Secure Content Distribution from Insecure Cloud Computing Systems

A Thesis

submitted to the designated

by the Assembly

of the Department of Computer Science and Engineering

Examination Committee

by

Evangelos Dimoulis

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN DATA AND COMPUTER SYSTEMS ENGINEERING

WITH SPECIALIZATION

IN ADVANCED COMPUTER SYSTEMS

University of Ioannina

School of Engineering

Ioannina August 2021

Examining Committee:

- **Stergios Anastasiadis**, Associate Professor, Department of Computer Science and Engineering, University of Ioannina (Advisor)
- **Evaggelia Pitoura**, Professor, Department of Computer Science and Engineering, University of Ioannina
- **Evangelos Papapetrou**, Assistant Professor, Department of Computer Science and Engineering, University of Ioannina

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my research advisor Prof. Stergios Anastasiadis for his invaluable assistance and guidance throughout this thesis. With his deep knowledge on the field of computer systems, he taught me the process of conducting research, how to understand, design and implement systems software. He was always available for long and creative brainstorming sessions, and it was a great privilege to work under his guidance.

I would also like to thank my friends who helped relax and clear my mind after many demanding working hours required to complete this research. Especially, Anargyros Katsoulieris with whom we worked together at the Computer Systems Lab, for the many hours of brainstorming sessions in the field of computer systems security, and his funny jokes that helped making time at the office even better.

Above all, however, I am grateful to my family for their support, love, care and for encouraging me to pursue and accomplish my goals.

Table of Contents

List of Figures

List of Tables

Abstract

Evangelos Dimoulis, M.Sc. in Data and Computer Systems Engineering, Department of Computer Science and Engineering, School of Engineering, University of Ioannina, Greece, August 2021.
Secure Content Distribution from Insecure Cloud Computing Systems.
Advisor: Stergios Anastasiadis, Associate Professor.

The problem we aim to solve in the current MSc thesis is that of secure content distribution in cloud computing environments. Our main goal is to protect sensitive user data stored in cloud storage services from a potentially honest-but-curious cloud storage service provider. We design and implement a prototype file-sharing service that leverages trusted execution environments, with the aid of specialized hardware utilizing Intel's SGX technology. We design secure protocols processing on user data inside isolated memory regions via the use of encryption. To this end, we achieve end-to-end security guarantees, by establishing secure channels between user client applications and the encrypted memory regions managing the data on the provider. We implement the file-sharing service by integrating the proposed protocols with an object storage system. We evaluate the overhead and security of our proposed solution when compared to the baseline storage system without the use of our protocols. In the end of this thesis, we present the findings of our work and propose future improvements.

Εκτεταμενη Περιληψη

Ευάγγελος Δημουλής, Δ.Μ.Σ. στη Μηχανική Δεδομένων και Υπολογιστικών Συστημάτων, Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πολυτεχνική Σχολή, Πανεπιστήμιο Ιωαννίνων, Αύγουστος 2021.
Ασφαλής διανομή περιεχομένου από ανασφαλή συστήματα υπολογιστικής νέφους.
Επιβλέπων: Στέργιος Αναστασιάδης, Αναπληρωτής Καθηγητής.

Το πρόβλημα που επιδιώκουμε να λύσουμε στην παρούσα μεταπτυχιακή εργασία είναι η ασφαλής διαμοίραση αρχείων σε συστήματα υπολογιστικής νέφους. Πρωταρχικός μας στόχος είναι να προστατεύσουμε ευαίσθητα δεδομένα χρηστών που αποθηκεύονται σε υπηρεσίες υπολογιστικής νέφους από έναν δυνητικά περίεργο αλλά έμπιστο πάροχο υπηρεσιών. Σχεδιάζουμε και υλοποιούμε ένα πρωτότυπο σύστημα αποθήκευσης και διαμοίρασης δεδομένων, το οποίο αξιοποιεί τα ασφαλή περιβάλλοντα εκτέλεσης με τη χρήση εξειδικευμένου υλικού της τεχνολογίας Intel SGX. Σχεδιάζουμε ασφαλή πρωτόκολλα τα οποία διαχειρίζονται τα δεδομένα σε απομονωμένες περιοχές μνήμης του συστήματος μέ τη χρήση κρυπτογράφησης. Ως αποτέλεσμα, παρέχουμε εγγυήσεις ασφάλειας από άκρο σε άκρο με ασφαλή κανάλια επικοινωνίας που μεταφέρουν την πληροφορία σε κρυπτογραφημένη μορφὴ μεταξύ των εφαρμογών στην πλευρά του χρήστη και των κρυπτογραφημένων περιοχών μνήμης στον πάροχο. Υλοποιούμε την υπηρεσία διαμοίρασης αρχείων ενσωματώνοντας τα πρωτόκολλα σε σύστημα αποθήκευσης αντικειμένων. Αξιολογούμε το κόστος και την ασφάλεια της προτεινόμενης λύσης σε σύγκριση με το βασικό σύστημα αποθήκευσης χωρίς την χρήση των πρωτοκόλλων μας. Στο τέλος της παρούσας εργασίας, παρουσιάζουμε τα ευρήματα της έρευνας μας και προτείνουμε μελλοντικές βελτιώσεις.

Chapter 1

Introduction

1.1 Motivation

Over the past years, the increase in popularity and easy accessibility of the Internet has led to an accelerated rate data is created. In 2020, approximately 2.5 quintillion data bytes was created every day, and on average every person created at least 1.7 MB of data per second. Interestingly enough, experts estimate that by 2025 people will generate 463 exabytes of data on a daily basis, with most data being related to social media, content sharing, finance and communications in general [1]. Even though a big part of the daily generated data is privately held by individuals, there are manifold cases in which data owners wish to share their information with other people they collaborate with. **Privacy**, generally means the ability of individuals or organizations, to determine how personal information they own is shared with or communicated to others. For example, patients have the right to keep private medical records regarding their health, unless they wish to disclose this information. With the term **data sharing**, we refer to the ability of individuals or organizations to contribute their privately held data with other people. Very common use cases of data sharing include sharing documents, sheets or other related data between employees

and departments of the same company, medical records with a doctor or health institute, or customer transactions using baking services.

However, this kind of data should be kept confidential and only be accessible to people approved by the data owner. **Data confidentiality** is about protecting against the disclosure of information by ensuring that the data is only limited to those authorized or to those who possess some critical information (e.g., keys, passwords). The privacy and confidentiality of online data is usually protected via the use of encryption. Before distributing information, data is transformed into a form so that only authorized parties can understand the information using encryption. Existing encryption techniques rely on strong mathematical models and are widely adopted by systems managing online data.

A great deal of effort has been dedicated by research institutions and companies to efficiently store, manage and distribute online confidential data. Cloud systems can be used to enable data-sharing capabilities over the Internet, and this can provide several benefits to users when the data is shared in the cloud. Leveraging cloud storage for data sharing, users can invite other users to view, edit, or download previously privately held files. Access to the files is feasible through almost any Internet-connected device such as personal computers, tablets or smartphones. Existing cloud storage platforms such as Amazon Simple Storage Service (S3) [2], Dropbox [3] or Google Drive [4] offer high level cloud storage services to millions of users in the world. Using these services, users are freed from the need of possessing high storage capabilities on their devices and bestow the management and security of their data to cloud services. Most of state-of-the-art cloud storage systems leverage the use of encryption techniques to protect the privacy and confidentiality of user data contributed to the cloud.

Despite the security mechanisms cloud providers embed into their systems to secure data, in 2020 the total number of compromised records resulting from data breaches at cloud services exceeded 37 billion leading to a 141% increase compared to 2019 [5]. Most data breaches result from inadequate data security measures, making user data committed to the cloud susceptible to external attackers. Moreover, another serious problem that arises is that data leakage is often caused by the people who operate the systems of cloud storage providers. Privileged employees such as administrators may not actively impose a threat to user data (e.g. destroy data), however a curious employee may view on sensitive user data such as medical records in the

clear, which goes against the privacy needs of users sharing their data online. Thus, a strong need emerges to introduce security measures that do not only protect data stored and shared in the cloud, but to guarantee the privacy and confidentiality of the data in the presence of a curious (but not malicious), or compromised cloud storage provider. The goal of this thesis is to introduce a cloud-based data-sharing service that mitigates these threats.

1.2 Summary of Contributions

In the era of big data, efficiently managing and storing data imposes a demanding task. Secure data sharing in the cloud is a challenging problem with several research effort having been made in this field by research facilities and companies. In our work, we examine an approach for securing data in the cloud with respect to fundamental security primitives and data-sharing requirements. Our ultimate goal, is to share data in the cloud in a way where an external attacker, or even a curious cloud storage provider is not able to gain insight to confidential user data stored on the cloud storage platform.

In this thesis, we introduce secure file-sharing protocols leveraging the use of the Intel SGX technology as a trusted execution environment to protect the confidentiality of user data in the cloud. We provide a prototype implementation of our proposed file-sharing service integrating our solution with the MinIO object store, which is responsible for storing user files in the cloud in the form of objects. Thereafter, we proceed to evaluating the performance of our implemented system. To summarize, our contributions are the following:

- Analysis of secure data storage requirements in object storage systems for secure data-sharing services.
- The design of secure file-sharing protocols leveraging the use of trusted execution environments.
- Implementation of a shielded data-sharing system using Intel's SGX technology over the MinIO object store.
- Performance evaluation and quantification of the overheads introduced by our hardware enforced data-sharing scheme.

1.3 Outline of the Thesis

The dissertation consists of 7 chapters in total.

In chapter 2 we talk about the need for encryption when storing data in the cloud, and present common encryption methods leveraged by cloud providers. After that, we describe how the communication between clients and servers is secured using cryptography. Finally, we discuss about object storage systems for storing data in the cloud, and the security guarantees such systems offer for protecting user data.

In chapter 3 we first discuss how specific hardware modules are used to establish trust in modern computer systems. Afterwards, we argue about the need for secure environments to perform operations on sensitive data. To this end, we present systems that offer secure computation environments and the technology we use in this thesis. To sum up, we present existing research aiming to offer secure data-sharing capabilities in the cloud.

In chapter 4 we analyze the requirements of a cloud-based data-sharing system and list the goals that we have set for our design. Furthermore, we present the entities participating in our proposed system and its overall architecture, we provide details about our trust and threat model, and conclude with describing the file-sharing protocols our systems supports and analyze the security of our design.

