KU LEUVEN

FACULTY OF
ENGINEERING SCIENCE

Private resources retrieval service for decentralized web ecosystem

Yinqi Liang

Thesis submitted for the degree of
Master of Science in Engineering:
Computer Science, option Secure
Software

Supervisors:
Prof. Dr. ir. Wouter Joosen
Prof. Dr. Bert Lagaisse

Assessors:
Jan Vermaelen
Sam Michiels

Assistant-supervisors:
Ir. Pieter-Jan Vrielynck
Dr. Emad Heydari Beni

Academic year 2022 – 2023

Preface

Spending two years in KU Leuven, I suffered, learned and grew up. KU Leuven develops not only my learning but also my capability to tackle study and work in a foreign country. It also gives me the courage to face difficult tasks in my upcoming life. It makes me realize there is nothing impossible to accomplish and always be humble. I shall remember every single day I spent in Leuven, those saded and depressed, and also joyful and crazy, sometimes feeling a sense of survival and sometimes achievement and success. I was thinking of the meaning of life in these two years though I have not got the answer. But now I got to know myself and knew what I actually wanted in my life better. For these two years, I would like to thank all my close friends who gave me mental support and accompanied me to survive those dark exam seasons. I would like to thank my mother for scarifying so much for me to pursue my study. I would like to thank all the people around me who have given me support for my personal life and study.

The thesis is the finale of the Master's study, and I do suffer a lot from it. First, I would like to thank my supervisor Mr Vrielynck who offered me so much patience and support for my thesis and guided me through nothing to everything. Without him, it would have been impossible to finish the thesis. Second, I would like to thank supervisors Wouter Joosen and Bert Lagaisse who provided me with suggestions to improve my work. Furthermore, without all those friends around me who distracted me from being frustrated with my thesis, I would not have continued my work in a good mental health. Finally, I would like to thank my boyfriend for providing some technical support and helping proofread my text.

Coming from Shenzhen, a city in south China, I would say Leuven is a charming little city.

Yinqi Liang

Contents

Abstract

The Internet exhibits a trend of centralisation as large web platforms, e.g., Google and Amazon, act as central communications points, and each has its user data database. The ongoing trend brings some downsides, including causing massive user data leakage, e.g. Netflix Data Breaches. It also depicts the fact that user data is seldom shared across applications, implying that users lack control over their data. A promising solution to mitigate these problems related to user privacy is the Solid protocol. Solid gives rise to a new concept called data Pods, a decentralised datastore that allows users to store their data and decide where the Pods are hosted. The protocol allows application-related data in the data Pods to be shared across applications, and users retain full control of how their data is handled. However, Solid does not provide any measure against data leakage once the servers hosting data Pods are compromised through attacks, e.g. database hacking. Another issue with web applications is that they might inspect HTTP requests to analyse user behaviour, compromising user privacy. The problem also exists with Solid applications.

The thesis proposes an architecture attempting to solve the mentioned problems of Solid. The solution can be applied to a scenario where users interact with the Solid applications while keeping their files and users' behaviour confidential to any third-party service provider. More specifically, when users download a file from their data Pods via a Solid application, the application gains no information on which file the users request. The method is achieved by encrypting the user request and performing the computation directly on encrypted data, adopting the public key version of the Somewhat Homomorphic Encryption scheme over the Integers. The architecture consists of two applications, the CryptoBox and the Dark Fetcher Solid application. The CryptoBox application provides encryption and decryption services and runs on the users' local machine. The Dark Fetcher Solid application runs on a remote server and provides private file retrieval service from users' data Pods, with no query information leaked to any third party.

The performance evaluation is performed for three critical operations, including encryption, decryption and the computation step to locate the targeted file secretly. The result indicates that the security level strongly influences the processing time of all the operations and that other factors, e.g. numbers of files, exhibit a linear relationship with the time needed for the computation step. It also demonstrated that the current solution is impractical for actual usage because the processing time is extremely high. Requesting a 1-byte file takes more than 3 minutes, even with a low-security level.

List of Figures and Tables

List of Figures

List of Tables

List of Abbreviations and Repositories

Abbreviations

HE	Homomorphic Encryption
FHE	Fully Homomorphic Encryption
SWHE	Somewhat Homomorphic Encryption
PHE	Partially Homomorphic Encryption
GCD	Greatest Common Divisor
LSB	Least Significant Bit
LWE	Learning with errors
RLWE	Ring Learning With Errors
BGV	Brakerski Gentry Vaikuntanathan

List of repositories

CryptoBox	`https://github.com/SeraphinaLiang/Thesis_CryptoBox`
DarkFetcher	`https://github.com/SeraphinaLiang/thesis_app`

Chapter 1

Introduction

1.1 Motivation

Over the past few years, a form of centralisation that becomes more present over the World Wide Web is that communication between all network nodes must pass through one or limited central nodes. Users and small platforms depend on large website platforms, e.g., Google, to manage their tasks or implement services on the internet. The ongoing trend gives rise to many troubles. It not only negatively influences Internet users but also shapes the structure of the Internet platform in a harmful way. The users suffer from three issues [31] due to the trend. First, big companies, e.g. Facebook, Google and Amazon gain control over the internet by holding a large percentage of users' attention. They control the content that could show to the users in the way that the other small companies who want to present their services and products to the users pay these companies for advertising. This way, users can not govern what they can read. Second, the trend shapes the internet infrastructure in a highly central way in that many small companies rely on the services provided by these large platforms, e.g. Amazon. This transition causes the single point of failure problem. In 2017, Amazon's massive outage caused the downtime of thousands of websites [30]. Third, since large platforms keep large amounts of user data, and each of these companies has its copy of user data, the chance of data leakage is high, and thus user privacy is threatened. Moreover, once a data breach occurs, the consequences are considered tremendous.

The last issue also implies that the user lacks control over their data. The users cannot control how other third parties use their data. Besides, the users have no insight into which of their data is held by which parties. Furthermore, the issue depicts the fact that different companies' platforms seldom share user data. The current situation does not comply with the GDPR [22] in that every individual does not have complete control over their personal data. A promising solution to mitigate the problem caused by Internet centralisation is the Solid protocol which enables users to gain full control over their data.

Solid[28] is a protocol that allows users to store their data in a structure called data Pods. The data Pods are hosted on servers of users' choice and give users complete access control of how applications access the data stored in their Pods. It creates a convention in that the application does not store user data, and the same user data could be shared among different parties. As a data-central platform with rich link among its data, the Solid platform has become a new attractive target to attackers who wants to get access to user personal data. However, the Solid platform does not provide any measure to protect against data leakage. If the attackers hack the servers hosting data Pods, all the user data are leaked all at once, since the Solid does not encrypt the stored files in Pods. The thesis solution attempts to address this problem.

Except for the above issues caused by the centralisation of the network, another problem exists with web applications and platforms. Most applications monitor and analyse user behaviour and utilise the behavioural data to improve their system, e.g. recommendation system, to offer a better user experience and compete against similar platforms. The collected data from monitoring user behaviour may compromise user privacy. If a user searches for cancer-related words on Google, the chance that the user is recommended the hospital advertisement is high. The user privacy is compromised in the request since the request content is transparent to the server. The same problem is also with the Solid that third-party service providers that interact with data Pods can inspect the user request without user authorisation. The thesis also attempts to solve this problem by presenting a solution to a demo case based on Solid protocol.

1.2 Problem statement

As a data-rich application, Solid has a problem that it does not have any measures to prevent data leakage. All the user data stored on Solid data Pods is unencrypted. The users' confidentially and privacy are seriously compromised if the server hosting data Pods is being attacked. Proper cryptographic schemes must be applied to the data Pods to protect user data from exposure in case of attacks. The user interacts with Solid data Pods by using Solid applications. When the user requests a file from its data Pod, the Solid application and the server hosting the Pod gain information on which file the user requests. This way, user privacy could be compromised to some extent. The Solid application could analyse user behaviour by inspecting the HTTP request content. A solution must be given to address the problem. The thesis presents an architecture based on Solid protocol that guarantees users' privacy and confidentially not only to user data but also to user behaviour.

1.3 Use case

The usage scenario is based on the Solid protocol but can also be applied to traditional client-server use cases. Suppose users intend to store their files on Solid data Pods and, at the same time, want to ensure the confidentially and integrity of their data.

That is to say, any third party cannot access the content of their file, and it could be detected when their files are illegally modified. Furthermore, the users also want to ensure that their behaviour, e.g. downloading a file, cannot be monitored by any third-party server to ensure their privacy. For example, when users download a file from their data Pods, the Solid application and the Solid server have no information on which files they request.

However, Solid currently does not provide any protection measures on confidentially and integrity to the user data stored in Solid data Pods. Not only could external attackers compromise the server and gain access to the user data, but internal curious service providers could also illegally access the data, which is harder to detect. Furthermore, it is possible that service providers that allow users to store files on their servers generate analysis data by monitoring and analyzing users' behaviour. A behaviour is logged whenever a user downloads and uploads a file.

The thesis system satisfies users' need for data confidentially and integrity and, at the same time, guarantees their privacy. The thesis solution assures that the Solid server and Solid applications gain no information about the file content and only access to limited file metadata. The third-party server blindly processes the file retrieval request of the users. All the user files that go beyond the point of the users' local machine stay confidential, and so does the result of the file retrieval request returning to the users.

1.4 Contributions

The thesis presents an architecture that puts security measures on the existing Solid platform to ensure user privacy and data confidentially. The solution encrypts all user data on the Solid data Pods and allows queries to be performed on the encrypted data. The precise contributions of the thesis are stated as follows.

First, the thesis studies the Homomorphic Encryption scheme, which enables computations to be performed directly on ciphertexts without knowing the secret key. Next, it examines how the scheme can be applied to the existing Solid architecture. Then, an functional and non-functional requirements analysis is conducted, giving rise to the basic architecture of the system.

Second, the detailed architecture of the system is settled down, and the system is implemented. The system consists of two applications. The CryptoBox running on the user's local machine is used for the encryption and decryption of data files. The Dark Fetcher Solid application running on a remote server enables file retrieval requests to be processed blindly by the third-party server.

Third, an performance evaluation is conducted against the thesis solution. The result illustrates how various factors, including the number of files, the identifier length, the average file size and the security level, influence the processing speed of several critical operations. Furthermore, it demonstrates that the current solution is not performant on the private laptop in that it takes several minutes to process a private file retrieval request, even with a low-security level and a tiny file size. Next, the solution is checked against compliance with the non-functional requirements. Finally, some future works to improve the current design are represented.

1.5 Structure

The structure of the thesis is presented as follows. First, chapter 2 provides some background information on Solid protocol and different Homomorphic Encryption schemes, focusing on introducing the Integer-based Somewhat Homomorphic Encryption scheme adopted in the thesis. Furthermore, it also mentioned several related works about private requests for videos and private database services as an inspiration for the thesis. Next, a high-level overview of the system design is presented in chapter 3. First, motivation is given, and an analysis of functional and non-functional requirements is conducted. A general system design is then proposed based on the requirements. Subsequently, chapter 4 introduces detailed system architecture design by describing the functionality of each module and then presents the system component design. Finally, sequence diagrams are shown to explain how users could interact with the system. The underlying mathematics reasoning of the thesis is explained in chapter 5. It also represents the procedures of processing cryptographic messages within a single request sent by the client. Next, chapter 6 first gives the system's setup. A performance evaluation is then performed to investigate how various factors influence the processing speed of the system. Furthermore, the compliance of the solution to the non-functional requirements is inspected. Finally, chapter 7 concludes the thesis and proposes future work to enhance the current solution.

Chapter 2

Background and related work

This chapter will give some background information to understand the rest of the thesis and review the related work. First, Section 2.1 gives an introduction to the Solid protocol. Second, Section 2.2 introduces Homomorphic Encryption. Finally, Section 2.3 shows the related work on the real-life applications.

2.1 The Solid protocol

The goal

Centralized Web application platforms rely on their protocol and APIs to handle user data and perform access control. It causes problems for users that they cannot fully control their data between different platforms. It also creates troubles for developers because they are confined to the existing data access APIs the platforms provided and have difficulties developing cross-platform applications. Solid[28], deriving from 'social linked data,' aims to build a decentralized platform for social Web applications based on Linked Data principles to solve the problems that centralized Web platforms bring to users.

Solid overview

In Solid[19][12], each user owns one or multiple data Pods, also called personal online datastores, that store user data and are deployed on private or public servers. All data files in user data Pods have a unique identifier and are organized by a structure called containers or directories, making it possible that every file can be identified through an unique URL as shown in figure 2.1. Users can choose and easily switch around different Pod providers or Pod servers with different degrees of privacy, reliability, and legal protection since Pod servers work transparently with Solid applications. The figure 2.2 illustrates the interaction between the user, the Solid server and the Solid application. The Solid applications include a wide variety, from file managers managing files in Pods to social applications discovering friends. Solid applications interact with the user data Pods according to the Solid protocols. The Solid protocols are based on W3C standards[3] to perform CURD operations and manage access control of the content on data Pods. Users have complete access control to their data in the Pods and can decide which Solid applications have to right to access their data at any time. When a user registers, a unique WebID is linked to that user for future authentication and locating resources purposes. Solid adopts decentralized authentication in that the user does not authenticate to the application directly but instead redirects the authentication request to Solid Identity Provider, which takes charge of the authentication of users for Pods, and login with the user's unique WebID linked to user data Pods. In this way, the Solid architecture ensures that users have full access control of their data used by the Solid applications, decoupling content from the application itself. Furthermore, application developers can reuse existing data created by other applications.

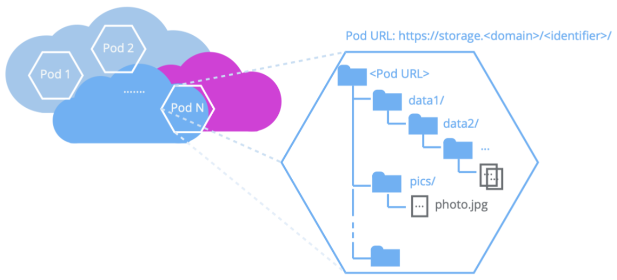


Figure 2.1: A Pod server hosting data Pods[19]

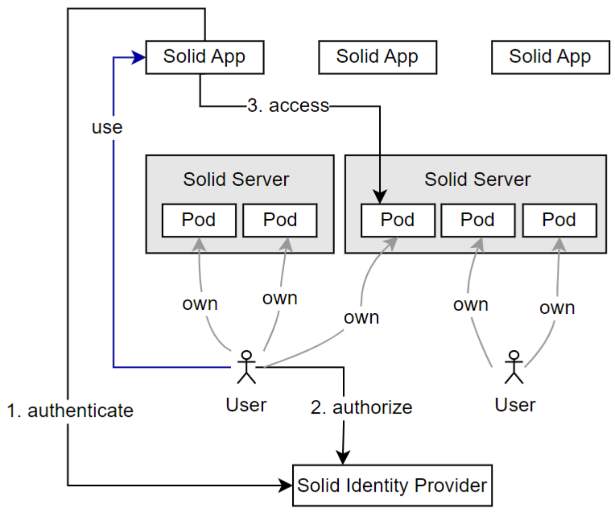


Figure 2.2: Interactions between users, servers and applications of the Solid ecosystem

2.2 Homomorphic Encryption (HE)

This section first introduces the concept of homomorphism, then presents the definition and properties of Homomorphic Encryption, together with its algorithms and allowed operations, and finally gives rise to three types of Homomorphic Encryption schemes.

Homomorphism

Homomorphisms[4] are the mapping relation between algebraic objects. A homomorphism between two algebraic objects A and B is a function $f : A \rightarrow B$, which maintains the algebraic structure of A and B. In other words, the algebraic equations, which include addition and multiplication, that elements in A satisfy, will also be satisfied by the images of the elements in B. The definitions of homomorphism rely on the algebraic structures of A and B in different contexts. To give an example, if the operations on A and B are both additions, the homomorphic condition is $f(a + b) = f(a) + f(b)$. If A and B are rings with addition and multiplication, the multiplication condition is $f(ab) = f(a)f(b)$ as an extra condition.

Definition and property

Homomorphic Encryption (HE)[1][15] is an encryption scheme that enables computations to be performed on encrypted data or ciphertexts without knowing the secret key. At the same time, the features of the function and the format of the encrypted data remain unchanged, meaning that the decryption of the resulting ciphertext is precisely the same as the result of computations performed directly over the plaintext.

Algorithms

A Homomorphic Encryption scheme[15][9] is a four-tuple algorithm. "

> **KeyGen** Given as input a security parameter λ, outputs a public and private key pair (pk, sk). The plaintext space $\mathcal{P}$ is related to pk, and the ciphertext space $\mathcal{C}$ is related to sk.
>
> **Enc** Given as input the public key pk and a plaintext $m \in \mathcal{P}$, outputs ciphertext $c \in \mathcal{C}$.
>
> **Dec** Given as input the private key sk and a ciphertext $c \in \mathcal{C}$, outputs a plaintext $m \in \mathcal{P}$ if the given sk is correct.
>
> **Eval** Given as input the public key pk , n ciphertexts $c_1, c_2, ..., c_n \in \mathcal{C}$ and a permitted function F, outputs $F(c_1, c_2, ..., c_n) \in \mathcal{C}$.
>
> The correctness is defined as $\mathbf{Dec}(\mathbf{Eval}(F, c_i, pk), sk) = F(m_i)$, where ciphertext $c_i = \{c_1, c_2, ..., c_n\}$ and the corresponding plaintext $m_i = \{m_1, m_2, ..., m_n\}$.

"

Allowed operations

Two types of mathematical operations[29] can be performed on ciphertext, namely addition and multiplication. In Addition Homomorphism, the encryption of the addition of plaintexts is equal to the addition of the encryption of plaintexts, shown as $Enc(m1 + m2) = Enc(m1) + Enc(m2)$, where the $Enc()$ denotes the encryption function. This property also applies to Multiplication Homomorphism that the encryption of the multiplication of plaintexts is equal to the multiplication of the encryption of plaintexts, shown as $Enc(m1 * m2) = Enc(m1) * Enc(m2)$.

Three types of Homomorphic Encryption

The Homomorphic Encryption scheme is divided into three categories, including Partially Homomorphic Encryption, Somewhat Homomorphic Encryption and Fully Homomorphic Encryption, according to the number of allowed operations performed on the encrypted data[1]. Partially Homomorphic Encryption (PHE) enables only one type of operation with unlimited usage. Somewhat Homomorphic Encryption (SWHE) allows both addition and multiplication but only with a limited number of times since its computations produce noise inhibiting the scheme's function. It can be transformed into Fully Homomorphic Encryption by applying some specific techniques. Fully Homomorphic Encryption (FHE) allows an unlimited number of operations at an unlimited number of times.

2.2.1 Somewhat Homomorphic Encryption Scheme over Integers

Gentry et al.[32] in 2020 first described a simple Somewhat Homomorphic Encryption scheme (SWHE) that adopts only basic arithmetic computations, including addition and multiplications, on a single-bit value. The original scheme is based on symmetric key encryption and can be easily converted into a public key encryption-based scheme. The authors reduce the security assumption of the SWHE scheme to finding an approximate integer greatest common divisor (GCD). The SWHE scheme functions correctly if the noise generated along the computation is below a certain level. The scheme is further converted into a fully homomorphic scheme by applying squash decryption and ciphertext refresh techniques to reduce the generated noise, which will be further explained in section 2.2.2.

Symmetric key version

Gentry et al. present a simple Somewhat Homomorphic Encryption Scheme[32] based on a shared secret key. The key generation, encryption, and decryption algorithms[23] which operate on a one-bit value are shown as follows. Let $\lambda \in \mathbb{N}$ be the security parameter, set integers $N = \lambda$, $P = \lambda^2$ and $Q = \lambda^5$, representing the bit length of secret components.

KeyGen(λ) The secret key p is an odd integer randomly chose from a P-bit integer.

Encrypt(p, m) The encrypted value of a bit $m \in \{0, 1\}$ is $c = pq + 2r + m$, where q, r are Q-bit and N-bit integers respectively.

Decrypt(c, p) The decrypted bit of the bit c is $m = (c \mod p) \mod 2$.

The scheme has the following properties[17]. First, the ciphertext is close to a multiple of p. And second, m is the least significant bit (LSB) of the distance to nearest multiple of p.

Security assumption : Approximate GCD problem

The approximate GCD problem[24] with parameters (γ, η, ρ) is the challenge of recovering the secret integer p given arbitrarily many samples in the form of $x_i = pq_i + r_i$, where integer p is η bits, the noise terms r_i's are sampled uniformly from the integer interval $[-2^\rho + 1, 2^\rho - 1] \cap \mathbb{Z}$, and the terms q_i's are sampled uniformly from $[0, 2^{\gamma-\eta}] \cap \mathbb{Z}$. It is proven in the paper[32] that the scheme is semantically secure under the Approximate GCD assumption. In other words, the security problem of the scheme can be reduced to the Approximate GCD problem. If Approximate GCD problem is hard, then the scheme is semantically secure[23].

Public key version

A weakly homomorphic encryption scheme[26] means that if homomorphically generated encryptions correctly decrypt, their lengths depend only on the security parameter and the plaintext length. According to Rothblum[27], any private-key encryption scheme that is weakly homomorphic for addition modulo two can be transformed into a public-key encryption scheme. The methodology[32][23] of converting the symmetric scheme mentioned above into a public key encryption scheme is as follows. The public key is a randomly chosen sum from a predefined set S of encryptions of zeros. The set S is defined as $S = \{2r_1 + pq_1, 2r_2 + pq_2, ..., 2r_n + pq_n\}$ where q_i, r_i are chosen from the same intervals as mentioned above and q is the symmetric secret key. Then to encrypt a bit m, the ciphertext is defined as $c = m + \sum_{i=0}^{\sharp T}(2r_i + pq_i)$ where $T \subseteq S$ is a subset of randomly selected elements from S. The complete scheme[10] is described as follows.

The scheme parameters[10] “Given the security parameter λ, the following parameters are used:

- γ is the bit-length of the x_i's.
- η is the bit-length of symmetric secret key p.
- ρ is the bit-length of the noise r_i.
- τ is the number of x_i's in the public key.
- ρ' is a secondary noise parameter used for encryption.”

Notation The notation is the same as in the original paper[10]. “ For a real number x, the notation $\lceil x \rceil$, $\lfloor x \rfloor$, and $\lceil x \rfloor$ are denoted as the rounding of x up, down, and to the nearest integer. For integers z and p , the reduction of z modulo p is denoted as $[z]_p$ with $-p/2 < [z]_p \leq p/2$. ”

The scheme is based on a set of public integers $x_i = pq_i + r_i$, where $0 \leq i \leq \tau$ and the integer p is the symmetric secret key[10]. “ For a specific η-bit odd integer p, the distribution over γ-bit integers is defined as $\mathcal{D}_{\gamma,\rho}(p) = \{x = q \cdot p + r \mid q \leftarrow \mathbb{Z} \cap [0, 2^\gamma/p), r \leftarrow \mathbb{Z} \cap (-2^\rho, 2^\rho)\}$.

> **KeyGen**(1^λ)
> Generate a random odd integer p of size η bits. For $0 \leq i \leq \tau$ sample $x_i \leftarrow \mathcal{D}_{\gamma,\rho}(p)$. Relabel so that x_0 is the largest. Restart unless x_0 is odd and $[x_0]_p$ is even. Let public key be $pk \leftarrow (x_0, x_1, ..., x_\tau)$ and secret key be $sk \leftarrow p$.
>
> **Encrypt**$(pk, m \in \{0,1\})$
> To encrypt a bit plaintext m, choose a random subset $S \subseteq \{1, 2, ..., \tau\}$ and a random integer r in $(-2^{\rho'}, 2^{\rho'})$, output the ciphertext as $c = [m + 2r + 2\sum_{i \in S} x_i]_{x0}$
>
> **Decrypt**(sk, c)
> To decrypt a bit ciphertext c, output $m \leftarrow (c \mod sk) \mod 2$.
>
> **Evaluate**$(pk, C, c_1, ..., c_t)$
> Given the circuit C with t input bits, and t ciphertexts c_i, apply the addition and multiplication gates of C to the ciphertexts, performing all the additions and multiplications over the integers, and return the resulting integer.

”

Noise component

The term $2r + m$ is the noise component[17] of the ciphertext. The noise grows along the computation process in that the addition operation doubles the noise, and the multiplication operation squares the noise. When the noise is sufficiently smaller than the secret key p, or precisely, smaller than $p/2$ in absolute value, the scheme works correctly due to its homomorphic property.

2.2.2 Fully Homomorphic Encryption schemes

Different types of Fully Homomorphic Encryption schemes[8] are derived from three main base problems: Learning with errors (LWE), Lattices based, and Integers based. The Learning With Errors problem[25] is to recover a secret $s \in \mathbb{Z}$ given a sequence of ‘approximate’ random linear equations on s. Lattices problem is a class of optimization problem on lattices which represent partially order set relationships. The Integers based problem is based on the Approximate GCD assumption mentioned above. The figure 2.3 shows how different schemes related to those base problems. Influential works from the branch of LWE are the Ring Learning With Errors

(RLWE) scheme[6] and the Brakerski Gentry Vaikuntanathan (BGV) scheme[5]. On the lattices-based direction, Gentry proposed an FHE scheme based on ideal lattices in his Ph.D. thesis[15] in 2009, establishing the foundation of Fully Homomorphic Encryption. Later, in 2010, Gentry and Halevi presented the primary system of the lattices-based FHE[16]. Finally, based on the integer arithmetic, the DGHV scheme[11] was proposed by Coron and derived two variants: the DGHV with smaller keys variant[10] and the DGHV with modulus switching variant[11]. The following paragraphs will detail the Gentry's method and the two DGHV variants in handling the noise component mentioned in section 2.2.1.

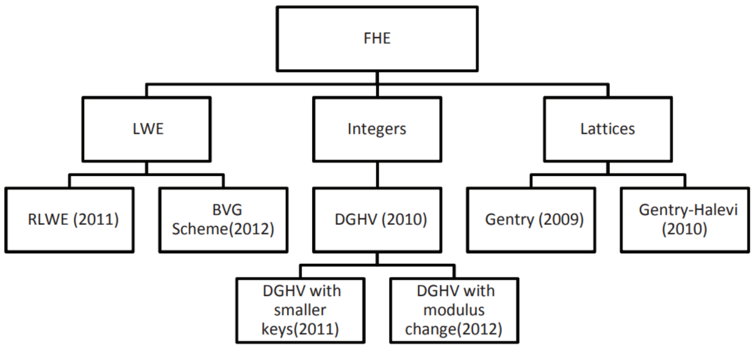


FIGURE 2.3: Problem-based FHE schemes overview[8]

Gentry's technique

In 2009, Gentry invented the first Fully Homomorphic Encryption (FHE) scheme over ideal lattices based on the following approaches[15][10]. First, Gentry proposed a Somewhat Homomorphic Encryption (SWHE) scheme as described in section 2.2.1, which only supports a limited number of operations since any computations on ciphertext increase the noise level, which limits the scheme from working correctly. Based on this scheme, Gentry performs the squashing procedure to the decryption process that enables it to be represented as a low-degree polynomial in which the ciphertext and the secret key are in bits format, making the scheme bootstrappable. Bootstrappability as the prerequisite of the following refresh procedure means that the degree of the polynomial that can be evaluated on ciphertexts is higher than the degree of decryption polynomial times two. The ciphertext refresh procedure is then performed by decrypting the encryption of ciphertext bits using the encryption of the secret-key bits, generating a new encryption of the original plaintext bit with a reduced noise level. The resulting noise in the new ciphertext of the same plaintext is lower than the original ciphertext. This way, the number of additions and multiplications that can be performed on ciphertext becomes unlimited, and the SWHE scheme can be converted into an FHE scheme.

DGHV with Shorter Public Keys

Based on the above technique developed by Gentry in 2009, van Dijk et al. presented an FHE scheme over the integers[32]. The scheme offers conceptual simplicity that all operation is performed on integers than on ideal lattices but at the cost of an overly large public key size impractical for any system. In 2011, Coron et al. presented a technique[10] that significantly reduces the public key size. " The two steps of the technique are first keeping only a small subset of the public key and then generating the whole public key on the fly by combining the elements in the small subset multiplicatively. The scheme is semantically secure under a more potent variant of the approximate GCD assumption." The authors also specify a secure set of concrete parameters[10] by simulating the known attacks and measuring their capability. The concrete parameters can be fixed according to the four desired levels of security, as shown in the figure 2.4. The four security levels are toy, small, medium and large, corresponding to 42, 52, 62, and 72 bits of security, respectively. It has the following constraints[10] in order to avoid specific attacks, e.g. brute force attacks, and enable the scheme to work correctly.

- $\rho = \omega(\log \lambda)$
- $\gamma = \omega(\eta^2 \cdot \log \lambda)$
- $\tau \geq \gamma + \omega(\log \lambda)$
- $\rho' = \rho + \omega(\log \lambda)$

The parameters in figure 2.4 and the constraints are consistent with the parameters in the public key version of the SWHE scheme as described in the section 2.2.1, since the FHE version is based on the SWHE version.

Parameters	λ	ρ	η	γ	β
Toy	42	16	1088	$1.6 \cdot 10^5$	12
Small	52	24	1632	$0.86 \cdot 10^6$	23
Medium	62	32	2176	$4.2 \cdot 10^6$	44
Large	72	39	2652	$19 \cdot 10^6$	88

Figure 2.4: Secure parameters set[10]

DGHV with Public Key Compression and Modulus Switching

Brakerski and Vaikuntanathan proposed a scheme established on the Learning with Errors (LWE) and Ring Learning with Errors (RLWE) problems. The scheme presents a new way to reduce dimension and to perform the modulus switching technique, shortening the ciphertext and reducing the decryption complexity[11]. In a recent paper, a new FHE framework[5] in which the noise ceiling grows only linearly with the multiplicative level indicates that bootstrapping is not necessarily required to achieve FHE. Based on the above findings, a new framework[11] of FHE has been invented. The new FHE system adopts the new modulus switching technique that reduces noise by efficiently transforming a ciphertext c under a modulus p into a ciphertext c' under a different modulus p'.

2.3 Related work

2.3.1 Private video streaming service

Contributions

Youssef Gahi et al. proposed an architecture[14] to achieve a fully secure and private video retrieval service. The system allows users to request a video from a service provider and the provider has no information about which video the user request for. At the same time, the video provider protects its resources by only allowing users to access authorized resources. The authors adopt the DGHV scheme mentioned in section 2.2.1 in its architectural design and point out that the security parameter λ is highly related to the system performance. Latter, based on the previous findings, Yacine Ichibane et al. discovered the association[18] between the security parameter λ and the maximum number of videos that could be requested successfully without the bootstrapping procedure.

Implementations

The secure video retrieval services[14][18] work as follows. The design assumes that the service provider or the server has n videos provided to the users. Every video has a unique identifier encoded on s bits.

1. **Identifier encryption** The user first requests a list of identifiers of the n videos and chooses an identifier ID. The user then encrypts the selected identifier ID bit by bit using the formula $Enc(ID_i) = ID_i + 2r + pk * q$ according to the public key version of DGHV scheme, where the ID_i is the i^{th} bit of the identifier, r and q are random generated integers and pk is the public key.

2. **Localizers calculation** The encrypted sequence $Enc(ID)$ is then sent to the server to be further processed. The server performs an exhaustive search comparing the encrypted sequence to all the existing video identifiers on the list to retrieve the target video since the server has no information on the encrypted sequence. The server compares bit by bit the encrypted sequence $Enc(ID)$ to each plaintext identifier v in the store, calculating the localizer I_v for each video using the formula $I_v = \prod_{i=0}^{s-1}((1 \oplus v_i) + Enc(ID_i))$, where v_i is the i^{th} bit of the identifier in the store, $Enc(ID_i)$ is the i^{th} bit of the encrypted sequence and $\oplus$ denotes addition modulo two. I_v is equals to $Enc(1)$ if and only if the identifier v is identical to the user selected identifier ID, or $Enc(0)$ otherwise. This will be further explained in chapter 5. The localizers

are computed for each video identifier on the server, generating a sequence of I_v in the form of one $Enc(1)$ and $(n-1)$ $Enc(0)$.

3. **Encrypted video stream computation** Finally, to compute the encrypted video stream R, every video V in the store is multiplying to its corresponding localizer I_v using formula $R = \sum_{i=0}^{n-1} I_{v,i} \times V$. In the equation, the target video stream is multiplied by $Enc(1)$, while other video streams become $Enc(0)$. Then the encrypted target video stream R is obtained by adding all the multiplication results bit by bit. The encrypted stream flow R is then sent back to the user and can be decrypted using the user's secret key.

Using the methods above, the server or the service provider also ensures its protected resources would not be illegally accessed since the server fully controls which video is involved in the resulted stream calculation.

Further discoveries

The first paper[14] points out that the length of the security parameter λ significantly impacts the system performance and that the security parameter λ should be carefully chosen to find a balance between the security level and the system's response delay. For a video of size 1 Megabyte and 10 bits identifier, the system processing time is 2 seconds with λ set to 3 and sharply rises to 12 seconds with λ set to 6. The second paper[18], based on the Somewhat Homomorphic Encryption scheme, discovers the relationship between the security parameter λ and the maximum number of videos n that could be processed. Under the maximum number n, the resulting noise is lower than the secret key's value, and thus the decryption works correctly without the bootstrapping step. The sufficient condition to keep the noise strictly lower than the value of secret key sk is $n \leq 2^{\lambda-2}$, where n is the number of the videos and λ is the security parameter. The equation indicates that n grows exponentially with the security parameter λ, with 128 videos when λ is set to 9 and up to 1024 videos when λ is set to 12.

2.3.2 Secure database queries

Original work

Y. Gahi et al. propose a solution[13] that the user can make secret queries to the database server and that the server acquires no information about the user's queries. The solution utilizes the DGHV Fully Homomorphic Encryption scheme with which the bootstrapping step is enabled. The adopted method is similar to the method in private video retrieval services, except that the identifiers in the database to be compared are in ciphertexts form, encrypted by the DGHV scheme. The comparison is operated bit by bit on two ciphertext identifiers while still generating the same outcome with $Enc(1)$ if equals or $Enc(0)$ otherwise. Also, all the files stored in the database are in ciphertext form, encrypted using the corresponding client's public key under the DGHV scheme.

Improvement

Latter, H. Usef et al. advance the previous identifiers comparison method[13], significantly reducing computation workload. Instead of comparing identifiers bit by bit, the authors develop a new approach[23] based on the Ring Based Fully Homomorphic Encryption scheme[7], allowing comparing blocks of data, e.g., integers, all at once rather than on a single bit.

Chapter 3

System design and overview

The chapter illustrates how the overall system design is modeled based on the result of requirement analysis. Section 3.1 introduces the motivation regarding the current situation and later briefly describes the solution to the existing problems. Section 3.2 analyzes the functional and non-functional requirements of the solution. Finally, section 3.3 derives the basic system architecture from the outcome of requirement analysis.

3.1 Motivation for system design

The concept of Solid arose to solve the current situation of decentralized user data storage owned by different applications. By collecting user data into a structure called Pods, users gain full access and control of their data, referring to section 2.1 for detail explanation. However, Solid currently does not provide any protection measures on confidentially and integrity to the user data stored in Solid data Pods. In the setting of Solid, the data leakage problem could be more severe than the original scenario once the Solid server that hosts the user data Pods is compromised. In the centralized scenario, all data in the data Pods related to a specific user is exposed, making the Solid server a more attractive target for the attackers. Instead of only partial information leakage in the decentralized scenario, this situation also leads to more attacks regarding the rich link between the leaked data. If an application, e.g., Netflix, is compromised, the leaked data only includes application-related user data and some of the user's personal and preferences data. While in the case of data Pods leakage, personal data used by all applications are exposed. Not only could external attackers compromise the server and gain access to the user data, but internal curious service providers could also illegally access the data, which is harder to detect. Even if the file stays secure and untouched on the service provider server, the file name itself could also compromise user privacy. Some commercial service providers that allow users to store files on their servers generate analysis data by monitoring and analyzing users' behavior. A behavior is logged whenever a user downloads and uploads a file. The generated extensive logging database is a rich resource for analyzing users' personal life.

The solution to the existing problems

The users have the need that not only their file storing on remote servers stay confidential but also the metadata of the file gives no information to the server when they interact with the service providers. The thesis application solves the current problem and satisfies the user's requirements. For confidentiality, the application allows the users to encrypt their files before uploading them to Solid Pods. For integrity, since the files are encrypted, a modification could be easily detected. In the solution settings, only the users have access to the secret key to encrypt and decrypt their files. A slight file change could lead to a decrypt error, and the service providers cannot re-encrypt the files themselves, therefore satisfying the integrity requirement in some aspects. This way, the Solid server and Solid applications gain no information about the file content. Users' activity is also kept secret to the Solid server and Solid applications in the design of the thesis application. The homomorphic encryption scheme is adopted to enable processing user request blindly. Solid File Manager[2] is a Solid application that allows users to manage their files on Solid data Pods. When a user attempts to download a file using Solid File Manager from data Pods, the application and the server that hosts the Pods have no information on the file content and which file has been requested by the user. The user could later decrypt the downloaded encrypted file. This way, the user's privacy has been massively preserved.

3.2 Requirement analysis

The section analyzes the thesis application requirements on functional and non-functional aspects based on the solution to the problem in section 3.1.

3.2.1 Function requirements analysis

The thesis application must fulfill the following functionalities and, at the same time, follow the Solid protocol convention.

- A crypto layer allows the user to encrypt files before uploading them to the Solid server via Solid File Manager and decrypt them after downloading from the server. The layer adopts symmetric or asymmetric cryptography from the scheme of Somewhat Homomorphic Encryption over Integers in section 2.2.1, since the later computation only works with this scheme. The allowed encryption and decryption methods are presented in the mentioned scheme.
- A computation layer allows secure computation to retrieve files on top of the Homomorphic encryption scheme. For simplicity and practicality, the filename is characterized by a n-bit integer file identifier. In other words, the whole calculation process does not reveal the actual value of the selected file identifier. The generated computation result also guarantees that the metadata of the target file, e.g., file length, is kept secret to the Solid server and Solid applications. The only information revealed through the computation includes

the length of the identifier, the length of each file, and the total number of files involved.

3.2.2 Non-functional requirements analysis

Non-functional requirements of the following aspects need to be considered in the overall system design.

Performance System performance or processing speed is crucial for systems that involve cryptographic applications since the primary overhead comes from processing cryptographic information. The generation of key materials and the encryption, decryption and homomorphic operations of gigabytes of bits take several seconds and even minutes to process. For a video size of one Megabyte with an identifier of 10 bits, retrieving the video from the server takes 2 to 6 seconds, with the security parameter varying from 3 to 6, respectively[14]. Many factors affect the speed of crypto function execution, including the choice of programming language, the way of implementing the algorithm, the level of parallelism adopted in the design of the algorithm, and the actual coding, etc. Therefore, for the system components comprising a high ratio of crypto operations, the architecture design should focus highly on system efficiency.

Configurability For configurability, users should be unrestricted to change the system's security level. The security level could be modified in two ways. First, the users could adjust the security parameter set that impacts key materials' generation regarding different security levels. This way, the users could make a tradeoff between the level of security and the processing speed. The users could also decide the length of the file identifier, only in the constraint that all the file identifiers should be at the same length, e.g., all in 8 bits or 16 bits. The system also remains configurable to the developer. Homomorphic encryption is based on symmetric or asymmetric cryptography. The developer could choose whether to adopt the symmetric key or the public key versions to the system design.

Security Encryption strengthens system security in that using a secret key to encrypt the plaintext file ensures the user's confidentially even if the file is leaked. Security is a crucial focus in the system architecture design in that properly managing key materials significantly improves system safety in preventing and imitating various attacks. The storage of key materials should be designed in a way that the keys are only accessible to the related users. The key materials should not be accessed by any third party or service provider, or transmitted through the public internet at all times.

Scalability For horizontal scaling, incorporating more servers to run the application of the computation layer scaled out the system in that more user requests could be processed simultaneously. However, vertical scaling is more crucial to the system design since it determines the speed of handling individual user requests. The system has a high potential for parallel processing since it involves many cryptographic computations. By adding more CPU or GPU units, the capability of parallelism in processing bit value significantly increases. The scaling up of the system considerably shortens the time for individual request processing.

3.3 The system design

The system architecture design is derived from the section Requirement analysis 3.2.

3.3.1 The rise of design decision to the computation layer

Concerning the Solid protocol convention, three design decisions are proposed for the design of the computation layer.

- **Building the computation layer on top of the Solid server** Since the server has no information regarding which file is being requested, the computation must iterate through all the existing files stored inside the data Pod. The considerable file transmission overhead is generated if the computation layer is built elsewhere, for example, on top of the Solid File Manager. All the files in the data Pod must be retrieved first by the Solid File Manager before the computation can start.

- **Adding the computation layer to the Solid File Manager** Instead of directly building the layer on top of the Solid server, the layer could also be added as a plugin to the existing Solid File Manager. The design decision provides more flexibility to users compared to the first decision. They could choose whether to use the plugin or not. However, the system design generated a massive file transmission workload.

- **Implementing the computation layer in terms of a Solid Application** The third design decision gives the most flexibility to users and also the developer. The computation layer is implemented as a separate Solid application that interacts directly with the Solid server. The users still interact with the Solid server and Solid File Manager in the old way. Compared to the two solutions above, the developer possesses higher freedom for implementation since the solution depends on fewer existing libraries and platforms. However, the solution generates a high overhead similar to the second proposal.

The third proposal is chosen for the thesis for some practical reasons after trying out all three design decisions. The first two decisions encounter technical implementation issues, including difficulty in extending functionality on a large existing project and lack of support from the cryptographic and math-related library of a specific programming language used to implement the computation process.

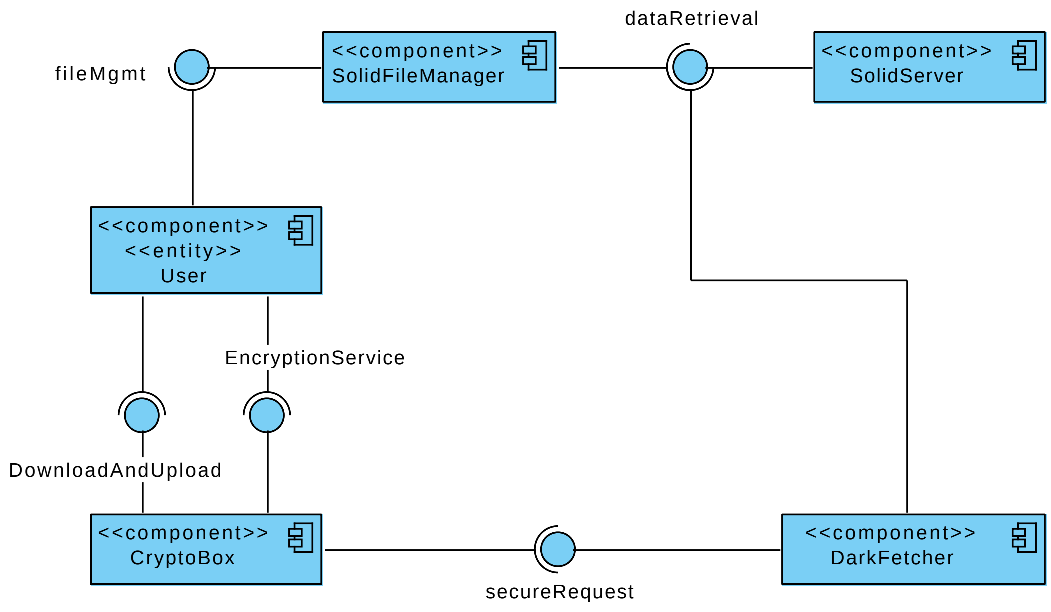
3.3.2 The overview of the system and the provides interfaces

Composition

The interaction between the user and the other three parties, including the thesis application, the Solid server, and the Solid File Manager, is shown in the figure 3.1. The thesis application consists of two components, the CryptoBox and the DarkFetcher Solid App, corresponding to the two layers in the functional requirements 3.2.1, the crypto layer and the computation layer.

Interaction

The user entity can use the DownloadAndUpload interface provided by the CryptoBox component to upload files to the CryptoBox, and then use the EncryptionService interface to encrypt the files. The user entity downloads the encrypted files from the CryptoBox, and uploads them to the SolidFileManager through the fileMgmt interface. The SolidFileManager then stores the files through the dataRetrieval interface to the user's data Pod hosted in the SolidServer. The user entity instructs the CryptoBox to make a secret file retrieval request using the secureRequest interface on the DarkFetcher component. The DarkFetcher first retrieves all the files from the data Pod through the dataRetrival interface on SolidServer, performs computation, and sends back the result. The CryptoBox decrypts the result and presents the requested file to the user entity.



fileMgmt
<<component>> SolidFileManager
dataRetrieval
<<component>> SolidServer
<<component>> <<entity>> User
EncryptionService
DownloadAndUpload
<<component>> CryptoBox
secureRequest
<<component>> DarkFetcher

FIGURE 3.1: System components overview and external applications

3.3.3 The deployment of the system

The deployment of the whole system is presented in figure 3.2. The CryptoBox component is deployed on the user's local machine for security consideration mentioned in the non-functional requirements 3.2.2. The DarkFetcher, SolidFileManager and SolidServer are deployed on different remote machines. However, deploying all three components on the same remote node is also possible. The tasks of the three components are executed separately. Thus the way of their deployment solely depends on the computing resources of the node. The quantity relationship between each node is also defined. The number of CryptoBox components running on a local machine is unconstrained, while there should be at least one SolidFileManager, SolidServer, and DarkFetcher running on remote machines.

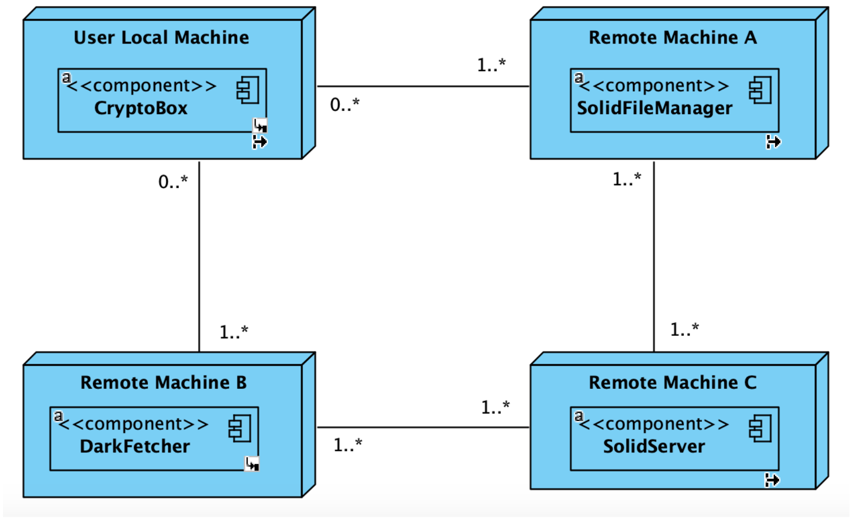


Figure 3.2: System deployment diagram

Chapter 4

System architecture and interaction

The chapter illustrates the detailed functionality and architectural design of the system and specifies the step of using the system from a user perspective. Section 4.1 describes the functionality module design of the CryptoBox and the Dark Fetcher Solid App. Section 4.2 represents the architectural solution to the system. Finally, section 4.3 shows the interaction of the system and users based on the architecture design.

4.1 Overview of functionality design

The section gives a high-level overview of the functionalities design of the system. First, it introduces the functionality module layout inside the CryptoBox component and its internal interaction. Next, the cooperation of each function module of the Dark Fetcher Solid app component is briefly illustrated.

4.1.1 CryptoBox functionalities modules overview

The detailed functionalities design of the CryptoBox component in figure 3.1 is illustrated in figure 4.1. The CryptoBox consists of three layers, the Core layer, the Service layer, and the Local Storage layer. The Core layer provides a basic cryptographic operations library and exposes the internal API calls to the Service layer. The Service layer connects to the Core layer, provides higher-level functionality, and launches API calls to the Dark Fetcher Solid App. The Local Storage layer acts as a local database for the system and the intermediate data transmission channel for the Core and Service layers.

The Core layer

The Core layer consists of three functionalities modules: the SWHE Library, the SWHE Utility, and the Core APIs. The SWHE Utility provides high-level operations built on top of the SWHE Library, a library for Homomorphic encryption. The Core

APIs operates on the SWHE Utility to provide internal API interfaces. Each module is explained in detail as follows.

- **SWHE Library** The SWHE Library provides a library for the public key version of the Somewhat Homomorphic encryption scheme on integers mentioned in section 2.2.1. The SWHE Library originated from an existing DGHV library [20], with the deletion of the bootstrapping step. It allows users to configure security parameters that affect the generation of the public and private key pair in the security-parameters-setting module. Then, the random-number-generation, key-initialization, and key-generation modules are responsible for public and private keys setup and generation. Next, the read-and-write-keys module is responsible for reading old keys from files and writing new keys to the files. In addition, the encryption-and-decryption module provides raw encrypting and decrypting services. The evaluated-add-or-multiple module allows evaluation functions to be applied to ciphertexts. After that, the converting-encoding-and-decoding module provides some helper functions for formatting. Finally, the ciphertext initialization module initializes the target value to a ciphertext format.

- **SWHE Utility** The SWHE Utility provides a higher-level service on top of the SWHE Library. The initialization module allows users to initialize the Core layer by setting up key material from local storage for future encryption and decryption operations. Next, the reset-parameter module enables users to reset the security parameters, generate new key pairs according to the new parameters, and store the new key pair in the local storage. After that, the encryption module encrypts a one-bit value and stores the generated ciphertext to the local storage. The decryption module reads a ciphertext from the local file and decrypts the ciphertext. Finally, the storing-pk0 module stores the first block of the public key to the local storage whenever a new key pair is generated. The stored pk0 is sent to the Dark Fetcher Solid App used in the computation process, where every evaluation step, including addition and multiplication, modulus the pk0 value.

- **Core APIs** The Core APIs acts as an API interface for the Core layer and wraps around the SWHE Utility. The Service layer makes use of the Core layer by sending HTTP requests to the Core APIs interfaces. The interfaces allow users to initialize the Core system, reset the security parameter k, encrypt a one-bit value n, and decrypt a ciphertext block. In addition, the data transmission of the Core APIs and the SWHE Utility is managed through the Local Storage layer.

The Service layer

The Service layer consists of four functionalities modules: the Core Connector, the Tools, the Encryptor, and the Secure Request Sender. The Tools, a utility library, makes use of the Core Connector, which launches internal API calls to the Core layer. The Tools also provides services for the Encryptor, which is used for sequence encryption, and the Secure Request Sender, which sends requests to the Dark Fetcher Solid app. Each module is explained in detail as follows.

- **Core Connector** The Core Connector launches HTTP requests to the Core APIs to use the Core layer service. First, the start and reset-parameter modules correspond to the /init and /reset interfaces that launch the initialization and the reset parameter services request to the Core layer. Next, the encrypt-one and encrypt-zero modules call the /encrypt interface with a parameter. Finally, the decrypt-one-bit module is linked to the /decrypt interface.
- **Tools** The Tools provides services to the Encryptor and the Secure Request modules by calling the Core Connector. First, the encrypt-sequence and the decrypt-sequence modules encrypt and decrypt the given sequence. Next, the decode-file module decodes the UNICODE sequence to characters. Finally, the get-pk0 module reads the first block of the public key from the file and returns it.
- **Encryptor** The Encryptor consists of two modules, the encrypt-identifier module, which encrypts the given plaintext identifier and stores the result in the local storage, and the encrypt-file module, which reads the file and generates an encrypted file.
- **Secure Request Sender** The Secure Request Sender makes secure requests to the Dark Fetcher Solid App by sending the encrypted identifier and the pk0, the first block of the public key.

The Local Storage layer

The Local Storage layer has three responsibilities. First, it stores the security parameter setting and the client's public and private keys. It also functions as an interface for clients to upload and download files. Finally, it is a medium for data exchange for multiple modules and layers.

4.1.2 Dark Fetcher Solid App functionalities modules overview

The high-level overview of the functionality design of the DarkFetcher Solid App component in figure 3.1 is presented in figure 4.2. The APIs Interface of Dark Fetcher receives the HTTP request sent by the CryptoBox from the client's local machine and starts processing the request. First, the Files Retrieval retrieves all the files in the client's data Pods hosting on the remote Solid server. Those files, together with the encrypted identifier and the first block of the public key, are then fed into the Private Computation. The Private Computation is made of a job chain consisting of four modules, Identifier-Processing module, Localizer-Generation module, Target-File-Selection module, and Result-Combination module, where the process result of the last module is fed into the next module. The processing step corresponds to steps 2 to 5 in section 5.2.

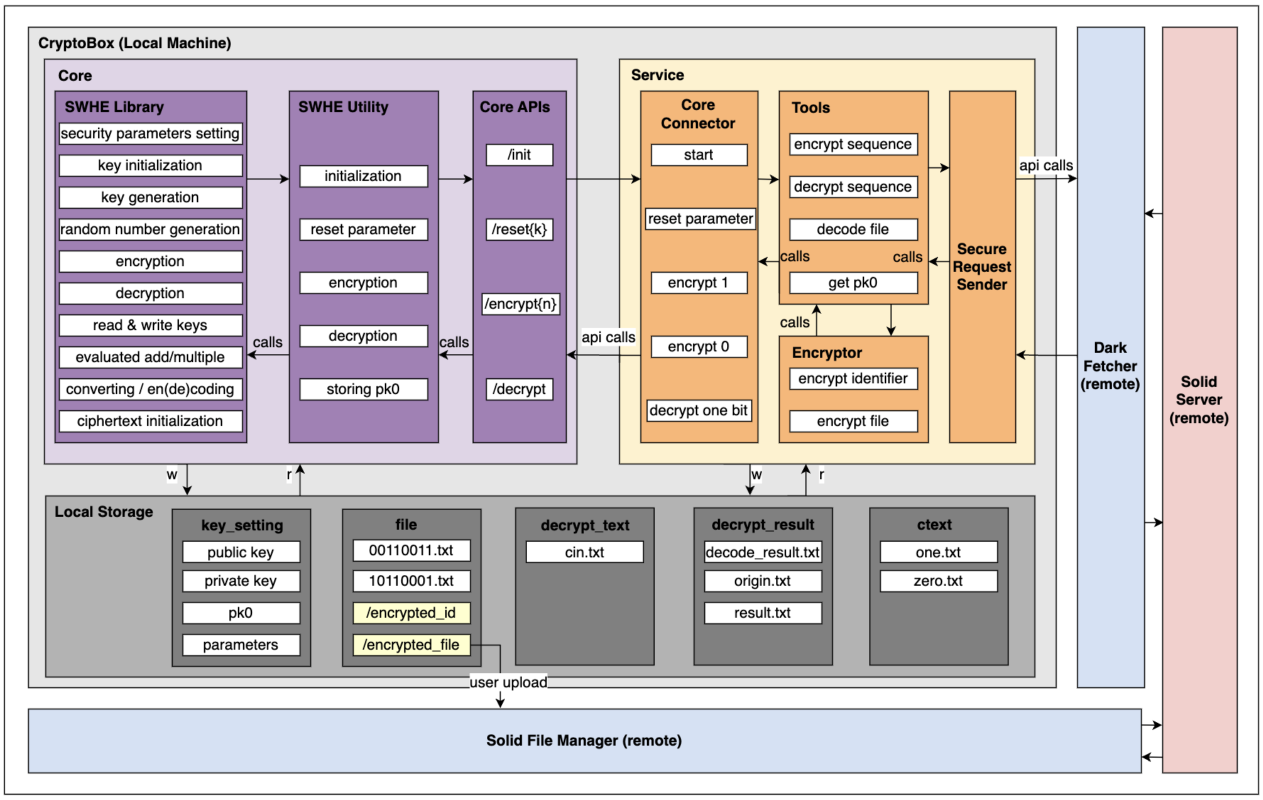


Figure 4.1: CryptoBox function modules

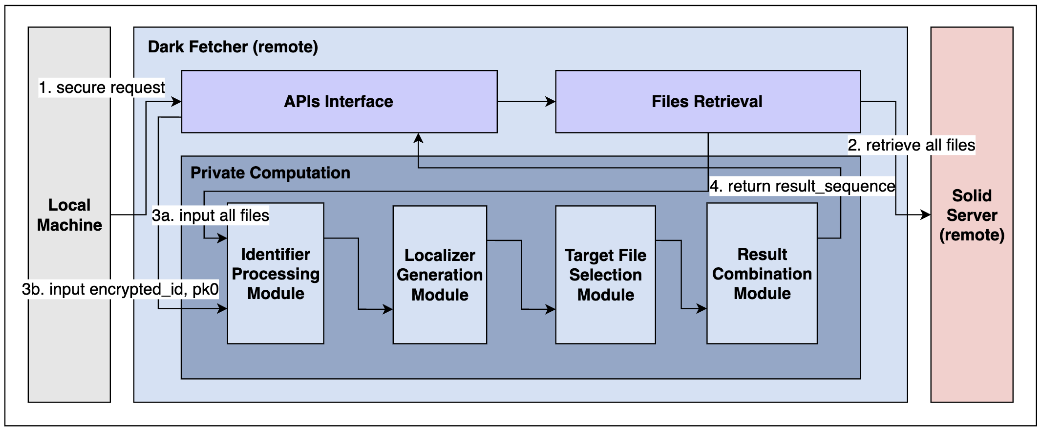

Local Machine
Dark Fetcher (remote)
1. secure request
APIs Interface
Files Retrieval
2. retrieve all files
Private Computation
4. return result_sequence
3a. input all files
3b. input encrypted_id, pk0
Identifier Processing Module
Localizer Generation Module
Target File Selection Module
Result Combination Module
Solid Server (remote)

Figure 4.2: DarkFetcher Solid App function modules

4.2 System components design

The section shows the system architecture design, which derives from the functionalities modules design in section 4.1, by presenting the components diagrams of the CryptoBox and the DarkFetcher Solid App.

4.2.1 CryptoBox components design

As shown in figure 4.3, the CryptoBox component has three subcomponents, the Core component, the Service component, and the LocalStorage component. The Core component is divided into three modules, the SWHE Utility module, the Core APIs module, and the SWHE Library module, as presented in figure 4.4. The Service component is split into four modules, the CoreConnector module, the Tools module, the SecureRequestSender module, and the Encryptor module, as shown in figure 4.5. The functionalities of each component and module are explained in section 4.1.1.

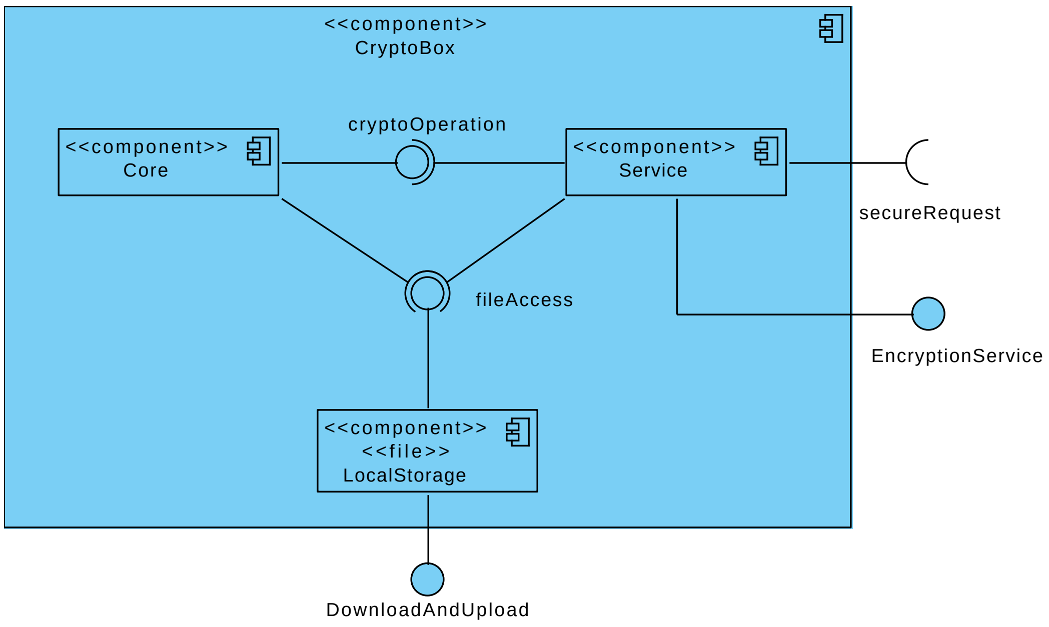


FIGURE 4.3: CryptoBox Component Diagram

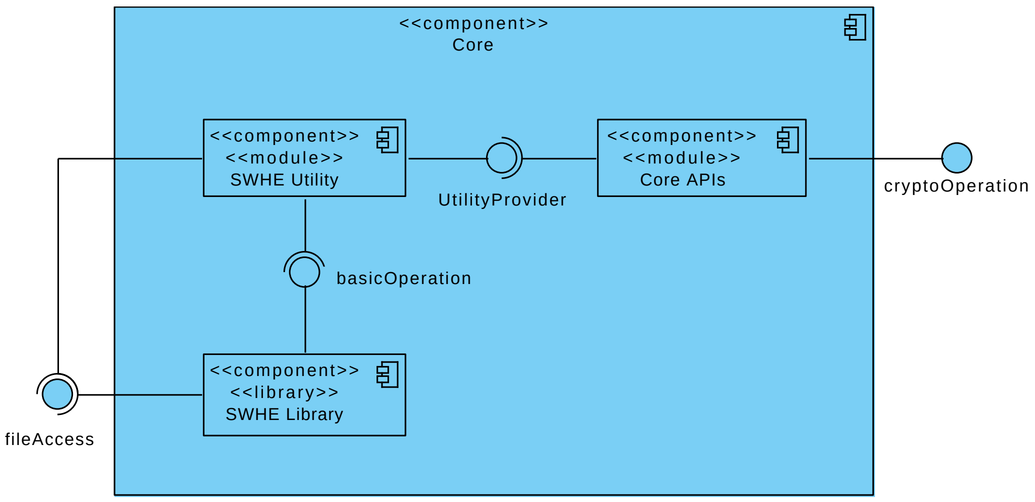


Figure 4.4: Core Component Diagram

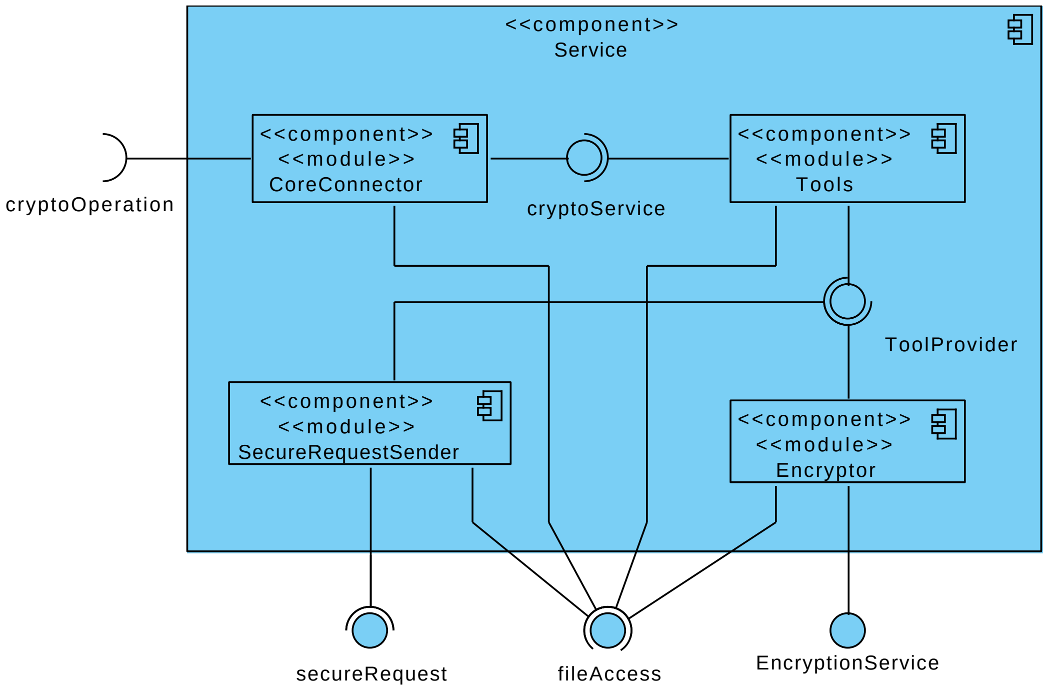


Figure 4.5: Service Component Diagram

4.2.2 DarkFetcher Solid App components design

The DarkFetcher component

As shown in figure 4.6, the Dark Fetcher Solid App component is divided into three subcomponents, the APIs Interface component, the FileRetrieval component, and the PrivateComputation component. The process of handling request by each component is described as follow. First, the client sends an HTTP request through the secureRequest interface. The APIs Interface component retrieves all the files in the client's data Pods through the FileProvider interface, and then the FileRetrieval component downloads all the files through the dataRetrieval interface provided by the SolidServer in figure 3.1. Next, the files and the information in the request are fed together into the PrivateComputation component through the blindComputation interface.

The PrivateComputation component

The PrivateComputation component is split into five modules, the ComputationController module, the IdentifierProcessing module, the LocalizerGeneration module, the TargetFileSelection module, and the ResultCombination module as presented in figure 4.7. The ComputationController acts as a central controller, which gathers the intermediate results of each step, launches the next job to the subsequent modules, and finally combines and returns the result. The other four modules work as a job chain. First, the IdentifierProcessing module performs identifier comparisons. Next, the LocalizerGeneration module generates localizers for each file involved in the computation. After that, the TargetFileSelection module filters out the targeted file requested by the user. Finally, the ResultCombination module combines the computation results from the last step. The detailed process of handling requests by the these four modules is illustrated in section 5.2.

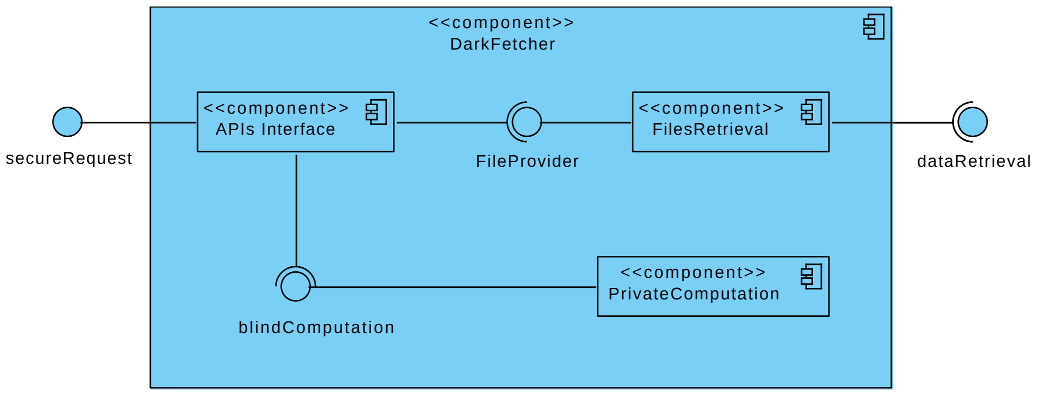


FIGURE 4.6: DarkFetcher Solid App Component Diagram

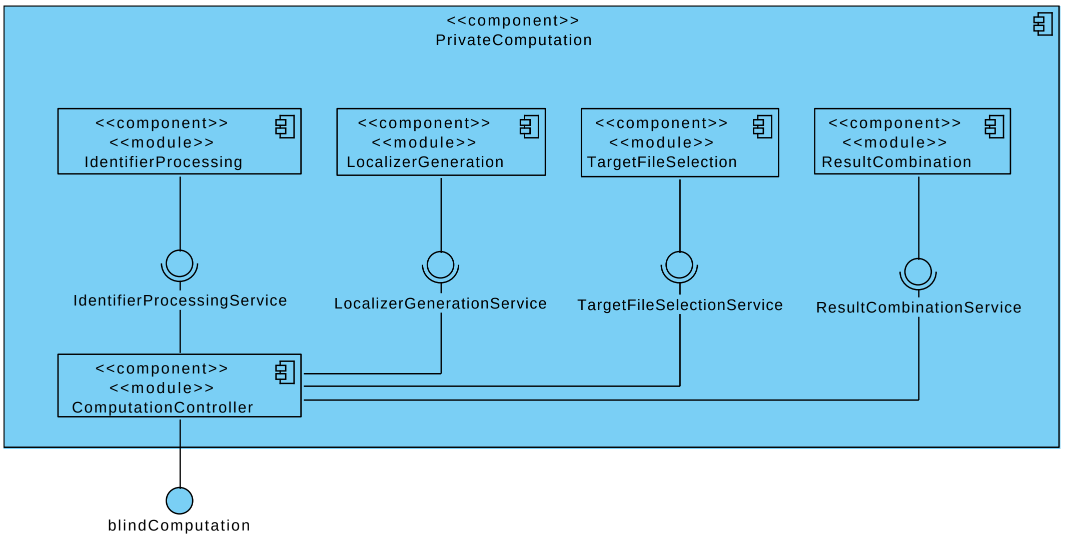

<<component>>
PrivateComputation
<<component>> <<module>> IdentifierProcessing
<<component>> <<module>> LocalizerGeneration
<<component>> <<module>> TargetFileSelection
<<component>> <<module>> ResultCombination
IdentifierProcessingService
LocalizerGenerationService
TargetFileSelectionService
ResultCombinationService
<<component>> <<module>> ComputationController
blindComputation

Figure 4.7: Private Computation Component Diagram

4.3 System interactions

The section represents the interaction between users and the system components, which is based on the architectural design in section 4.2, through sequence diagrams and depicts the proper steps of using the system from a user perspective.

4.3.1 User initializes the system

The process of setup the system from last configuration by the user is demonstrated in the sequence diagram 4.8. The user indicates the system to start the initialization process. A new random state is generated, and the parameter setting and keys are initialized to a proper format with a NULL value. Later, the security parameter setting from the last configuration and the public and private keys are restored from local files.

4.3.2 User modifies security parameters setting

Instead of restoring the system from the last configuration, the user can also choose to configure a new security parameter set. The process of modifying security parameters setting is represented in sequence diagram 4.9. The user indicates a security level, an integer number from 1 to 3, where the larger the number becomes, the bigger the keys' size. The system generates a new random state and then initializes and generates a new parameter set and a new key pair. Next, the new parameter set and the generated key pair are stored locally, overwriting the previous files. The pk0, the first block of the public key, is stored separately in the local storage.

4.3.3 The bit sequence encryption process

The process of encrypting a bit sequence using the client's public key and generating a sequence of ciphertext blocks is represented in sequence diagram 4.10. Each bit is encrypted bit by bit independently by sending a local HTTP request from the Service to the Core. The encrypted result of every bit is different since the public key used by the encryption process varies every time. The resulting ciphertext of each bit is first stored in a local file and then collected by the Service from the local storage. The Tools combines the collected ciphertext blocks and generates the final encrypted sequence.

4.3.4 The file identifier encryption process

Before a user can send a secure request to the Dark Fetcher Solid app, the target file identifier must first be encrypted by the user as shown in sequence diagram 4.11. First, the user provides a plaintext identifier sequence and specifies the length of the identifier. The Encryptor then triggers the Core to perform an encrypted sequence operation as described in section 4.3.3. Next, the Encryptor saves the result to the local storage.

4.3.5 The file encryption process

Before a user can upload an encrypted file to the Solid data Pods, the file needs to be first encrypted by the CryptoBox. The file encryption process is shown in sequence diagram 4.12. The user uploads the file to the Service, the Encryptor in the Service reads the file, encrypts the file sequence as described in section 4.3.3, and saves the encrypted file to the local storage. The user can then download the encrypted file from the CryptoBox.

4.3.6 The process of a user uploading an encrypted file to the Solid data Pod

The procedure of a user storing an encrypted file on a data Pod is described in sequence diagram 4.13. The user encrypts a file using the CryptoBox and then downloads the encrypted file. Next, the user uploads the encrypted file to the data Pod hosting on a Solid server through the Solid File Manager.

4.3.7 The process performing secure computation

The process of a user making a secure file retrieval request to the Dark Fetcher Solid App and finally decrypting and decoding the retrieved result is explained in sequence diagram 4.14. First, the user specifics the plaintext identifier of the target file to the Secure Request Sender. The Secure Request Sender will prepare the pk0, the first block of the user's public key, from a local file and then read the encrypted identifier sequence from the local storage. The encrypted identifier sequence is already prepared in section 4.3.4. The pk0 and the encrypted identifier are then sent to the Dark Fetcher for computation. Upon receiving a request, the Dark Fetcher first downloads all the files inside the user's data Pods. The downloaded files, together with two pieces of information in the HTTP request, are then fed into the Private Computation for further processing, which secretly filters out the target files from all the files. The generated result is then returned to the CryptoBox. The CryptoBox uses the Tools to decrypt and decodes the result. The detailed procedure of decrypting a file is described in section 4.3.8. The final result is then saved to local storage, ready for downloading.

4.3.8 The process of decrypting a ciphertext block sequence

The process of decrypting an encrypted block sequence using the private key is shown in sequence diagram 4.15. First, the Tools read the file from the local storage given a filename. The Tools decrypts the encrypted file block by block using the Core. For each ciphertext block in the file, it saves the block to the local storage, sends a decryption request to the Core, and collects the result. Finally, it combines the collected results and returns the plaintext file.

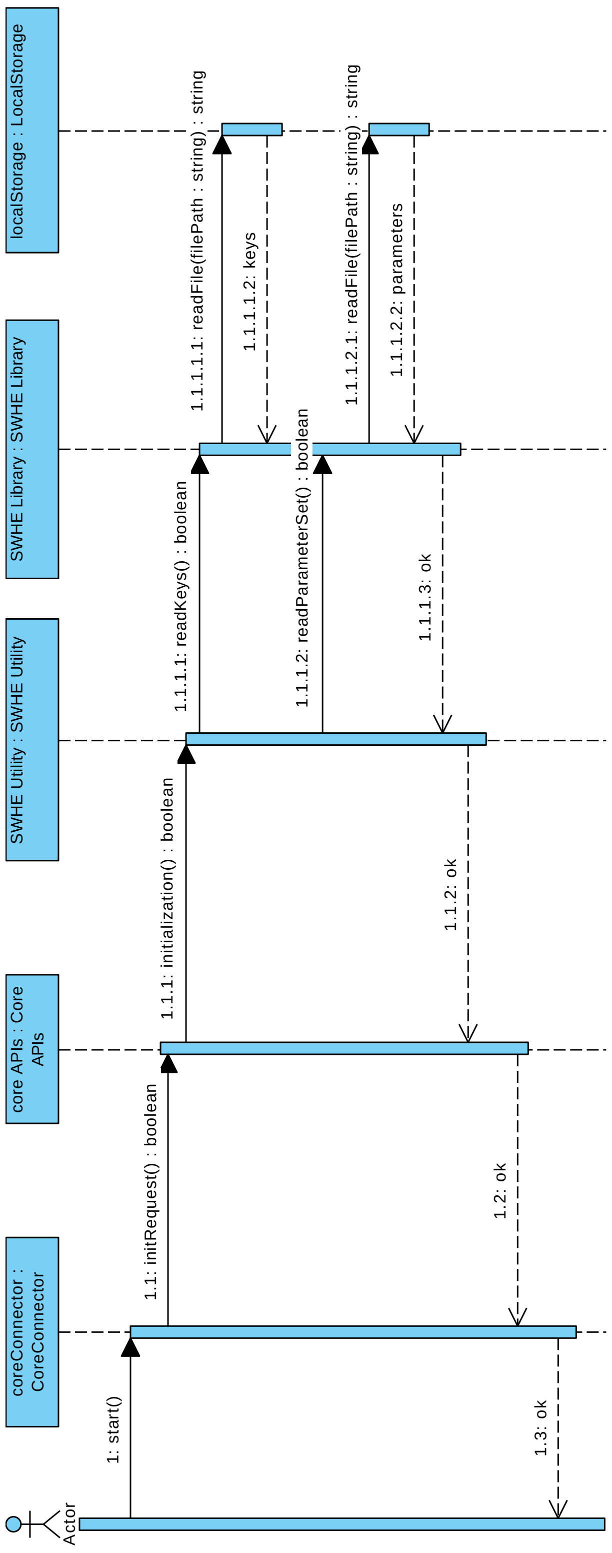

localStorage : LocalStorage
SWHE Library : SWHE Library
SWHE Utility : SWHE Utility
core APIs : Core APIs
coreConnector : CoreConnector
Actor
1: start()
1.1: initRequest() : boolean
1.1.1: initialization() : boolean
1.1.1.1: readKeys() : boolean
1.1.1.1.1: readFile(filePath : string) : string
1.1.1.1.2: keys
1.1.1.2: readParameterSet() : boolean
1.1.1.2.1: readFile(filePath : string) : string
1.1.1.2.2: parameters
1.1.1.3: ok
1.1.2: ok
1.2: ok
1.3: ok

Figure 4.8: System setup initialization

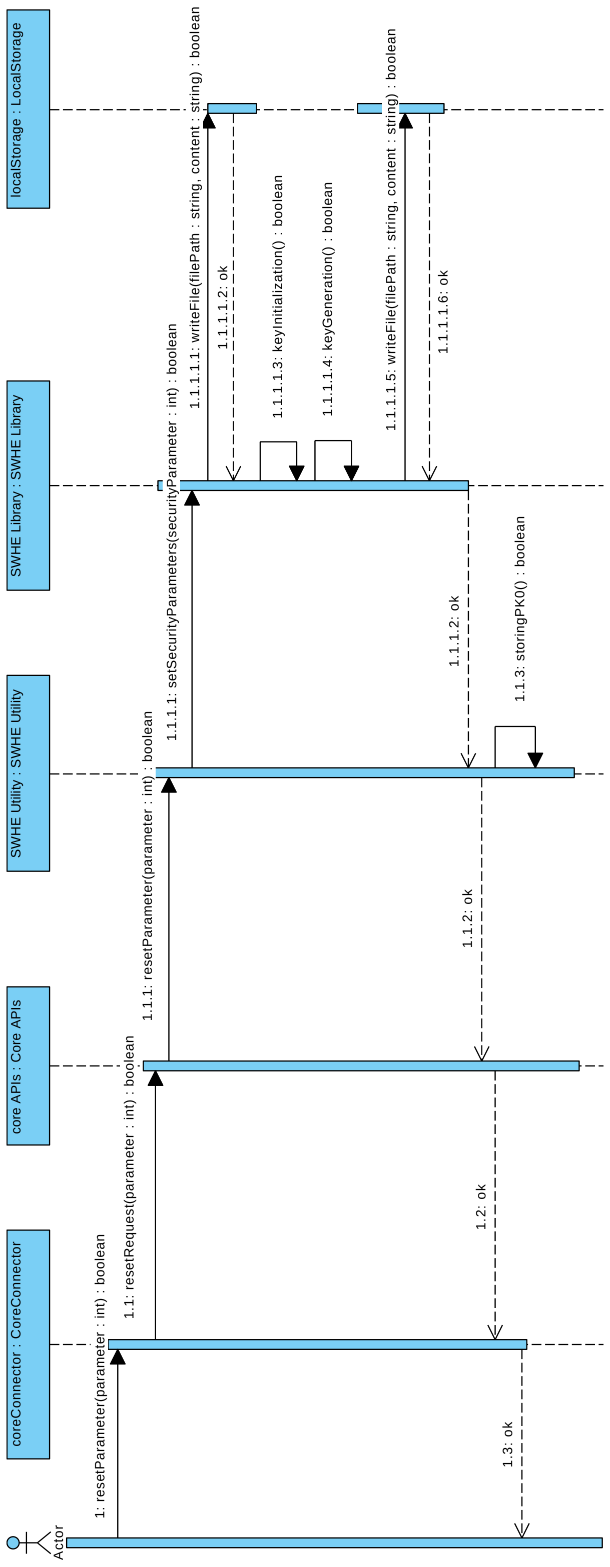

localStorage : LocalStorage
SWHE Library : SWHE Library
SWHE Utility : SWHE Utility
core APIs : Core APIs
coreConnector : CoreConnector
Actor
1: resetParameter(parameter : int) : boolean
1.1: resetRequest(parameter : int) : boolean
1.1.1: resetParameter(parameter : int) : boolean
1.1.1.1: setSecurityParameters(securityParameter : int) : boolean
1.1.1.1.1: writeFile(filePath : string, content : string) : boolean
1.1.1.1.2: ok
1.1.1.1.3: keyInitialization() : boolean
1.1.1.1.4: keyGeneration() : boolean
1.1.1.1.5: writeFile(filePath : string, content : string) : boolean
1.1.1.1.6: ok
1.1.1.2: ok
1.1.3: storingPK0() : boolean
1.1.2: ok
1.2: ok
1.3: ok

Figure 4.9: Reset security parameters and generate new keys

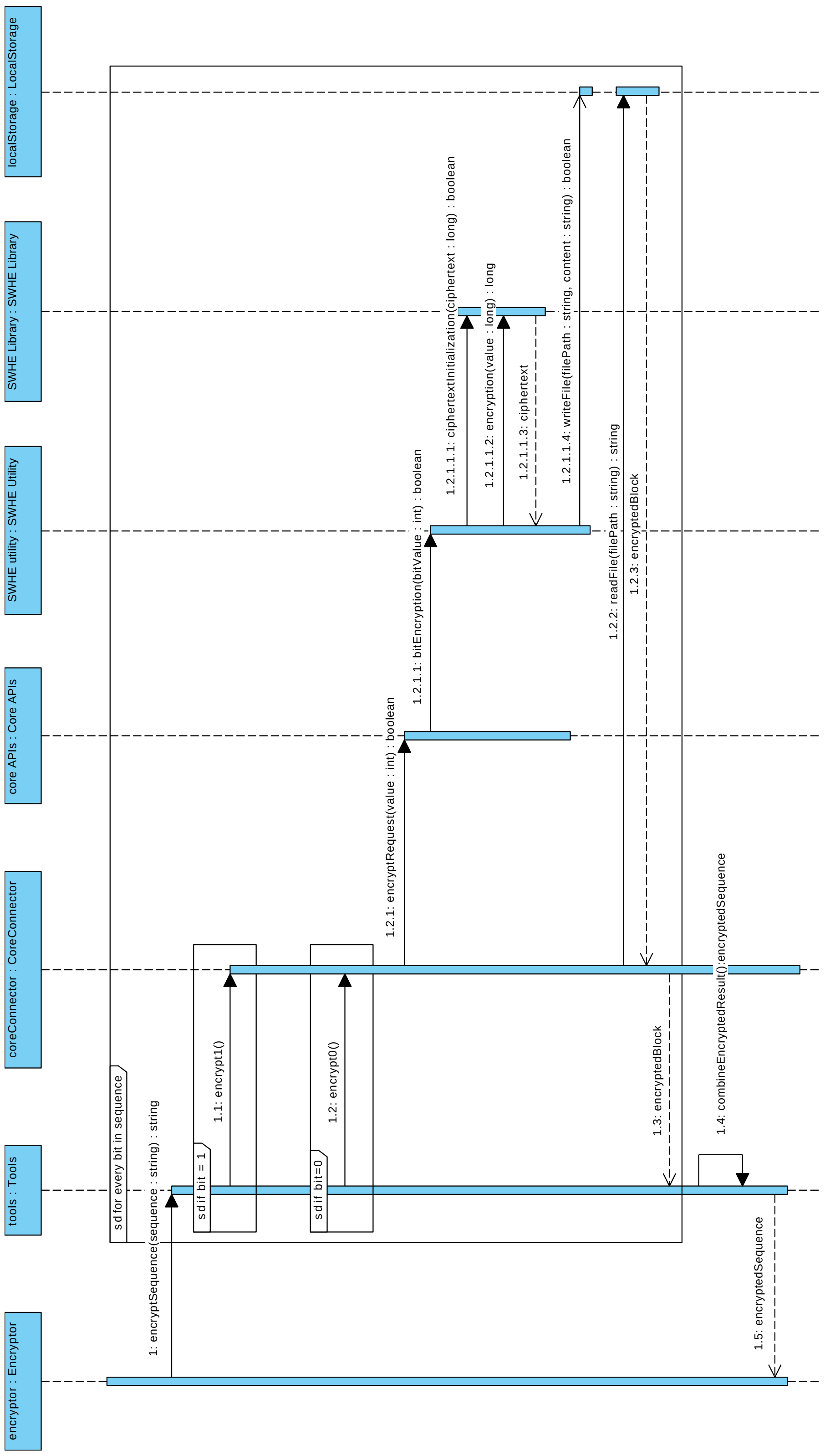

encryptor : Encryptor
tools : Tools
coreConnector : CoreConnector
core APIs : Core APIs
SWHE utility : SWHE Utility
SWHE Library : SWHE Library
localStorage : LocalStorage
1: encryptSequence(sequence : string) : string
sd for every bit in sequence
sd if bit = 1
1.1: encrypt1()
sd if bit=0
1.2: encrypt0()
1.2.1: encryptRequest(value : int) : boolean
1.2.1.1: bitEncryption(bitValue : int) : boolean
1.2.1.1.1: ciphertextInitialization(ciphertext : long) : boolean
1.2.1.1.2: encryption(value : long) : long
1.2.1.1.3: ciphertext
1.2.1.1.4: writeFile(filePath : string, content : string) : boolean
1.2.2: readFile(filePath : string) : string
1.2.3: encryptedBlock
1.3: encryptedBlock
1.4: combineEncryptedResult():encryptedSequence
1.5: encryptedSequence

FIGURE 4.10: The bit sequence encryption

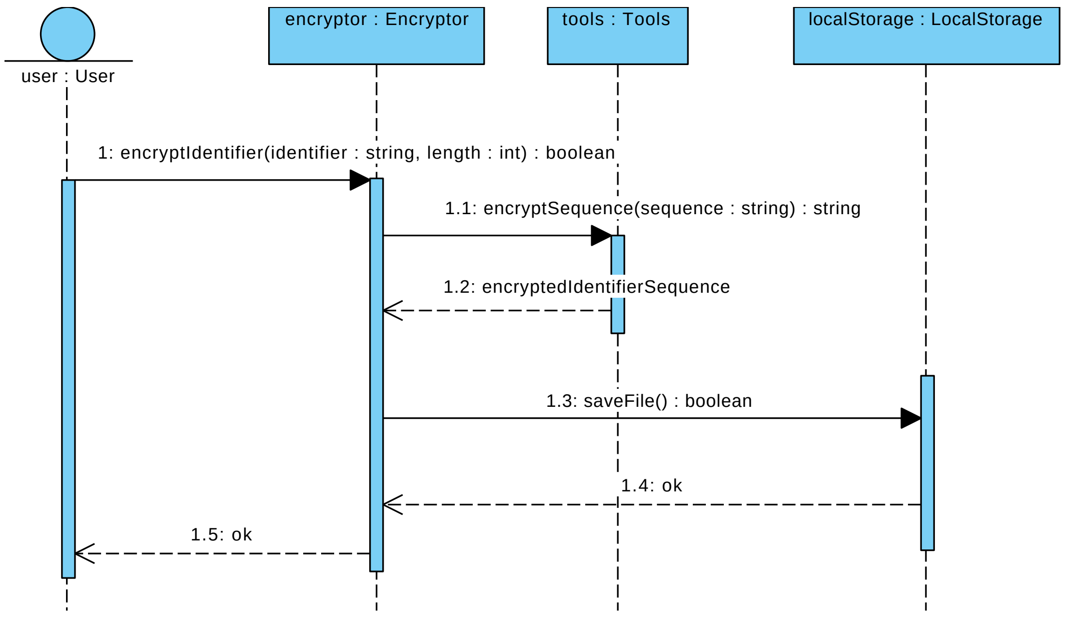

user : User
encryptor : Encryptor
tools : Tools
localStorage : LocalStorage
1: encryptIdentifier(identifier : string, length : int) : boolean
1.1: encryptSequence(sequence : string) : string
1.2: encryptedIdentifierSequence
1.3: saveFile() : boolean
1.4: ok
1.5: ok

Figure 4.11: The file identifier encryption

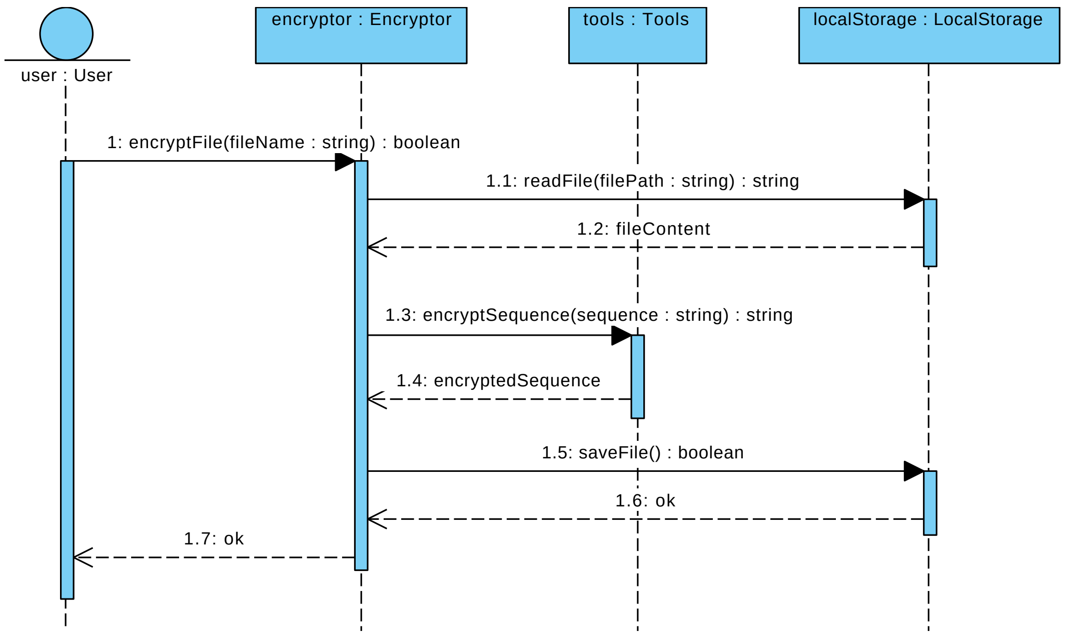

user : User
encryptor : Encryptor
tools : Tools
localStorage : LocalStorage
1: encryptFile(fileName : string) : boolean
1.1: readFile(filePath : string) : string
1.2: fileContent
1.3: encryptSequence(sequence : string) : string
1.4: encryptedSequence
1.5: saveFile() : boolean
1.6: ok
1.7: ok

Figure 4.12: The file encryption

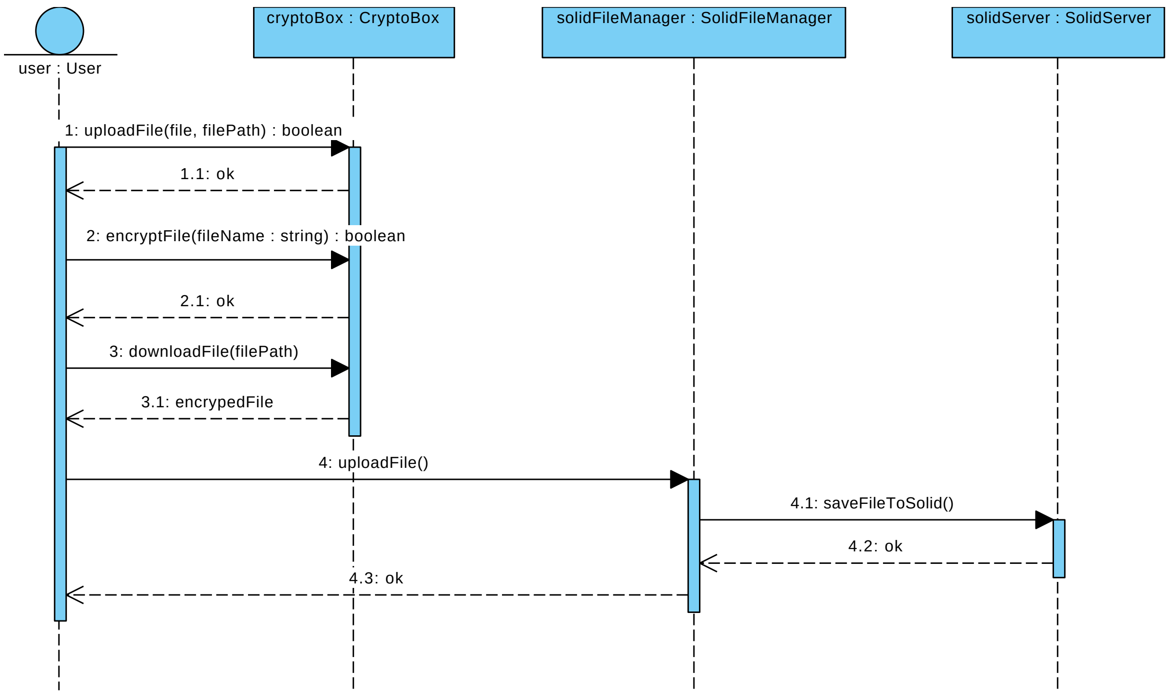

user : User
cryptoBox : CryptoBox
solidFileManager : SolidFileManager
solidServer : SolidServer
1: uploadFile(file, filePath) : boolean
1.1: ok
2: encryptFile(fileName : string) : boolean
2.1: ok
3: downloadFile(filePath)
3.1: encrypedFile
4: uploadFile()
4.1: saveFileToSolid()
4.2: ok
4.3: ok

FIGURE 4.13: User Upload Encrypted File To data Pod

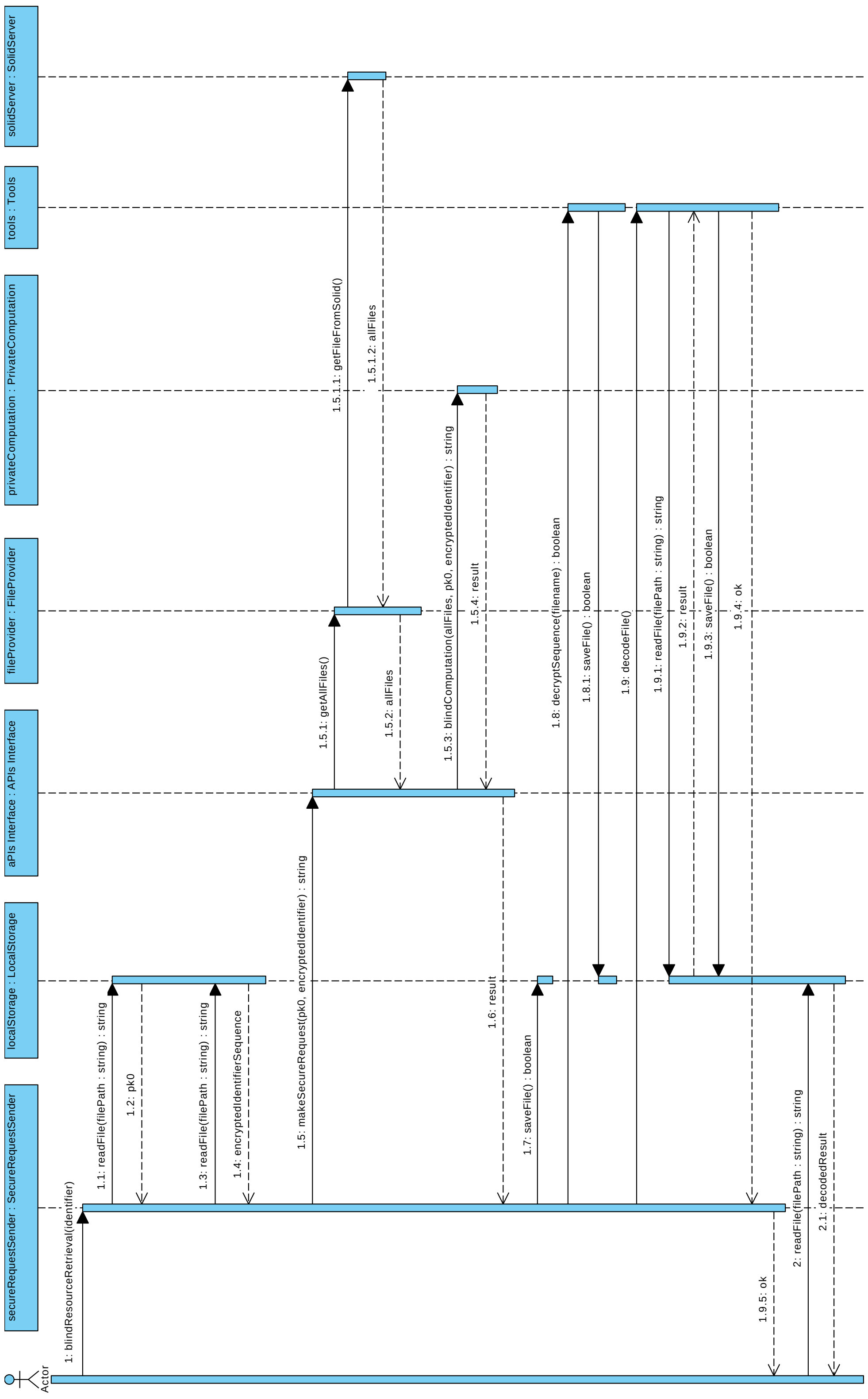

Actor
secureRequestSender : SecureRequestSender
localStorage : LocalStorage
aPIs Interface : APIs Interface
fileProvider : FileProvider
privateComputation : PrivateComputation
tools : Tools
solidServer : SolidServer
1: blindResourceRetrieval(identifier)
1.1: readFile(filePath : string) : string
1.2: pk0
1.3: readFile(filePath : string) : string
1.4: encryptedIdentifierSequence
1.5: makeSecureRequest(pk0, encryptedIdentifier) : string
1.5.1: getAllFiles()
1.5.1.1: getFileFromSolid()
1.5.1.2: allFiles
1.5.2: allFiles
1.5.3: blindComputation(allFiles, pk0, encryptedIdentifier) : string
1.5.4: result
1.6: result
1.7: saveFile() : boolean
1.8: decryptSequence(filename) : boolean
1.8.1: saveFile() : boolean
1.9: decodeFile()
1.9.1: readFile(filePath : string) : string
1.9.2: result
1.9.3: saveFile() : boolean
1.9.4: ok
1.9.5: ok
2: readFile(filePath : string) : string
2.1: decodedResult

Figure 4.14: Performing secure computation

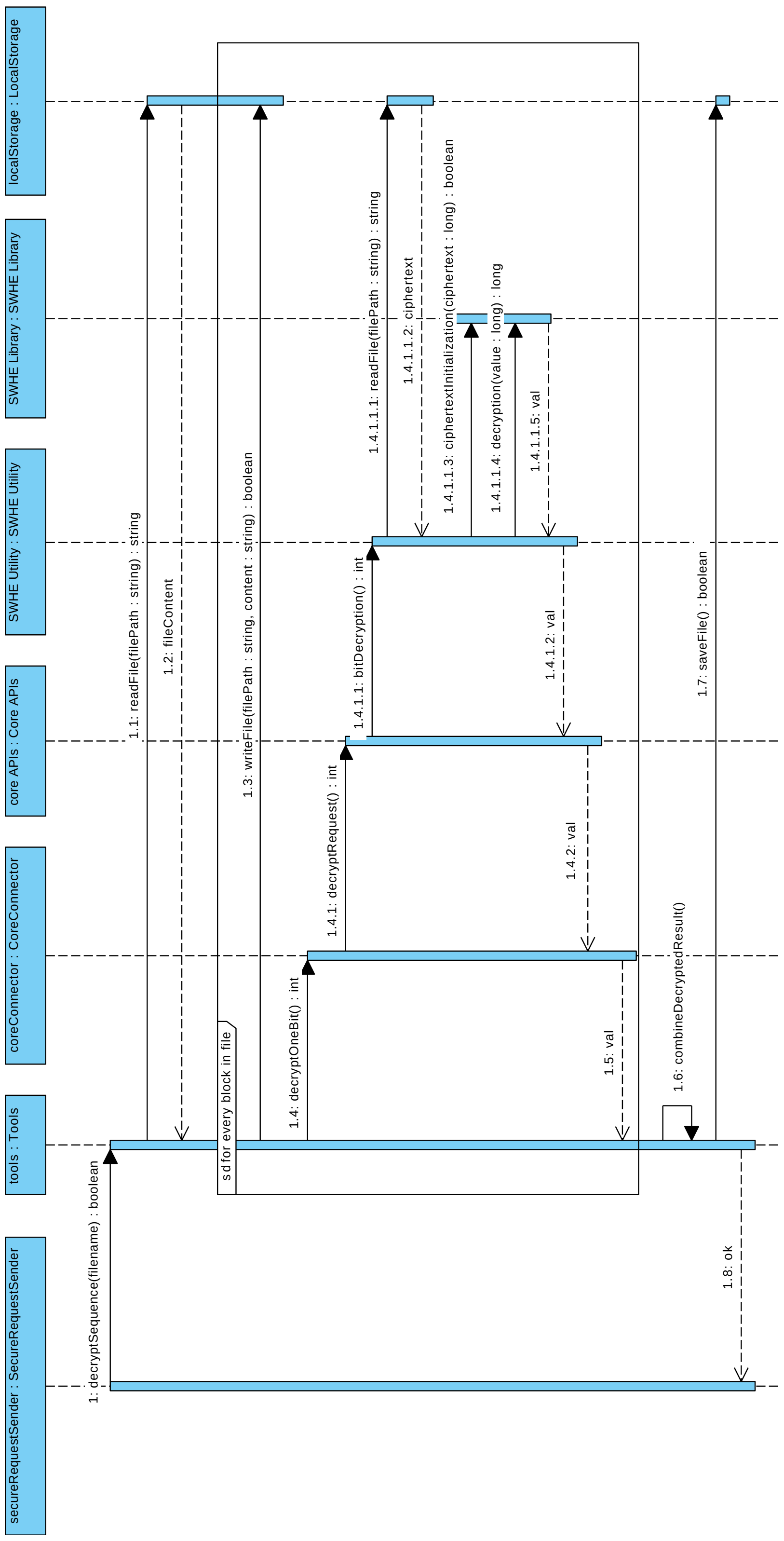

secureRequestSender : SecureRequestSender
tools : Tools
coreConnector : CoreConnector
core APIs : Core APIs
SWHE Utility : SWHE Utility
SWHE Library : SWHE Library
localStorage : LocalStorage
1: decryptSequence(filename) : boolean
1.1: readFile(filePath : string) : string
1.2: fileContent
sd for every block in file
1.3: writeFile(filePath : string, content : string) : boolean
1.4: decryptOneBit() : int
1.4.1: decryptRequest() : int
1.4.1.1: bitDecryption() : int
1.4.1.1.1: readFile(filePath : string) : string
1.4.1.1.2: ciphertext
1.4.1.1.3: ciphertextInitialization(ciphertext : long) : boolean
1.4.1.1.4: decryption(value : long) : long
1.4.1.1.5: val
1.4.1.2: val
1.4.2: val
1.5: val
1.6: combineDecryptedResult()
1.7: saveFile() : boolean
1.8: ok

FIGURE 4.15: Decrypting a ciphertext sequence

Chapter 5

Mechanism Explanation

The chapter explains the underlying mathematical reasoning behind selecting the targeted file from the client's data Pods and gives a detailed representation of how messages and files between different system components are processed. First, section Mathematics foundation 5.1 represents the mathematical proof of how localizers are calculated in the DarkFetcher Solid App. After, section Cryptographic messages processing 5.2 illustrates the procedures for handling bit-level cryptographic information and file data in one typical client's request.

5.1 Mathematics foundation

Step one and two of DarkFetcher Solid App computation in figure 5.1 is based on the following principles, in reference to the reasoning in the paper Homomorphic Evaluation of Database Queries[23]. Let c_i be the i^{th} block of the encrypted file identifier in the secure request, and m_i be the i^{th} bit of a plaintext file identifier in the data Pod, where the encryption is based on the Somewhat Homomorphic Encryption Scheme over Integers in section 2.2.1. Assume that the plaintext bit of the targeted retrieving file's identifier corresponding to c_i is n_i, then one can have

$$c_i = n_i + 2r_i + pq_i$$

, where r_i and q_i are random numbers and p is the secret key.

It can be observed that

$$1 + c_i + m_i = 1 + (n_i + m_i) + 2r_i + pq_i \ .$$

If $n_i = m_i$, then $n_i + m_i \equiv 0 \mod 2$, thus

$$1 + c_i + m_i = 1 + 2r_i + pq_i = Enc(1).$$

If $n_i \neq m_i$, then $n_i + m_i \equiv 1 \mod 2$, thus

$$1 + c_i + m_i = 2(1 + r_i) + pq_i = Enc(0).$$

For each file F_k with index k in the data Pods, the DarkFetcher Solid App computes a localizer I_k based on its s-bit identifier,

$$I_k = \prod_{i=0}^{s-1}(1 + c_i + m_i).$$

Based on the deduction above, it can be computed that $I_k = Enc(1)$ if two identifiers n and m match or $I_k = Enc(0)$ if two identifiers do not match.

5.2 Cryptographic messages processing

The procedures of processing cryptographic messages in a single request is shown in figure 5.1.

Step 1 : Identifier encryption and send secure request

In this scenario, an 8-bit plaintext identifier chosen by the clients is encrypted separately by the core layer and generates eight encrypted ciphertext blocks. It is essential to notice that the encrypted results are different for the same plaintext value. The service layer retrieves and then sends those blocks to the DarkFetcher Solid App to make a secure request.

Step 2 : Identifier processing

The computation component on the app start processing the request. First, the encrypted blocks c_i are compared to plaintext identifier v_i of all the files in the corresponding client data Pods, using the formula $I_v = \prod_{i=0}^{8}(c_i + v_i + 1)$. If the corresponding plaintext bit of the c_i is the same as the compared identifier bit v_i, it will generate an encryption of one, if not, an encryption of zero otherwise. The computation is implemented by Identifier Processing module 4.7.

Step 3 : Localizers generation

The generated eight compare results are then multiplied together, generating a ciphertext block, a localizer I_v, which is an encryption of one if two identifiers are equal or an encryption of zero otherwise. Detail explanation is shown in section 5.1. The computation is implemented by Localizer Generation module 4.7.

Step 4 : Target file Selection

The generated localizer I_v of each compared file identifier is then multiplied by the corresponding encrypted file blocks by blocks, where each block represents one bit of value. If the localizer value is Enc(0), the multiplied result of the encrypted file would become a block sequence of encryption of zero. If the localizer is equal to Enc(1), the result remains equal to the encrypted file content in the way that they both decrypt to the same result. The computation is implemented by Target File Selection module 4.7.

Step 5 : Result combination

The results are then added together according to the order of every block, where every first block of the computed results is added together, and the remaining blocks are also added in their order. The length of the added result depends on the file that has the largest length. The final result consists of a sequence of encryption of zero and an encrypted target file block sequence. The computation is implemented by Result Combination module 4.7.

Step 6 : Decrypt and decode the result

The final result is then returned to the service layer for decryption. Every ciphertext block is decrypted separately in the core layer and collected by the service layer. A file decoding procedure is performed after the file has been decrypted. The decoded plaintext file is returned to the clients, and the process is finished. It is necessary to mention that the length of the identifier and the file length could be arbitrary integer numbers.

Special scenario: illegal request with an invalid file identifier

When users make illegal requests to ask for a resource that does not exist in their data Pods by sending an invalid file identifier, the requests are made ineffective. Suppose users request a resource that does not exist on the data Pods. In that case, all of the localizers generated by comparing the identifiers will all be an encryption of zero since the requested file identifier does not match any of the identifiers of the files in the Pods. The comparing process is described in steps 2 and 3, and the result generation is illustrated in step 4. This way, the return result will be a sequence of encryption of zeros according to step 4.

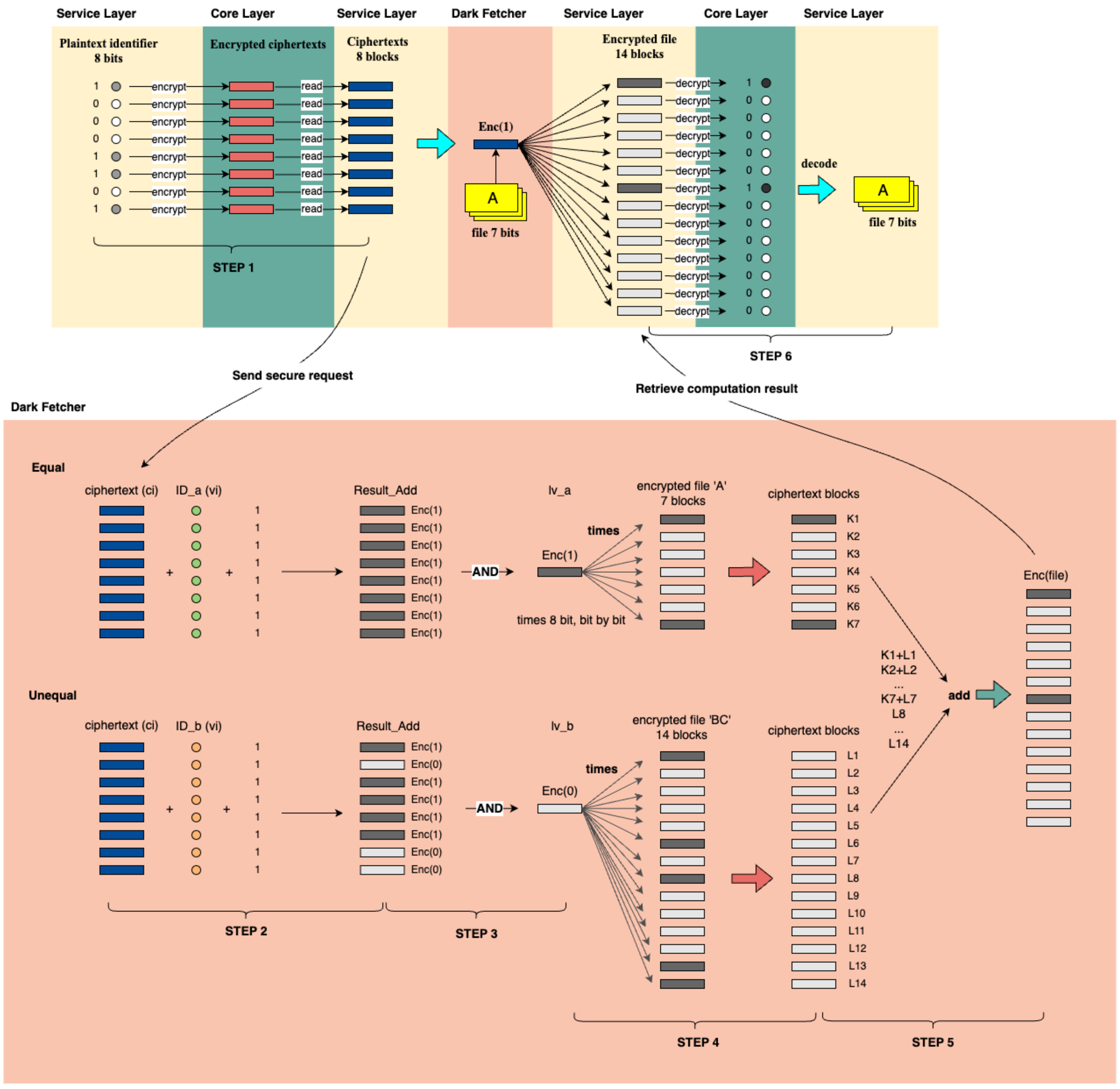


Figure 5.1: The processing of cryptographic information in a single request

Chapter 6

Evaluation

This chapter presents the setup of the system and the evaluation of the system from both performance and non-functional requirements aspects. First, section 6.1 gives the basic requirements for running the system and presents the setup for the evaluation. Next, section 6.2 evaluates factors that influence the system performance of several critical operations. Finally, section 6.3 discusses to which extent the non-functional requirements listed in section 3.2.2 are achieved.

6.1 Setup

The CryptoBox[1] implements two layers, the Core layer and the Service layer. The Core layer is implemented in C and compiled with GNU 11, using libraries facil.io[2] and DGHVlib[3], with dependencies on GNU Arithmetic Library[4]. The DGHVlib library for the homomorphic encryption over integers is chosen for the reason of easability of use. Other libraries either generate bad results, e.g., Coron441[5], or are too difficult to extend, e.g. fhe[6]. The service layer is written in Python, which depends on the gmpy[7] library. The Dark Fetcher Solid App[8] is written in Java with dependencies on Inrupt Java Client Libraries[9] using the SpringBoot framework[10]. Currently, the Inrupt Java Client Libraries only supports single user scenario, with data Pods hosting on an Inrupt PodSpaces server and managing by the PodBrowser Solid application. The evaluation is performed on a M1 Mac machine with 8GB RAM and 256GB storage.

[1]`https://github.com/SeraphinaLiang/Thesis_CryptoBox`
[2]`https://github.com/boazsegev/facil.io`
[3]`https://github.com/limengfei1187/DGHVlib`
[4]`https://gmplib.org/`
[5]`https://github.com/lCardosoSantos/Coron441`
[6]`https://github.com/coron/fhe`
[7]`https://pypi.org/project/gmpy/`
[8]`https://github.com/SeraphinaLiang/thesis_app`
[9]`https://docs.inrupt.com/developer-tools/java/client-libraries`
[10]`https://spring.io/guides/gs/spring-boot/`

6.2 Performance evaluation

The system performance is mainly affected by four kinds of operations, the encryption of file and file identifier, the decryption of the result, the computation process of the Dark Fetcher Solid App, and the files retrieval process from the Solid server to the Dark Fetcher Solid App. First, the performance of bit sequences encryption is evaluated in subsection 6.2.1. The encryption process is used for encrypting the file identifiers and content of the files. Second, subsection 6.2.2 examines how different factors influence the speed of decrypting ciphertext blocks. Next, the computation process to filter out the target file corresponds to step 2 to step 5 in the section 5.2. The operation is evaluated in subsection 6.2.3 by measuring how various factors have an impact on the processing speed. Finally, the time to execute the last operation is uncontrollable by the system if the Solid servers are hosted on remote machines since the network is unstable. If the servers are hosted on local machines, the time to execute the operation is stable and only affected by the size and number of files. Thus, the necessity to measure the performance of the retrieval operation is considered low. Therefore, the evaluation will only measure the performance of the first three operations. Every experiment to these operations, including encryption, decryption, and blind computation, is executed 50 times on the mentioned machine to minimize the influence of uncertainty in the testing environment.

The security parameter set

As mentioned in section 2.2.1, the security parameter λ affects the generation of the key pair in that the higher the λ value, the larger the values of the items in the security parameter set, and thus the larger the key size. The security parameter set of the public key version of the Somewhat Homomorphic encryption scheme includes five terms $\lambda, \rho', \rho, \eta, \gamma$ and τ as described in the scheme parameters of the public key version in section 2.2.1. Coron et al.[10] proposed a suggested parameter set value as shown in figure 2.4. The toy level is corresponding to a 42-bit λ value. In the evaluation, three parameter sets shown in table 6.1 are designed for testing. It is necessary to mention that the small level has a security parameter λ lower than the λ in the toy security level mentioned in the suggested parameter set. The parameter sets are chosen for several reasons. First, the machine capability is taken into account because the memory space used by the Core layer is considered limited. This is due to the reason that the SWHE library the Core layer adopted does not automatically free unused memory space. Second, it should roughly satisfy the constraints mentioned in the DGHV with Shorter Public Keys in section 2.2.2. Third, relatively low compared to the toy level, but the maximum possible values concerning the performance of the current implementations are chosen to enable the evaluation to be conducted successfully. The small file size and low number of files used in the evaluation are selected for the same reason. The current implementation is not practical for actual usage since it performs poorly. In future, the implementation needs to be improved. The security parameter λ has a considerable impact on the system performance as described in section 2.3.1 in that the higher the λ value, the more secure the system become, however, the longer time is needed for processing. The three chosen parameter sets are regarding as an underlying influence factor for performance evaluation in measuring the three operations.

Level	λ	ρ'	ρ	η	γ	τ
Tiny	24	8	8	872	300000	200
Mini	28	12	18	1400	1000000	300
Small	32	16	24	2216	2000000	500

TABLE 6.1: The security parameter sets used in evaluation

6.2.1 Evaluation of the encryption operation

The encryption operation is evaluated on three security parameter levels as listed in table 6.1. The tiny, mini and small levels correspond to security parameter level α equaling to 1,2 and 3, respectively. The processing time of encryption operation for three security levels is shown in figure 6.1, for bit sequences ranging from 1 to 50 bytes. Additionally, the time in seconds required to encrypting every ten more bits is presented in table 6.2. The time and the number of bits encrypted display a linear relationship since every bit is encrypted separately. The line graph also shows that the higher the security level, the more time it takes to encrypt one bit. With the security level increasing, the length of cryptographic components grows longer; thus, the system takes more time to process larger values. For the small level, it already takes 1.8 seconds to encrypt a 50-bit value, which is around seven characters. At this speed, for a text file of size 1 KB, the encryption of the file takes almost 5 minutes. Even at the tiny level, where α equals 1, the same file needs 46 seconds to process, exceeding the limit of 10 seconds to keep users attention[21]. The encryption speed is impractical for actual usage.

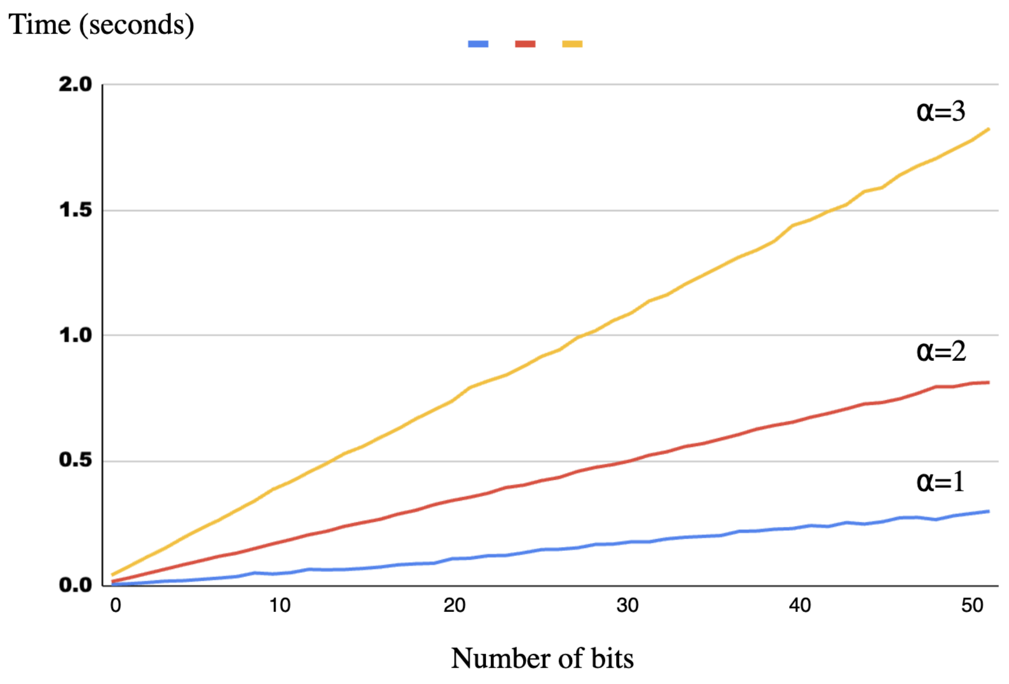
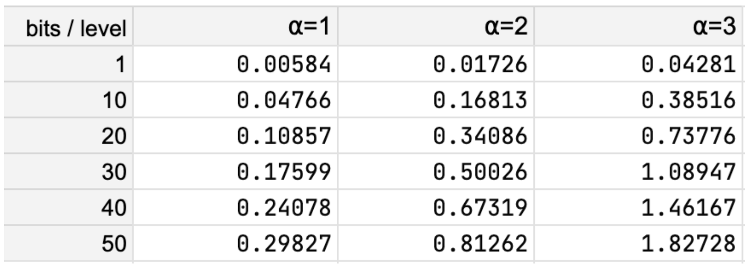


Figure 6.1: The encryption from 1 to 50 bits under 3 security parameter levels



bits / level	α=1	α=2	α=3
1	0.00584	0.01726	0.04281
10	0.04766	0.16813	0.38516
20	0.10857	0.34086	0.73776
30	0.17599	0.50026	1.08947
40	0.24078	0.67319	1.46167
50	0.29827	0.81262	1.82728

Figure 6.2: The selection of encryption experiment data

6.2.2 Evaluation of the decryption operation

The decryption operation is evaluated on the same security parameter levels with the same α corresponding relationship. The processing time of decryption operation for three security levels is shown in figure 6.3, for bit sequences ranging from 1 to 50 bytes. Additionally, the time in seconds required to decrypting every ten more bits is presented in table 6.4. The number of decrypted bit and the time needed also indicates a linear relationship since every bit is decrypted individually. Also, the higher the security level, the more time takes to decrypt one bit. With the security level increasing, the length of cryptographic components grows, consuming more time for the system to handle. Surprisingly, the decryption takes less time than the encryption process. For the small level, it requires 1.5 seconds to decrypt 50 bits. At this speed, a text file of size 1 KB takes 4 minutes to process. Even at the tiny level, 43 seconds are needed for decryption, exceeding the limit of 10 seconds to keep users attention[21]. Thus, the processing speed is impractical for actual usage.

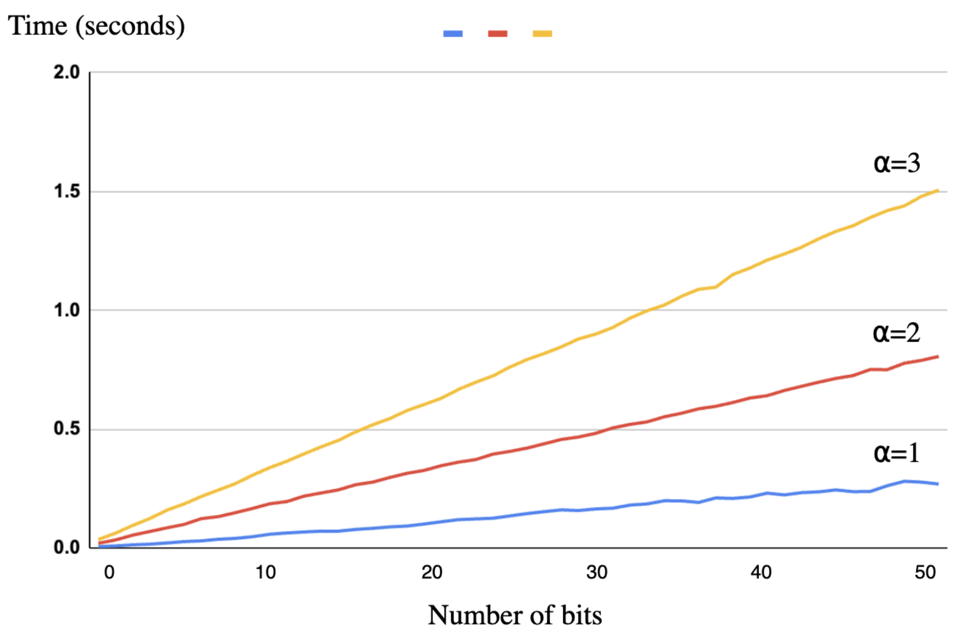
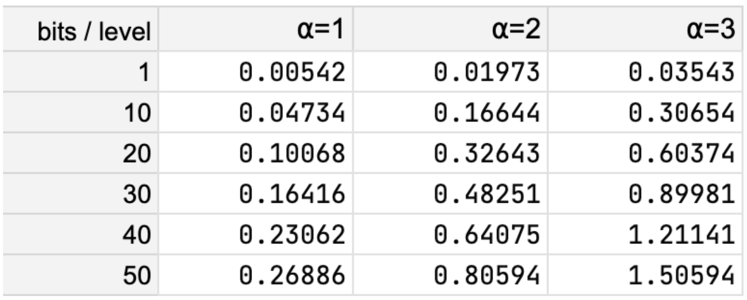


FIGURE 6.3: The decryption from 1 to 50 bits under 3 security parameter levels



bits / level	α=1	α=2	α=3
1	0.00542	0.01973	0.03543
10	0.04734	0.16644	0.30654
20	0.10068	0.32643	0.60374
30	0.16416	0.48251	0.89981
40	0.23062	0.64075	1.21141
50	0.26886	0.80594	1.50594

FIGURE 6.4: The selection of decryption experiment data

6.2.3 Evaluation of the blind computation operation

The speed of blind computation operation is affected by various factors, including the length of identifiers of files, the number of files involved in the computation process, the average file size of the plaintext file and the security parameter levels. Those mentioned factors are examined using the variable control method to decide to which extent each factor influences the processing time. The combination of variable values is chosen considering two reasons. First, the processing time should be under a certain limit, e.g. 5 minutes. Second, in considering the first reason, the experiments attempt to select the average values for each variable.

The computation time

The time needed for the computation process, as shown in steps 2 to 5 in figure 5.1, can be computed as follow. Assumed the length of identifier be l_i, the average file size be l_f, the number of file be n_f, the time needed for addition operation be t_{ADD}, the time needed for multiplication operation be t_{MUL}, the time T needed for computation is

$$T = l_i \cdot t_{ADD} \cdot n_f + (l_i - 1) \cdot t_{MUL} \cdot n_f + l_f \cdot t_{MUL} \cdot n_f + l_f \cdot t_{ADD} \cdot (n_f - 1).$$

Reformatting the T equation, the relationship between the time T and the length of identifier l_i can be expressed as

$$T = l_i \cdot (n_f \cdot t_{ADD} + n_f \cdot t_{MUL}) + (l_f - 1) \cdot t_{MUL} \cdot n_f + l_f \cdot t_{ADD} \cdot (n_f - 1).$$

The relationship between the time T and the number of file be n_f can be expressed as

$$T = n_f \cdot (l_i \cdot t_{ADD} + l_i \cdot t_{MUL} - t_{MUL} + l_f \cdot t_{MUL} + l_f \cdot t_{ADD}) - l_f \cdot t_{ADD}.$$

The relationship between the time T and the average file size l_f can be expressed as

$$T = l_f \cdot (t_{MUL} \cdot n_f + t_{ADD} \cdot n_f - t_{ADD}) + l_i \cdot t_{ADD} \cdot n_f + (l_i - 1) \cdot t_{MUL} \cdot n_f.$$

Only changing l_i, n_f or l_f, the experiment results are expected to be linear functions with different slopes and intercepts according to the equations above.

The length of the identifier

The experiment examines how the length of identifiers affects the processing time for the computation step. The correlation between the length of identifiers in bits and the time consumed is presented in figure 6.5. The other variables are fixed, where the security parameter level α is set to 1, the number of files is set to 5, and the average file size is set to 5 bytes. The experiment does not adopt α equals 2 since it took extremely high amount of time to process even one test. Based on the above equations, the result illustrates that the time increases linearly as expected, from around 40 seconds to 60 seconds, with the identifier length growing from 3 to 15 bits.

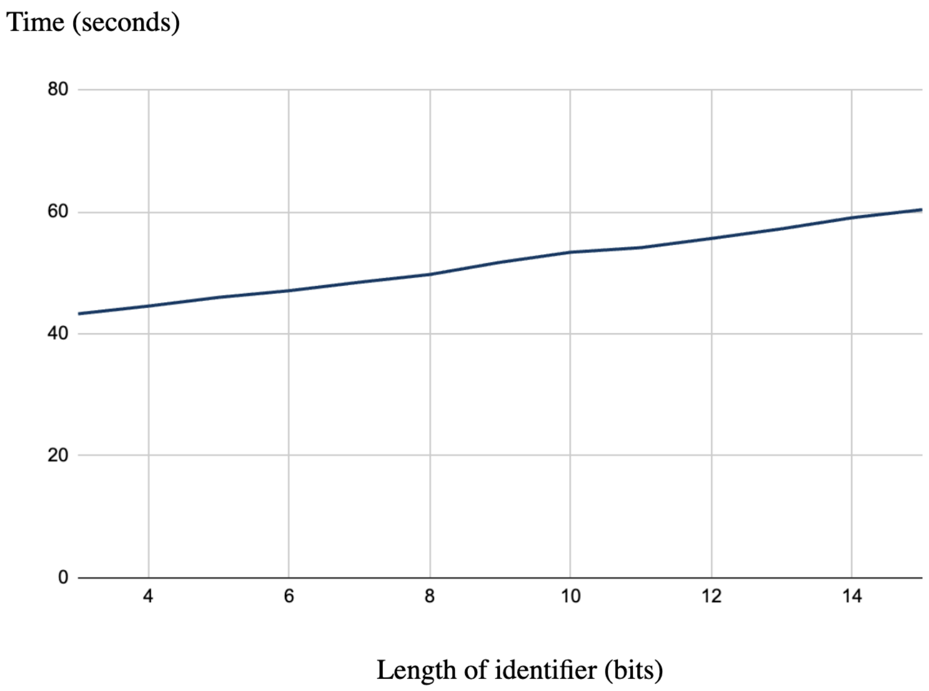


Figure 6.5: The variation in the processing time based on the length of the identifier

The number of files

The correlation between the number of files involved in the blind computation operation and the processing time is illustrated in figure 6.6. The number of files increases from 1 to 15 while other variables are fixed to static values. The security parameter level α is set to 1. The length of identifiers equals 8 bits, and the average file size is set to 5 bytes. The result demonstrates that the number of files considerably influences the processing time, which increases linearly with the file number, as expected based on the above equations. The result also shows that it takes around 140 seconds to filter out one file from 15 files, which is impractical within actual setting.

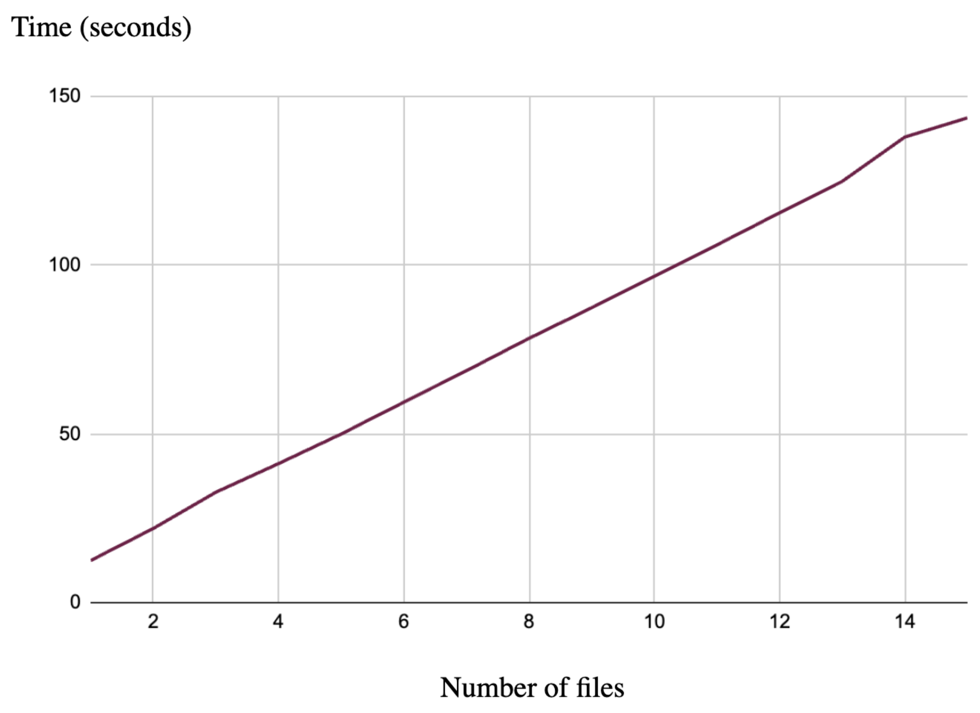


Figure 6.6: The variation in the processing time based on the number of files

The file size

The processing time increases linearly, with the average file size of the plaintext file increasing from 1 byte to 10 bytes as presented in figure 6.7. The linear behaviour is as expected based on the above equations. The experiment also adopts α equals one and takes eight as the identifier length and five as the number of files. The result indicates that the file size also largely influences the processing time. For a file size of 2 bytes, filtering the targeted file already takes around 25 seconds. When the file size increases to 10 bytes, more than 88 seconds are needed for the computation step.

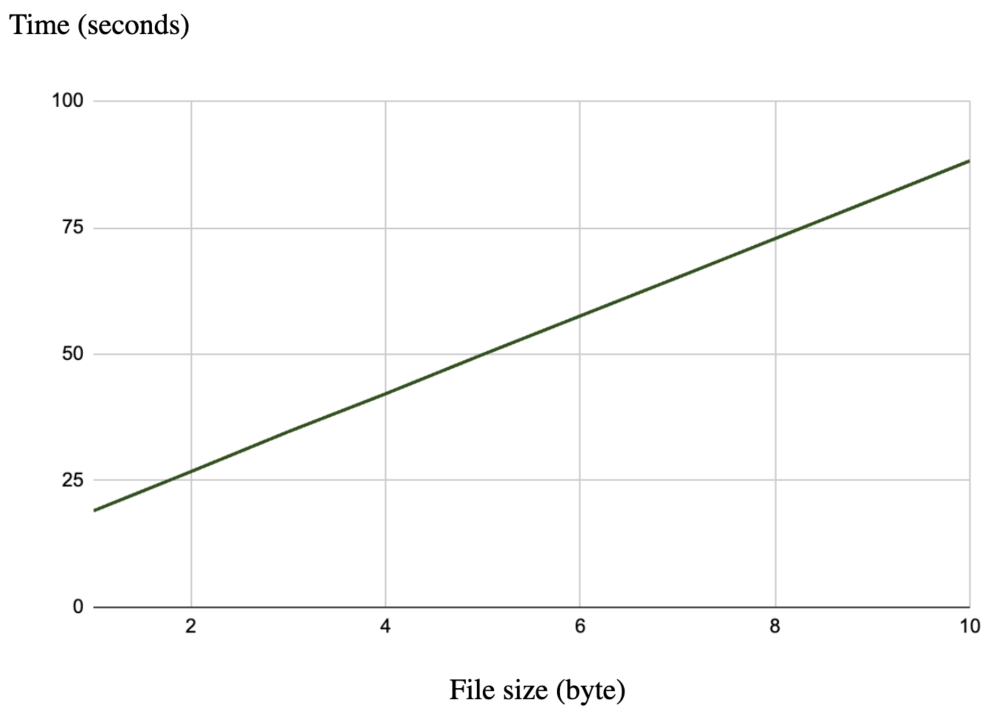


FIGURE 6.7: The variation in the processing time based on the average file size

The security parameter level

The evaluation is based on the security parameter levels as mentioned in table 6.1 with the same α corresponding relationship. Figure 6.8 exhibits how different security parameter levels α affect the processing time. Very small values are selected for the other variables since a high amount of time needed for measuring security parameter level α equals 2 and 3. The identifier length is set to 4. The file number is set to 2, and the average file size equals 1 byte. As expected, the result illustrates that the time rises with an increase in the security parameter levels, as explained in section 6.2 in that it takes more time to process bigger values. It takes only 6 seconds to process the tiny level ($\alpha = 1$), while up to 218 seconds for the small level ($\alpha = 3$).

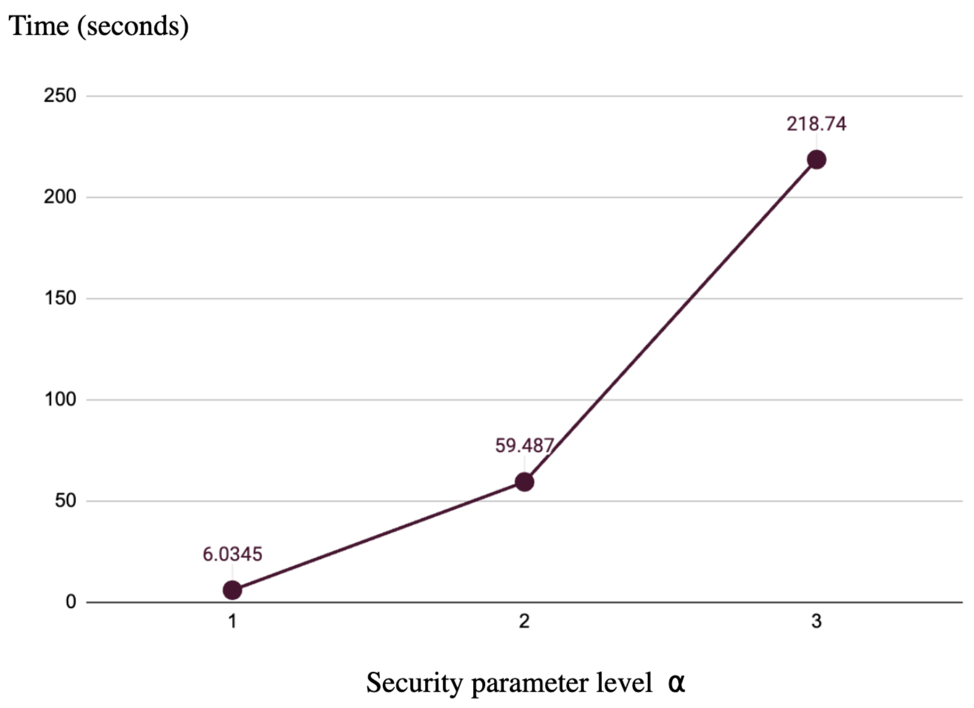


Figure 6.8: The variation in the processing time based on the security parameter level

Conclusion

As demonstrated in the above evaluation results, the security parameter level has the most significant impact on system performance within the four factors. The number of files is the second largest factor affecting the processing time. The average file size also influences the processing speed to some extent. Lastly, the length of identifiers slightly impacts the system's performance.

The evaluation result aligns with the findings of the related work [14], which points out that the security parameter λ strongly influences the system performance. With the λ values increasing, the length of cryptographic components increases. Thus, the system processes bigger values which is time-consuming. The linear behaviour of the results of the other three factors aligns with the expectation according to the equations of the computation time.

6.3 Non-functional requirements achievement

The section analysis to which extent the non-functional requirements mentioned in subsection 3.2.2 are achieved.

Performance The encryption and decryption operations take 1 to 2 seconds to process 50 bits, that is, 7 or 8 characters, depending on the actual encoding. The processing speed is relatively slow, even for the small security level. The Core layer in charge of encryption and decryption operations did not adopt parallel computing in actual implementation. Instead, the Core layer implements the operations in a serial sequence in that the second bit is handled after the first bit, despite no dependence relationship between the two bits. The design vastly lowers the processing speed. If parallelism is applied, the processing speed could be improved at a maximum of N times if N bits are processed simultaneously and at least N CPU cores are used for computation. The private computation step takes high amount of time to process which is impractical for actual usage. The private computation of the Dark Fetcher Solid app is implemented by Java which, to some extent, lowers the processing speed. Java was chosen for two reasons. First, only Javascript and Java have corresponding library support for Solid app development. Second, Javascript does not support the GMP library use for cryptographic operations, while the Java BigInteger library provides the same utility and is easy to use. The actual implementation also does not adopt parallelism. If parallelism computing is applied to step 2, step 4 and step 5 in figure 5.1, the processing time could be significantly reduced. The speed can be improved at a maximum of N times if enough computing resources are provided, and N bits or blocks are processed at the same time. No faster algorithm for addition or multiplication is applied to the private computation, which only performs these two operations.

Configurability The system enables users to configure the security level by changing the parameter set. Also, the user could specify the identifier length only in the constraint that all files should have an identifier of the same length. The system adopts public key cryptography in the actual implementation. However, the developer could also change the underlying implementation to asymmetric cryptography in the future.

Security Confidentiality is achieved since all the files are encrypted with the user's private key before exposing them to any third parties. The accessibility of the private key is strictly restricted to the user since the key is only stored locally on the user's machine. However, if the key leaks, all previous files, queries, and processed results encrypted with the private key are all exposed to the attacker. As mentioned in section 4.3, the user encrypts file content and identifier through the bit sequence encryption process in subsection 4.3.3 using the same private key. The encrypted identifier is then sent to the Dark Fetcher Solid app for querying and generating a result that the same private key can decrypt. If the attacker gets access to the private key, the attacker could use the key to recover those encrypted materials.

Scalability Vertical scaling has a substantial impact on the system's performance. The processing time can be reduced for most operations since those operations can be handled in parallel.

Chapter 7

Conclusion

The chapter concludes the thesis and discusses potential future work. First, section 7.1 presents the thesis overview from the goal, approach and contributions aspects. Next, section 7.2 suggests future improvements that could be made to the existing work.

7.1 Overview

In the thesis, we focus on achieving a scenario in which the client uses Solid services and, at the same time, wants to keep the content of the files and user behaviour unknown to any third-party Solid service provider. An existing problem with the Solid platform and applications is that they might compromise user privacy and not ensure user confidentially in case of attacks. The thesis aims to tackle these two problems by designing and implementing a framework that incorporates the Integer-based Somewhat Homomorphic Encryption scheme.

The approaches of the thesis for finding the solution are described as follows. First, the functional and non-functional requirements are analyzed and defined. The functional requirements give rise to the system's basic architecture, while the non-functional requirements provide decisions for the system's detailed design. The general components design is thus presented. The thesis application consists of two main components. First, a CryptoBox application running on the user's local machine provides encryption and decryption services. Second, a Dark Fetcher Solid application provides private file retrieval service from user data Pods, with no query information leaked to any third party. Next, the detailed view of each upper component is analyzed and designed, and the functional modules are defined for each component. The system logic flow is settled down. Finally, the system is implemented, and its performance is evaluated. The performance evaluation studies various factors that impact the processing speed of the system's critical operations. The result exhibits that the security parameter highly influences system performance. While other factors, including the number of files, the identifier length and the average file size, all show a linear relationship with the processing time but with different slopes. It also analyses to which extent the non-functional requirements are covered. The evaluation result shows that the system is impractical for actual usage since the processing time is extremely high. Requesting a 1-byte file takes more than 3 minutes, even for a small security parameter level.

To summarise, the main contributions of the thesis are as follows. First, the thesis developed two applications, the CryptoBox and the Dark Fetcher Solid application, enabling users to request a file from the data Pods without letting any third party know the exact content of the query. Second, the thesis evaluation result illustrates how various factors influence processing speed. It also indicates that the current solution is not performant and thus can not be applied to actual use.

7.2 Future work

The section discusses possible future improvements that could be made to the system design and actual implementations.

Integer-based comparison

As mentioned in section 2.3.2, H. Usef et al. established a new method[23] based on the Ring Based Fully Homomorphic Encryption scheme that allows comparing identifiers in the form of integers rather than bits. This way, the number of encryption and operation steps is largely reduced. The improvement can be applied to the Private Computation component in figure 4.2 by replacing the current implementation.

Paralleling operation steps

The encryption, decryption and computation steps referring to steps 1, 2, 4, 5 and 6 in figure 5.1 can be made parallel for all bits and ciphertext blocks since each bit or block is handled independently. This way, the time used for processing could be largely reduced if given sufficient computing resources. In the current implementation, the bit sequence is processed bit by bit in a loop structure. The block sequence is handled in the same way.

Adopting Bootstrapping technique

The maximum number of files involved in the private computation of the Darker Fetcher Solid app is limited, as mentioned in the further discoveries of the related work in section 2.3.1. The limitation is caused by a property of Somewhat Homomorphic Encryption (SWHE) that its operation generates noise when processing cryptographic messages. Changing the scheme from SWHE to Fully Homomorphic Encryption by adopting the bootstrapping step, the limitation problem does not exist. The app can process any number of files in one computation process.

Multi-users scenario

The Inrupt Java Client Libraries[1] only support single client scenarios without incorporating third-party libraries for client authentication. The Darker Fetch Solid app based on this libraries is currently a single-user app which only accesses one user's data Pod. The current solution is not practical for actual usage. The app could be extended to a multi-user application if a third-party library or framework with OpenID Connect[2] Authentication integration is integrated with the Inrupt Java Client Libraries. This way, multi-user authentication could be achieved, and thus more users could use the same Dark Fetcher Solid app to make private requests.

Lowering down overhead

The private computation component is currently designed as a Solid application that fetches client's files from data Pods each time when a request comes in. The fetching files operation generates considerable overhead since all files in the data Pods need to be transferred through the network. The design is due to the technical implementation limitation that it is challenging to integrate the private computation component directly into the Solid Server. If implemented this way, the unnecessary overhead is reduced. However, the new solution also causes an issue in that the Solid server provider layer is no longer transparent to the Solid application layer.

Faster multiplication implementation

The private computation in the Dark Fetcher Solid app comprises a high amount of multiplication and addition operations. Some fastest available multiplication algorithms, e.g.Schönhage-Strassen algorithm, can be applied to the multiplication operation[18]. This way, the time complexity could be largely reduced.

Future use case and the brute force attack

In a future scenario, it is possible that the data Pod owner is not the same entity as the person that tries to fetch files from the data Pod. It is also possible that the person does not have permission to access all the files in the Pod. In this case, the brute force attack can occur when an attacker constantly tries out different combinations of identifiers and illegally access the forbidden files. Thus, the server should limit the maximum times a user can make requests within a specific period. However, more measures must be applied to prevent files from being illegally accessed.

[1]https://docs.inrupt.com/developer-tools/java/client-libraries
[2]https://openid.net/connect/

Bibliography

[1] A. Acar, H. Aksu, A. S. Uluagac, and M. Conti. A survey on homomorphic encryption schemes: Theory and implementation. *ACM Comput. Surv.*, 51(4), jul 2018. https://doi.org/10.1145/3214303.

[2] Anonymous. Solid file manager, 2019. https://github.com/Otto-AA/solid-filemanager.

[3] Anonymous. W3c standards, 2021. https://www.w3.org/standards/.

[4] Anonymous. Homomorphism. brilliant.org., 2023. https://brilliant.org/wiki/homomorphism/.

[5] Z. Brakerski, C. Gentry, and V. Vaikuntanathan. Fully homomorphic encryption without bootstrapping. Cryptology ePrint Archive, Paper 2011/277, 2011. https://eprint.iacr.org/2011/277.

[6] Z. Brakerski and V. Vaikuntanathan. Efficient fully homomorphic encryption from (standard) lwe. In *2011 IEEE 52nd Annual Symposium on Foundations of Computer Science*, pages 97–106, 2011.

[7] Z. Brakerski and V. Vaikuntanathan. Fully homomorphic encryption from ring-lwe and security for key dependent messages. In P. Rogaway, editor, *Advances in Cryptology – CRYPTO 2011*, pages 505–524, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[8] L. Cardoso dos Santos, G. Rodrigues Bilar, and F. Dacêncio Pereira. Implementation of the fully homomorphic encryption scheme over integers with shorter keys. In *2015 7th International Conference on New Technologies, Mobility and Security (NTMS)*, pages 1–5, 2015. https://ieeexplore.ieee.org/document/7266495.

[9] A. Carey. On the explanation and implementation of three open-source fully homomorphic encryption libraries. 2020. Computer Science and Computer Engineering Undergraduate Honors Theses https://scholarworks.uark.edu/csceuht/77.

[10] J.-S. Coron, A. Mandal, D. Naccache, and M. Tibouchi. Fully homomorphic encryption over the integers with shorter public keys. Cryptology ePrint Archive, Paper 2011/441, 2011. https://eprint.iacr.org/2011/441.

[11] J.-S. Coron, D. Naccache, and M. Tibouchi. Public key compression and modulus switching for fully homomorphic encryption over the integers. Cryptology ePrint Archive, Paper 2011/440, 2011. `https://eprint.iacr.org/2011/440`.

[12] T. B.-L. et al. The solid project., 2017. `https://solid.mit.edu/`.

[13] Y. Gahi, M. Guennoun, and K. El-Khatib. A secure database system using homomorphic encryption schemes. *ArXiv*, abs/1512.03498, 2011.

[14] Y. Gahi, M. Guennoun, Z. Guennoun, and K. El-Khatib. A fully private video on-demand service. In *2012 25th IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*, pages 1–4, 2012.

[15] C. Gentry. *A Fully Homomorphic Encryption Scheme*. PhD thesis, Stanford, CA, USA, 2009. AAI3382729 `https://crypto.stanford.edu/craig/craig-thesis.pdf`.

[16] C. Gentry and S. Halevi. Implementing gentry's fully-homomorphic encryption scheme. In K. G. Paterson, editor, *Advances in Cryptology – EUROCRYPT 2011*, pages 129–148, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[17] S. Halevi. Fully homomorphic encryption, winter school on secure computation and efficiency, 2011. `https://shaih.github.io/pubs/FHE-winter-school.pdf`.

[18] Y. Ichibane, Y. Gahi, Z. Guennoun, and M. Guennoun. Private video streaming service using leveled somewhat homomorphic encryption. In *2014 Tenth International Conference on Signal-Image Technology and Internet-Based Systems*, pages 209–214, 2014.

[19] I. Inc. Inrupt developer tools documentation, 2023. `https://docs.inrupt.com/developer-tools`.

[20] L. Mengfei. Library of fully homomorphic encryption scheme on integers, 2018. `https://github.com/limengfei1187/DGHVlib`.

[21] J. Nielsen. Response times: The 3 important limits. nn group., 2022. `https://www.nngroup.com/articles/response-times-3-important-limits/`.

[22] O. J. of the European Union. Regulation (eu) 2016/679 of the european parliament and of the council of 27 april 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing directive 95/46/ec (general data protection regulation) (text with eea relevance), 2016. `https://eur-lex.europa.eu/eli/reg/2016/679/oj`.

[23] S. Palamakumbura and H. Usefi. Homomorphic evaluation of database queries, 2016. `https://arxiv.org/pdf/1606.03304.pdf`.

[24] H. V. L. Pereira. The approximate common divisor problem, 2021. https://www.esat.kuleuven.be/cosic/blog/the-approximate-common-divisor-problem/.

[25] O. Regev. The learning with errors problem (invited survey). In *2010 IEEE 25th Annual Conference on Computational Complexity*, pages 191–204, 2010.

[26] R. Rothblum. Homomorphic encryption: From private-key to public-key. volume 17, page 146, 01 2010.

[27] R. Rothblum. Homomorphic encryption: From private-key to public-key. In Y. Ishai, editor, *Theory of Cryptography*, pages 219–234, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. https://www.iacr.org/archive/tcc2011/65970216/65970216.pdf.

[28] A. V. Sambra, E. Mansour, S. Hawke, M. Zereba, N. Greco, A. Ghanem, D. Zagidulin, A. Aboulnaga, and T. Berners-Lee. Solid : A platform for decentralized social applications based on linked data. 2016. http://emansour.com/research/lusail/solid_protocols.pdf.

[29] R. Shrestha and S. Kim. Chapter ten - integration of iot with blockchain and homomorphic encryption: Challenging issues and opportunities. In S. Kim, G. C. Deka, and P. Zhang, editors, *Role of Blockchain Technology in IoT Applications*, volume 115 of *Advances in Computers*, pages 293–331. Elsevier, 2019. https://www.sciencedirect.com/science/article/pii/S0065245819300269.

[30] Y. Sverdlik. Aws outage that broke the internet caused by mistyped command, 2017. https://www.datacenterknowledge.com/archives/2017/03/02/aws-outage-that-broke-the-internet-caused-by-mistyped-command.

[31] P. Tummalacherla. The top 3 issues of the centralized internet, 2019. https://iampremt.medium.com/the-top-3-issues-of-the-centralized-internet-1db59d5e495e.

[32] M. van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan. Fully homomorphic encryption over the integers. In H. Gilbert, editor, *Advances in Cryptology – EUROCRYPT 2010*, pages 24–43, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. https://eprint.iacr.org/2009/616.pdf.