QATAR UNIVERSITY

COLLEGE OF ENGINEERING

CLIENT-BASED CONFIDENTIAL DATA SHARING USING UNTRUSTED CLOUDS

BY

NAHEEL FAISAL KAMAL

A Thesis Submitted to

the College of Engineering

in Partial Fulfillment of the Requirements for the Degree of

Master of Science in Computing

June 2021

COMMITTEE PAGE

The members of the Committee approve the Thesis of
Naheel Faisal Kamal defended on 22/04/2021.

Prof. Qutaibah Malluhi
Thesis Supervisor

Prof. Roberto Di Pietro
Committee Member

Dr. Abdelkarim Erradi
Committee Member

Dr. Elias Yaacoub
Committee Member

Approved:

Khalid Kamal Naji, Dean, College of Engineering

ABSTRACT

KAMAL, NAHEEL, F., Masters : June: 2021, Master of Science in Computing Title: Client-Based Confidential Data Sharing using Untrusted Clouds

Supervisor of Thesis: Prof. Qutaibah Malluhi.

Cloud storage has been used widely by organizations and individuals. However, using known cloud providers is not a solution that can fit the needs of many entities that need to store private and sensitive data. This is due to the fact that the data stored in the cloud is not hidden from the cloud providers themselves. This issue can be critical for example in use cases including the usage of governmental data, health care and patients data, or even for individual users who are careful about their privacy. A simple solution to this problem can be encrypting the data with a symmetric key before uploading it to the cloud and decrypt to reuse. However, this raises several issues including the lack of the ability to share files with different users. The proposed solution tackles the issue of sharing data confidentially by designing and implementing a system that allows encrypted data sharing and revocation between users. The clouds are considered untrusted where all computations are performed on the client-side with no trusted third party. The scheme is analyzed and the implementation is evaluated and compared to existing solutions showing that it outperforms them. Two practical prototypes were implemented using the proposed scheme including a cloud storage application and an IoT cloud system. Those applications show that the work presented in this thesis is applicable in real-life scenarios.

DEDICATION

To my family.

ACKNOWLEDGMENTS

I would like to thank my supervisor, Prof. Qutaibah Malluhi for his great efforts in providing continuous support, advice, and feedback. Suggestions and support from many faculty members of the Computer Science and Engineering department are also strongly appreciated. I deeply appreciate the help and support of my family and friends. Without all of this support, this work would not have been possible.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1: INTRODUCTION

Sharing of data is a common function of most traditional cloud storage systems. However, attempting to confidentially share data on untrusted clouds increases the complexity of the problem. This chapter motivates the issue this thesis is attempting to solve, lists the objective and the contributions of this thesis, and presents an overview of the structure of the rest of the chapters.

1.1. Motivation

Cloud storage comes with a lot of benefits like scalability, sharing, and cost-effectiveness. Because of that, organizations of various sectors are choosing to use cloud storage for different applications [1], [2]. However, relying on a third-party cloud provider might not always be suitable when the stored information is highly confidential.

Secure sharing is a critical requirement of many conventional cloud applications [3]. Examples range from direct file-sharing between cloud storage users to sharing between nodes in IoT systems. IoT systems typically need to share, for example, sensory data that it has collected through cloud storage. This data is then accessed by other nodes to be analyzed or to be used to control any kind of actuator device. Whether it is IoT system data or a shared set of users' files, the data is ideally expected to be hidden from other parties including the cloud providers themselves. However, traditional cloud services do not provide such levels of data confidentiality and user privacy.

One way to secure confidential data on public clouds is to encrypt it with a symmetric cipher. However, this limits the ability to use the data until it is decrypted back. In such a scenario, it is not possible to share data between different users in the system.

Attempting to share one file with one user means that the symmetric key must be shared, which gives the recipient user access to all the encrypted files in the cloud. In addition, the only way to revoke such access is to re-encrypt all files with a new key which can be very expensive. Such a symmetric cipher is not sufficient to allow different users to access different sets of files. Another obvious solution can be using an asymmetric cipher. An asymmetric cipher can be used to encrypt a file with the recipient's public key and decrypted with his/her private key. However, this process needs to be repeated between every pair of users as asymmetric ciphers typically work between two parties only. Thus, neither symmetric nor asymmetric ciphers are good solutions to the problem of confidential sharing in typical settings. Many researchers have approached this issue providing novel solutions as presented in Chapter 2. However, each of those solutions faces different challenges; most notably, the key escrow problem. The key escrow problem is when a system relies on a central trusted or a semi-trusted third party. This third party is usually the cloud provider who is given access to stored user's data.

Several different approaches were used to address the key escrow problem. Most notably, two techniques are found in the literature, certificate-based schemes and decentralized schemes [4]. Certificate-based schemes work by relying on an authority to generate a partial secret key [5], [6]. This key is then modified by the users with some randomness to generate the full key. These kinds of schemes partially solve the key escrow problem as the authority knows a part of the key and it still acts as a central point of failure. Decentralized schemes, also known as multi-authority schemes, do not rely on a single authority [4], [7], [8]. Instead, multiple users in the system act as authorities. Those authorities generate partial secret keys and collaborate to build a full secret key. This scheme improves over certificate-based schemes as it would take as many authorities used to generate the key to expose the full secret key.

This thesis proposes a method that allows data to be shared confidentially. The main idea is to have a different key for every file. Sharing a file, in this case, can be done by sharing the key of that specific file. However, storing a key for every corresponding file can be very expensive for a large number of files. Instead of storing every key for every file, the proposed scheme derives keys when needed. The protocol behind this process is explained in-depth in Section 3.2. The system allows sharing and revoking access to a file between different users seamlessly. All of the system operations rely on client-side computations only with no collaboration, central authority, multi-authority, nor any trusted third-party. Therefore, the system is decentralized and can be easily deployed by any user. Deploying such a system can be extremely beneficial for entities to be able to share confidential data between different sectors (e.g. governments, health care institutions, etc. . .) while using cost-effective public clouds.

1.2. Objectives and contribution

The main objectives of this thesis can be summarized as follows:

1. Designing decentralized encryption and key management scheme that allows sharing and access revocation of encrypted data using client-side computations with no central authority.

2. Demonstrate the applications of the proposed scheme by employing it in two scenarios; cloud storage and sharing in IoT cloud system.

3. Evaluate the system and compare it to existing literature.

The contribution of this work is in introducing a scheme that allows sharing and revocation while being fully decentralized and not dependent on any third party. In addition, practical prototypes of two applications were built on top of this scheme to showcase the capabilities of the protocol in real-life scenarios. These applications function as a proof-of-concept for the proposed scheme. Developers and researchers can use those provided applications with minor modifications to come up with new solutions to match their needs in any scenario that requires confidential data sharing. The scheme and its implementation are also analyzed, evaluated, and compared to other works found in the literature.

1.3. Thesis overview

This chapter introduces and motivates the work. The rest of the chapters are organized as follows. Chapter 2 introduces basic concepts and cryptographic primitives. Related work of systems and encryption schemes is also presented and contrasted to this thesis in Chapter 2. Chapter 3 goes over formal definitions, main scheme construction, and security analysis of the formulated solution. In Chapter 4, the implementation details of the system are described. Applications of that implementation are listed and explained in Chapter 5 that discusses employing the proposed scheme to build prototypes for a cloud storage application and as an IoT system application. Chapter 6 evaluates the proposed scheme and the implemented system. The thesis is then concluded in Chapter 7 that summarizes the main results and proposes future improvements to this work.

CHAPTER 2: BACKGROUND AND RELATED WORK

Secure cloud sharing has been an area of interest for many researchers and inventors. Some researchers have addressed the topic from a purely theoretical point of view by designing new encryption schemes. Others tried to build system implementations for cloud storage and IoT systems that provide such features. Besides, several companies attempted to invent techniques that achieve secure sharing using existing art. For this reason, the work of this research tries to address some aspects that the existing literature lacks by introducing a new scheme that matches the objectives listed in Section 1.2.

2.1. Background

Several researchers in the literature have been trying to solve the issue of secure data sharing on public untrusted clouds. However, those solutions suffer from the problem of having a trusted or semi-trusted third party for keys generation and distribution. Such systems rely on a cryptographic primitive known as broadcast encryption [9]. It is based on the idea that some public parameters are shared in the initial phase of the system setup and only authorized users can decrypt. This method was applied mainly to share encrypted TV content for subscribed users.

Based on broadcast encryption, several cryptographic schemes have been developed throughout the years. Most of them fall under the concepts of Identity-Based Encryption (IBE) [10] and Attribute-Based Encryption (ABE) [11]. IBE has been introduced back in 1984. It works by encrypting the data using keys that are related to the users' identity (e.g. email). ABE was introduced after several years in 2005. The keys in ABE schemes are not only based on the identity, but are also based on the attributes related to the user.

ABE has been used to build many other schemes used for cloud storage applications [12].

Another cryptographic primitive that has been used to allow multiple users to access shared data is proxy re-encryption [13] to allow secure sharing of data. The ciphertext in such schemes is transferred from using one public key to another using the proxy to be able to share data between users. However, this proxy needs to be semi-trusted in order to function.

Using one of the techniques mentioned above can be sufficient to allow encrypted data to be accessed by different users. However, most papers in the literature introduce a third party to the system to manage the keys among the users and act as a central authority [5], [14]–[16]. This can be problematic because the central authority needs to be trusted or at least semi-trusted. It also raises the issue of having a central point of failure. For this reason, some research efforts attempted to decentralize it with the addition of extra features [8], [17].

2.2. Related work

Related work to this thesis varies between different areas of research. Section 2.2.1 presents cloud storage systems that provide secure cloud sharing. Section 2.2.2 shows different implementations of secure data sharing in IoT systems. General cryptographic schemes are then presented in Section 2.2.3.

2.2.1. Systems for secure data sharing

A system named Mona was introduced in 2013 providing cloud sharing with multiple owners with the existence of a group manager [15]. ABE was used to allow secure sharing for cloud computing [18]. This system also allows revoking access to data. However, the revocation is done by the cloud server. Also, a trusted attribute authority is needed for key generation. CloudDocs was another application introduced in 2015 that attempt to provide secure cloud sharing using another system named TrustStore [14]. In 2017, SeShare was another similar work allowing secure cloud sharing [16]. The focus of this work was targeting file updates collision prevention and the authors managed to achieve that using blockchain. Another IBE system was presented in 2019 using Certificateless Hybrid Signcryption [5]. This implementation required no pairing and thus lower cost and complexity. However, a third party is needed to manage the system.

However, all of the previously mentioned implementations require a trusted third party for user management and key distribution. This raises the key escrow problem which arises when an authorized entity is responsible for the management of the system. This problem has two major impacts. First, it allows the authority to have access to the users' information as it generates the keys and distributes them to users. Secondly, it acts as a single point of failure where the system stops in case the main authority goes down.

The work [19] was done for secure sharing with untrusted cloud providers in mind. A new progressive elliptic curve encryption scheme was utilized in this work. It was designed to make it possible to encrypt a file with multiple keys but only decrypt once with one key. Sharing between parties is done by encrypting the already encrypted file by the cloud provider and decrypting by the receiver. This process reveals nothing to the cloud provider. However, this protocol assumes the ability to do computations on the cloud. In addition, the scheme was not implemented but the authors mentioned that the performance of the scheme is not as good as existing schemes.

Secure sharing is an important part of many existing commercial solutions provided by different companies. Because of that, several patents exist on this topic.

Dropbox, Inc. is one of the companies that targeted the issue of sharing. In one of their inventions, they describe the process of sharing a file [20]. Their mechanism of sharing is based on sending the file from the client to the content management system (i.e. online servers). The content management system then generates a link which the user can send to the recipient(s) to allow them to access the file. The described process in this system is a simple method of sharing. However, it describes no particular way of securing the shared data. The only mention of securing user's data is that standard transmission protocols are used to secure the communication channel (e.g. SSL, HTTPS, etc. . .)

Another invention that is assigned to Dropbox, Inc. is a secure protocol for sharing and synchronizing data over local area network (LAN) [21]. The protocol starts with the first client (i.e. the sender) communicating with the content management system to generate an identifier and a secret key associated with the shared folder. The identifier is then announced over the LAN and the second client (i.e. the recipient) sends a request for synchronization. The second client is then authenticated by the first and the data is shared and synchronized securely. This method is good for sharing data between users on a LAN securely while keeping it hidden from unauthorized users. However, the security of this method is limited to the LAN. The cloud management system, which is provided by the cloud provider, is involved in generating the secure key. Thus, the cloud provider still has access to the shared files.

Box, Inc. has also presented efforts on the topic of secure sharing. A patented method was assigned to them focusing on secure sharing and deduplication of data on the cloud [22]. The deduplication of the files is performed first by the cloud server before sharing. Sharing is then done over several steps assuming that the client has an enterprise key. First, the shared file is hashed using a hash function to form a content-based encryption key. The file intended for sharing is then encrypted using the content-based encryption key and stored on the cloud. Next, the content-based encryption key is encrypted using the enterprise key and stored on the cloud. Such a protocol can be considered secure under the assumption that the enterprise key is secured. However, the enterprise key is known by the cloud provider. This means that the cloud provider can decrypt the content-based key and the encrypted files.

Several other companies have been patenting different methods of secure sharing. In a method for securely storing and sharing files assigned to Invenia [23], a client has a username and a password. An asymmetric key pair is then generated and the private key is encrypted with the hash of the password. A file-specific symmetric key is then generated and used to encrypt the file intended for sharing. The file-specific key is then encrypted using the previously generated public key. The encrypted file-specific key is then set as the header of the encrypted file and stored on the cloud server. The problem with such a technique is that it is limited to sharing between only two parties.

Another patented work focusing on secure cloud data sharing was done by MyMail Technology, LLC. [24]. The process of sharing in this method starts by generating a file-specific key associated with the file to be shared. This assumes that the file is already stored encrypted in the cloud server. The file-specific key is then encrypted using the identification key where the identification key to associated with the recipient. The recipient is then notified and the data can be downloaded and decrypted using the recipient's identification key. The issue in this method is that the identification key is known by the cloud server. This means that the cloud provider can access and decrypt the shared files.

Similar work on secure data sharing was patented for Masimo Corporation [25]. This system was intended mainly for the health care sector. However, the system can be applied to other domains. The method of this system is based on having a hierarchy of keys to access a master key. This was done to offer the users alternatives like having a password or a security question. Sharing of files is done in this process by encrypting the data with the recipient's public key and decrypting with the private key once received. In other words, sharing on this system is based on typical asymmetric encryption between only two parties.

Most of the prior art that was found assumes the existence of a cloud server that can generate the keys and/or manages the encryption process. Other techniques were limited to sending and receiving between two parties closer to being typical public-key cryptography methods. This shows that more work needs to be done in the area of secure cloud sharing without exposing the data to the cloud provider.

2.2.2. IoT-based solutions

Secure sharing is a significant part of IoT systems. It is a common scenario in IoT that sensor devices read data and store it in cloud services. These sensory data are then accessed by other devices to be analyzed or by actuator devices to take actions based on the data. This should be done ideally while keeping the data hidden from other parties that should not have access to the data, including the cloud providers. For this reason, several researchers have designed various methods to secure this kind of communication.

Some researchers approached the problem by designing new encryption schemes. Partially Homomorphic Encryption was used to implement sharing with revocation capability [26]. This technique relies on proxy re-encryption where a third party transforms the ciphertext from a public key to another without learning the plaintext. Revocation in this work is done under the assumption that a trusted proxy or a semi-honest cloud exists. Also, the cloud provider is required to perform some computation. This work has been showcased using two applications with smartphones and IoT devices.

Another special-purpose encryption scheme was made to work for IoT devices particularly on the edge [27]. The purpose of this protocol is to reduce the computation of IoT devices. However, the edge servers need to be semi-trusted. This work also supports searching along with data sharing. However, user revocation was not considered in this research.

Information-centric IoT is a kind of network where the information gets cached on the nodes. A secure sharing scheme was designed for Information-centric IoT [28].

This scheme was built on the top of Ciphertext-policy Attribute-Based Encryption (CP-ABE). The problem with CP-ABE is that it requires a dedicated server for attributes management. The work in this research provides a similar scheme to CP-ABE while being distributed and publisher-driven scheme with revocation capabilities. However, a trusted third party is still needed to act as a data sharing authority.

Blockchain has been used to aid in implementing secure data sharing systems in IoT [29], [30]. Blockchain was used as an access control system for IoT data in [29]. This work uses key regression where new keys are frequently generated over time and only the last key is shared. Another work has used blockchain along with proxy re-encryption to dynamically distribute the re-encryption keys [30]. Even though the process of re-encryption does not reveal any information, a third party is still needed to perform the computation. Another big trade-off while using blockchain is that the transactions are very slow. For example, in the latter research, the process of sharing takes around 28.01 seconds [30].

2.2.3. Sharing-enabled encryption schemes

In addition to papers focusing on cloud systems and IoT scenarios, many papers were focused on building novel encryption schemes that can allow secure sharing of data.

Ciphertext-Policy ABE (CP-ABE) was introduced in 2007 [31]. It introduces a new ABE scheme where the policy is stored within the ciphertext. Many researchers have been attempting to make such encryption schemes distributed and decentralized. A distributed ABE scheme was introduced in 2008 [32]. In this scheme, multiple authorities co-exist on the system to distribute the secret attribute keys. However, only one master authority distributes user keys. Besides, this scheme only allows disjunctive normal form attributes.

Later in 2011, a decentralized ABE scheme was introduced [7]. This scheme shares many similarities with the previously mentioned work [32] Improvements were made where any party in the system can act as an authority. Also, if an authority fails, other authorities can still function. In addition, any Boolean formula can be accepted as the attributes. However, the creation of the initial parameters requires a trusted setup.

Another similar scheme based on CP-ABE allowed multiple authorities [33]. Multiple central authorities distribute users' keys and multiple attribute authorities distribute attribute keys. However, the initial setup also needs to be trusted.

Anonymous IBE allowed better revocation for file sharing scenarios [34]. The idea of this scheme is that revocation does not need decryption and re-encryption due to hiding users' identities. However, the revocation process is done by the cloud server.

Certificate-based broadcast encryption is also considered as an anonymous scheme [6]. Having certificates and certificate authorities in the system allows constant decryption cost. Also, certificate authorities know only a portion of the keys in this scheme.

Another work focused on using CP-ABE for lightweight devices [8]. This is done by having a constant key size with optimized ciphertext size and optimized computation time. This paper implements two schemes where one of them supports revocation and the other does not. Each of those schemes has its advantages and disadvantages. Those schemes are implemented, benchmarked, and compared against other schemes found in the literature. A similar decentralized scheme was built as a broadcast encryption scheme [4]. This work improves the performance over the previous scheme.

Blockchain was used to introduce a decentralized privacy-preserving storage and sharing system [35]. This system relies entirely on blockchain to perform its operations. Being a scheme that is based on CP-ABE, an attribute authority is needed in the system. Instead of using a single trusted attribute authority, attribute authorities are considered as special kinds of blockchain users. This design has the advantage of being fully decentralized with no central authority. However, attribute authority users are still needed to generate the attributes for other users. Revocation is also not included in the scheme. In addition, heavily depending on the blockchain can introduce long delays.

Overall, schemes have been improving with time towards more secure solutions. However, schemes that allow both sharing and revocation still rely in a way or another on some third party or a trusted authority.

CHAPTER 3: PROBLEM FORMULATION AND SCHEME CONSTRUCTION

For users U, cloud files F, and file owner A, files are shared with B, where $A \in U$ and $B \subseteq U$. Table 3.1 lists a summary of the various terms used in this thesis.

Table 3.1: Terminology used in the scheme construction

Terminology	Description
A	Data owner user (i.e. Alice)
B	Set of other users (i.e. Bob(s))
U	Set of users where $A \cup B \subseteq U$
B_i	Sample other users (i.e. Bob)
F	Set of files
F_A	Set of files owned by A
F_{Bi}	Set of files shared with B_i
f	Example file where $f \in F$
f'	Encrypted version of f
$H()$	Cryptographic hash function (e.g. SHA)
$E_{sym}()$, $D_{sym}()$	Symmetric crypto functions (e.g. AES)
$E_{asym}()$, $D_{asym}()$	Asymmetric crypto functions (e.g. RSA)
$X\|Y$	Concatenation of variables X and Y
$X - Y$	Delete Y from the beginning of X
k	Encryption key
$"X"$	String value of X (e.g. user name, file name, etc. . .)
$S(X)$	Byte size of X

3.1. Assumptions and constrains

For the system to function, two statements must be valid in all cases:

1. For files F_{Bi} shared by A with every user $B_i \in B$, a user B_i must always have access to files F_{Bi} while keeping those files hidden from other users in U.
2. When a user B_i is revoked from accessing a certain file f, B_i will no longer be able to access f without affecting the sharing state of other users in U.

In order to keep the data hidden from any adversary and from the cloud providers themselves, the statements above must hold a certain context of constraints:

1. The cloud is treated as a passive storage system where only basic file operations can be performed (e.g. downloading, uploading, listing, etc. . .)
2. No coordination is involved and computation is only possible on non-communicating clients.

Having such cloud usage restriction adds the advantage that the system can function on any abstract cloud interface. This includes using simple cloud storage or using a more sophisticated system that allows multi-cloud storage [36].

The lack of central coordination and constraining the system to be fully decentralized transforms the problem of data sharing from a simple operation in traditional clouds to a more complex problem. Another issue that is introduced is the fact the users either get full read and write privileges or none. This is due to the limitation that only basic cloud file operations are possible. This means that preventing an adversary from destroying or corrupting the data is out of the scope of this work. Such issues can be solved by improving the availability like using a multi-cloud storage setup [36]–[38]. Instead, this thesis focuses on constructing a scheme that ensures data confidentiality under the assumptions and constraints listed above.

3.2. Scheme construction

The proposed scheme tries to achieve confidential sharing of data on untrusted clouds. The escrow problem is also an important consideration of the design where no central authority exists in the system. Figure 3.1 presents the general operations that the scheme needs to support, which include adding a file, retrieving a file, retrieving a shared file, and revoking a user. The steps of these operations are formally defined below.

- **KeyGen**: Given A's symmetric key k_A and an initialization vector (IV) IV_f used later while encrypting f, the key k_f for file $f \in F$ is computed as

$$k_f = H(IV_f | k_A) \tag{3.1}$$

  Note that IV_f is used here instead of any other generic file identifier to enable changing the key whenever the file is re-encrypted. This is important for revocation.

- **Encrypt**: Given k_f and a random IV IV_f, we encrypt the plaintext file $f \in F$ to get the ciphertext f' as follows

$$f' = IV_f | E_{sym}(f, k_f) \tag{3.2}$$

Note that the IV is also used for revocation.

- **Decrypt**: Given k_f and encrypted file f'

$$f = D_{sym}(f' - IV_f, k_f) \tag{3.3}$$

Note that in case k_f is not available, it can be reconstructed only by the owner who has k_A. Using IV_f extracted from f', k_f can be reconstructed as shown in the KeyGen section.

- **Share**: In order for A to share file f with B_i, file f is added to the set F_{Bi} (i.e. the set of files shared with user B_i). This process is done given A's symmetric key k_A and B's asymmetric keys k_{Bipriv} and k_{Bipub}.

$$\begin{aligned} T_A &= E_{sym}(\{"B_0" \mapsto F_{B0}, \ldots, "B_i" \mapsto F_{Bi}, \ldots\}, k_A) \\ T_{A,Bi} &= E_{asym}(\{"f" \mapsto k_f, \ldots\}, k_{Bipub}) \end{aligned} \tag{3.4}$$

Where T_A is used by A to keep track of shared files with other users and T_{A,B_i} is provided by A for B_i to access files F_{Bi} where $f \in F_{Bi}$.

- **Decrypt shared file**: Given that f is shared by A with B_i, to decrypt f', B_i downloads f' and $T_{A,Bi}$ and perform the following.

$$\begin{aligned} &D_{asym}(T_{A,Bi}, k_{Bipriv}) = \{"f" \mapsto k_f, \ldots\} \\ &D_{sym}(f', k_f) = f \end{aligned} \tag{3.5}$$

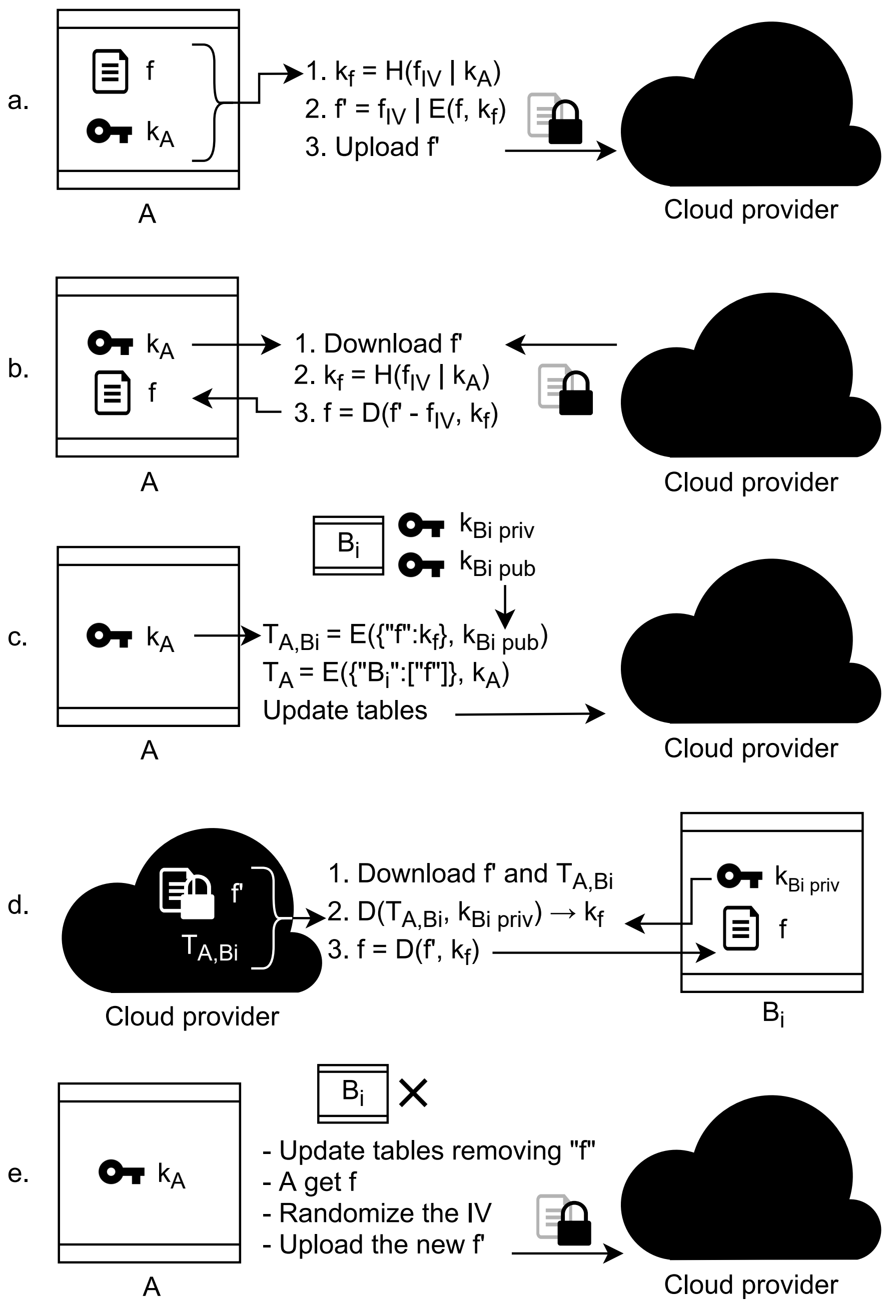
- **Revoke**: To revoke user B_i from accessing file f owned by A

1. A downloads T_A and decrypts it using k_A, i.e. $D_{sym}(T_A, k_A) = \{"B_i" \mapsto F_{Bi}, \ldots\}$.
2. Remove "f" from the F_{Bi}.
3. Encrypt the table again and upload it, i.e. $E_{sym}(T_A, k_A)$.
4. Regenerate $T_{A,B}$ and upload it replacing the old table. Where the new regenerated table does not include k_f.
5. Download f' and decrypt it to get f.
6. Randomize IV_f, generate new k_f, and encrypt again to get f''.
7. Upload f''.
8. Regenerate $T_{A,Bj}$ where $B_j \in$ the set of users A shares the file f with.

Note that removing the table entry is done before the re-encryption of the file because it is assumed that the tables are smaller than the files in most cases. This means that the revocation will be faster in case B is honest. The remaining steps ensure revocation regardless of B's honesty. Also, the steps in this process need to be atomic. A copy of the tables must be kept in case of failure for recovery. The last step is performed to update the new key for other users who have the same file shared by A.

3.3. Security analysis

The scheme is constructed using symmetric and asymmetric ciphers in addition to a cryptographic hash function. Every file f in this model is assumed to have a different key k_f generated using the hash function $H()$ as shown in Section 3.2. This key combines the



a.
f
k_A
1. $k_f = H(f_{IV} | k_A)$
2. $f' = f_{IV} | E(f, k_f)$
3. Upload f'
A
Cloud provider
b.
k_A
f
1. Download f'
2. $k_f = H(f_{IV} | k_A)$
3. $f = D(f' - f_{IV}, k_f)$
A
Cloud provider
c.
B_i
$k_{Bi\ priv}$
$k_{Bi\ pub}$
k_A
$T_{A,Bi} = E(\{"f":k_f\}, k_{Bi\ pub})$
$T_A = E(\{"B_i":["f"]\}, k_A)$
Update tables
A
Cloud provider
d.
f'
$T_{A,Bi}$
1. Download f' and $T_{A,Bi}$
2. $D(T_{A,Bi}, k_{Bi\ priv}) \rightarrow k_f$
3. $f = D(f', k_f)$
$k_{Bi\ priv}$
f
Cloud provider
B_i
e.
B_i
k_A
- Update tables removing "f"
- A get f
- Randomize the IV
- Upload the new f'
A
Cloud provider

Figure 3.1: Scheme operations including a. adding a file, b. retrieving a file, c. sharing a file, d. retrieving a shared file, and e. revoking a user from a shared file

symmetric key k_A of the file owner A and the initialization vector IV_f of the file f. The key k_f is then used to encrypt and decrypt the file using the symmetric cryptographic functions $E_{sym}()$ and $D_{sym}()$. This means that the key generation step is as secure as the used hash function and file access is as secure as the used symmetric cryptographic functions.

Sharing a file f owned by A with a user B_i, in this scheme, is the process of adding the key k_f to a table $T_{A,Bi}$ that maps the files A allows B_i to access to their corresponding keys. This table functions like a capability list of B_i to access files of A. It is also encrypted with the asymmetric cipher $E_{asym}()$ using the public key of B_i, k_{Bipub}. When B_i wants to access f, he decrypts $T_{A,Bi}$ with the asymmetric cipher $D_{asym}()$ using his private key k_{Bipriv} to get the key k_f and be able to access the shared file f. This means that accessing files shared between users is as secure as the used asymmetric cipher. If A decides to revoke B_i from accessing f, she follows the scheme revocation steps where IV_f is regenerated, a new k_f is generated, and f is re-encrypted. Building on top of this model, the assumptions can be listed as follows.

1. Functions $H()$, $E_{sym}()$, $D_{sym}()$, $E_{asym}()$, and $D_{asym}()$ are assumed to be cryptographically secure.

2. Clients trust no party other that themselves.

3. All users have read and write access to the encrypted data stored in the cloud storage.

The main advantage of using this scheme is the ability to **share** and **revoke** access to files between users. For that, the statements below must always hold.

Property 1: When a file f is shared by A with user B_i, the file is always accessible by B_i while staying hidden from any adversary.

Proof: To be able to access file f, its key k_f must be accessible. Those keys are previously encrypted using k_{Bipub} and stored as $T_{A,Bi}$ by A. B_i is the only user who has his private key k_{Bipriv}. Using the asymmetric cipher, B_i can decrypt $T_{A,Bi}$ with his private key k_{Bipriv} and obtain k_f to be able to decrypt f' and get the shared file f. For an adversary to access f, either k_A or k_{Bipriv} must be known. However, those keys are kept private in their owners' local machines. Therefore, f cannot be accessed by any adversary while always being accessible to A and B_i.

Property 2: If A revokes B_i from accessing file f that was previously shared with B_i, the file can never be accessed by B_i unless it is shared again.

Proof: When a file f is already shared with B_i, its key k_f is exposed to B_i. Assuming that B_i is still able to access f after he has been revoked by A, he attempts to download f' and decrypt it with his k_f. However, the decryption results in garbage data. This is because when A revoked B_i, a new IV_f was generated. Consequently, the generated key k_f has changed and the file f is re-encrypted with a new key different than the old k_f owned by B_i. Therefore, by contradiction, B_i can never access f again unless A decides to share it again with him.

CHAPTER 4: IMPLEMENTATION

The implementation of the system is divided into different modules. The core module of the system provides a direct implementation of the scheme described in Section 3.2. This core module functions independently from the cloud storage method and the intended application. Instead, it acts as an abstract interface that other modules can make use of to implement more specific applications. Public key sharing is another module that is discussed in Section 4.3. Other modules of the system are the sample applications discussed in Section 5. The complete implementation is publicly available as a git repository [1]

Table 4.1 shows the specifications of the hardware used for testing and evaluating the implemented software. Table 4.2 lists the commands for the implemented software modules, other than the core, and their description. Assuming that the main program is in the current directory and is named `main.py`, executing commands should be as `./main.py [COMMAND] [ARGUMENTS]`. These commands are further explained in their corresponding sections.

Table 4.1: Hardware specifications of devices used for testing and evaluation

Device	Raspberry Pi Zero W	Intel-based laptop
CPU	ARM1176JZF-S 32-bit @ 1GHz	Intel® Core™ i7-5500U 64-bit @ 2.40GHz
RAM	512MB	16GB
OS	Raspberry Pi OS Lite	Arch Linux

[1] `https://github.com/Naheel-Azawy/ccdsuc`

Table 4.2: List of commands implemented for the prototype applications

Command	Description
`bcpki`:	Control the blockchain public key infrastructure described in Section 4.
`fs-mount`:	Mount storage filesystem as shown in Section 5.1.
`fs-cmd`:	Execute a command on a mounted filesystem.
`iot-server`:	Start the prototype TCP server for the IoT system explained in Section 5.2.
`iot-sens-ldr`:	LDR sensor with raspberry pi.
`iot-act-led`:	LED with raspberry pi.
`iot-sens-sim`:	Simulated IoT sensor for testing.
`iot-act-sim`:	Simulated IoT actuator for testing.

4.1. Cryptographic configuration

The system was implemented using the cryptographic library `pycryptodome` for Python [39]. For the purpose of this system, both symmetric and asymmetric ciphers are needed. So, AES and RSA were chosen for our implementation. In practice, these can be replaced by any other symmetric and asymmetric techniques. The keys sizes selected for AES and RSA are 256-bits and 2048-bits, respectively. AES is set to be used in the counter mode (CTR). Using CTR is particularly useful while reading and writing large a file where only portions of the file are accessed as blocks. The cloud storage application in Section 5.1 benefits from using CTR by enabling seamless filesystem operations. The block size of AES is set to be 16 bytes. This block size is also used as the size of the IV across the system.

The asymmetric cipher needs to accommodate data with arbitrary plaintext size. For this reason, RSA was only used to encrypt a 256-bits random session key. This session key is then used to encrypt the plaintext with AES. The encrypted session key is then concatenated along with the ciphertext generated from AES. To decrypt, the encrypted session key and the AES ciphertext are first separated. Then the session key is decrypted with the RSA cipher. Finally, the session key is used to decrypt the ciphertext.

One issue with this implementation is that it can be inconvenient for the user to keep the three keys. To solve this issue, the keys need to be generated from a passphrase a user can memorize. To generate the AES key, the passphrase is hashed using SHA-256 where SHA-256 generates a 256-bits hash that fits as an AES key. RSA keys, on the other hand, require randomness to be generated. However, the keys in the case of this solution need to be deterministic based on the passphrase given to the user. So, the solution was to modify the random number generator used by the RSA keys generator making it seed from the user passphrase. This is done thanks to the used library which allows passing a custom function as the random number generator to be used internally for key generation. The implemented function uses the system's random number generator that is initially seeded with the user's passphrase. This way, RSA keys can be regenerated deterministically using only the user's passphrase. Using this process is less secure than using randomly generated keys as it moves the complexity to the passphrase. However, it introduces the convenience needed for this prototype.

4.2. Integrity checking

The main focus of this work is preserving the confidentiality of data. However, ensuring the integrity can be relatively easy to add. Message authentication codes (MACs) are usually used to check the integrity of data for various applications [40]. Several traditional options of MACs exist such as HMAC [41] and GMAC [42]. GMAC is widely used not only as a MAC, but as an encryption mode. This is usually referred to as Galois/Counter Mode of Operation (GCM) and is typically used with AES. The system implemented presented in this chapter is constructed using AES with CTR mode. This means that adding authentication to the system can be done by changing the AES mode of operation to one offering authenticated encryption (such as GCM).

Introducing authentication to the system comes with some costs in storage and performance. Using a typical GCM, for example, adds a tag to the encrypted data. The size of this tag varies based on the implementation but it typically ranges between 4 and 16 bytes [39]. Performance can be also affected as verifying the integrity would require going over the whole file. Such an effect can significantly reduce the performance, especially for large files. Therefore, this could lead to losing the important CTR mode property of being able to encrypt and decrypt individual blocks as discussed in Section 5.1. However, some newer authentication techniques have the promise for achieving higher performance [43].

4.3. Public keys distribution

In order to share a file with a particular user, the public key of that user is needed to be known by the sharer. However, sharing the public keys using the cloud provider or any other third party might not be safe. This is because the third party or any malicious user can act as a man in the middle and change the public keys allowing them to access data that is not shared with them. One solution could be to statically add the public keys of the users with whom we intend to share our data. However, this would introduce a lot of hassle every time a new device is added. For this reason, a public key infrastructure (PKI) is needed to dynamically share public keys with digital certificates.

Using a classical PKI has some drawbacks. Requesting a digital certificate from a certificate authority (CA) can be a long process. It also introduces the CA as a third party that needs to be trusted to manage the public keys honestly. Because of that, the implemented prototype relies on a simple blockchain-based PKI. This implementation choice was done to match with the decentralized nature of the scheme in this thesis. Similar techniques can be found in the literature [44]. Etheruem is used with its smart contract to build such systems [45]. The CA interface is implemented in Solidity to implement a simplified certificate structure acting as a simplified version of the X509 standard. A snippet of the smart contract is shown below.

```
struct Certificate {
    uint    version;
    string  valid_to;
    string  public_key;
    address issuer_id;
    string  subject_id;
    string  subject_name;
    bool    exist;
    address wallet_owner;
}
mapping(string => Certificate) certs;
mapping(string => bool)        crl;
```

The smart contract is capable of enrolling, revoking, and verifying a certificate. It can also return a particular certificate, the list of certificates, and the certificate revocation list (CRL). These operations are done by calling the contract functions. The certificates and the CRL are stored in the contract as mappings using the hash of the certificate. In addition, enrolling and revoking certificates are functions that are restricted only to the certificate owner, which is the CA in this system. These operations can be invoked as listed in Table 4.2 by running `./main.py bcpki add`, `get`, and `revoke`.

4.4. Limitations

Prototypes implemented in this work do lack some features that are taken for granted in typical cloud systems. However, different solutions and workarounds can be applied to add more features to the system. One of those features is the ability to search the content of the files. Searchable encryption schemes can be used as the symmetric cipher instead of AES to solve this problem. Several schemes have been developed and discussed in the literature to address this problem [46].

Another issue is the lack of access control. This is due to the assumption that the system deals with a passive server as discussed in section 3.1. One way to solve this issue can be by relying on a cloud server that supports access control. But in this case, the cloud server will have more control over the system. Using multiple clouds can enhance the reliability and thus reduce the risk of corrupting the data by a malicious user [36]. Based on that, multi-cloud storage systems can be used as a workaround to the issue of access control. Some other features were not implemented in this system such as notifying the users when a file is shared or when a shared file is modified. Such features can be implemented in the future without any significant compromise. The system can be also ported to other platforms to support mobile and web platforms.

CHAPTER 5: APPLICATIONS

Based on the implementation of the proposed scheme, applications can take advantage of the core implementation to allow confidential sharing in any cloud-based application. As a proof-of-concept, two applications were developed; secure cloud storage and IoT systems. These applications are bundled together as a command-line program to keep the prototype simple and to the point. The commands implemented can be seen in Table 4.2.

5.1. Cloud storage application

Building a cloud storage application is usually dependent on the API provided by the cloud provider. However, the built system is expected to work with any cloud provider. Well-known cloud storage providers usually provide a desktop sync application. Based on that, a virtual filesystem can be implemented to store encrypted data on the cloud and expose a decrypted view of the data for the user. This means that the system can be easily deployed on top of many existing cloud providers like Dropbox, Box, Google Drive, One Drive, etc. . . In addition, it can be used on top of other systems like SafeDrive where multiple clouds are used [47]. Example usage of this application can be found in appendix B.

5.1.1. User-space encrypted filesystem

Similar ideas can be found in user-space disk encryption techniques [48], [49]. Such systems implement a user-space filesystem acting on two directories. One directory is usually a physical filesystem that stores the encrypted data to be considered as the root directory. The other directory is the virtual mount directory. The mount directory shows a decrypted view of the root directory and cryptographic operations are performed on-demand per data block. This kind of virtual filesystem provides a seamless user experience while keeping the data encrypted in the root directory.

An approach similar to encrypted virtual filesystems was chosen in this application, where the root directory is the cloud sync directory. The mount directory represents the decrypted user's logical view of files. This means that the virtual filesystem will be responsible for encrypting and decrypting data while reading and writing, listing files shared with the users as a logical directory, and performing commands on files such as sharing and revoking.

5.1.2. Filesystem implementation

This implementation is built using filesystem in user-space (FUSE) [50]. FUSE is installed as a kernel module that provides a bridge to the user-space through `libfuse` to implement filesystem operations. Using FUSE to implement a filesystem requires implementing the basic filesystem operations such as opening, closing, listing, reading, writing, etc. . . In this particular application, calling `read()` decrypts the data from the root directory. Calling `write()` encrypts data and writes it to the root directory after reading and decrypting unaligned blocks if any. These operations run on segments of the data given the requested offset and size. In addition, file listing and other filesystem operations are deeply integrated with the sharing system to allow a seamless user experience. Figure 5.1 shows the block diagram of the filesystem implementation with the rest of the system components of one user device. The figure shows `sharing-fs` as the interface controlling the filesystem operations based on the sharing core implementation.

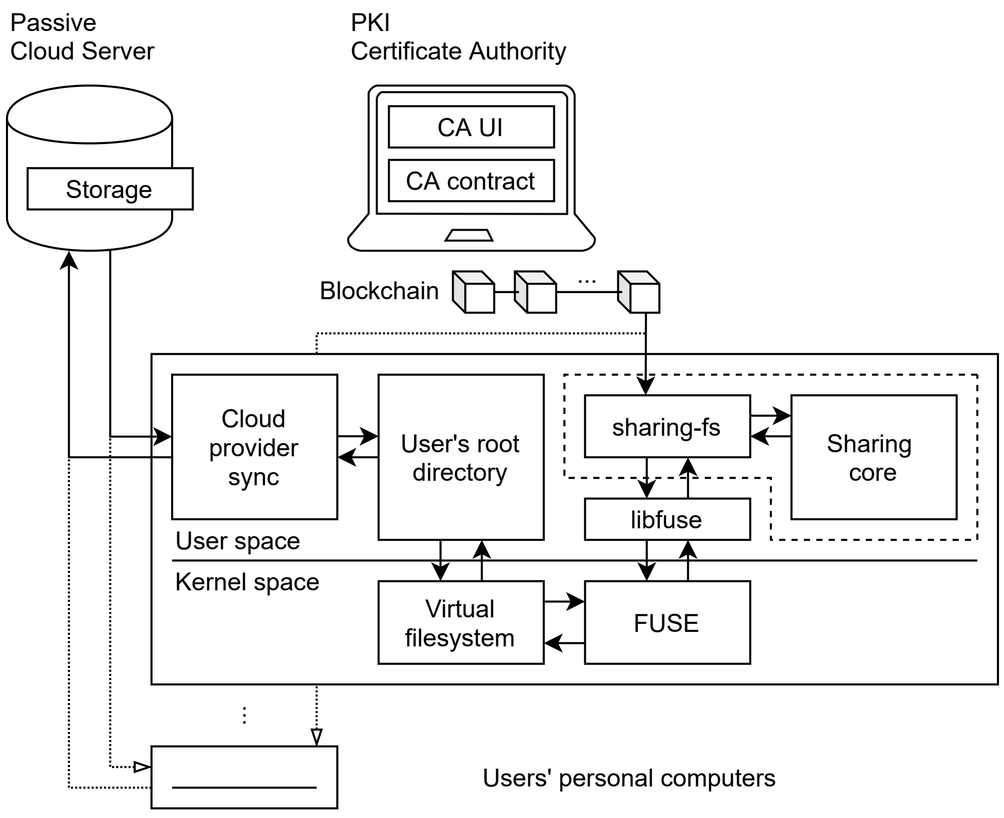


Figure 5.1: Cloud storage filesystem block diagram

Table 5.1: Structure of encrypted files stored in the filesystem

Data	Size in bytes
IV	16
Padding size	1
Encrypted data	Size of the plaintext
Padding	0–15

When a new file f is created, it gets encrypted with its key k_f as described in Section 3.2 with a new randomly created IV. The file is encrypted using CTR mode to allow `read()` and `write()` operations to be run on any given file offset and size without reading the whole file. To make the file size a multiple of the block size, padding is usually added to the end of the file if needed. In this case, the size of the padding can be at most equal to the block size -1 (i.e. between 0 and 15). Table 5.1 shows the structure of how a file is stored. The IV is added to the beginning of the file as it is used to generate the key k_f. Then the size of the padding is specified in cleartext. This is necessary to be able to know the original size of the file without decrypting the file. When f is modified, only corresponding blocks are encrypted and replaced. Unaligned blocks are also decrypted and concatenated to the new data if needed. In case of appending, padding is added as needed and padding size is updated.

5.1.3. Organization of shared files between users

Alice and Bob can be taken as a usage example of this application. Assuming that their public keys have already been added and that files `foo.txt` and `bar.txt` are owned by Alice and Bob respectively. Mounting the filesystem can be done as follows for Alice.

```
$ ./main.py fs-mount /path/to/root/ /path/to/mount-alice/
Enter username: alice
Enter password: 123
Server started at :47585
...
```

The root directory is shown above as `/path/to/root/`. This path is expected to be where the cloud provider sync directory is located. All of the data stored in the root directory is encrypted as explained in Section 3.2. The mount point

/path/to/mount-alice/ is what Alice uses to read and write her files and files shared with her. Knowing that Alice has a file named foo.txt, she can share it with any other user. For instance, below, Alice shares her file foo.txt with Bob and then lists the users that she shares foo.txt with.

```
$ ./main.py fs-cmd share foo.txt bob
true
$ ./main.py fs-cmd ls-shares foo.txt
["bob"]
```

Bob, on the other hand, can log in just like Alice with his username and password. He is expected to have access to the same cloud Alice has used as her root directory.

```
$ ./main.py fs-mount /path/to/root/ /path/to/mount-bob/
Enter username: bob
Enter password: abc
Server started at :45975
...
```

To visualize how Alice and Bob see their files and how they are actually stored in the root directory, below is a tree of the directories root, mount-alice, and mount-bob. The root directory organizes data based on the users' sub-directories. It also stores the sharing tables in a dedicated sub-directory. Every file under the root directory is encrypted as explained in the 3.2 Section and is decrypted on demand under the mount directories. Shared files are listed in the mount directories under shared sub-directory. File foo.txt is an example of that where it is listed under mount-bob.

```
root/                        | mount-alice/
|-- alice                    | |-- foo.txt
|   |-- foo.txt              | \-- shared
|   \-- shared               +-----------------------
|-- bob                      | mount-bob/
|   |-- bar.txt              | |-- bar.txt
|   \-- shared               | \-- shared
\-- __sharing_tables         |     \-- alice
    |-- table_alice_bob      |         \-- foo.txt
    \-- table_alice_others   |
```

To build the prototype of an IoT system that uses the proposed scheme, several components needed to be implemented. Figure 5.2 shows a generic block diagram of the proposed architecture. This implementation differs from the virtual filesystem by being lighter and built to be specifically compatible with IoT scenarios [51].

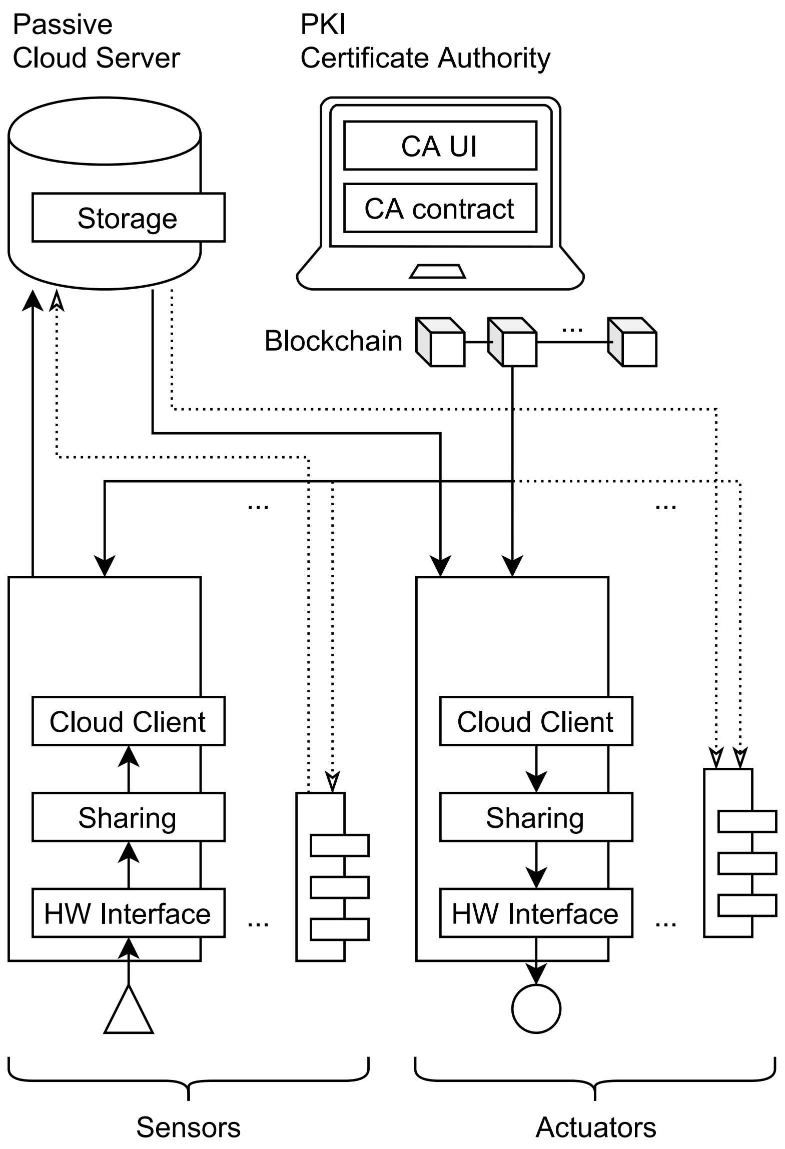


Figure 5.2: IoT system architecture block diagram

A basic passive cloud server is implemented using TCP communication for the purpose of demonstration. This can be replaced with API calls of any chosen cloud provider. One important aspect of the system is that the server performs no computations on the data. It is a passive server that only stores encrypted files and performs basic filesystem operations such as reading, writing, removing, and listing files.

5.2.1. Software interface

IoT devices in the system interface with the hardware sensors and actuators. They read and/or write data from the connected hardware and use the proposed scheme to confidentially store and share the data between devices. The software is installed on the IoT node and the hardware interface can be configured based on the needs of the user. Code snippets for an example sensor and actuator hardware interface are shown below. Implementations of `read_sensor()` and `write_actuator()` can be used to directly communicate with the physically connected hardware. Other methods are used to configure the node properties and their sharing relationships with other nodes.

```
class ExampleDevice(IoTDevice):
    def __init__(self):
        super().__init__(
            device_passphrase = ...,
            device_type       = ...,
            device_id         = ...,
            update_period     = ...,
            log_count         = ...)
    def file_name(self): ...
    def shared_with_list(self): ...
    def read_sensor(self): ...
    def write_actuator(self, values): ...
ExampleDevice().run()
```

5.2.2. Hardware prototype

The hardware implementation of the described design was done using four nodes. Each node is a Raspberry Pi Zero W single-board computer. This particular model was chosen because of its low cost, small size, and onboard WiFi support, which make it suitable for IoT applications. Table 4.1 shows the specifications of the used hardware. Each node was connected to a simple sensor or actuator. The implemented software was installed on all of the nodes and configured to interface with the attached hardware.

As a proof-of-concept, 4 nodes were deployed. Two of them as LDR light sensors and two of them as LED lights (i.e. actuators). Each sensor shares its data with one actuator. Based on the code snippets above, the hardware interface is implemented such that the darker the reading from the LDR, the brighter the LED lights. Those interfaces are named `iot-sens-ldr` and `iot-act-led` as shown in Table 4.2. Figure 5.3 presents the connected prototype. More details about the prototype can be found in appendix A.

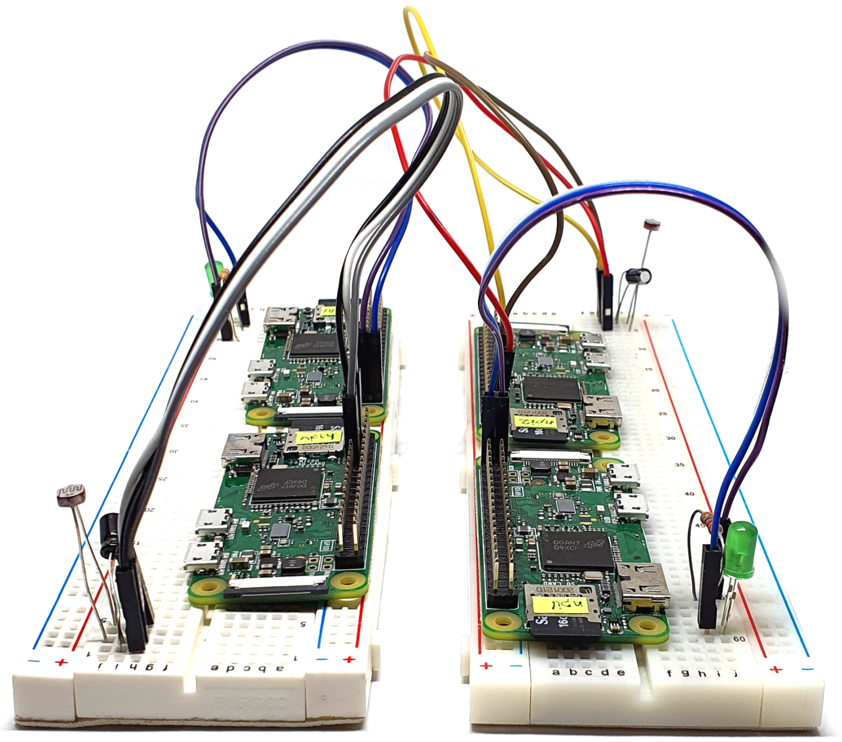


Figure 5.3: IoT system prototype

CHAPTER 6: EVALUATION

This chapter addresses the aspects where this scheme differs from using traditional cryptographic methods in terms of space and time. The overhead is discussed theoretically as well as measured based on the implemented system.

6.1. Public storage overhead

In the set of users U, every user $A \in U$ has a set of files F_A and can share any file $f \in F_A$ with an arbitrary number of other users B_i. Let $b_{A,Bi,f}$ be a bit which has a value of 1 iff A shares file f with B_i, otherwise it has a value of 0. Let $S()$ be a function that returns the size in bytes, k_f be the key for file f, $M_{A,Bi,f}$ be the additional space for the mappings data structure of the tables, and ϵ be the extra bytes (e.g. for padding) needed to encrypt the two tables (i.e. T_A and $T_{A,Bi}$). IV is a value concatenated to the plaintext before encryption as explained in Section 3.2. Equation 6.1 calculates the storage overhead, Θ, in bytes of implementing sharing on the cloud system.

$$\begin{aligned}\Theta = &\sum_{A \in U} \sum_{B_i \in U} \Bigg[\Big(\bigvee_{f \in F_A} b_{A,Bi,f} \Big) \\ &\Big(\sum_{f \in F_A} b_{A,Bi,f} [2.S("f") + S("B_i") + S(k_f) + S(M_{A,Bi,f})] \Big) + \epsilon \Bigg] + \\ &\sum_{u \in U} \sum_{f \in F_u} S(IV)\end{aligned} \tag{6.1}$$

This equation goes over every user A sharing files with other users B_i. The value of the term $\bigvee_{f \in F_A} b_{A,Bi,f}$ is 1 if A shares any file with B_i, otherwise it is 0. $"f"$ is present in both, T_A and $T_{A,Bi}$. $"B_i"$ is only present in T_A and k_f is only present in $T_{A,Bi}$.

To simplify equation 6.1, A is assumed to be sharing all of her x files with y users. This means $x = S(F_A) = S(F_{Bi}); i \in [0, y-1]$. Note that $S(k_f)$, $S(M_{A,Bi,f})$, $S(IV)$, and ϵ are constant parameters of the system, x varies as files are added or deleted, y varies as users join or leave the system, and b changes whenever a file is shared with or revoked from a user. Based on the previous assumption, b can be eliminated. The equation can be rewritten as in shown equation 6.2.

$$\begin{aligned} \Theta = & \sum_{B_i \in B} \Big(\sum_{f \in F_A} \big[2.S("f") + S("B_i") + S(k_f) + S(M_{A,Bi,f})\big] + \epsilon \Big) + \\ & \sum_{f \in F_A} S(IV) \end{aligned} \tag{6.2}$$

To further simplify the equation, $S("f")$, $S("B_i")$, $S(k_f)$, and $S(M_{A,Bi,f})$ are assumed to be fixed and are replaced with $S("f_s")$, $S("B_s")$, $S(K)$, and $S(M)$. Substituting with the variables x and y, the equation can be further simplified.

$$\begin{aligned} \Theta = & y.\Big(x.\big[2.S("f_s") + S("B_s") + S(K) + S(M)\big] + \epsilon\Big) + x.S(IV) \\ = & xy.\big[2.S("f_s") + S("B_s") + S(K) + S(M)\big] + y.\epsilon + x.S(IV) \end{aligned} \tag{6.3}$$

In equation 6.3, fixing y and varying x then fixing x and varying y shows that the cost increases linearly with respect to both, the number of users and the number of files in the system. This can be seen if some variables are replaced with constants as follows.

$$\begin{aligned} & c_1 = 2.S("f_s") + S("B_s") + S(K) + S(M),\ c_2 = \epsilon,\ c_3 = S(IV) \\ & \Rightarrow \Theta = \ xy.c_1 + y.c_2 + x.c_3 \end{aligned} \tag{6.4}$$

Fixing y results in a linear equation of x where the overhead equals to $x(yc_1 + c_3) + y.c_2$. Similarly, fixing x results in a linear equation of y where the overhead equals $y(xc_1 + c_2) + x.c_3$. Further explanation can be found in Section 6.2 with the implemented model.

Figure 6.1 shows the measured storage overhead in the implemented system. It illustrates the storage cost while varying to the number of shared files (x) and the storage cost while varying the number of users, i.e. devices, (y) in the system while having all files shared by one owner with all other users. These results reflect equation 6.3 showing a linear increase with respect to both, the number of users and the number of shared files. An important observation is that the size increases faster while varying y than varying x. This is because increasing y means creating more tables for more users while increasing the size of F only adds additional entries on the existing tables. For example, sharing 100 different files with another user added about 10KB of storage, and sharing one file with 100 different users added about 40KB of storage. Another way to visualize the storage overhead is presented in Figure 6.2. This plot represents the measured sizes as a 3D graph. More extreme cases can be presented in this plot where different numbers of users share different numbers of files. For example, sharing 100 different files with 100 different users adds about 1MB of storage.

Figure 6.3 presents the storage overhead as a percentage of the total storage with respect to different file sizes. For testing purposes, this was done in a scenario where there are 10 files shared with 10 users. The plot starts with a high percentage of almost 100% when the files are too small and decreases rapidly as the file size increases. The percentage drops down to 1% when the size of the files is around 100KB and to 0.26%

when the size of the files is 512KB and keeps getting smaller for larger files.

Such results are expected since the storage overhead is independent of the size of the file itself. Moreover, the overhead is very reasonable in practice since for small file sizes, the storage size is not an important issue. For larger file sizes (i.e., when the storage cost is important), the storage overhead is very small compared to file sizes. Moreover, in practice, the file sizes required usually in applications are not typically very small. Therefore, files can be shared confidentially between users with a negligible storage overhead in the usual use cases.

The data shown in Figure 6.1 reflect the model represented by the equations listed above. Appendix C goes over connecting the measured data with the model equations to find the unknown constants.

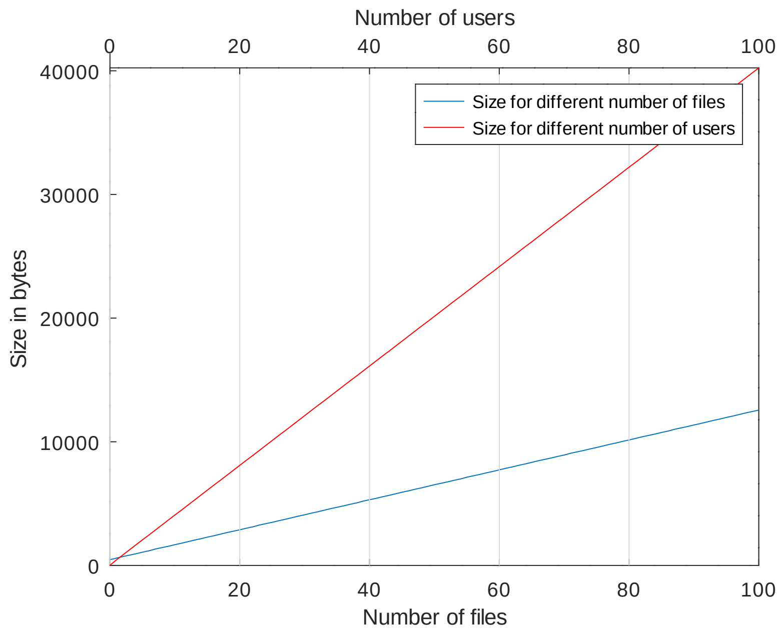


Figure 6.1: Public storage overhead with respect to the number of files and the number of users

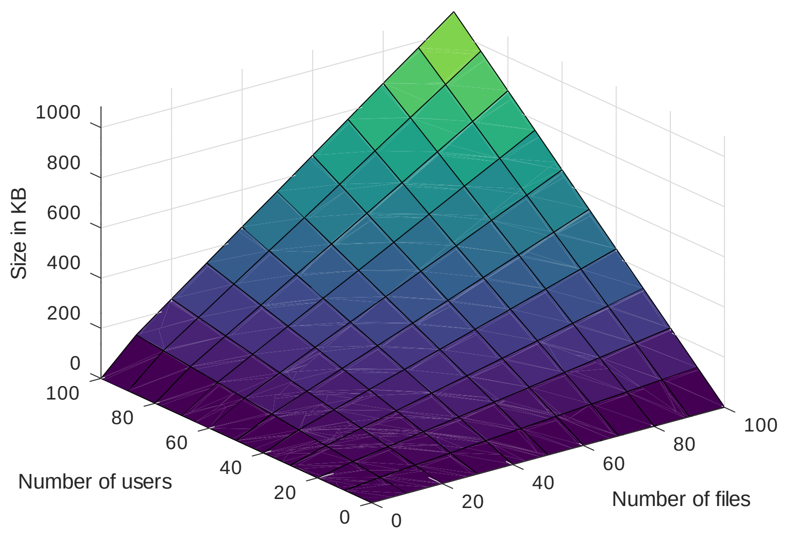


Figure 6.2: 3D representation of the public storage overhead

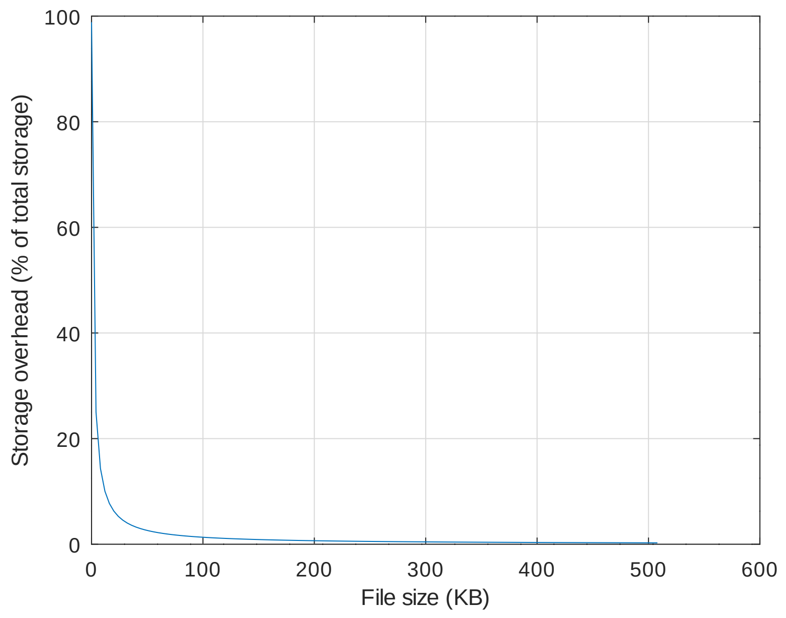


Figure 6.3: Public storage overhead as a percentage of the total storage

6.2. Performance evaluation

Implementing this system is based on the well-known AES and RSA ciphers. However, sharing comes with additional costs in both time and space. The time cost mainly comes out of the key generation before encryption and decryption and the space cost is the size of the sharing tables. The performance measurements were performed on the Raspberry Pi where it is suitable for IoT scenarios as explained in Section 5.2 and on an Intel-based laptop where it is suitable for basic cloud storage scenario as explained in Section 5.1. The specifications of those devices are listed in Table 4.1.

The speed of encryption and decryption is dependent on the library implementation of AES. However, the key generation time is an important factor in the scheme that affects the speed of the process. These keys are generated every time files get encrypted. As it was shown in Section 3.2, the key for file f is $k_f = H(IV_f|k_A)$. The process of concatenating the IV and the key and hashing them adds a small constant extra time. Based on the run tests, the key generation process added about 60 extra nanoseconds on average to the encryption and decryption speeds on the Raspberry Pi and about 1 nanosecond on the Intel-based machine.

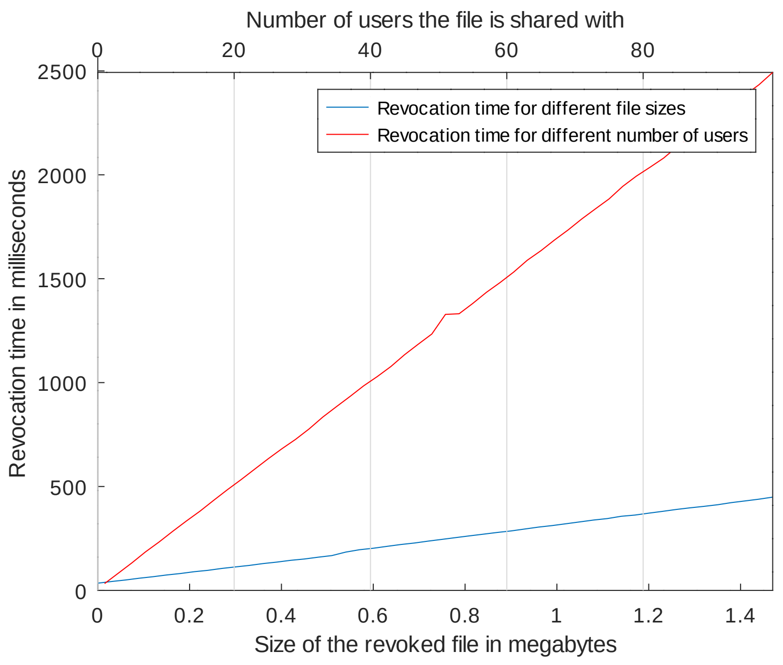


Figure 6.4: Revocation performance with respect to the size of the file and the number of file shares. Performed on the Raspberry Pi

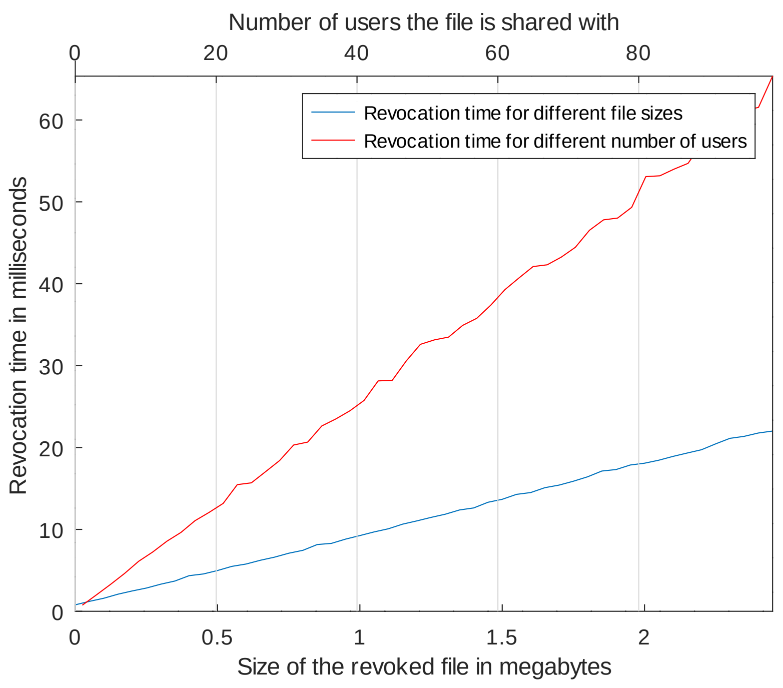


Figure 6.5: Revocation performance with respect to the size of the file and the number of file shares. Performed on the Intel-based machine

Revocation is another important factor in the designed scheme having its performance varying based on different variables. Figures 6.4 and 6.5 show the performance of revocation while changing the file size and the number of users the file is shared with. These metrics were chosen as they directly affect the revocation time. The process of revoking a shared file involves re-encryption of the file by the user owning the file. Because of that, the larger the size of the file the longer it will take to revoke it. Also, the last step of the revocation process, as shown in Section 3.2, is to update the users who have that particular file shared with them with the new key after revoking. This means that the more users the file is shared with, the longer it will take to revoke the file from one user.

The revocation cost can vary based on the network bandwidth in a real-life scenario. Also, this performance cost is reasonable in practice as far as the number of users a file is shared with is not too large. In addition, revoking a user is not a process that happens at a high frequency. For this reason, such a delay may not be critical in real-life scenarios.

6.3. Baseline evaluation

The literature has many examples of schemes that allow secure sharing as shown in Section 2.2. However, each scheme differs in the way it functions and the way it is evaluated. Finding a common base to compare all schemes cannot be done using all the variables in each system. Encryption time, decryption time, and storage overhead size were chosen to evaluate this work against two other related schemes. The first one is a certificate-based scheme [6] and the second is a decentralized scheme [4]. Table 6.1 and Figure 6.6 present the evaluated metrics. Data shown were measured on the Intel-based machine as shown in Table 4.1. The source code of the evaluated software in [4], [6] is publicly available as a git repository [1] and the source of our scheme is also available as described in Chapter 4. Encryption and decryption times are recorded in milliseconds and the storage sizes are recorded in kilobytes. Those metrics are evaluated with respect to y, the number of users, and r, the number of revoked users.

Encryption and decryption in [4], [6] are performed as specialized scheme operations. This increases the complexity of encryption and decryption and results in slower performance. The performance is also affected by y and r in [6]. In [4], y encryption times are almost constant. Decryption times, however, are slightly affected by y and r. Our scheme, on the other hand, can use any symmetric cipher for file encryption. This gives it the advantage of having faster encryption and decryption times that are not dependent on y and r. Both encryption and decryption times are identical because

[1]`https://github.com/tranvinhduc/dbe`

the implemented system uses AES-CTR as shown in Section 4.1. The storage size for all three schemes increases linearly with y. In [4], [6], r has no effect to the storage size. Our scheme shows a small decrease in storage cost with higher values of r. This is because entries are removed for the tables whenever a user is revoked as explained in Section 3.2.

Table 6.1: Performance and storage size baseline evaluation

	[6]			[4]			Ours		
y, r	Enc.	Dec.	Storage	Enc.	Dec.	Storage	Enc.	Dec.	Storage
100,10	13.59	3.23	6.695	2.64	1.98	71.76	0.048	0.038	40.192
100,20	11.87	3.29	6.695	2.70	1.99	71.76	0.055	0.051	39.040
100,30	10.61	3.28	6.695	2.77	2.04	71.76	0.043	0.040	37.872
200,20	25.47	3.44	13.195	2.80	2.17	143.26	0.043	0.038	80.336
200,40	23.53	3.59	13.195	2.83	2.10	143.26	0.060	0.043	78.016
200,60	19.81	3.40	13.195	3.02	2.23	143.26	0.063	0.051	75.696
400,40	49.43	3.22	26.195	2.87	2.09	286.26	0.041	0.040	160.624
400,80	43.85	3.23	26.195	3.50	2.88	286.26	0.045	0.039	155.984
400,120	38.67	3.53	26.195	3.34	2.55	286.26	0.055	0.048	151.344
800,80	96.36	3.32	52.195	4.11	2.37	572.26	0.081	0.079	321.184
800,160	87.87	3.23	52.195	3.59	2.87	572.26	0.077	0.075	311.904
800,240	75.36	3.35	52.195	4.31	3.55	572.26	0.072	0.061	302.624

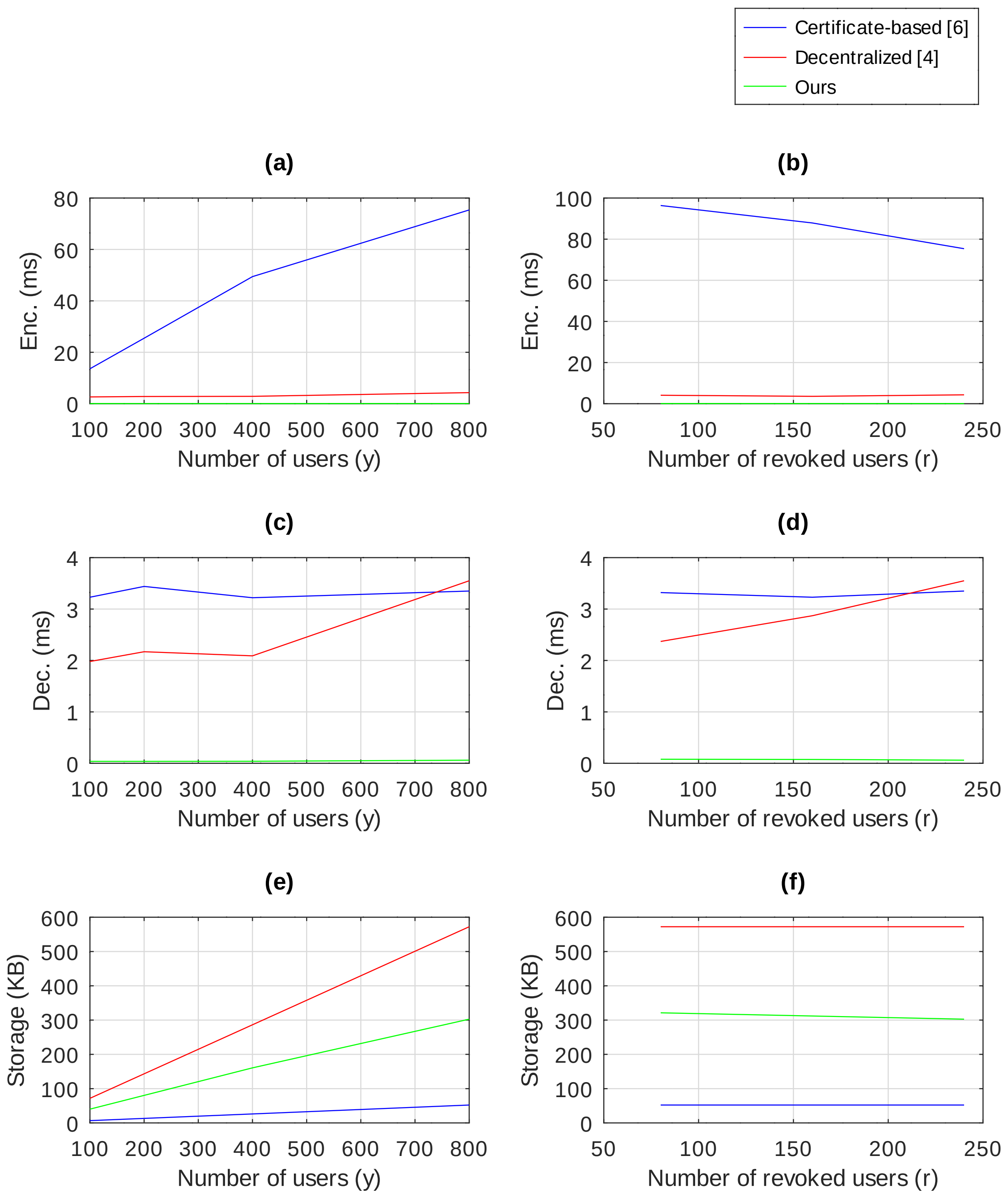


Figure 6.6: Performance and storage size baseline evaluation plots. (a) and (b) show the encryption time (in milliseconds) with respect to y and r, (c) and (d) show the decryption time with respect to y and r, and (e) and (f) show the storage with respect to y and r

CHAPTER 7: CONCLUSION

In this thesis, a scheme of encrypted data sharing was developed. The scheme developed allows sharing and revoking files between users. The major contribution of this work is that all operations performed in the proposed protocol require no trusted third party. The cloud server is treated as a passive storage server and all needed computations are performed on the client side. The protocol has been analyzed and proven to achieve its security objectives.

This scheme has been implemented on top of the well-known AES, RSA, and SHA-256 cryptographic primitives inheriting their cryptographic strength. The implementation includes a complete system presenting a proof-of-concept of the scheme. The core implementation provides an interface to be implemented by applications to be able to introduce confidential sharing of data. Two applications were implemented to showcase the possibilities of the system. The first application is a generic cloud storage interface acting as a virtual filesystem. Such an application can be used by individuals and organizations to securely share files between users on public clouds. The second application is a method to share confidential data between IoT devices. This implementation differs from the first one by being lighter and optimized for IoT applications and by providing an easy-to-use interface to be integrated with any hardware. In addition to these applications, a system of public-key sharing has been built using blockchain. This was added to simplify the process of accessing user's public keys.

Several future improvements to the proposed system are possible. The limitations discussed in section 4.4 can be addressed to improve the system. The scheme could be further improved to allow sharing with groups instead of individual users only. In this case, every group could have a set of keys and other users could share data with that group. Members of a group should then be able to access the group data as well instead of only accessing data shared with them directly. Implementing this can further reduce the storage overhead in scenarios where many groups are needed. Another improvement to the scheme can be developing a new method for revocation that has better time complexity. Hierarchical structures of keys can be used as one to achieve group sharing and simultaneously improve revocation performance.

More research can be done as well on the implementation of the protocol. One direction of improving the system can be in using elliptic curve cryptography (ECC) instead of RSA. Using ECC, smaller keys can achieve an equivalent level of security compared to RSA. This makes ECC a better option for resource constraint applications [52]. Another way to improve the system for resource constraint IoT scenarios is to investigate implementing this protocol on a fog computing setup (a.k.a. on the edge). This could reduce the cost for large numbers of IoT devices and allow more complex computation using the more capable edge devices [53].

Using the scheme proposed in this thesis is not restricted to the research area. Deploying this protocol as a system for production in real-life applications is the ultimate goal of this work. Individuals and organizations can take advantage of this technology in their day-to-day life. Cloud storage and IoT systems are only examples of how this work can be deployed. Many other applications can be tuned to adopt the proposed scheme. One example could be online cloud projects and document collaboration systems. Using the protocol of this thesis can keep the data on the cloud completely hidden from the cloud providers while still being able to function like any traditional collaboration service. Users could be editing the same document on the cloud at the same time by modifying the corresponding encrypted blocks in the cloud. Another example could be building a social media platform where the service providers cannot see the users' data. Several social media platforms already support encryption. However, they still can access data on the servers. This can be solved by using the proposed protocol where the data is only visible to the intended users. This means, for instance, posts that a user shares can be only visible to the followers of that user. This is because the user shares the decryption keys of the posts with his/her followers using their public keys. Unfollowing, in this case, would be equivalent to revoking. Other operations like commenting, liking, etc. can be in similar manners.

This thesis is only a step forward towards more privacy-focused data sharing systems. Many other areas of improvement and applications can be targeted other than what has been mentioned in this section. Such advancements can benefit organizations and individuals by providing a safe way of sharing information without violating their privacy.

REFERENCES

[1] P. P. Ray, “A survey of iot cloud platforms,” *Future Computing and Informatics Journal*, vol. 1, no. 1-2, pp. 35–46, 2016.

[2] P. Yang, N. Xiong, and J. Ren, “Data security and privacy protection for cloud storage: A survey,” *IEEE Access*, vol. 8, pp. 131 723–131 740, 2020.

[3] D. Thilakanathan, S. Chen, S. Nepal, and R. A. Calvo, “Secure data sharing in the cloud,” in *Security, privacy and trust in cloud systems*, Springer, 2014, pp. 45–72.

[4] Q. Malluhi, V. D. Tran, and V. C. Trinh, “Decentralized broadcast encryption schemes with constant size ciphertext and fast decryption,” *Symmetry*, vol. 12, no. 6, p. 969, 2020.

[5] W. Luo and W. Ma, “Secure and efficient data sharing scheme based on certificateless hybrid signcryption for cloud storage,” *Electronics*, vol. 8, no. 5, p. 590, 2019.

[6] J. Li, L. Chen, Y. Lu, and Y. Zhang, “Anonymous certificate-based broadcast encryption with constant decryption cost,” *Information Sciences*, vol. 454, pp. 110–127, 2018.

[7] A. Lewko and B. Waters, “Decentralizing attribute-based encryption,” in *Annual international conference on the theory and applications of cryptographic techniques*, Springer, 2011, pp. 568–588.

[8] Q. Malluhi, A. Shikfa, V. Tran, and V. Trinh, “Decentralized ciphertext-policy attribute-based encryption schemes for lightweight devices,” *Computer Communications*, vol. 145, pp. 113–125, 2019.

[9] A. Fiat and M. Naor, “Broadcast encryption,” in *Annual International Cryptology Conference*, Springer, 1993, pp. 480–491.

[10] A. Shamir, “Identity-based cryptosystems and signature schemes,” in *Workshop on the theory and application of cryptographic techniques*, Springer, 1984, pp. 47–53.

[11] A. Sahai and B. Waters, “Fuzzy identity-based encryption,” in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, Springer, 2005, pp. 457–473.

[12] C.-W. Liu, W.-F. Hsien, C. C. Yang, and M.-S. Hwang, “A survey of attribute-based access control with user revocation in cloud data storage.,” *IJ Network Security*, vol. 18, no. 5, pp. 900–916, 2016.

[13] Z. Qin, H. Xiong, S. Wu, and J. Batamuliza, “A survey of proxy re-encryption for secure data sharing in cloud computing,” *IEEE Transactions on Services Computing*, 2016.

[14] C. Wise, C. Friedrich, S. Nepal, S. Chen, and R. O. Sinnott, “Cloud docs: Secure scalable document sharing on public clouds,” in *2015 IEEE 8th International Conference on Cloud Computing*, IEEE, 2015, pp. 532–539.

[15] X. Liu, Y. Zhang, B. Wang, and J. Yan, “Mona: Secure multi-owner data sharing for dynamic groups in the cloud,” *IEEE transactions on parallel and distributed systems*, vol. 24, no. 6, pp. 1182–1191, 2012.

[16] L. Huang, G. Zhang, S. Yu, A. Fu, and J. Yearwood, “Seshare: Secure cloud data sharing based on blockchain and public auditing,” *Concurrency and Computation: Practice and Experience*, vol. 31, no. 22, e4359, 2019.

[17] D. H. Phan, D. Pointcheval, and M. Strefler, “Decentralized dynamic broadcast encryption,” in *International Conference on Security and Cryptography for Networks*, Springer, 2012, pp. 166–183.

[18] J.-y. FU, Q.-l. HUANG, Y.-x. YANG, *et al.*, “Secure personal data sharing in cloud computing using attribute-based broadcast encryption,” *The Journal of China Universities of Posts and Telecommunications*, vol. 21, no. 6, pp. 45–77, 2014.

[19] G. Zhao, C. Rong, J. Li, F. Zhang, and Y. Tang, “Trusted data sharing over untrusted cloud storage providers,” in *2nd IEEE International Conference on Cloud Computing Technology and Science*, IEEE, 2010, pp. 97–103.

[20] V. Mody, A. Subramani, D. Ryazanov, T. Wen, and S. Cayre, *Automatic file storage and sharing*, US Patent 10,506,046, 2019.

[21] A. Mityagin and D. Litzenberger, *Advanced security protocol for broadcasting and synchronizing shared folders over local area network*, US Patent 10,425,391, 2019.

[22] T. Luthra and R. Malhotra, *Secure cloud-based shared content*, US Patent 10,402,376, 2019.

[23] A. Andersen, O. Pedersen, and T. Wold, *Method for secure storing and sharing of a data file via a computer communication network and open cloud services*, US Patent 9,224,003, 2015.

[24] T. D. Selgas and J. D. Heintz, *Secure cloud data sharing*, US Patent 9,767,299, 2017.

[25] A. Haider and A. Ahmed, *Secure and zero knowledge data sharing for cloud applications*, US Patent 10,608,817, 2020.

[26] H. Shafagh, A. Hithnawi, L. Burkhalter, P. Fischli, and S. Duquennoy, "Secure sharing of partially homomorphic encrypted iot data," in *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, 2017, pp. 1–14.

[27] M. B. Mollah, M. A. K. Azad, and A. Vasilakos, "Secure data sharing and searching at the edge of cloud-assisted internet of things," *IEEE Cloud Computing*, vol. 4, no. 1, pp. 34–42, 2017.

[28] R. Li, H. Asaeda, and J. Li, "A distributed publisher-driven secure data sharing scheme for information-centric iot," *IEEE Internet of Things Journal*, vol. 4, no. 3, pp. 791–803, 2017.

[29] H. Shafagh, L. Burkhalter, A. Hithnawi, and S. Duquennoy, "Towards blockchain-based auditable storage and sharing of iot data," in *Proceedings of the 2017 on Cloud Computing Security Workshop*, 2017, pp. 45–50.

[30] A. Manzoor, M. Liyanage, A. Braeke, S. S. Kanhere, and M. Ylianttila, "Blockchain based proxy re-encryption scheme for secure iot data sharing," in *2019 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, IEEE, 2019, pp. 99–103.

[31] J. Bethencourt, A. Sahai, and B. Waters, "Ciphertext-policy attribute-based encryption," in *2007 IEEE symposium on security and privacy (SP'07)*, IEEE, 2007, pp. 321–334.

[32] S. Müller, S. Katzenbeisser, and C. Eckert, “Distributed attribute-based encryption,” in *International Conference on Information Security and Cryptology*, Springer, 2008, pp. 20–36.

[33] Z. Liu, Z. Cao, Q. Huang, D. S. Wong, and T. H. Yuen, “Fully secure multi-authority ciphertext-policy attribute-based encryption without random oracles,” in *European Symposium on Research in Computer Security*, Springer, 2011, pp. 278–297.

[34] J. Lai, Y. Mu, F. Guo, W. Susilo, and R. Chen, “Anonymous identity-based broadcast encryption with revocation for file sharing,” in *Australasian Conference on Information Security and Privacy*, Springer, 2016, pp. 223–239.

[35] G. Li and H. Sato, “A privacy-preserving and fully decentralized storage and sharing system on blockchain,” in *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, IEEE, vol. 2, 2019, pp. 694–699.

[36] N. Mhaisen and Q. M. Malluhi, “Data consistency in multi-cloud storage systems with passive servers and non-communicating clients,” *IEEE Access*, vol. 8, pp. 164 977–164 986, 2020.

[37] Q. M. Malluhi and W. E. Johnston, “Coding for high availability of a distributed-parallel storage system,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, no. 12, pp. 1237–1252, 1998.

[38] Q. Zhang, S. Li, Z. Li, Y. Xing, Z. Yang, and Y. Dai, “Charm: A cost-efficient multi-cloud data hosting scheme with high availability,” *IEEE Transactions on Cloud computing*, vol. 3, no. 3, pp. 372–386, 2015.

[39] H. Eijs. (2014). “Pycryptodome.” (Accessed on 06/03/2020), [Online]. Available: `https://www.pycryptodome.org`.

[40] M. A. Simplicio Jr, B. T. De Oliveira, C. B. Margi, P. S. Barreto, T. C. Carvalho, and M. Näslund, “Survey and comparison of message authentication solutions on wireless sensor networks,” *Ad Hoc Networks*, vol. 11, no. 3, pp. 1221–1236, 2013.

[41] H. Krawczyk, M. Bellare, and R. Canetti, *Hmac: Keyed-hashing for message authentication*, 1997.

[42] D. McGrew and J. Viega, “The galois/counter mode of operation (gcm),” *submission to NIST Modes of Operation Process*, vol. 20, pp. 0278–0070, 2004.

[43] M. A. Simplicio Jr, P. d. F. Barbuda, P. S. Barreto, T. C. Carvalho, and C. B. Margi, “The marvin message authentication code and the lettersoup authenticated encryption scheme,” *Security and Communication Networks*, vol. 2, no. 2, pp. 165–180, 2009.

[44] A. Yakubov, W. Shbair, A. Wallbom, D. Sanda, *et al.*, “A blockchain-based pki management framework,” in *The First IEEE/IFIP International Workshop on Managing and Managed by Blockchain (Man2Block) colocated with IEEE/IFIP NOMS 2018, Tapei, Tawain 23-27 April 2018*, 2018.

[45] V. Buterin *et al.*, “A next-generation smart contract and decentralized application platform,” *white paper*, vol. 3, no. 37, 2014.

[46] R. Handa, C. R. Krishna, and N. Aggarwal, “Searchable encryption: A survey on privacy-preserving search schemes on encrypted outsourced data,” *Concurrency and Computation: Practice and Experience*, vol. 31, no. 17, e5201, 2019.

[47] W. A. Mansour and Q. M. Malluhi, "Safedrive: A reliable and secure distributed cloud storage," 2020.

[48] V. Gough, *Encfs: An encrypted filesystem for fuse*, 2017.

[49] E. Acri, *Portable aes-ctr encryption using fuse*, `https://crossbowerbt.github.io/fuse_ctr_encryption.html`, (Accessed on 02/19/2021), 2016.

[50] M. Szeredi, *Fuse: Filesystem in userspace*, `http://fuse.sourceforge.net`, (Accessed on 02/19/2021), 2010.

[51] N. F. Kamal and Q. M. Malluhi, "Client-based secure iot data sharing using untrusted clouds," 2021.

[52] D. Mahto and D. K. Yadav, "Rsa and ecc: A comparative analysis," *International journal of applied engineering research*, vol. 12, no. 19, pp. 9053–9061, 2017.

[53] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash, "Internet of things: A survey on enabling technologies, protocols, and applications," *IEEE communications surveys & tutorials*, vol. 17, no. 4, pp. 2347–2376, 2015.

[54] J. W. Eaton, D. Bateman, S. Hauberg, *et al.*, *Gnu octave*. Network thoery London, 1997.

APPENDIX A: IOT CIRCUIT DIAGRAM AND HARDWARE INTERFACE

The prototype of the IoT system shown in Figure 5.3 in Section 5.2 consists of 4 Raspberry Pi Zero W nodes. Two of them have an LED connected and two of them have RC circuits with an LDR. Figure A.1 shows the circuit schematics of two nodes.

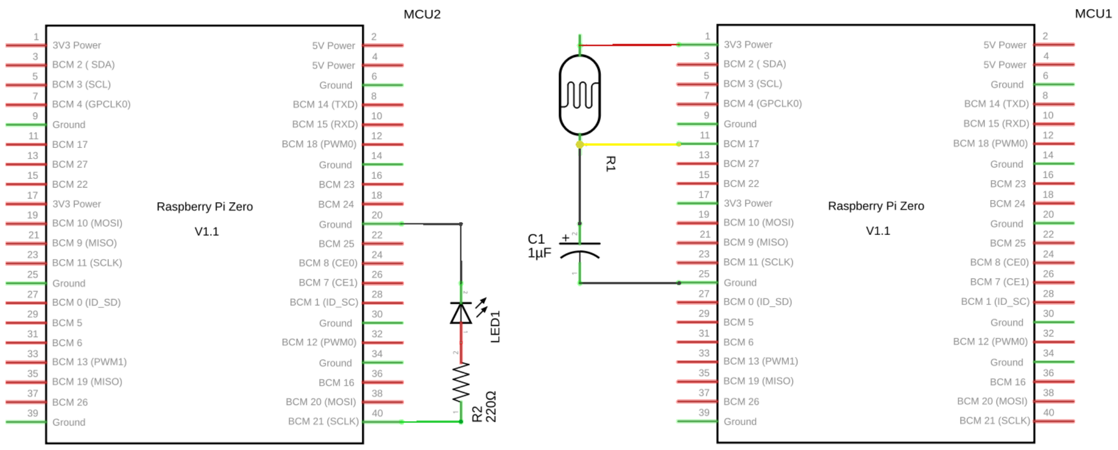


Figure A.1: Schematic of two nodes of the IoT prototype

An RC circuit is needed for the LDR nodes as the Pis do not provide analog inputs. Instead, the RC circuit timing is measured. The `read_sensor()` interface is implemented like what is shown below.

```
count = 0
while (GPIO.input(self.pin) == GPIO.LOW and count <= darkest):
    count += 1
return count
```

The LED nodes implement the `write_actuator(value)` interface by changing the pulse width modulation (PWM) cycle as shown below.

```
self.pwm.ChangeDutyCycle(value)
```

APPENDIX B: CLOUD STORAGE USAGE EXAMPLE

Below is an example showing how to operate with the cloud storage application mentioned in sections 5.1.

First, a test blockchain is started and the smart contract is deployed on it.

```
$ ganache-cli --host 0.0.0.0 --port 8545 &
...
Listening on 0.0.0.0:8545

$ ./main.py bcpki deploy
...
Deploying 'CA'
...

$ export ETH_ADDR='http://localhost:8545@0xF691E550AD7Fc3cB5192...'
```

Adding and revoking users.

```
$ ./main.py bcpki add alice 123 2030-01-01
0x4e1eb5aa1d29cc0761500b6ea925fdb66efa18e350cfcdf027e274c8901d5cb4

$ ./main.py bcpki add bob abc 2030-01-01
0x50334dbbd7087da60ddb099ffabf301d5b878e6d0bc2f74b52efbfcca9d8e7b2

$ ./main.py bcpki add eve bad 2030-01-01
0x2f426bf3a10f6225ce5ef1ac51d30fc043ccfd22a1a416047633e3fffb69b39e

$ ./main.py bcpki revoke eve
0xd4b46238ffb65c9d585f1450de681d7f6fc8a3bb2761a8697bca6a14bfd41ff5

$ ./main.py bcpki ls
['alice', 'bob']
```

Mounting the filesystem to Dropbox directory and signing in as Alice.

```
$ mkdir ~/Dropbox/secure
$ mkdir -p ~/mount/alice
$ mkdir -p ~/mount/bob

$ ./main.py fs-mount ~/Dropbox/secure/ ~/mount/alice/
Enter username: alice
Enter password: 123
Server started at :34941
```

Creating a file by Alice and sharing it with Bob.

```
$ echo 'Hello from Alice' > ~/mount/alice/text.txt
$ ./main.py fs-cmd share text.txt bob
true
$ ./main.py fs-cmd ls-shares text.txt
["bob"]
```

Signing in as Bob and checking the shared file.

```
$ ./main.py fs-mount ~/Dropbox/secure/ ~/mount/bob/
Enter username: bob
Enter password: abc
Server started at :48417

$ tree ~/mount/bob/
/home/naheel/mount/bob/
\-- shared
    \-- alice
        \-- text.txt

2 directories, 1 file

$ cat ~/mount/bob/shared/alice/text.txt
Hello from Alice
```

APPENDIX C: PUBLIC STORAGE OVERHEAD REGRESSION

This appendix lists the steps to find the unknowns in the storage overhead equations mentioned in Section 6.1. From equation 6.3:

$$\Theta(x,y) = xy.\big[2.S(\text{"}f_s\text{"}) + S(\text{"}B_s\text{"}) + S(K) + S(M)\big] + y.\epsilon + x.S(IV) \tag{C.1}$$

x := number of shared files

y := number of users the files are shared with

From Section 4, $S(K) = 32$ bytes, $S(IV) = 16$ bytes

$S(\text{"}f_s\text{"})$ is set to be 32 bytes and $S(\text{"}B_s\text{"})$ is set to be 4 bytes.

Unknowns: $S(M)$ and ϵ

let $c_1 = 2.S(\text{"}f_s\text{"}) + S(\text{"}B_s\text{"}) + S(K) + S(M) = 2*32+4+32+S(M) = 100+S(M)$

let $c_2 = \epsilon$

let $c_3 = S(IV) = 16$

This simplifies to:

$$\Theta(x,y) = xy.c_1 + y.c_2 + x.c_3 \tag{C.2}$$

Fixing y and varying x, let $y = 1$, $m_1 = c_1 + c_3$, and $d_1 = c_2$

$$\Theta(x,1) = x(c_1 + c_3) + c_2 = m_1 x + d_1 \tag{C.3}$$

Fixing x and varying y, let $x = 1$, $m_2 = c_1 + c_2$, and $d_2 = c_3$

$$\Theta(1,y) = y(c_1 + c_2) + c_3 = m_2 x + d_2 \tag{C.4}$$

Using the data shown in Figure 6.1 and GNU Octave [54], linear regression can be performed on the data to obtain m_1, d_1, m_2, and d_2.

```
vs_x = csvread("./benchmarks/test_sharing_x_vs_cost.csv"); % files
vs_y = csvread("./benchmarks/test_sharing_y_vs_cost.csv"); % users

function [slop, intercept] = regression(x, y)
  X = [ones(length(x), 1) x];
  theta = (pinv(X' * X)) * X' * y;
  intercept = theta(1);
  slop = theta(2);
end

[m1, d1] = regression(vs_x(:,1), vs_x(:,2))
[m2, d2] = regression(vs_y(:,1), vs_y(:,2))
```

The results are approximated to

$m_1 = 121$, $d_1 = 347$, $m_2 = 413$, and $d_2 = 56$

From equation C.3,

$$m_1 = c_1 + c_3 \tag{C.5}$$

$$m_1 = [100 + S(M)] + 16 \tag{C.6}$$

$$m_1 = S(M) + 116 \tag{C.7}$$

$$d_1 = c_2 = \epsilon \tag{C.8}$$

From equation C.4,

$$m_2 = c_1 + c_2 \tag{C.9}$$

$$m_2 = [100 + S(M)] + \epsilon \tag{C.10}$$

$$m_2 = S(M) + \epsilon + 100 \tag{C.11}$$

$$d_2 = c_3 = 16 \tag{C.12}$$

Using equation C.7 and $m_1 = 121$, $S(M)$ can be found:

$S(M) = 121 - 116 = 5$ bytes

And ϵ can be found using equation C.8 where $\epsilon = d_1 = 347$ bytes.

The numbers do not perfectly match between the linearly regressed fitted values and the model. For example, using equation C.11, $\epsilon = 413 - 100 - 5 = 308$ bytes which is different than the previous result. Another mismatch can be seen in equation C.12. The equation shows that $d_2 = S(IV) = 16$. However, the regressed values show that $d_2 = 56$. This is again due to the error in the fitted values. Figure C.1 shows that the real data do match the model and is equal to 16 bytes.

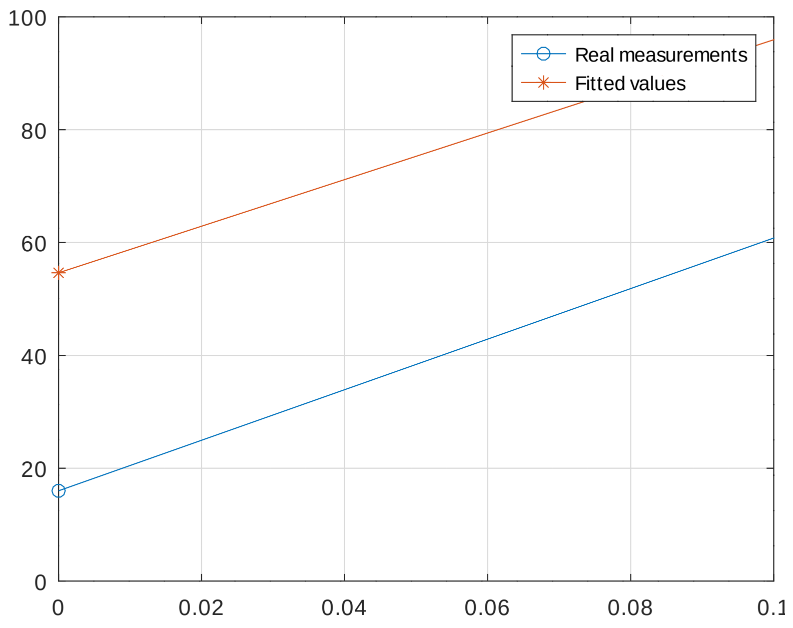


Figure C.1: Difference in the values of d_2 between real and regressed data

This part of the appendix is used to check if the system built-in Chapter 4 matches the model described in Chapter 6. The numbers above show that the model and the implementation do match.