In chapter 5 we specify our implementation of the proposed file-sharing service's protocols over object-based storage systems, leveraging the use of trusted execution environments to secure user data. Moreover, we describe the data structures we introduce, and mention the system's components. Finally, we provide significant implementation details on the implementation of our proposed file-sharing protocols.

In chapter 6 we experimentally evaluate our prototype implementation of the proposed file-sharing system. First, we present the system setup and the necessary configurations applied before running our experiments. Then, we evaluate and compare the performance of our file-sharing service with the baseline performance of the object storage system we integrated into our implementation. To conclude, we summarize the results of our experiments and reason about the performance overheads introduced by our hardware enhanced scheme.

In chapter 7, we discuss the main findings we obtained from our research, we indicate the conclusions of our proposed file-sharing service, and we suggest future improvements.

Chapter 2

Background

In this chapter we present an introduction to fundamental security mechanisms achieved through the use of encryption techniques for protecting private user data in cloud computing systems. Furthermore, we describe object storage systems and analyze the security mechanisms applied in the object storage system we utilize for the completion of this thesis.

2.1 Data Encryption in the Cloud

Online data privacy has been one of the most prominent topics in the field of information technology. In today's cyber-world there is an unceasing risk of unauthorized access to all forms of data. Most at risk is sensitive information including payment system data that can expose the personal identifying information (PII), and patient's medical records. Cloud storage allows users to save data and files in an off-site location that they can access either through the public internet or a dedicated private network connection. Encryption is a critical component needed by cloud storage systems providing an additional layer of protection above basic access control, and its main aim

is to secure and protect confidential information as it is transmitted through the Internet and other computer systems. Encryption works by using an *algorithm* with a *key* to convert data into unreadable data (*ciphertext*) that can only become readable again with application of the right key. Encryption algorithms rely on mathematical properties to produce ciphertext that can't be decrypted using any practically available amount of computing power without the necessary key. Next we will present some of the common encryption methods leveraged by cloud providers.

2.1.1 Symmetric Encryption

Symmetric encryption is a type of encryption where only one key (a secret key) is used to both encrypt and decrypt digital information. The entities communicating via symmetric encryption must exchange the key so that it can be used in the decryption process. By using symmetric encryption algorithms, data is converted to a form that cannot be understood by anyone who does not possess the secret key to decrypt it. Once the intended recipient who possesses the key has the message, the algorithm reverses its action so that the message is returned to its original and understandable form. Symmetric encryption algorithms are broken down into two main types: *stream* and *block ciphers*. Block ciphers encrypt data in chunks (blocks), whereas stream ciphers encrypt data one bit at a time.

The most commonly-used symmetric algorithm is the Advanced Encryption Standard (AES), which was originally known as Rijndael, with two variants that operate on 128-bit blocks using 128-bit keys or 256-bit keys. Overall, security experts consider AES safe against brute-force attacks, in which all possible key combinations are checked until the correct key is found. AES is used widely for protecting data at rest. Applications for AES include self-encrypting disk drives, database encryption and storage encryption. A major risk to AES encryption comes from *side-channel attacks*, which aim at collecting leaked information from the system (e.g. monitoring the cipher's shared use of the processor's cache tables). AES comes with a variety of operation modes. In this thesis we leverage AES with Galois/Counter Mode (AES-GCM) [6]. AES-GCM provides both authenticated encryption (confidentiality and authentication), and the ability to check the integrity and authentication of additional authenticated data (AAD) that is sent in the clear. Implementation and usage details will be given in Chapter 5.

2.1.2 Public-Key Cryptoystems

Public-key cryptography, or asymmetric cryptography, is an encryption scheme that uses two mathematically related, but not identical keys, a public key and a private key. In contrast to symmetric cryptography, each key performs a unique function. The public key is used to encrypt messages and verify digital signatures, while the private key is used to decipher encrypted messages. Digital signatures serve as proof to the recipient of a signed message that the message originated from the sender. The private key is held confidential, whereas the public key is given to any party who wishes to securely communicate with the private key's holder. The most deployed asymmetric key block cipher is the *Rivest-Shamir-Adelman* (RSA) [7] algorithm. The idea behind public-key cryptosystems is attributed to Whitfield Diffie and Martin Hellman. The Diffie-Hellman [8] key exchange algorithm was the first widely used method of safely exchanging keys over an insecure channel. The main purpose of the Diffie-Hellman key exchange is to securely develop shared secrets that can be used to derive keys. These keys can then be used in conjunction with symmetric-key algorithms to transmit information in a secure manner.

2.1.3 Homomorphic Encryption

Traditional encryption algorithms, such as AES, are generally fast allowing data to be stored conveniently in encrypted form in the cloud. However, in order to perform even simple analytics on data, either the cloud server needs access to the secret key, which leads to security concerns, or the owner of the data needs to download, decrypt, and operate on the data locally which can be a costly series of operations. *Fully homomorphic encryption*, or simply *homomorphic encryption* refers to a class of encryption methods constructed by Craig Gentry [9], that differ from typical encryption methods, allowing computation to be performed directly on encrypted data without requiring access to a secret key. Using homomorphic encryption, the data owner encrypts their data and sends it to the cloud server. The server may perform computations on the data without decrypting it, and sends the encrypted results to the data owner. Thus, only the data owner is able to view the computed results in clear, since they alone possess the secret key. Nonetheless, despite the novel way homomorphic encryption operates on encrypted data current homomorphic encryption schemes are assumed to be relatively slow, and require large memory compared to plaintext operations

considering them impractical for some use cases (e.g. database queries).

2.1.4 Data Re-Encryption

Data re-encryption is an encryption scheme that enables secure data sharing and confidential data sharing in the cloud. Re-encryption allows a user to convert a ciphertext under the data owner's secret key into another ciphertext that can only be decrypted by other user's secret key. Existing re-encryption techniques involve the use of hybrid encryption schemes [10] where: (1) *symmetric encryption* techniques to encrypt the actual data due to enhanced performance, and (2) *public-key cryptography* for stronger security guarantees.

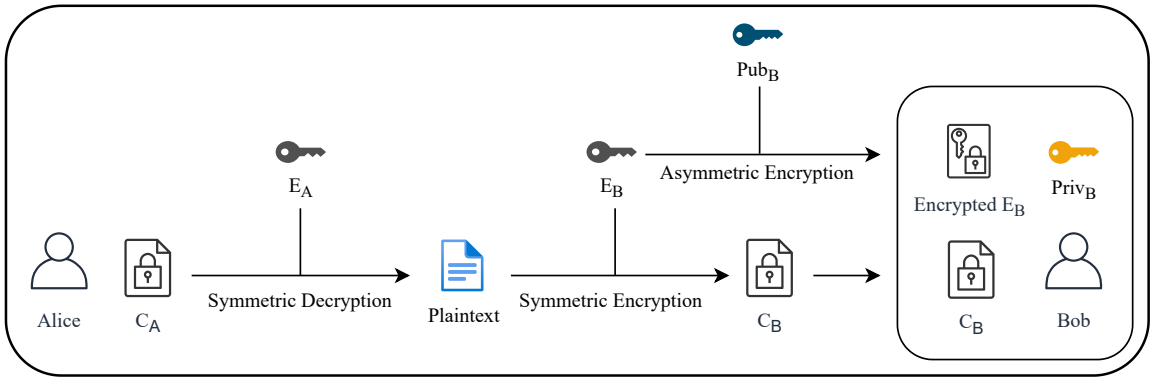


Figure 2.1: Data sharing example using a hybrid re-encryption scheme.

Figure 2.1 illustrates a basic re-encryption example combining symmetric and asymmetric cryptography. A user, Alice, wants to share her data with another user, say Bob. Alice owns an encrypted file called C_A which is encrypted using a symmetric key algorithm and the encryption key E_A, and Bob owns a public-private key pair (Pub_B,$Priv_B$). Alice also owns a public-private key pair, which is not depicted to simplify the procedure followed in the example. First, Alice decrypts C_A and produces the underneath plaintext of the file. Afterwards, Alice encrypts the original file with a newly generated symmetric key E_B and produces a new ciphertext called C_B. Then, the symmetric encryption key E_B is encrypted using Bob's public key (Pub_B). The re-encrypted file and the encrypted symmetric key are sent to the receiver Bob. Bob is now able to decrypt the encryption key E_B using his private key ($Priv_B$), and with it decrypt C_B. Finally, Bob has access to the original file owned by Alice, and using this particular re-encryption scheme Bob is the only user able to gain access to the file through the use of his private key, which he keeps secret at all times.

2.2 Transport Layer Security (TLS)

The primary use of TLS is encrypting the communication between web applications and servers, such as web browsers loading a website. TLS evolved from a previous encryption protocol called Secure Sockets Layer (SSL), which was developed by Netscape. HTTPS is an implementation of TLS encryption on top of the HTTP protocol which is used by all websites as well as other web services. Any website that uses HTTPS is therefore employing TLS encryption. TLS accomplishes the enforcement of three fundamental security primitives: 1) **encryption** obfuscates the data that is sent from one host to another, 2) **authentication** ensures that the parties exchanging information are who they claim to be, and 3) **integrity** verifies that the transferred data has not been forged or tampered with. In order to establish a cryptographically secure data channel, the connection peers must agree on which cipher suites will be used and the keys used to encrypt the data. The TLS protocol specifies a well-defined handshake sequence to perform this exchange. During a **TLS handshake** [11], the client and server corporate to do the following:

1. Agree on the version of TLS (TLS 1.0, 1.2, 1.3, etc.) they will use.
2. Decide on which set of encryption algorithms (e.g., RSA, Diffie-Hellman) they will use for establishing a secure communication channel.
3. Verifies the identity of the server (sometimes also the client) via the server's public key and the SSL certificate authority's digital signature.
4. Use asymmetric encryption techniques to generate shared session keys used for symmetric encryption of messages, after the handshake is complete.

2.2.1 WolfSSL

The wolfSSL [12] embedded TLS library (formerly CyaSSL) is the SSL/TLS we use in the implementation of our prototype. wolfSSL is a lightweight open-source SSL/TLS library written in ANSI C targeted at embedded, RTOS (Real-Time Operating System), and resource-constrained environments, primarily because of its size, speed, and feature set. It works seamlessly in desktop, enterprise, and cloud environments offering closs platform support. wolfSSL supports industry standards up to the current TLS 1.3, is up to twenty times smaller than its widely used alternative OpenSSL [13],

offers a simple API detailed in the wolfSSL manual [14], an OpenSSL compatibility layer, and is backed by the wolfCrypt cryptography library. The wolfCrypt cryptography engine supports cryptographic services leveraged by wolfSSL such as RSA, Diffie-Hellman, AES-GCM, Random Number Generation etc. Recent advents [15] of wolfSSL include integrating TPM 2.0 (Trusted Platform Module) support, providing API calls to the underlying TPM 2.0 module which will be described in the next section. wolfSSL includes a port for Intel® SGX (Software Guard Extensions) with Linux, isolating the wolfSSL library from potentially malicious application running on the host machine, which we make use of in our implementation.

2.3 Object Storage Systems

Designing and implementing a cloud storage service requires enforcing security mechanisms to safely handle user data. However, another crucial step involves determining the type of storage system to utilize so as to efficiently store data. There are three main types of data storage: object storage, file storage, and block storage. File storage is a hierarchical storage methodology involving the existence of a file system organizing data as a single piece of information, namely files. On the other hand, block storage breaks up data in blocks, allowing data to spread across multiple platforms. The solution we propose in this thesis integrates object storage as the main storage system. Object-based storage systems [16] process data as objects when compared with the other state-of-the-art distributed storage system, blocks and files. Both files and blocks are converged to be called Objects. An object has variable length, and can be used to store all types of data, such as files, database records, audio/video images, medical records. A single object could even be used to store an entire file system or database. Unlike block I/O, creating objects on a storage device is accomplished through an interface similar to a file system, allowing object storage systems to easily manage files and metadata.

There are several reasons for which to prefer an object-storage-based solution to store massive volumes of data. Scalability and reduced complexity are two of the main advantages of object storage. Objects are stored in a structurally-flat data environment within the storage cluster. By adding more servers to an storage cluster, higher throughput and additional processing capabilities can be supported given that

object storage also removes the complexity that comes with a hierarchical file system with folders and directories. Furthermore, object storage goes hand in hand with cloud or hosted environments that deliver multi-tenant storage as a service allowing many companies or departments within a company or institution to share the same storage repository [17].

2.3.1 Amazon Simple Storage Service (S3)

Amazon Simple Storage Service (Amazon S3) [2] is an object storage service that offers industry-leading scalability, data availability, security and performance. Amazon S3 provides easy-to-use management features in order to organize data and configure finely-tuned access controls to meet specific business, organizational, and compliance requirements, In Amazon S3, objects are uploaded and stored into collections referred to as *buckets*. Amazon S3 provides RESTful APIs for creating and managing buckets which requires writing code and authenticating the requests. The requests are initiated and executed via clients. Alternatively, managing buckets and objects can be achieved using high-level S3 commands provided by command line tools used to establish communication to an Amazon S3 instance.

Amazon S3 Objects

Amazon S3 is essentially an object store that uses unique key-values to store as many objects as a client desires to. Objects can be handled across one or more buckets and each object can be up to 5 TB in size.

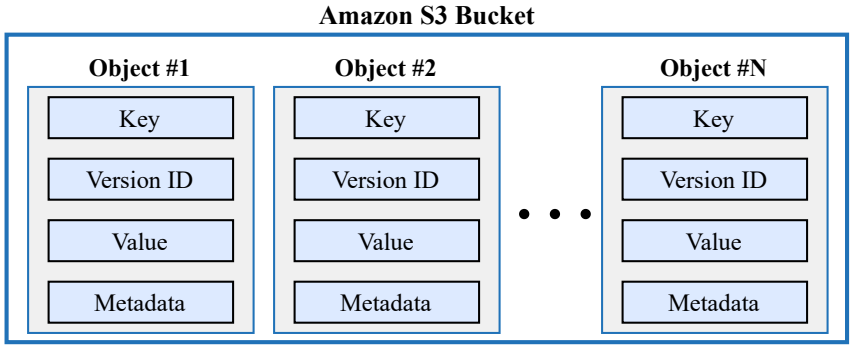


Figure 2.2: A collection of objects stored in a Amazon S3 bucket.

Figure 2.2 illustrates an example bucket which consists of a collection of total N distinct objects. Each object is composed of the following (basic) attributes which

uniquely describe the object in a bucket:

- **Key**: The name that you assign to an object. You use the object key to retrieve the object.
- **Version ID**: Within a bucket, a key and version ID uniquely identify an object.
- **Value**: The content that is stored. An object value can be any sequence of bytes. Objects can range in size from zero to 5 TB.
- **Metadata**: A set of name-value pairs with which information regarding an object can be stored.

2.3.2 MinIO

MinIO [18] is a high-performance object storage system released under Apache License v2.0 and written in Google's Go programming language [19]. It has API compatible with Amazon S3 cloud storage service. MinIO is used to build high performance infrastructure for machine learning, analytics, and various application data workloads. MinIO provides different configuration of hosts, nodes, and drives depending on the needs and infrastructure of the organization hosting the object storage service. There are three distinct types of deployment modes for MinIO: a) Standalone, b) Distributed, and c) Cloud Scale. For the purpose of this thesis, we integrate and leverage the Standalone deployment mode of MinIO.

Accessing MinIO Server with TLS

In order to securely access MinIO server, all communication between a client and a MinIO server instance is protected via TLS. To securely establish communication, we need to provide a private key and a public certificate that have been obtained from a certificate authority (CA). Alternatively, if the server is used for development or research purposes and not for actual production release, the hosting provider can generate a self-signed certificate using the OpenSSL library [13] which is full-featured toolkit for the TLS and SSL protocols. First, the administrator generates an SSL configuration file named openssl.conf, stating information such as the location of the provider, domain name, IP address etc. By using the OpenSSL library the administrator generates the private key and the public certificate, the certificate is

signed using the private key and both files are stored under the MinIO configuration directory.

Server Side Encryption (SSE)

The communication between client and MinIO server is securely established through TLS/HTTPS protocol. Therefore, all data transmitted through HTTP requests is encrypted and integrity protected. However, upon successfully uploading an object into a bucket the data resides in plaintext without any form of encryption, making the data publicly accessible to any client that gains access to the MinIO server even with simple collaboration with the storage provider. In order to further protect the privacy of sensitive data we want to securely store, MinIO supports two different types of Server-Side-Encryption (SSE) schemes:

- **SSE-C**: The MinIO server en/decrypts objects using a secret key provided by the S3 client as part of the HTTP request headers. SSE-C requires TLS/HTTPS.
- **SSE-S3**: The MinIO server en/decrypts objects with a secret key managed by a Key Management System (KMS).

MinIO uses a unique, randomly generated secret key per object known as *Object Encryption Key* (OEK). Neither the client provided SSE-C key nor the KMS-managed key is directly used to en/decrypt an object. Instead, the OEK is stored as part of the object metadata next to the object in an encrypted form. To en/decrypt the OEK another secret key is needed also known as *Key Encryption Key* (KEK). To summarize, for any encrypted object there exists three different keys:

- **Object Encryption Key (OEK)**: A secret and unique key used to encrypt the object, stored in an encrypted form as part of the object metadata and only loaded to RAM in plaintext during en/decrypting the object.
- **Key Encryption Key (KEK)**: A secret and unique key to en/decrypt the OEK and never stored anywhere. It is (re-)generated whenever en/decrypting an object using an external secret key and public parameters.
- **External Key (EK)**: An external secret key, either the SSE-C client-provided key or the KMS-managed key.

For content encryption MinIO server leverages AES-256-GCM. Any secret key is 256 bits long. The secret key is never stored by the server however, it resides in RAM during en/decryption process. The client provided key is not required to be unique and multiple objects may be en/decrypted using the same secret key.

Chapter 3

Trusted Execution Environments & Related Work

In this chapter we argue about trust in modern computer systems, present hardware-based solutions for this purpose, and introduce the technology used in this thesis.

3.1 Establishing Trust in Commodity Computers

Throughout the past decades, numerous software solutions employing the use of strong encrypting algorithms have been deployed to protect computer security. However, software is vulnerable by its nature and writing secure software is a difficult task even for experienced developers. Businesses and individuals entrust progressively greater amount of security-sensitive data to computer platforms. Data breaches pose enormous threats to the privacy of individuals and the integrity of companies whose responsibility it is to safeguard sensitive information. Before entrusting a secret to a computer, a user needs some assurance that the computer can actually be

trusted. Measuring code identity and establishing trust requires some functional *root of trust* [20]. One way to bootstrap trust in a computer is to use secure hardware modules embedded by CPU and hardware vendors to monitor and report on the state of the software stack running on the platform. In a *trusted boot* a hardware-based root of trust initiates the chain of trust by measuring the initial BIOS code, which in turn measures the bootloader, and finally the bootloader measures and executes the operating system. With that in mind, we have to note that a trusted boot does not guarantee the trustworthiness of the software that is running, but merely that it must be trusted because the platform itself is considered to be trustworthy.

3.1.1 Trusted Platform Module (TPM)

The Trusted Computing Group (TCG) has the goal of developing and promoting open standards for trusted computing. The main contribution of the TCG is the specification for the Trusted Platform Module (TPM). TPM technology is designed to provide hardware-based, security-related functions. A TPM chip is a security chip embedded in both personal computers and servers, designed to carry out cryptographic operations. The chip provides a hardware-based tamper-resistant environment in which malicious software is not able to tamper with the security functions of the TPM. Most common TPM functions include generating, storing, protecting encryption keys, and the primary scope of TPM is to ensure the integrity of the platform. In a trusted platform, the TPM provides: **attestation** mechanisms reporting the platform state, and the **sealing** operation using the platform state to authorize access to keys and data.

During the boot process of a system, the firmware and the operating system components can be measured and recorded in the TPM. The integrity measurements can be used as evidence for how a system started and to make sure that a TPM-based key was used only when the correct software was used to boot the system. TPM-based keys can be protected by making the keys unavailable outside the TPM to mitigate phishing attacks, or they can also be configured to require an authorization value to use them. If too many incorrect authorization guesses occur, the TPM will activate its dictionary attack logic and prevent further authorization value guesses. The TPM 1.2 standard was incorporated into billions of PCs and server platforms, and the increasing demand for security has led the TCG to develop a new TPM specification. To this end, TCG designed TPM 2.0 [21] with a library-oriented approach supporting

flexibility of application developing, allowing users to choose applicable aspects of TPM functionality for different implementation levels and security.

3.2 Overview on Trusted Execution Environments

TPM's main shortcoming is that it does not provide an isolated execution environment of arbitrary code as its functionality is reduced to a set predefined APIs. Large projects like the Linux kernel have million lines of code exposing a large attack surface to adversaries, however no software based security technique ensures full protection. Thus, there is need for an extra layer of security, in order to prevent sensitive content compromise. *Trusted Execution Enviroments* (TEEs) [22] are encrypted and integrity-protected memory regions, which are isolated by the CPU hardware from the rest of the software stack, including privileged system software, such as the OS or Hypervisor. TEEs enable the construction of shielded environments in which sensitive application code and data is isolated and integrity protected. In the continuation of this section we present the most prominent commercially available TEE implementations.

3.2.1 AMD Memory Encryption

AMD has recognized the serious challenges arising by the increase in system complexity and has developed memory encryption technologies to address the need for protecting complex computing environments. **Secure Memory Encryption (SME)** [23] is a general-purpose mechanism, integrated into CPU architecture for main memory encryption. SME is performed via on-die memory controllers each including an AES engine for encryption. Encryption is done with a 128-bit key, and the key is managed entirely by the AMD Secure Processor (AMD-SP). With *Full Memory Encryption* all DRAM contents are encrypted providing strong protection, while *Partial Memory Encryption* selectively encrypts only a subset of memory if desired by the OS or Hypervisor. AMD introduces **Secure Encrypted Virtualization (SEV)** to support encrypting virtual machines. SEV integrates main memory capabilities with the existing AMD-V virtualization architecture to protect virtual machines from physical threats, other virtual machines hosted on the same system, and against privileged software including the hypervisor. Software is executed in secure environments which are typically hardware VM constructs (or even Docker containers), cryptographically

isolated from the host system leveraging AES 128-bit encryption. The security of SEV depends on the security of memory encryption keys. The SEV firmware administers a secure key management interface enforcing three main security properties: 1) *authenticity* of the platform, 2) *attestation* of the guest, and 3) *confidentiality* of the guest's data. Nevertheless, a major drawback is SEV is that the system needs to create a seperate VM or container for each application the guest launches.

3.2.2 Arm TrustZone

ARMS's TrustZone [24] is a collection of hardware modules that can be used to enable trusted computing built into every modern ARM processor. TrustZone partitions a system's resources between two worlds: a *secure world* which hosts a secure container, and the non-secure execution environment referenced as *normal world*, in which the untrusted software stack runs. The trust of a computation system is usually bootstrapped from some elemental *root of trust*, that is typically the secrecy of a private key. However, TrustZone does not directly provide any kind of root of trust but a system-wide isolation of the two environments. TrustZone's Trusted Computing Base (TCB) includes a boot loader, which initializes the platform, sets up the TrustZone hardware to protect the secure container from untrusted software, and loads the normal world's bootloader. To switch between the Secure and Non-Secure World, TrustZone introduces a secure state to the ARM architecture, the Secure Monitor. This mode determines whether the system is operating within the secure or non-secure world. The *Secure Monitor Interrupt* (SMI) instruction is used to implement transitions between the two worlds. Software for a TrustZone-enabled device consists of both non-secure elements, such as the OS and common applications, and the protected software components including the Secure Monitor, secure drivers and boot loader. The software architecture supports flexibility allowing the secure world to range from being a security sensitive process (e.g. pseudo-random number generation),even to a full operating system. Developers are allowed to incorporate their own application-specific security measures through a number of key API's publicly available. Common use cases of TrustZone include mobile devices such as smartphones and the protection of smartphone applications that handle sensitive information (e.g., health apps, financial software).

3.2.3 Sanctum

Sanctum [25] introduces a straightforward software/hardware co-design exposing resilience against an important class of software attacks, and adds protection against attacks that infer information from memory access pattern leaks, such as cache timing attacks. Sanctum is mostly implemented in trusted software targeting an open RISC-V architecture allowing researchers to reason about its security properties, and does not perform cryptographic key operations. Sanctum uses a cache partitioning scheme, where a computer's DRAM is split into equally-sized contiguous DRAM regions. Each region is allocated to exactly one isolated container referred to by the author as an **enclave**. Enclaves are protected from compromised system software through isolation, manage their own page tables mapping their DRAM regions and handle their own page faults. Sanctum security relies on a trusted **security monitor** that operates on the highest privilege level (machine level in RISC-V), and is responsible for verifying the system's resource allocation decisions. However, Sanctum does not protect against hardware based attacks as its threat model targets exclusively software attacks.

3.3 Intel Software Guard Extensions (SGX)

By the end of 2015 Intel released processors containing Intel's Software Guard Extensions (SGX) technology [26]. Intel's SGX is a set of extensions to the Intel architecture that aims to shield code execution against attacks from privileged code and certain physical attacks. Similarly to ARM Trustzone [24] or Sanctum [25], SGX's main responsibility is to provide integrity and confidentiality guarantees to security sensitive computation performed on a computer where all privileged software (e.g., kernel, hypervisor, infected operating system) is potentially malicious.

3.3.1 Intel SGX Background

Commodity applications usually tend to have a large attack surface due to the inclusion of hardware, hypervisor, OS, and the application itself in the *Trusted Computing Base* (TCB). A unit of application code protected by SGX is called an *enclave*. Enclave is the trusted execution environment embedded in a process. Application code can be

put into an enclave via special instructions and software made available to delvelopers via the Intel® SGX SDK [27]. The TCB of an SGX enclave is composed of the CPU of the platform, and the code running within. As a result, the TCB is restricted to the limits of the CPU and the enclave interface, reducing the attack surface. The assumption is that we trust Intel for securely implementing SGX.

To implement this concept, SGX introduces a new set of instruction to the x86 architecture. It allows the BIOS to allocate a memory region restricted for processor usage called the *Processor Reserved Memory* (PRM). Enclave data and code are placed in a processor reserved memory area called the *Enclave Page Cache* (EPC), which is a subset of PRM. To ensure its confidentiality, the EPC is kept encrypted by the *Memory Encryption Engine* (MME), a CPU component which ensures that enclave data runs clear only within the CPU limits. The encryption key is randomly generated by the CPU, is changed at each power cycle and never crosses its boundaries. Unauthorized enclave memory requests are blocked by the CPU and treated as non-existent memory addresses. Thus, only the enclave has access to its own information. Furthermore, operations involving the EPC are checked by the CPU utilzing the *Enclave Page Cache Map* (EPCM) structure. EPCM is essentially a matrix that contains an entry for each EPC page, listing information whether a page is being used, the enclave which the page belongs to, and the type of the page. Thus, loading multiple enclaves at once is enabled without conflicts among the memory regions belonging to each enclave.

3.3.2 Enclave Signing

To ensure that only authentic enclaves will be able to run, Intel provides an *enclave signing mechanism*. It is a self-signed certificate with enough information for the SGX architecture to identify tampered or modified enclaves. The main information is the *enclave measurement*, a 256-bit hash of enclave code and data. The signature structure contains a hash of the author's public key, a product ID, and a security version number (ISVSVN) used for application updates. An enclave's measurement is computed using a secure hash algorithm, so the system software can only build an enclave that matches an expected measurement. The SGX design uses the SHA256 secure hash function to compute its measurements, and each enclave's measurement is stored to a dedicated register named MRENCLAVE. Another register, called MRSIGNER is also reserved to hold the author's key hash. This signed enclave signature is verified

in the target system during the enclave initialization. The measurement data that is obtained during enclave launch is used to verify the signature. If they are identical the enclave will be able to load successfully. If there is any modification detected in the measurement value, signature mismatch will occur and the enclave will not be allowed to run. Enclave signatures are generated using Intel's *enclave signing tool*.

3.3.3 SGX Application Architecture

The application of an Intel SGX application is split into parts: a secure one and a non-secure one. Access to a specific enclave by the untrusted application is done through a set of pre-defined functions by the enclave author. *ECALLs* are the interface that the untrusted application can use to call executions within enclaves. Instead, *OCALLs* are a set of declared functions that the enclave uses to access the non-secure world. Both types of functions are defined in a special file using the *Enclave Definition Language* (EDL). This file contains function declaration, similar to the C programming language but with special attributes to specify the direction, size, and type of data that will cross enclave's boundaries. Using this file, a special tool (*Edger8er*) generates the edge routines to safely manipulate the forementioned parameters.

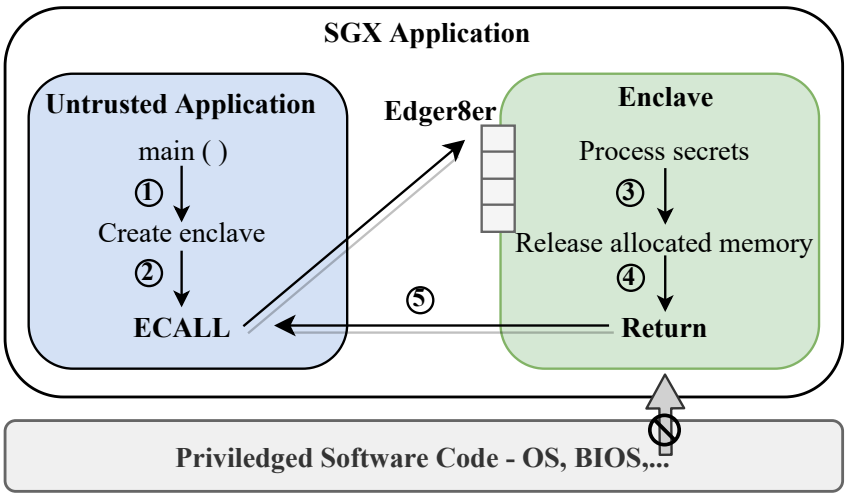


Figure 3.1: Intel SGX application architecture.

To begin with, the untrusted application's main function creates an enclave (1), and once it needs to process on sensitive data, an ECALL is used to transfer the execution to the enclave (2). The enclave executes the trusted function call to process secrets (3), releases allocated memory buffers after the processing occurs(4), and returns the execution flow to the untrusted application (5). System resources are

not directly accessible inside enclaves, and enclaves cannot execute functions from dynamic libraries either. If any of these actions is imperative, an OCALL should be performed triggering an enclave exit. During enclave operation, privileged software such as the OS, BIOS or the Hypervisor do not have access to the enclave code and data. At the end of its operation, the enclave is destroyed and its data is erased.

3.3.4 Data Sealing

During the operation time of an enclave, the safety of data and secrets placed within the enclave is protected, but once the enclave is destroyed data safety is not guaranteed. Intel SGX introduces a *data sealing mechanism* which is the process of encrypting enclave secrets for persistent storage in disk. When protecting data using cryptography, a potential security issue arises when it comes to the secret keys storage. However, the SGX architecture performs data sealing using a private seal key unique to platform running an enclave. The CPU generates a unique 128bit AES-GCM key at runtime to seal or unseal data. The key is unknown to any other entity and never leaves the processor boundaries, thus key storage is no more a relevant issue. The enclave developer is responsible for attaching the sealing key to either the MRENCLAVE or to the MRSIGNER register. A sealing key attached to MRENCLAVE restricts access to sealed data only to instances of the enclave that performed the sealing operation, while MRSIGNER allows all enclaves on the same platform and signed by the same authority to unseal the data. Data sealed by a specific CPU can only be unsealed by this particular CPU on the same platform.

3.3.5 Attestation

Attestation is the process of demonstrating that a piece of software has been properly instantiated on the platform, and proving your identity to a relying party. Intel provides a way for enclaves to *attest* each other. After the attestation process, enclaves will be sure that each other is running code that they are meant to execute. Furthermore, the attestation process results in a secure channel through which the participants can exchange secrets with each other. Remote attestation is supported through an anonymous group signature scheme called *Intel Enhanced Privacy ID* (Intel® EPID). In the case of Intel SGX, a group is a collection of Intel SGX-enabled platforms. Utilizing EPID, each group member is given a unique private key for signing. Signatures gen-

erated by a member of the group are verified using the group's public key. During attestation, the processor's provisioned EPID signature is validated, establishing that it was signed by a member of a valid EPID group. Next we describe the two forms of attestation Intel currently supports, *local* and *remote*.

1. **Local Attestation**: Local attestation allows two enclaves on the same platform to attest each other. The enclaves perform a local message exchange employing the platform's *Root Seal Key* to generate a shared secret (called *Report Key*) for symmetric authentication. Authentication is performed using the Intel SGX report mechanism by applying a report based Diffie-Hellman key exchange. The successful result of local attestation, offers a secure channel between two enclaves of the same platform, that can corporate with each other to perform higher level functions.

2. **Remote Attestation**: Remote attestation [28] involves generating a report that can be verified by any report party. The enclave generates a report (enclave measurements enclave attributes, software version etc.) that summarizes the enclave and the platform state. Next, the enclave local attests to a special enclave called the *Quoting Enclave*, sending it the report. The Quoting Enclave verifies and signs this report and returns it to the enclave application. The signed report is called a *quote*. The quote contains the data listed in the report but is signed with a private key for Intel's EPID remote attestation. Verifying these signatures involves contacting the *Intel Attestation Service* (IAS), though in principle this could be done by any verifier that has the group public key. As part of the attestation process, it is possible to provision the enclave with secrets. They will be securely transmitted to the enclave only if the remote attestation process was successful.

3.3.6 Memory Limitations in SGX

One of the critical limitations of SGX is the capacity of the protected memory region that is provided by the EPC component. Hardware and performance overheads of securing the memory has led to limit the EPC to 128 MB (consisting of 4KB page chunks) in current SGX implementations (e.g. SGX_LKL [29]) for all the enclaves of a platform, which is relatively small considering the large main memory capacities

of server systems. Applications can approximately use 90 MB of the EPC. The EPC region allows efficient fine-grained accesses with cacheline-granularity encryption and integrity protection by the MME. If enclave memory usage exceeds the EPC limit, some pages are evicted from EPC, and remapped to the non-EPC memory region. Accessing data in the enclave memory pages mapped in EPC involves a costly demand paging step which maps the page back to the protected region, causing significant performance penalties.

3.3.7 Vulnerabilities

Intel SGX does not consider side channel or reverse engineering attacks in its threat model. Side-channel attacks exploit implementation details of an algorithm to learn unauthorized information via side channels, as they are not monitored by the system, while reverse engineering attacks aim to understand the application execution though deductive reasoning. Several side-channel attacks that the SGX design is susceptible to include cache-timing attacks [26] and page-fault attacks. Additionally, Foreshadow [30] attacks and Spectre [31] attacks have been announced leveraging out-of-order execution to subvert the confidentiality of SGX enclaves. These kind of attacks take advantage of speculative execution bugs embedded in modern Intel processors, and ultimately aim to extract and steal SGX keys (e.g., Root Seal Key, Attestation Key) by leaking enclave contents. This allows for example, an attacker to access private CPU keys used for remote attestation, or even decrypt secrets that were previously encrypted using the Root Seal Key.

3.4 Related Work

Securing and sharing data over untrusted storage services has been a demanding research challenge over the past years. Since the emergence and wide adoption of cloud computing systems for data storage and sharing, significant research effort has been contributed to design and implement systems that facilitate secure data sharing, and enable fine-grained access control to encrypted data.

3.4.1 Secure Data Sharing on Untrusted Storage

Secure file systems like **Plutus** [32] leverage the use of cryptographic primitives to protect and share data as files in untrusted environments. In Plutus, all user data is stored in an encrypted form and Plutus features scalable encryption key management, where keys are handled in a decentralized manner. Cryptographic operations and key management operations are performed by clients, allowing users to manage access control to their stored files, thus users are assumed to be frequently online. Files in Plutus are divided into several blocks, and each block is encrypted with a symmetric key called *file-block key*. File-block keys of the same file are kept in a *lockbox* that is protected via encryption. Plutus groups files into *filegroups* so that keys are shared among files in a file group, and each filegroup is associated with an RSA key pair. Requirements for server trust are almost eliminated in Plutus (the server ought to preserve and not delete data) delegating secret keys distribution to individual data owners.

3.4.2 Multitenant Access Control

Dike [33] is an authorization architecture aiming to facilitate multitenant access control for distributed filesystems deployed in the cloud in a scalable and secure manner, using virtual machines. Dike's design introduces secure protocols to natively support multitenant access control compatible with object-based filesystems. An attestation server is used to bootstrap system trust, and tenants that wish to access the shared filesystem use a *tenant authentication service (TAS)* to authenticate clients and users. Public keys are used to identify the system's entities, and tenants are identified though their TAS keys. Users access shared files through the use of *data tickets*, that are issued by clients on users behalf. The data tickets contain *handles* to files and the *permissions* applying to the requesting user, and access is further restricted by the use of *access control lists (ACLs)*. Moreover, Dike introduces a *multiview authorization methodology* to selectively grant access of metadata to tenants. Inheritance of access permissions is also supported to simplify file access, using *tree permissions* for folder permissions and file permission for the files in the folder.

3.4.3 Cryptographically Enforced Access Control

Access control systems like **Sieve** [34] allows end-users to selectively expose their private data stored in the cloud to third party services. In Sieve, users upload their privately held data to a cloud store, and initially only the owner of the data has access to it and is in possession of the data encryption keys. Users define *access policies* to control access to the their data. Using *attribute-based encryption (ABE)* where encrypted data is associated with attributes, Sieve translates policies into cryptographically-enforced restrictions. Additionally, Sieve introduces *key homomorphism* which is a technique to refresh encryption keys. Key homomorphism leverages homomorphic encryption to re-encrypt data in situ at the storage provider without leaking the new encryption keys. Finally, Sieve protects cryptographic secrets against user device loss and is compatible with legacy web applications.

3.4.4 Data Sharing using TEEs

The recent advent of hardware-based trusted execution environments, has led researchers to put on significant effort in order to implement systems that securely share data in the cloud, by integrating TEEs in their design.

ShieldStore [35] is a secure in-memory-key-value store designed for shielded execution using Intel's SGX technology. In ShieldStore, the authors propose a hash-based key-value store that is designed for SGX and overcomes the memory restrictions of SGX. ShieldStore maintains the majority of data structures used in key-value stores (e.g., pointers, keys) in the non-enclave memory region by encrypting each key-value pair inside enclaves with a 128-bit global secret using AES counter-mode encryption. Only the encrypted and integrity-protected data are placed in the non-enclave memory, and data is read in the clear only inside the enclave.

IRON [36] is a practical implementation of *functional encryption (FE)*, which is a cryptographic tool that allows authorized entities to compute on encrypted data, and learn results in the clear. IRON holds a master secret as root for key derivations. IRON consists of: (1) a *single trusted authority (Authority)* that is responsible for secrets provisioning, and (2) several *decryption node platforms*. Decryption nodes operate a decryption enclave receiving client requests indicating a particular computation, and forward client requests to *function enclaves* that compute on user data by loading the related function code.

IBBE-SGX [37] is a cryptographic access control extension that allows collaborative editing on data. IBBE-SGX is built upon Identity-Broadcast-Encryption (IBBE), a scheme where a single public key can be paired with several private keys, one per user, in order to facilitate data sharing among a group of users. In IBBE-SGX the system interacts with standard *users* and *administrators*. Users form *groups* and group data is encrypted through AES using a symmetric *group key (gk)*. IBBE-SGX manages keys inside SGX enclaves and establishes trust with the TEE leveraging Intel SGX attestation services. All group management operations (e.g., create group, add user to group) that involve the use of secret keys are handled inside enclaves.

Always Encrypted with secure enclaves [38] is a security extension that aims to shield sensitive data managed in SQL database systems from high-privileged unauthorized users. Initially, the Always Encrypted method protected data by encrypting it on the client side, however this scheme was impractical for higher level functionalities (e.g., key rotation, complex database queries) since users needed to move data back to the client side to perform these operations. Always Encrypted with secure enclaves runs secure enclaves within the database engine process to allow computation on plaintext data and access to cryptographic keys inside enclave memory directly on the server side. SQL transactions that involve operations on encrypted data make use of the secure enclaves. Column encryption keys are sent over secure channels to the enclave by the client, and all cryptographic operations on encrypted columns are performed inside the shielded memory region of the enclave directly on the plaintext data. Before passing any secrets to the enclave, the process verifies the code running in the enclave via the attestation services described in 3.3.5.

Chapter 4

Design

In this chapter we explain the security requirements of a cloud-based data-sharing system and list the goals we have set for our proposed design. Furthermore, we explain the threat model, we describe the main entities participating in the system as well as the secret encryption keys involved, and we illustrate the system's overall architecture. Additionally, we describe the protocols our system supports and the key management policy we apply in order to securely handle secret keys. To conclude, we summarize about the system's design and analyze the security of our proposed protocols.

4.1 Design Goals

In the proposed system's design, we set the following goals:

1. **Data Sharing** Support secure protocols for fundamental data-sharing operations in untrusted cloud environments, leveraging object based storage in a flexible and confidential manner.

2. **Confidentiality** Protect sensitive data from other users that should not have access to it unless permission is granted by the data owner. The data is not visible in the clear even by the cloud provider and only authorized users must be allowed to access it.

3. **End-to-end Security** The system must ensure that all critical user data is secure (e.g. encrypted) when transmitted from or to the storage provider at all stages of the data-sharing operations. Data is never revealed in plaintext to any entity besides certified users.

4. **Availability** The system must be able to interact with authenticated users, and serve data-sharing requests. Thus, a user ought to have access to her private data or data shared by other users of the service at all times.

5. **Performance** Achieve data-intensive throughput for secure file-sharing requests, while maintaining low latency even for high data-intensive workloads.

4.2 System Entities

In this section we describe the entities participating in the establishment of the secure data-sharing protocols, the responsibilities carried out by each entity, and the type of storage we leverage in our design.

4.2.1 Users

The user is an entity (e.g. individual or application) that wants to securely store or share their data in the cloud. Each user owns a *public-private key pair*, and is uniquely identified by the system using their public key. In order to interact with the system

and forward requests to the file-sharing service, users are equipped with a client application that establishes secure channels (e.g. TLS) with the service. Data propagated to the service is sent in encrypted form through secure channels established by the client application. Furthermore, users' key pairs are generated and managed by the client application. The client application facilitates only outcoming connections to the service, however it efficiently receives data transmitted to users by the service. The client application does not need to run on an Intel SGX-enabled platform, and is responsible for managing any secret keys it stores locally.

4.2.2 Cloud Storage Provider

A *Cloud Storage Provider* is an organization that offers other organizations or individuals the ability to place and retain data on the Internet in an off-site storage system that manages user data. The provider offers storage capacity in a pay-as-go model to the cloud consumers. The provider's organization usually consists of several *privileged employees* such as administrators, that are responsible for managing, maintaining, and supporting the cloud storage services. The provider's employees can posses crucial security sensitive information related to the storage service (e.g., passwords, encryption keys), manage privileged software running on the provider's side, and may even have physical access to the service's storage systems.

4.2.3 Tenant Authorization Server

We call tenant an independent organization whose users consume networked services from a cloud provider. The tenant is in possession of secret keys that are used for bootstrapping the file-sharing service. Moreover, the tenant's computing nodes are equipped with Intel's SGX extension, and are able to run secure enclave code for managing the service's secret keys. We assume that the tenant runs an enclave that has passed the local attestation test before delivering its services. That enclave is responsible for remote attesting enclaves running on remote computing nodes, and has access to the tenant's keys. Our system consists of a single trusted server that has the role of distributing secrets to remote computing nodes, and operates the tenant's enclave. The tenant's enclave authorizes enclaves operating on remote servers and provisions them with secrets via remote attestation. We call the server operating the tenant's enclave *Tenant Authorization Server* (TAS).

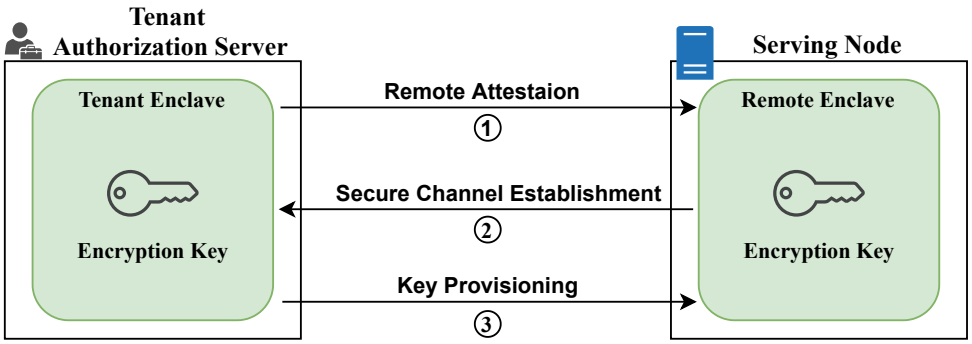


Figure 4.1: Secret key provisioning by tenant enclave to remote computing node leveraging remote attestation.

Figure 4.1 illustrates the process of distributing secret keys to remote computing nodes. The tenant's enclave initiates a remote attestation test with a remote computing node. We assume that the enclave initiating the process has locally attested itself to the platform's Quoting Enclave. By successfully executing the remote attestation protocol as described in 3.3.5, a secure channel is established between the initiator enclave and the remote enclave. Through this channel, the enclaves can safely exchange messages and secrets with each other, and the tenant's enclave is able to provision the remote enclave with encryption keys related to the execution of the file-sharing service's protocols.

4.2.4 Serving Nodes

Our proposed system consists of an arbitrary number of *serving nodes* that belong to the tenant, with a genuine Intel SGX component installed on all serving nodes so as to participate in the proposed system's protocols. For compatibility reasons, we assume that all computing nodes run the same operating system, are equipped with the same hardware, and that they are directly connected through the datacenter network. All computing nodes execute the same locally-attested enclave code and are signed using Intel's signing mechanism. Prior to participating in the system's protocols, each enclave passes the remote attestation procedure initiated by the Tenant Authorization Server. TAS delivers the service's secrets through its enclave and the serving node is integrated to the system. Consequently, the system is now able to forward user requests to the serving node.

4.2.5 Object Store

Object storage is the main type of storage we leverage in our proposed design. Our object-store server is used for safely storing and sharing user's data in the cloud. In our design, the object-store server deployed in standalone mode on a single host. The object-store server is accessible through a dedicated port in order to forward requests to the server, and can also be accessed via a web browser through its web interface to facilitate client requests. Nevertheless, in our design all network traffic and user requests are passed through the serving nodes that participate and facilitate the secure file-sharing protocols.

Objects are stored and managed inside collections. There are three types of object collections we define to support our protocols. First, we have created a collection where all principals can store their public keys. We refer to this collection as *User Keys Collection*. Objects stored in the User Keys collection are not encrypted and any user can gain access to the public keys in order to identify other users, and selectively share their data with them. Moreover, we introduce a second collection to manage and store objects containing the list of folders users are owners of. We denote this as the *User Folders Collection*. All other collection of the system are used as common *Data Folders* where users can upload and/or share files with other users. Each data folder includes a user defined *Access Control List* (ACL), declaring the set of users that are authorized to access the folder's files.

4.3 User Requests

Before we proceed to analyze the protocols and the encryption keys we introduce in the design of the proposed file-sharing service, we briefly mention the requests users can make to the service and the entities involved. All user requests are propagated to the service using the user's client application. Each request is forwarded to a specific serving node of the platform, and the serving node collaborates with the object storage server which is responsible for handling data storage. Thus, the service acts as an intermediary receiving and processing user requests and data, and leverages the object store as a back-end for storing user data. Users may submit a specific set of data-sharing requests to the service to manage their data. The types of user requests we support in our design are listed as follows:

1. **User Registration**: Users need to be able to register to the service. The registration process itself is the first request submitted by a user to the service, and is essential in order to use the services file-sharing features.

2. **Data Folder Creation**: Registered users are allowed to create data folders which they own, to privately store or even selectively share their files with other users of the service by declaring access control lists for the specified folder.

3. **Access Control List Update**: Data folder owners are given the privilege of updating the access control list of a specific folder they own. Thus, they may restrict access to the folder's contents to previously authorized users, or grant access to other users who priorly did not have access.

4. **Content Upload**: Users are able to upload their privately held data to data folders they own, or to folders they are granted access to by other users of the service.

5. **Content Download**: Users are allowed to download content stored in data folders they are authorized to access and store it locally for private use.

6. **Content Delete**: Stored/Shared files residing in data folders can be removed by folder owners, or by authorized users declared in the access control list.

7. **List Folder Content**: Users can request a list of the contents stored in a specific data folder they have access to.

4.4 Encryption Keys

Each user accessing the service owns a public-private key pair. The public key is used to identify the user to the system, and the private key is essential to perform specific user transactions. Therefore, in order to implement the protocols that will be described in section 4.7, we introduce a set of encryption keys. Table 4.1 lists the encryption keys used by the system, provides a brief description for each type of key, and states the keys length in bits.

- **Service Encryption Key (SEK)**: The *Service Encryption Key* is securely distributed among the platform's SGX-enabled serving nodes by the tenant's enclave operating on TAS, via the remote attestation service. Within each serving node, SEK is protected using the machine's Root Seal Key. Its main responsibility is en/decrypting user's folder encryption keys. Thus, users can share their files without the need for file owners to be online at object request time from other users, delegating the access control to the service.

- **Folder Encryption Key (FEK)**: The *Folder Encryption Key* is generated by an attested SGX enclave operating on a serving node, each time a new collection is created by a principal participating in the system at the object store. Access to the folder's ACL is cryptographically enforced by each folder's FEK, and FEK's are encrypted by the SEK. Each is stored as part of the respective ACL's metadata field.

- **Object Encryption Key (OEK)**: An *Object Encryption Key* is generated by an attested SGX enclave operating on a serving node, each time a user requests to upload a new object to a collection. The object is encrypted using OEK, OEK is encrypted using the folder's FEK, and the the encrypted OEK is stored as part of the object's metadata. All encryption logic is securely handled inside SGX enclaves.

Key	Description	Type	Size
OEK	Encrypts distinct data objects	Symmetric	128bit
FEK	Encrypts access control lists	Symmetric	128bit
SEK	Encrypts folder encryption keys	Symmetric	128bit
Public-Private Keys	Identify users to the system	Key-Pair	1024 bit

Table 4.1: Encryption keys overview for implementing secure file-sharing protocols.

4.5 Overall System Architecture

In the previous sections we introduced and analyzed the individual entities that our system is composed of, and the encryption keys involved. Figure 4.2 illustrates

the overall architecture of our proposed system. The Tenant Authorization Server is responsible for provisioning the SEK to each attested server participating in the Intel SGX server farm via remote attestation. The server farm is comprised of N individual servers that are directly connected through the datacenter network switch achieving fast data transfer rates between the servers. As mentioned in 4.2.1, users own a public-private key pair and are equipped with a client application to interact with the service via secure channel. Using her public-private key pair and any additional information required depending on the type of the request, a user transmits her requests through the client application via a secure channel to the server farm. Each incoming request is handled by an SGX-enabled server, that interacts with the object storage server. The object store may contain an arbitrary number of data collections M, plus the User Keys Collection and the User Folders Collection as described in 4.2.5. The object-store server processes the SGX server request and returns a response indicating the request's execution result. Communication between server nodes and the object store is also protected by secure channels. Finally, the system provides a response to the initial request submitted by the user.

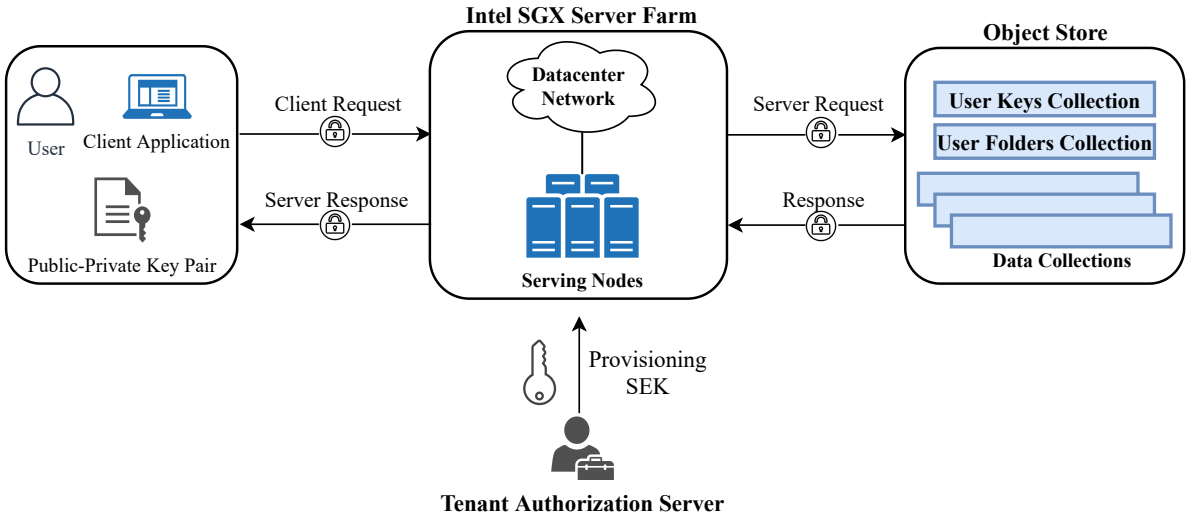


Figure 4.2: Overall architecture displaying the interaction between the system's components.

4.5.1 Trust Assumptions

The tenant operates the Intel SGX server farm nodes at which the data-sharing service runs. Moreover, the Tenant Authorization Server also runs on an SGX-enabled

platform. Prior to gaining access to the tenant's keys, the enclave responsible for distributing secrets to the serving nodes uses Intel's enclave signing tool to perform the enclave's signature and ensure the authenticity of the enclave. The serving nodes are being remotely attested by the tenant authorization server and run secure, signed, and attested enclave code. The remote attestation process involves contacting Intel's EPID Provisioning and Attestation Services guaranteeing the correctness of the procedure. Secure hardware (e.g. TPM 2.0 support) is provisioned with each server machine to perform static measurements and verify the integrity of the system's software stack. Client communication with the file-sharing service, and message exchange between the service's servers and the object store is integrity protected, authenticated, and encrypted by the use of secure protocols (TLS 1.2, TLS 1.3 etc.).

4.6 Threat Model

In our work, we focus on three kinds of principals. A **user** is someone who wants to store data online and may selectively expose the data to a **third-party service** or another user of the system. The user is allowed to upload and store her data on an object storage server which claims the role of a **cloud storage provider**. All users store their data on the same storage provider, but may selectively share their data with other users of the system which request access. The considered adversary, such as *an employee of the provider*, may already have control of the host machine and may also have physical access to it. In consequence, the BIOS, OS kernel, and OS subsystems may be compromised. We assume that the ultimate goal of the adversary is not to disrupt the service's operations, but may attempt to read user's confidential data in the clear (*honest-but-curious*). Furthermore, we consider the case where *a user of the same system* (or even an external attacker) endeavors to obtain another user's files. This is possible by launching masquerade attacks using fake identities to gain legitimate access to otherwise not accessible data, or eavesdrop on transmitted packets to the file-sharing service.

Initially, users upload their data to the storage provider in plaintext through secure channels without any sort of client-side encryption. Leveraging server-side encryption schemes with user provided keys supported by commodity object storage solutions, user data resides on the server-side in an encrypted form. Therefore, it is not acces-

sible without being in possession of the client provided key. However, this does not guarantee the security of users' data in case of storage server compromise. During en/decrypting an object the encryption keys are loaded to RAM in plaintext. Additionally, even though the communication between client and the provider is encrypted, during data upload the server ultimately decrypts the encrypted data chunks and gains insight to the actual plaintext of the transmitted packets. Thus, a *malicious user which has access to the server*, may launch cold boot attacks [39] to retain DRAM data by freezing the memory chip even with simple cooling techniques (e.g. spray liquid nitrogen). Using cold boot attacks attackers can gain access to object encryption keys, or even read the actual data having being transmitted by client applications.

The set of encryption keys we employ to encrypt user files is solely managed by the system. Object encryption keys are never revealed to users of the system and are encrypted using the service's keys. This scheme contrasts the methodology followed by existing commodity storage systems, where authorized users are allowed to locally store encryption keys. With our scheme we relieve users of the responsibility to manage data encryption keys. Moreover, unauthorized users can never gain access to a data folder's encryption keys because the keys are never revealed in plaintext to any entity besides the enclave's encrypted memory area. Thus, we eliminate the threat of unauthorized access by other users of the service or privileged personnel at the cloud provider.

Nevertheless, we do not provide any security guarantees if the user's client machine gets compromised. We do not protect against denial-of-service (DoS) attacks by compromised system software: a malicious OS may deny service by refusing to allocate any resources to an SGX enclave. Moreover, Intel's SGX technology is susceptible to specific types of side-channel attacks such as power analysis or cache-timing attacks. Protection against against such kinds of attacks is out of the scope of this thesis, and we bestow this to future improvements of the SGX technology.

4.7 Protocols

In this section we describe the protocols we designed in order to facilitate secure file-sharing services utilizing an object-based store, with respect to fundamental security primitives and design goals described in 4.1.

4.7.1 Requests Management

Anterior to describing the protocols we introduce in the system's design, we briefly propose a scheme to handle, distribute, and serve incoming user requests to the service. In this thesis, we do not analyze the workload distribution among the SGX server farm nodes. We assume that one node of the system is acting as a coordinator, and delegates an incoming request to the first available node of the system, or to the node which has less incoming requests waiting at its connection queue. Server workload distribution is bestowed to future developments of the system. Upon receiving the incoming request at a computing node, the SGX enclave processes the user's request. The system parses the request and after applying validity checks (contains all necessary information) on the request, the system proceeds to follow the necessary steps to serve the user request.

4.7.2 User Registration

Initially, to grant access to the file-sharing protocols a user needs to register to the file-sharing service. The user constructs a registration request containing her public key and forwards the request to the service.

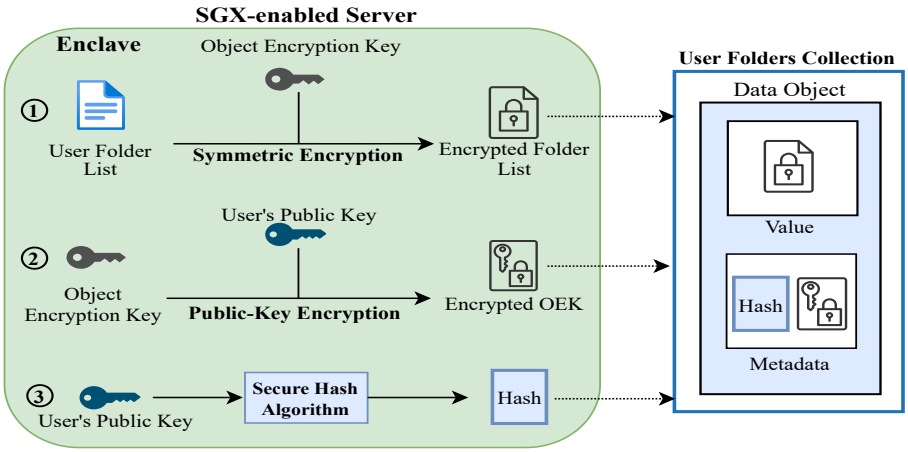


Figure 4.3: Registering a new user to the file-sharing service.

The user's public key is securely transmitted to the cloud server, and is only accessible inside the enclave memory. The user's public key is used in order to prove the user identity to the system, and identify other users of the system. Moreover, the public key is leveraged in the establishment of the secure communication channels with the service (e.g. TLS), thus only the user possessing the corresponding private

key is able to read data transmitted by the service to her. Ultimately, all public keys are stored inside the User Keys Collection without any form of encryption, and are available to all users of the system in order to populate ACLs with user identities.

The user registration process is depicted in Figure 4.3. Initially, the enclave constructs an empty file (*User Folder List*) which will we be used to store the list of folders the user will potentially create in her future transactions with the system. First, (1) the file is encrypted with a randomly generated *Object Encryption Key* (OEK) inside the enclave. Next, (2) the file's OEK is encrypted with the user's public key, and then (3) the enclave generates the public key's hash signature using a secure hash algorithm. The enclave does not preserve any of the forementioned encryption keys, and the encrypted list is uploaded to the object store's User Folders Collection. Each list is stored at the object store as an **object** (as with all files uploaded to the object store), and objects are accompanied with the user's public key hash and the encrypted OEK in their **metadata** field. At the end of this process, a new object is stored that contains the encrypted folder list file the requesting user owns, along with the user's public key hash and the OEK used (encrypted with the user's public key) to encrypt the list in its metadata field.

4.7.3 User Folder Creation

Users registered to the service are given the privilege of creating folders in the object store to securely store and share their data. To create a new folder, the user needs to specify the following information in her request: (1) the name of the folder to create, (2) her public-private key pair, and (3) a file containing the folder's initial *Access Control List* (ACL).

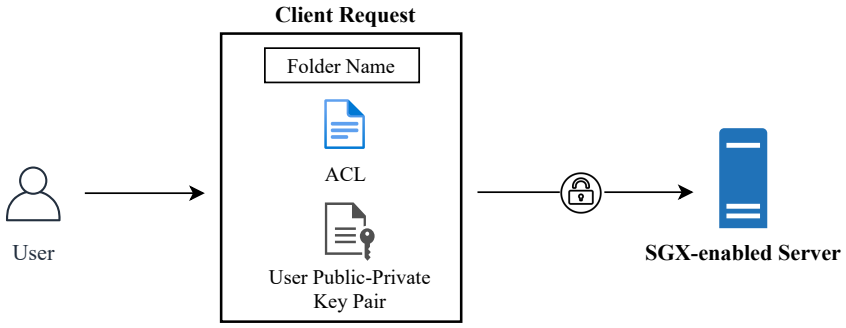


Figure 4.4: User request specification for creating a new folder, and delegation to the file-sharing service.

Figure 4.4 illustrates the process of creating a new user folder. The user selects a name for the new folder to be created. Consequently, the user locally constructs an ACL for the folder, and attaches her key pair to the request. This bundle of information is forwarded to the service, and is delegated to a specific server node to process the request information. A new *Folder Encryption Key* (FEK) is randomly generated by the enclave for the new folder to be created. With the newly derived FEK, we encrypt the user provided ACL. The ACL must contain at least the user's public key, otherwise the ACL is not valid. The system is not responsible for scrutinizing the folder's ACL for invalid public key population. Other users' public keys are publicly accessible in the User Keys Collection due to the user registration process, or can be obtained through any means of user interaction (e.g, public announcements, publicly available directory, public-key authority). Following the creation of the collection, the system uploads the folder's ACL, which is essentially an object. The FEK is encrypted with the *Service Encryption Key* (SEK). Instead of storing the FEK as a separate object, we attach the FEK as one of the ACL's metadata field to automatically gain access to the key whenever we require to parse the ACL.

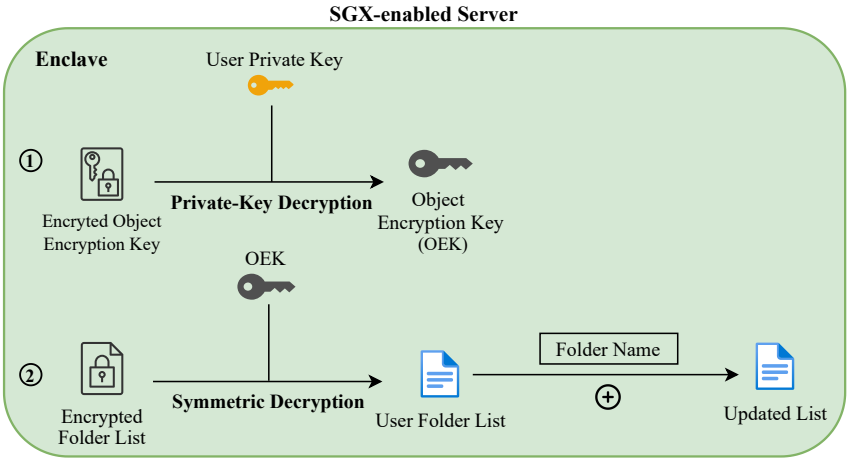


Figure 4.5: Appending the new folder's name to the user's folder list file.

To conclude, the system needs to update the user's folder list file. All users registered to the service own a folder list file that is stored in the User Folders Collection. The system computes the hash signature of the requesting party's public key to identify the user's folder list file. We retrieve the specified object and its encrypted *Object Encryption Key* (OEK) from its metadata fields. Figures 4.5 and 4.6 depict the steps executed to update the file. Inside the SGX enclave:

1. The system decrypts the folder list file's OEK using the user's private key in order to gain access to the symmetric key the file is encrypted with.

2. The folder list file is decrypted using the retrieved OEK, and the folder name is appended to the plaintext of the file. Hence, the user added a new folder she owns to store or share her data to the list.

3. The system re-encrypts the file with its OEK to secure the data, and then replaces the folder list file in the object store's User Folders Collection with the updated file, thus enabling the use of the newly created user folder.

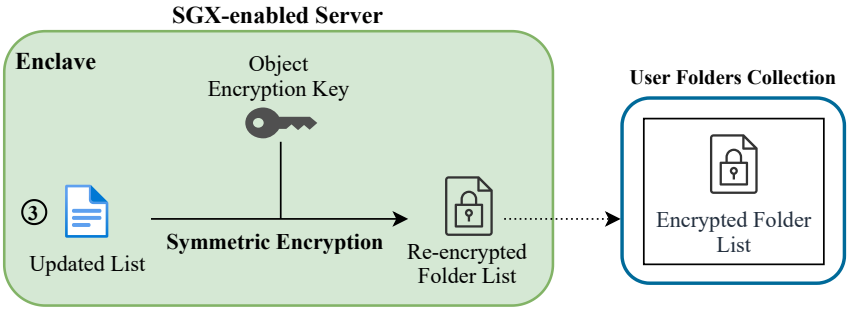


Figure 4.6: Re-encrypting the folder list file and updating it at the object store.

4.7.4 Uploading Data to the Storage Provider

Suppose a user wants to upload her data to the storage provider. Before uploading the file, the user needs to construct a request providing her public key, the name of the destination folder, and the file to be uploaded. The end-to-end upload protocol is illustrated in Figure 4.7. First, the file is transferred to the server in chunks of equal size by the client application, using the maximum available record size for each chunk. Files with size less than or equal to the max record size require only sending a single file chunk to the service. The chunks are accumulated inside a temporary buffer for encryption. To grant upload access, the system fetches the specified folder's ACL along with the folder encryption key. FEK is decrypted using the service encryption key, and the ACL is unsealed using FEK. We search the ACL file for the user's public key.

If the ACL does not contain the user's public key, the request is rejected and the system prevents unauthorized access. A user is allowed to upload just by being

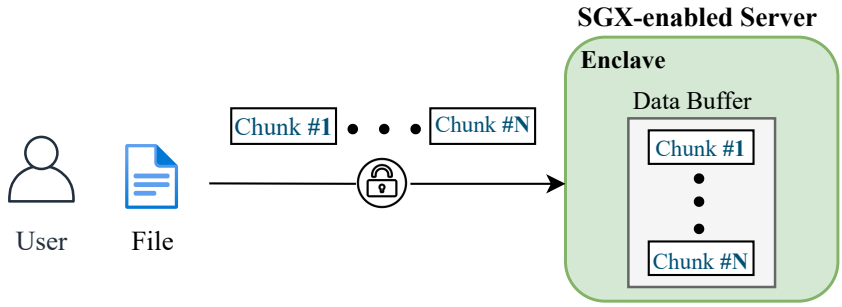


Figure 4.7: Transferring the user's file in data chunks through a secure channel to an enclave that accumulates the chunks in a data buffer.

enlisted to the folder's ACL, without necessarily being the specified folder's owner. Upon successfully passing the ACL check, an new OEK is randomly generated. The client's file is encrypted using OEK, and OEK is encrypted using FEK. Finally, the file is uploaded to the specified folder, with the associated metadata block containing the encrypted OEK.

4.7.5 Accessing Shared Data

Accessing a shared or private file stored in our object store requires a series of steps. First, when a user who is not the owner of the folder desires to access user files, the user does need to interact with the owner. Instead, the user sends an access request directly to the storage provider. The user needs to send a request to the system specifying the folder and file name to download, and provide her public key. Similar to the principle described in 4.7.4, the system accesses the folder's ACL file and metadata, decrypts the object encryption key and the file itself, and then parses the ACL for the requesting user's public key. If the ACL contains the designated public key, the system retrieves the file listed in the request and decrypts its contents inside the enclave. Afterwards, the enclave sends back a notification to the client application indicating the file to receive, and as a last step begins streaming the file back to the user in data chunks of equal size. The user receives and decrypts the data chunks, and re-constructs the original file locally in plaintext. The file format is retained during this process since we transfer bytes and we do not apply any transformations to the original encoding of the data.

4.7.6 Access Control List Revocation

The ACL is initially populated with the owner's public key. In future transactions with the system, the owner may want to declare an updated version of the ACL populated with a user defined set of user public keys. Furthermore, a user may selectively exclude a set of public keys from the ACL to restrict access to specific users that previously where able to view/update the folder's contents. To update the ACL, the user needs to provide: (1) her public-private key pair, (2) the relevant folder name, and (3) the updated ACL file. Initially, the system needs to verify the user's ownership of the declared folder. In a similar fashion to 4.4, the system retrieves the user's folder list file, and decrypts the file's OEK using the user attached private key. Thereafter, we replace the previous ACL with the updated file encrypted with the folder's FEK. In this scheme, we re-use the folder's FEK and do not generate a new as with any new file upload to the storage service. This is due to the fact that objects of the same folder have their OEK encrypted with FEK. Thus, replacing the existing FEK would require re-encrypting all OEK's (*Key Rotation*) with a new FEK.

4.7.7 Removing Shared Data

Users are allowed to remove objects that are stored in the service's data folders. To perform this type of request, a user must be either the specified data folder's owner, or an authorized user the folder owner lists in the folder's ACL. To remove a stored object from the object store, the user constructs a request in which she specifies: (1) her public key, (2) the folder name, (3) the name of the object to remove. The request is forwarded to the service by the requesting party's client application and then served by the system. To begin, the system needs to verify that the user is indeed listed in the data folder's ACL. To this end, the system performs an ACL check to determine if the the user's public key is listed in the folder's ACL file. To perform the ACL check, the ACL is decrypted inside the enclave that handles the request and consequently the system parses the ACL for the requesting user's public key. Upon, verifying that the user is authorized to remove content from the data folder, the system proceeds to remove the specified object from the object store along with the object's metadata.

4.7.8 Listing Folder Contents

Users are able to obtain a list of the objects that are stored in a data folder. The requesting user must be authorized to access the folder's contents, and proves her identity to the system using her public key. In order to retrieve the list of objects managed in a specific data folder, the user specifies the following information in her request: (1) her public key, and (2) the folder name. Afterwards, the request is propagated to the file-sharing service by the user's client application. As described in the preceding file-sharing protocols, the system receives and parses the incoming request, and performs an ACL check on the folder's ACL file. Thereupon, the system fetches the list of objects stored in the folder, and constructs a file containing the name of the objects as well as the size of each object. At the end of this process, the system sends back the file holding the folder contents to the requesting user.

4.8 Security Analysis

The proposed SGX file-sharing strategy leaves three basic approaches for the adversary to read on users' data given the above threat model. The first one is to crack the sealed file containing the SEK which is encrypted with an AES-GCM 128bit key. For that, she may capture the file and mount an offline attack in a high performance environment. The sealing key is not stored anywhere in the serving nodes, and each serving node uses a different private Seal Key that is unique to that particular platform and enclave [40]. