the derived key. This value is in bytes, so in our case, the length of 32 will generate a key, which is 256 bits long. Salt is a random value used to increase the security of the key derivation process. It should be unique for each password and in our case, it is specified in the initialization of `PBKDF2Utils` class. Parameter iterations specify the number of iterations that determine the computational effort required to derive the key. A higher number of iterations generally leads to stronger security but also increases computational cost. In this case, we are using 100000 iterations. Lastly, the backend specifies the cryptographic backend to use for the key derivation process.

Cryptographic backends handle low-level cryptographic operations. In our case, the default backend from `cryptography.hazmat.primitives.kdf.pbkdf2` is used.

When the `PBKDF2HMAC` is initialized with the parameters described above. Function `PBKDF2HMAC.derive(password.encode())` can be finally used. This key will be of length 256 bits and will be later used to construct an AES256-CBC symmetric key used for encryption and decryption of the Kyber1024 master private key. The whole function can be seen in the code snippet below, see figure 4.2

```
def _generate_key(self, password):
    """Generates a key using PBKDF2."""
    kdf = PBKDF2HMAC(
        algorithm=hashes.SHA256(),
        length=32,
        salt=self.salt,
        iterations=self.iterations,
        backend=self.backend
    )
    return kdf.derive(password.encode())
```

Fig. 4.2: PBKDF2Utils function for generation of the kdf secret

Once the PBKDF2Utils has been initialized and we have derived a key from the password, the class instance can be then called to encrypt and decrypt the provided plaintext and ciphertext. In the process of decryption, there is created AES256-CBC key from the derived key and the random 16bit nonce. After this, the encryptor/decryptor can be created and text can be encrypted/decrypted. Note that in the encryption process, the 16-bit nonce has to be added into the generated ciphertext in order to reconstruct the same AES symmetric key later on, see figure 4.3. And similarly, in decryption this nonce has to be extracted from the ciphertext in order to get the plaintext back, visualized in figure 4.4.

```
def encrypt(self, plaintext):
    """Encrypts the plaintext (bytes) with the generated key."""
    cipher = Cipher(algorithms.AES(self.key), modes.CBC(os.urandom(16)),
        backend=self.backend)

    encryptor = cipher.encryptor()
    padder = padding.PKCS7(128).padder()
    padded_data = padder.update(plaintext) + padder.finalize()
    # Initialization Vector is required for security purposes in AES.CBC
    iv = cipher.mode.initialization_vector
    final_ciphertext = encryptor.update(padded_data) + encryptor.finalize() + iv
    return final_ciphertext
```

Fig. 4.3: PBKDF2 encryption function used to encrypt Kyber Master Key

```
def decrypt(self, ciphertext):
    """Decrypts the ciphertext with the generated key."""
    # Extract Initialization Vector
    ciphertext, extracted_iv = ciphertext[:-16], ciphertext[-16:]
    cipher = Cipher(algorithms.AES(self.key), modes.CBC(extracted_iv),
        backend=self.backend)

    decryptor = cipher.decryptor()
    unpadder = padding.PKCS7(128).unpadder()
    padded_plaintext = decryptor.update(ciphertext) + decryptor.finalize()
    return unpadder.update(padded_plaintext) + unpadder.finalize()
```

Fig. 4.4: PBKDF2 decryption function used to decrypt Kyber Master Key

KyberUserUtils

KyberUserUtils.py provides an implementation for operations for usage, creation, and manipulations of User Key Pairs. Each Nextcloud user in the system that sends a file for archivation will have his own Kyber1024 key pair created and stored on the archiving system. To protect these keys and ensure the attackers can not use them for decryption, all of the user keys are encrypted with the Master Private Key and AES256. The class contains methods for the generation of the keys, encapsulation of shared secrets, decapsulation of ciphertext, saving the keys, loading the keys, encryption, and decryption.

The class utilizes the previously mentioned Liboqs library, specifically it is KeyEncapsulation module to provide basic functionality for Kyber1024 keys. Base functions with Kyber keys can be seen in the code snippet below 4.5. Note that Liboqs function `self.kem.generate_keypair()` generates just a public key and in order to generate a private key one has to call `self.kem.export_secret_key()`. The naming seems confusing, but this is how Kyber KEM is in Liboqs currently implemented. Encapsulate and decapsulate secret are functions that provide keying material for the creation of AES256-GCM symmetric key used to encrypt files stored in the system.

When encapsulate_secret is called, it provides a unique shared secret and ciphertext that can be later decapsulated with decapsulate_secret function, recovering the unique shared secret. This way we can simulate key exchange between archivation system in time of archiving and archivation system in the future when archived data are requested. This provides post quantum safe long term confidentiality of the system while also encrypting data with much more effective AES algorithm.

```
def generate_user_keys(self):
    public_key, private_key = self.generate_public_key(),
        self.generate_private_key()
    return public_key, private_key

def generate_public_key(self):
    # generates public key
    return self.kem.generate_keypair()

def generate_private_key(self):
    # generates private key
    return self.kem.export_secret_key()

def encapsulate_secret(self, public_key):
    self.ciphertext, self.shared_secret = self.kem.encap_secret(public_key)
    return self.ciphertext, self.shared_secret

def decapsulate_secret(self, private_key):
    self.secret = self.kem.decap_secret(private_key)
    return self.secret
```

Fig. 4.5: KyberUserUtils key pair generation

The function encrypt defined within the KyberUserUtils class provides an encryption mechanism, integrating post-quantum cryptography principles with symmetric encryption techniques to secure plaintext data, in this case, the whole files. Initially, the function retrieves the public key associated with a specified owner, emphasizing the use of a non-private key to maintain confidentiality. The cryptographic process begins with the generation of a KEM (Key Encapsulation Mechanism) ciphertext and a shared secret using the Kyber algorithm, a candidate in the NIST post-quantum cryptography project known for its security against quantum computer attacks.

Following the key encapsulation, the function utilizes the AES-GCM (Advanced Encryption Standard in Galois/Counter Mode) symmetric encryption algorithm to secure the plaintext. It generates a random nonce (number used once) to ensure the uniqueness of the encryption process in each instance, thus preventing replay attacks. The plaintext is then encrypted using AES-GCM with the previously generated shared secret, ensuring data confidentiality and integrity. Finally, the function concatenates the KEM ciphertext, the nonce, and the encrypted data, which it then encodes in base64 format to produce a string output suitable for safe storage. This method effectively layers the robustness of post-quantum secure key encapsulation with the efficiency and reliability of symmetric encryption, leading to the creation of a hybrid encryption approach suitable for the safeguarding of sensitive data in a potentially quantum-computational future.

```
def encrypt(self, plaintext, owner):
    public_key = self.load_key(owner, private_key=False)
    kem_ciphertext, shared_secret_enc = self.encapsulate_secret(public_key)
    self.ciphertext_length = len(kem_ciphertext)
    aesgcm = AESGCM(shared_secret_enc)
    nonce = os.urandom(12)  # Generate a nonce
    encrypted_data = aesgcm.encrypt(nonce, plaintext, None)
    # Combine KEM ciphertext, nonce, and AES-GCM encrypted data then encode
    full_ciphertext = kem_ciphertext + nonce + encrypted_data
    encrypted_text = base64.b64encode(full_ciphertext).decode('utf-8')
    return encrypted_text
```

Fig. 4.6: KyberUserUtils encryption function

The decrypt function in the KyberUserUtils class is designed to reverse the encryption process outlined in the encrypt method, enabling the recovery of plaintext

from encrypted data. This function initiates by loading and decrypting the private key associated with the specified owner, ensuring that only the intended recipient with the correct credentials can decrypt the data. It underscores the importance of key security by leveraging the master utility's decryption functionality, subsequently logging the key length for verification and auditing purposes.

The core decryption workflow begins with decoding the base64-encoded input, encrypted_text, to retrieve the full binary ciphertext. This ciphertext is dissected into three parts: the KEM ciphertext, a nonce, and the AES-GCM ciphertext, each extracted based on predefined byte lengths. The KEM ciphertext is processed using the private key to decapsulate and retrieve the shared secret. This secret is then employed to initialize an AES-GCM instance, which is used to decrypt the AES-GCM ciphertext using the extracted nonce, ensuring both the confidentiality and integrity of the data. The function ultimately returns the decrypted plaintext, effectively completing the secure communication cycle established by its encryption counterpart. This method exemplifies a robust approach to decryption that complements the advanced encryption techniques, maintaining a high security standard necessary in a post-quantum cryptographic environment.

```
def decrypt(self, encrypted_text, owner):
    private_key = self.load_key(owner, private_key=True)
    private_key = self.master_utils.decrypt(private_key)
    full_ciphertext = base64.b64decode(encrypted_text)
    kem_ciphertext = full_ciphertext[:self.ciphertext_length]
    nonce = full_ciphertext[self.ciphertext_length:self.ciphertext_length + 12]
    # Decapsulate to get the shared secret
    aes_ciphertext = full_ciphertext[self.ciphertext_length + 12:]
    shared_secret_dec = self.decapsulate_secret(kem_ciphertext, private_key)
    aesgcm = AESGCM(shared_secret_dec)
    decrypted_data = aesgcm.decrypt(nonce, aes_ciphertext, None)
    return decrypted_data
```

Fig. 4.7: KyberUserUtils decryption

KyberMasterUtils

KyberMasterUtils.py class is similar to KyberUserUtils.py, however, its intention is only to work with Master Key Pair and its protection. The functionality for Kyber keys and how AES keys are created and used are very similar. However, the class introduces some changes and also utilizes PBKDF2Utils. This subsection will therefore not go through the code again. The reader should just note that the private key in the class is encrypted with PBKDF2Utils before it is saved and similarly it is decrypted after being loaded and used. Therefore, decryption of the files in the system is not possible without the knowledge of the parameters and password used in KDF.

4.1.2 Block Schemes of Processes

To further clarify how the encryption and decryption processes in the archiving system work, this section introduces and describes multiple block schemes. To better understand block schemes, the mathematical descriptions will be added to describe certain parts of block schemes to enhance comprehensibility. Following is a list of mathematical descriptions of operations used in this subsection.

- P_{file}: Plaintext data to be encrypted.
- $KEM_{encapsulate}$: Key encapsulation mechanism using the public key to generate random shared secret and ciphertext.
- $KEM_{Decapsulate}$: Key decapsulation mechanism using the private key to recover shared secret from ciphertext.
- C_{KEM}: Ciphertext generated by the key encapsulation mechanism.
- SS_{KEM}: Shared secret generated by KEM.
- $AES_{generate}$: AES in Galois/Counter Mode. Taking shared secret and nonce to generate a symmetric AES key.
- $AES_{encrypt}$: Encryption operation with AES key
- $AES_{decrypt}$: Decryption operation with AES key
- $PBKDF2_{generate}$: Creation of AES-CBC key with PBKDF2 and password
- $PBKDF2_{encrypt}$: Encryption operation AES from PBKDF2
- $PBKDF2_{decrypt}$: Decryption operation AES from PBKDF2
- N: Nonce, a randomly generated number for AES-GCM.
- C_{file}: Encrypted file contents

Encrypting files

The block diagram in figure 4.8 illustrates a multi-step process for secure file encryption using public-key and symmetric-key cryptography. Initially, a public key for a user identified as Kyber1024 is either loaded or created. Using this public key, a shared secret and a ciphertext are encapsulated, meaning the shared secret is securely established using the kyber public key. Following this, an AES256-GCM symmetric key is generated from the shared secret and a random nonce, they are used to encrypt the file securely. Finally, the ciphertext and the nonce used in the encryption are appended to the beginning of the encrypted file, ensuring that all of the necessary components for decryption are stored together. This process leverages both the security of post-quantum safe public-key cryptography and the symmetric key encryption for performance.

Mathematical description of the encryption process omitting first and last block is following:

1. $C_{KEM}, SS_{KEM} = KEM_{encapsulate}(UserPublicKey)$
2. $AES_{generate}(SS_{KEM}, N)$
3. $C_{file} = AES_{encrypt}(P_{file})$

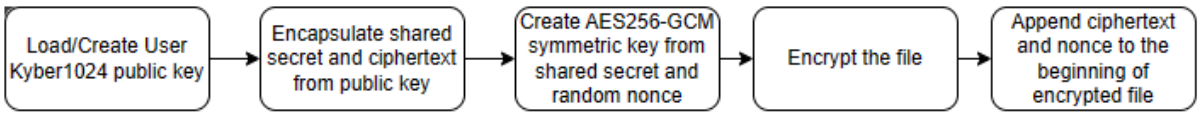

Fig. 4.8: Block scheme of file encryption process

Decrypting Files

The block diagram in figure 4.9 illustrates a decryption process similar to the diagram before it uses a combination of Kyber and AES. Initially, the Kyber1024 private key of a user is loaded. Then, the ciphertext and nonce are extracted from the start of the encrypted file, which were appended during the encryption phase. Using the Kyber1024 private key, the shared secret is retrieved from the ciphertext through a decapsulation function. This shared secret, along with the nonce, is then used to reconstruct the AES256-GCM symmetric key. Finally, this symmetric key decrypts the file, restoring the original data. This process effectively reverses the encryption sequence, allowing for the secure retrieval of the encrypted information. A mathematical description of the decryption process, omitting loading of the key and extraction of ciphertext and nonce is this:

1. $SS_{KEM} = KEM_{decapsulate}(UserPrivateKey, C_{KEM})$
2. $AES_{generate}(SS_{KEM}, N)$
3. $P_{file} = AES_{decrypt}(C_{file})$

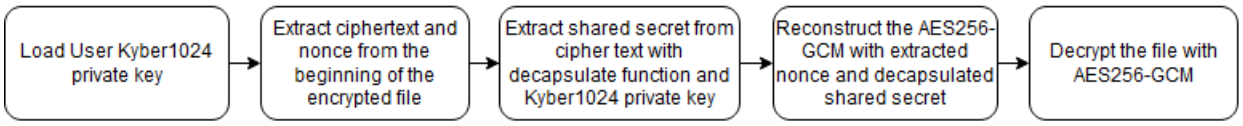

Fig. 4.9: Block scheme of file decryption process

Overall Process of Encryption

The next block scheme in figure 4.10 outlines a whole process for securely archiving a file through a REST API using cryptographic methods previously described. Upon the file's arrival to the archiving system, it first verifies the file owner. Then the check if the owner has an existing Kyber1024 key pair happens. If not, a new Kyber1024 key pair is generated for the owner. If a key pair exists, the file is encrypted with an AES256-GCM key, a detailed process of encryption as described in the previous section, see figure 4.8. In a case, where the owner's key pair does not exist yet, the key pair is automatically generated and the user's private key is encrypted using an AES256-GCM key derived from the Kyber1024 public key. After encrypting the file or the private key as necessary, a hash of the encrypted file along with other required operations for archiving is created. The process finishes with the completion of the archiving task.

Encrypt file with AES256-GCM derived from UserKyber1024 public key in mathematical description means this:

1. $C_{KEM}, SS_{KEM} = KEM_{encapsulate}(UserPublicKey)$
2. $AES_{generate}(SS_{KEM}, N)$
3. $C_{file} = AES_{encrypt}(P_{file})$

Similarly Encrypt the User Private Key with AES-256-GCM key derived from Kyber1024 public key, in mathematical description:

1. $C_{KEM}, SS_{KEM} = KEM_{encapsulate}(MasterPublicKey)$
2. $AES_{generate}(SS_{KEM}, N)$
3. $C_{UserPrivateKey} = AES_{encrypt}(P_{UserPrivateKey})$

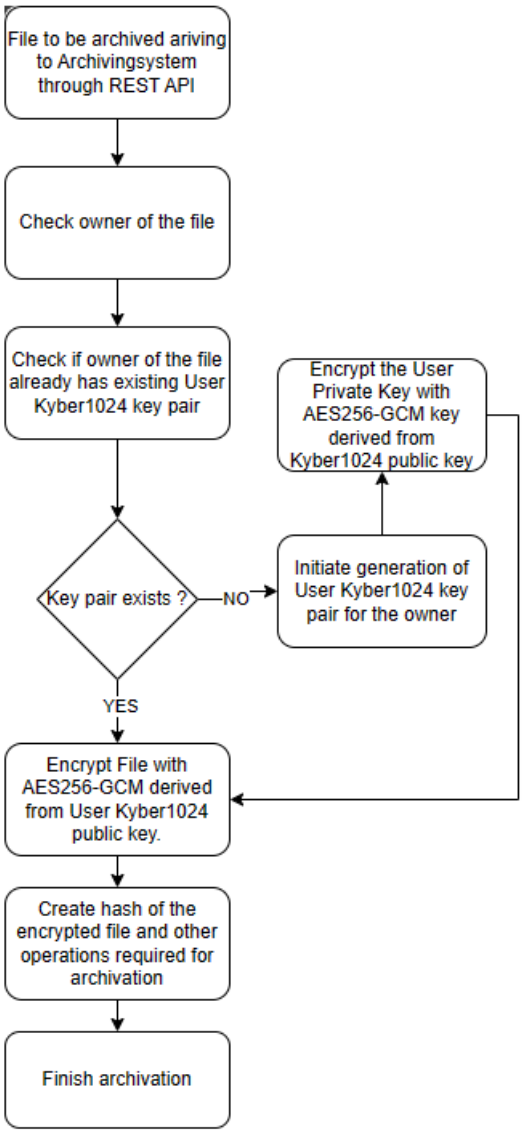

Fig. 4.10: Block scheme of entire encryption process in the system

Overall Process of Decryption

The last block scheme in figure 4.11 describes the overall process of decrypting a file archived with previously described cryptographic methods. The decryption starts with a request via REST API or CLI. Initially, the User Kyber1024 private key is loaded, right after that also Master Kyber1024 private key is loaded. When both private keys are in memory, the master key is decrypted using a PBKDF2 key, reconstructed from a twelve-word mnemonic. Following this, the AES256-GCM key that was initially used to encrypt the user's private key can be reconstructed. This key is then used to decrypt the encrypted user's private key in memory. Finally, an AES256-GCM key protecting the archived file is also reconstructed and this key is

used to decrypt the archived file, securely restoring and returning the original data to the owner.

Mathematical description for all of the cryptographic operations in the process would look like this:

1. $PBKDF2_{generate}(Password)$
2. $P_{MasterPrivateKey} = PBKDF2_{decrypt}(C_{MasterPrivateKey})$
3. $SS_{MASTER-KEM} = KEM_{decapsulate}(P_{MasterPrivateKey}, C_{USER-PRIVATE-KEY-KEM})$
4. $AES_{generate}(SS_{MASTER-KEM}, N)$
5. $P_{UserPrivateKey} = AES_{decrypt}(C_{UserPrivateKey})$
6. $SS_{USER-KEM} = KEM_{decapsulate}(P_{UserPrivateKey}, C_{KEM})$
7. $AES_{generate}(SS_{USER-KEM}, N)$
8. $P_{file} = AES_{decrypt}(C_{file})$

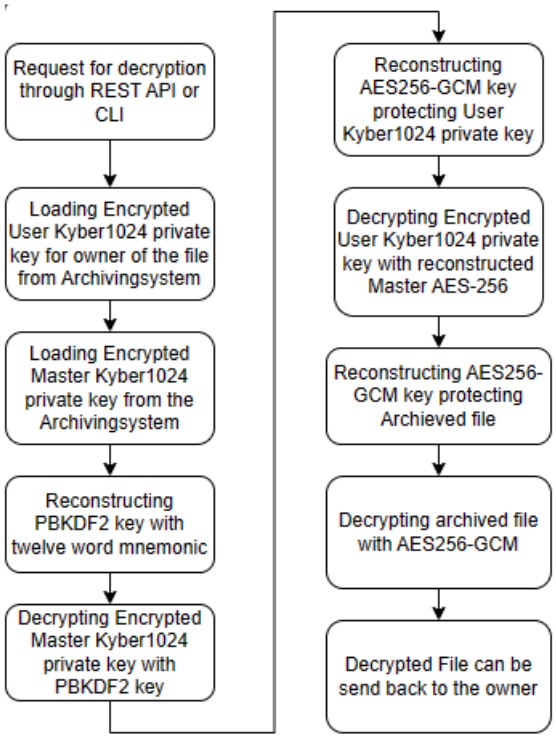

Fig. 4.11: Block scheme of entire decryption process in the system

Command Line Interface

The project solution also provides and extends a command line interface that allows one to manually do cryptographic operations. It is intended for usage by archiving system administrator however, in this thesis, it was also used to make the testing and bench-marking more effective. To display a help message, which describes in detail available commands, the administrator can use the command `archivingsystem -h`. In order for CLI to be available administrator has to be in the folder `/opt/archivingsystem/`.

There are 4 newly added commands, which provide the option to decrypt files based on fileID and owner and to decrypt files based on filePath and owner. Similarly, there are also commands to encrypt files based on fileID and the owner or to encrypt file based on filePath and the owner. Examples of commands look like this:

1. `archivingsystem -e {fileID} -o {owner}`: Encrypt file based on fileID and owner
2. `archivingsystem -ef {filePath} -o {owner}`: Encrypt file based on filePath and owner
3. `archivingsystem -d {fileID} -o {owner}`: Decrypt file based on fileID and owner
4. `archivingsystem -df {filePath} -o {owner}`: Decrypt file based on filePath and owner

New Archiving System Architecture Scheme

The previous architecture was extended by cryptographic operations, which help to provide long-term confidentiality of archived files in the system. This is done with functions implemented in the cryptography module, described earlier in this chapter. The main differences from the previous system are encrypting, re-encrypting, and key storage blocks. The new architecture of the system is illustrated in the figure on the next page 4.12.

1. `Encrypting:` Extends archivation process with processes described above to provide confidentiality of archived data in the archiving system.
2. `Re-Encrypting:` Extends re-timestamping process with processes of re-encryption and key rotation of the AES key protecting the file.
3. `Keystorage:` Specifically marked space in the filesystem reserved for all of the keys necessary for the functioning of the key management system. All private keys are protected by encryption.

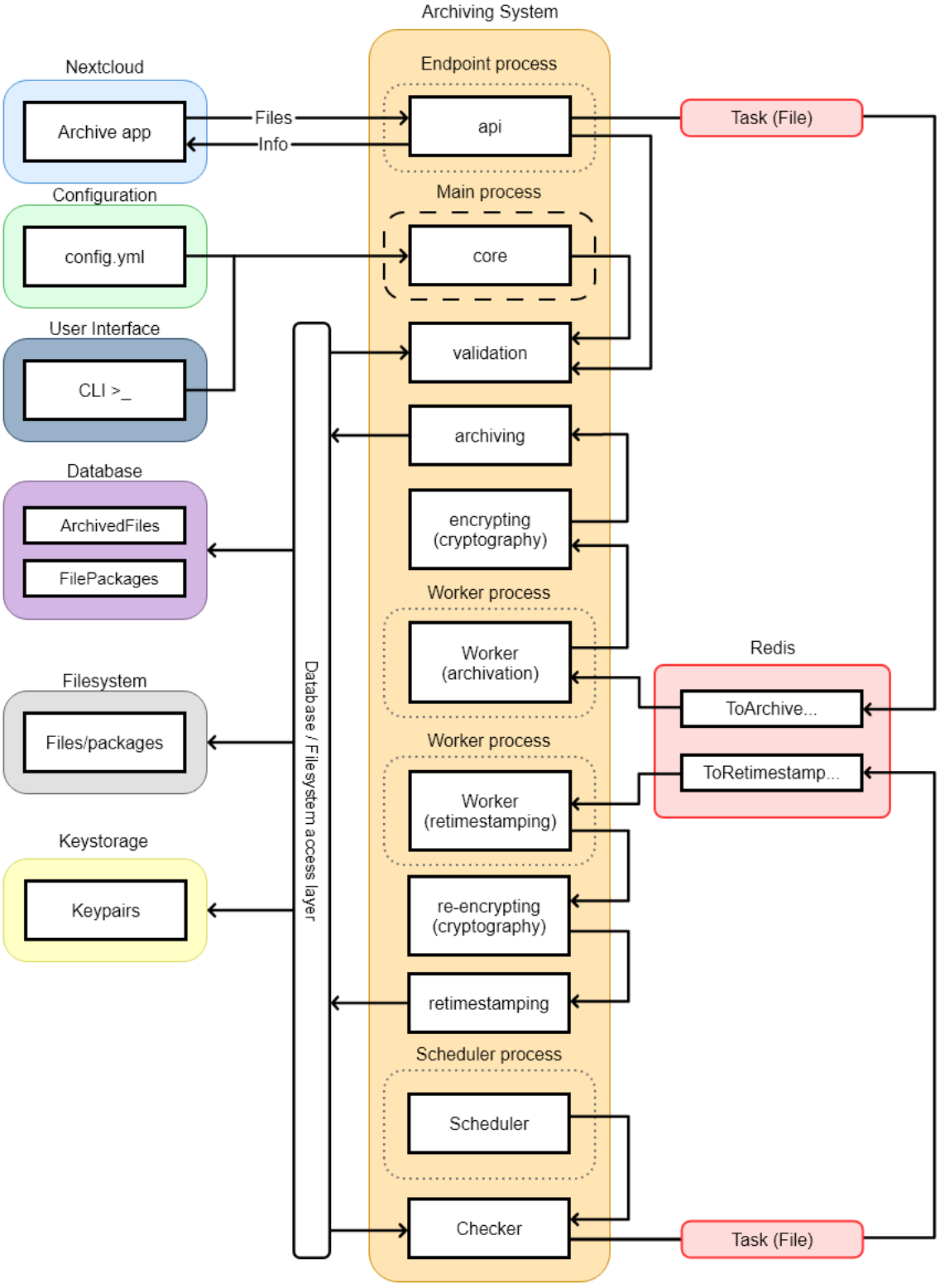

Fig. 4.12: Extended version of archiving system architecture previously mentioned in chapter 3.1 from [23]

4.2 Performance Benchmarking

This section is focused on providing some benchmark values, showing the reader how long the various cryptographic operations in the system take. The benchmark will be done on files of various sizes ranging from 5MB to 10GB.

Machine Used To Benchmark

The whole project was developed and bench-marked on Lenovo ThinkPad X1 Carbon Gen 9. Specs of the laptop can be seen in table 4.1. As all of the cryptographic processes happen on the archiving system, times could be improved by providing better hardware for the bare metal hosting the server, but at the same time, if the system gets deployed in the real scenario with multiple users, the use case should count of multiple operations happening at the same time and should be stress tested appropriately.

Table 4.1: Lenovo ThinkPad X1 Carbon Gen11th Specifications

Component	**Specification**
Processor	11th Gen Intel Core i7
Processor frequency	3 GHz
Processor cores	4
RAM	32GB LPDDR4
Storage	256GB PCIe SSD
Graphics Card	Integrated Intel Iris Xe Graphics
Operating System	Fedora 39 Linux

Script and Tools Used to Benchmark

This is the shortened version of one of the bash scripts used for benchmarking using Python CLI. The script runs the encryption and decryption operations 100 times and averages the times required for the operations to get more stable and meaningful results.

```
for ((i = 1; i <= $ITERATIONS; i++)); do
    echo "Iteration $i:"

    # Encrypt file
    command="archivingsystem -ef $file_path -o $owner_name"
    start_time=$(date +%s.%N)
    $command
    end_time=$(date +%s.%N)

    encrypt_duration=$(calculate_time_difference $start_time $end_time)
    encrypt_total_duration=$(awk "BEGIN {print $encrypt_total_duration
            + $encrypt_duration}")

    # Decrypt file
    command="archivingsystem -df $file_path -o $owner_name"
    start_time=$(date +%s.%N)
    $command
    end_time=$(date +%s.%N)

    decrypt_duration=$(calculate_time_difference $start_time $end_time)
    decrypt_total_duration=$(awk "BEGIN {print $decrypt_total_duration
            + $decrypt_duration}")
done
```

Fig. 4.13: Bash script used to benchmark cryptographic operations

Benchmarking Results

For each file size, the encryption and decryption times are quite close, suggesting that the computational overhead for both processes is similar. This indicates that the encryption and decryption processes are similarly efficient, which is often desirable and expected in symmetric encryption schemes like AES combined with post-quantum cryptographic algorithms like Kyber. Additionally, both encryption and decryption times increase approximately linearly with the size of the files. This is expected as larger files contain more data to process. It would be preferable to also test larger files, for example up to 1TB, but this unfortunately is not possible due to the size of the hard disk on the machine used for benchmarking. With growing file sizes the decryption time is getting a bit longer than then encryption time by approximately 8%, see table 4.2.

Table 4.2: Encryption and Decryption times for different file sizes

File Size	5MB	50MB	100MB	250MB	500MB	1GB	2GB	5GB	10GB
Encryption Time (s)	0.05s	0.28s	0.52s	1.42s	2.60s	5.11s	9.81s	23.87s	45.79s
Decryption Time (s)	0.05s	0.30s	0.54s	1.69s	2.68s	5.27s	10.23s	25.14s	48.54s

The results illustrate the practical performance of combining Kyber, a post-quantum cryptography algorithm, with AES. For typical usage scenarios involving small to medium-sized files, the times are quite reasonable, suggesting good practicality. For very large files, the time taken is still within a manageable range, though optimizations might be needed in case the application expects higher amounts of simultaneous requests at the time or if the archive system will be used to archive file sizes in TB ranges. Results can also be seen in a form of graph in figure 4.14

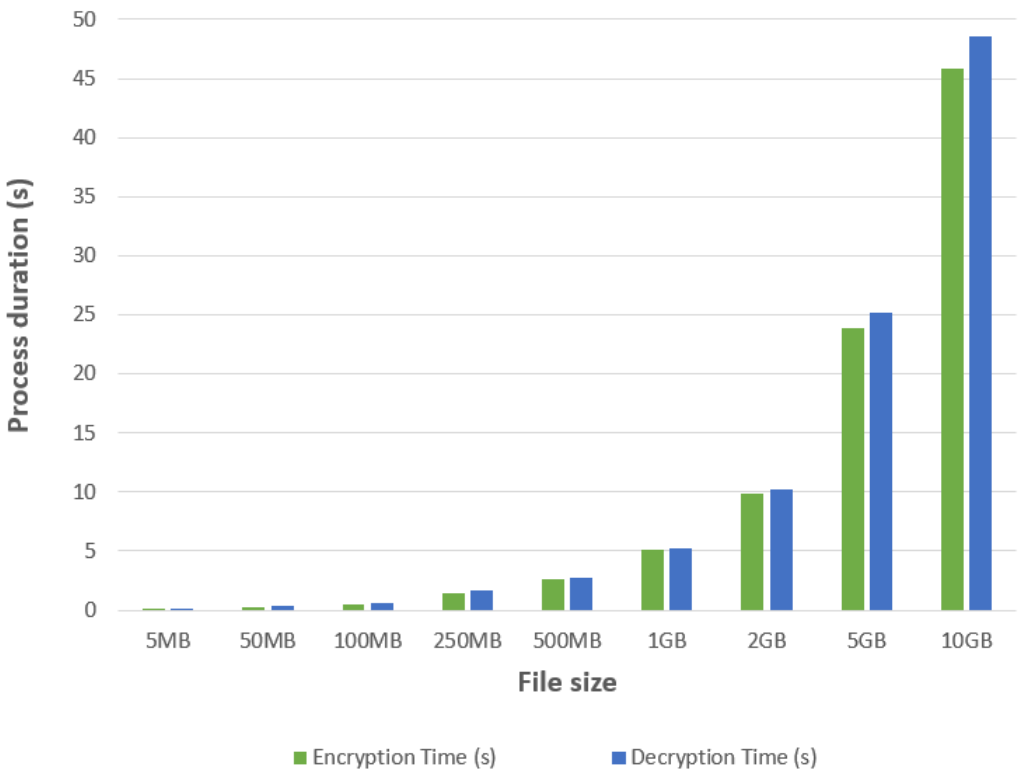

Fig. 4.14: Graph of encryption and decryption times for files of various sized

More detailed breakdown of operations that influence encryption and decryption time:

- **Encryption time** - loading user public key, encapsulating user shared secret, constructing AES key, encrypting file with AES key.
- **Decryption time** - loading master private key, loading user private key, generating PBKDF2 key, decrypting master key with PBKDF2 key, decapsulating master shared secret reconstructing master AES key, decrypting master private key, decapsulating user shared secret, reconstructing user AES key, decrypting file with user AES key.

Every operation from the list except for encryption and decryption file takes constant time regardless of file size. Most of the time spent in the process is connected to those operations and this operation is also the reason for growing times with growing file sizes.

4.3 Future work

As was mentioned in chapters 1.4 and 1.5, the security of the key management system is highly dependent on the security of its master key. Currently, there is a solution in the form of protecting the master key with PBKDF2. The safer and more functional solution, which will enable the archiving system to be more flexible, would be securing the master key through a standalone HSM (Hardware Security Module) which could store the master key and use it for cryptographic operations, for example through REST API.

Another beneficial improvement for the future would be to implement a stronger mechanism for the authentication of users through OAuth2 or OIDC protocols. As of now, there exists authentication in the Nextcloud instance. However, this does not extend to the archiving system.|

Last and probably the most important recommendation, is to create a safer environment while working with encrypted private keys in the memory of the server. The archiving system would ideally work with the keys only within a certain part of memory with additional safeguards and try to ensure the plaintext keys are completely removed from memory once they are no longer needed. This could be done by, for example, overwriting the memory with zeros.

Conclusion

The thesis delved into the foundational recommendations provided by the National Institute of Standards and Technology (NIST) concerning the Key Management lifecycle. It explained the essential terms and principles necessary for comprehending the secure and efficient execution of key management practices. The end of chapter one focused on exploring key management options tailored for cloud solutions, emphasizing the pivotal role of the master key in securing cloud-based storage environments.

Moving on to the second chapter, the thesis conducted an in-depth analysis of the threats posed by quantum computers to current cryptographic protocols. It then delved into the exploration of post-quantum secure alternatives, conducting a comprehensive comparison based on both performance metrics and safety considerations. Ultimately, the thesis identified Crystals-Kyber as the best-performing Key Encapsulation Mechanism (KEM) and Crystals-Dilithium as the most suitable Digital Signature Algorithm (DSA) option. Additionally, the chapter looked into the Advanced Encryption Scheme (AES) and how to combine it with Crystals-Kyber.

In the third chapter, the thesis introduced a novel key management system that utilizes the knowledge acquired from chapters one and two. The current state of the system exhibits robust design principles, ensuring the long-term safety of data at rest, even when confronted by Grover's algorithm. While the system excels in automating data saving and encryption processes, certain aspects still require manual intervention, such as the decryption of archived data. This could be solved by securing the master key with HSM in the future and automating those tasks through REST API between the archiving system and HSM.

Finally, in the fourth chapter, the specifically developed solution was in detail explained and described with block schemes and snippets of Python code. Benchmarking and measuring the performance of the solution was done and analyzed. The thesis was successful in its goals and managed to deliver a functioning key management system, which provides confidentiality of archived files on the archiving system. The results showed that the solution can achieve encryption and decryption in reasonable times tested on files of size up to 10GB. Additionally, the results showed that the encryption and decryption times are pretty similar with decryption operation taking approximately 8% more time.

Bibliography

[1] NIST Computer Security Resource Center. (2001). *Key Management Lifecycle - NIST Computer Security Resource Center*. Retrieved from https://csrc.nist.gov/csrc/media/events/key-management-workshop-2001/documents/lifecycle-slides.pdf

[2] NIST Computer Security Resource Center. (2001). *Key Management Guidelines Overview Notes - NIST Computer Security Resource Center*. Retrieved from https://csrc.nist.gov/CSRC/media/Events/Key-Management-Workshop-2001/documents/guideline-overview-notes.pdf

[3] NIST Computer Security Resource Center. (2001). *Key Management Guideline Workshop - CSRC*. Retrieved from https://csrc.nist.gov/CSRC/media/Events/Key-Management-Workshop-2001/documents/key-management-guideline-workshop.pdf

[4] Peter Mell and Timothy Grance. *The NIST Definition of Cloud Computing*. 2011-09-28. Special Publication (NIST SP), National Institute of Standards and Technology, Gaithersburg, MD. https://doi.org/10.6028/NIST.SP.800-145.

[5] A. R. Buchade and R. Ingle, "Key Management for Cloud Data Storage: Methods and Comparisons," 2014 Fourth International Conference on Advanced Computing & Communication Technologies, Rohtak, India, 2014, pp. 263-270, doi: 10.1109/ACCT.2014.78.

[6] Gorjan Alagic, David Cooper, Quynh Dang, Thinh Dang, John M. Kelsey, Jacob Lichtinger, Yi-Kai Liu, Carl A. Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, Daniel Smith-Tone, and Daniel Apon, *Status Report on the Third Round of the NIST Post-Quantum Cryptography Standardization Process*, *NIST Interagency/Internal Report (NISTIR), National Institute of Standards and Technology, Gaithersburg, MD*, July 5, 2022, https://doi.org/10.6028/NIST.IR.8413, Language: English.

[7] Lukas Malina, Patrik Dobias, Jan Hajny, and Kim-Kwang Raymond Choo, *On Deploying Quantum-Resistant Cybersecurity in Intelligent Infrastructures*, *Proceedings of the 18th International Conference on Availability, Reliability and Security (ARES '23)*, Association for Computing Machinery, New York, NY, USA, 2023, https://doi.org/10.1145/3600160.3605038, doi = 10.1145/3600160.3605038, ISBN: 9798400707728, Booktitle: Proceedings of the

18th International Conference on Availability, Reliability and Security, Article Number: 131, NumPages: 10, Location: Benevento, Italy, Series: ARES '23

[8] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé, *CRYSTALS-Kyber (version 3.02) – Submission to round 3 of the NIST post-quantum project*, Specification document (update from August 2021), August 4, 2021. https://pq-crystals.org/kyber/data/kyber-specification-round3-20210131.pdf,

[9] Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé, *CRYSTALS-Dilithium: A Lattice-Based Digital Signature Scheme*, *IACR Transactions on Cryptographic Hardware and Embedded Systems*, **2018** (1), 238–268, Feb., https://tches.iacr.org/index.php/TCHES/article/view/839, 10.13154/tches.v2018.i1.238-268.

[10] Cankun Zhao, Neng Zhang, Hanning Wang, Bohan Yang, Wenping Zhu, Zhengdong Li, Min Zhu, Shouyi Yin, Shaojun Wei, and Leibo Liu, *A Compact and High-Performance Hardware Architecture for CRYSTALS-Dilithium*, *IACR Transactions on Cryptographic Hardware and Embedded Systems*, **2022**(1), 270–295, Nov., https://tches.iacr.org/index.php/TCHES/article/view/9297, 10.46586/tches.v2022.i1.270-295,

[11] K.V. Pradeep and V. Vijayakumar, "Survey on the Key Management for Securing the Cloud," *Procedia Computer Science*, vol. 50, pp. 115-121, 2015. *Big Data, Cloud and Computing Challenges*, ISSN: 1877-0509, DOI: https://doi.org/10.1016/j.procs.2015.04.072, URL: https://www.sciencedirect.com/science/article/pii/S1877050915005736, keywords: Cloud Security, Cloud Services, Key Management.

[12] Xiaolong Huang and Ruining Chen, "A Survey of Key Management Service in Cloud," in *2018 IEEE 9th International Conference on Software Engineering and Service Science (ICSESS)*, pp. 916-919, 2018. DOI: https://doi.org/10.1109/ICSESS.2018.8663805

[13] Keiko Hashizume, David G. Rosado, Eduardo Fernández-Medina, and Eduardo B. Fernandez, "An analysis of security issues for cloud computing," *Journal of Internet Services and Applications*, vol. 4, no. 1, pp. 5, 2013, Feb. 27. ISSN: 1869-0238, DOI: https://doi.org/10.1186/1869-0238-4-5, URL: https://doi.org/10.1186/1869-0238-4-5

[14] Vlastimil Clupek, Lukas Malina, and Vaclav Zeman, "Secure digital archiving in post-quantum era," in *2015 38th International Conference on Telecommunications and Signal Processing (TSP)*, pp. 622-626, 2015. DOI: https://doi.org/10.1109/TSP.2015.7296338

[15] B. Mishra and D. Jena, *Security of Cloud Storage: A Survey*, in *2019 International Conference on Information Technology (ICIT)*, Bhubaneswar, India, 2019, pp. 109-114, doi: 10.1109/ICIT48102.2019.00026.

[16] Yaser Mansouri, Adel Nadjaran Toosi, and Rajkumar Buyya, *Data Storage Management in Cloud Environments: Taxonomy, Survey, and Future Directions*, *ACM Comput. Surv.*, vol. 50, no. 6, article 91, November 2018, pp. 1-51, issn: 0360-0300, doi: 10.1145/3136623, url: https://doi.org/10.1145/3136623, publisher: Association for Computing Machinery, address: New York, NY, USA, month: December 2017,

[17] Farrukh Shahzad, *State-of-the-art Survey on Cloud Computing Security Challenges, Approaches and Solutions*, *Procedia Computer Science*, vol. 37, pp. 357-362, year: 2014, note: The 5th International Conference on Emerging Ubiquitous Systems and Pervasive Networks (EUSPN-2014)/ The 4th International Conference on Current and Future Trends of Information and Communication Technologies in Healthcare (ICTH 2014)/ Affiliated Workshops, issn: 1877-0509, doi: https://doi.org/10.1016/j.procs.2014.08.053, url: https://www.sciencedirect.com/science/article/pii/S1877050914010187,

[18] D. Kossmann and T. Kraska, *Data Management in the Cloud: Promises, State-of-the-art, and Open Questions*, *Datenbank-Spektrum*, vol. 10, no. 3, pp. 121–129, Dec. 1, 2010. https://doi.org/10.1007/s13222-010-0033-3

[19] Galinina, Olga and Andreev, Sergey and Balandin, Sergey and Koucheryavy, Yevgeni Internet of Things, Smart Spaces, and Next Generation Networks and Systems 2020 Springer International Publishing Cham 80–94 978-3-030-65729-1

[20] 2020 IEEE International Conference on Problems of Infocommunications. Science and Technology (PIC S&T) 2020 0 515-520 10.1109/PICST51311.2020.9467909

[21] 2019 Blekinge Institute of Technology https://urn.kb.se/resolve?urn=urn:nbn:se:bth-18225

[22] Eric Chovanec, *Řízení přístupu k datům v cloudu*, Bachelor thesis, Brno University of Technology, 2021, Brno, Czech Republic.

[23] Martin Nohava, *Zajištění dlouhodobé integrity dat v cloudových úložištích*, Bachelor thesis, Brno University of Technology, 2023, Brno, Czech Republic.

[24] Stathis Mavrovouniotis and Mick Ganley, *Hardware Security Modules*, in *Secure Smart Embedded Devices, Platforms and Applications*, Springer New York, 2014, pp. 383–405, New York, NY, https://doi.org/10.1007/978-1-4614-7915-4_17.

[25] Hannes Salin and Dennis Fokin, *Mission Impossible: Securing Master Keys*, arXiv preprint arXiv:2107.00580, 2021.

[26] Xiaolong Huang and Ruining Chen, *A Survey of Key Management Service in Cloud*, in *Proceedings of the 2018 IEEE 9th International Conference on Software Engineering and Service Science (ICSESS)*, DOI: 10.1109/ICSESS.2018.8663805, 2018.

[27] Douglas Stebila and Michele Mosca, *Post-quantum Key Exchange for the Internet and the Open Quantum Safe Project*, in *Selected Areas in Cryptography – SAC 2016*, edited by Roberto Avanzi and Howard Heys, pages 14–37, Springer International Publishing, Cham, 2017, ISBN 978-3-319-69453-5.

[28] Panos Kampanakis, Matt Campagna, Eric Crocket, Adam Petcher, and Shay Gueron, *Practical Challenges with AES-GCM and the need for a new cipher*, presented at the *Third NIST Workshop on Block Cipher Modes of Operation 2023*, March 2024.

[29] Sandeep Kumar Rao, Dindayal Mahto, Dilip Kumar Yadav, and Danish Ali Khan, *The AES-256 Cryptosystem Resists Quantum Attacks*, in *International Journal of Advanced Research in Computer Science*, vol. 8, no. 3, pp. 404–408, 2017.

[30] *Performance comparison of AES-GCM-SIV and AES-GCM algorithms for authenticated encryption on FPGA platforms* 2017 51st Asilomar Conference on Signals, Systems, and Computers 2017 1331–1336 10.1109/ACSSC.2017.8335570

[31] *Using Crystals Kyber KEM for Hybrid Encryption with Java*, 2023 https://medium.com/@hwupathum/using-crystals-kyber-kem-for-hybrid-encryption-with-java-0ab6c70d41fc

5 Content of the electronic attachment

The electronic attachment consists of files added or modified in the scope of this thesis. The full code can be found at gitlab repository of research team Brno AXE `https://axe.vut.cz`. As the whole project is 204MB large and the maximum for electronic attachment of the thesis is 35MB only newly added files and files modified in order to extend the archiving file by cryptographic operations are included in the attachment. Attachment additionaly contains some examples of encrypted and decrypted files and keys, plus it contains short video demonstrating some functionality from the project. The structure of the electronic attachment is following:

- `/` ..... root folder of the attachment files
  - `archivingsystem` ..... files for archiving system docker container
    - `src/`
      - `PBKDF2Utils.py`
      - `KyberMasterUtils.py`
      - `KyberUserUtils.py`
      - `MyLogger.py`
      - `archiving.py`
      - `retimestamping.py`
      - `api.py`
      - `core.py`
      - `console.py`
    - `benchmark-kyber.sh`
    - `requirements.txt`
    - `setup.py`
  - `examples` ..... examples of encrypted/decrypted files and keys
    - `files`
    - `keys`
    - `video`
  - `docker-compose-nextcloud.yml`
  - `docker-compose.yaml`
  - `Dockerfile.yaml`
  - `README.md`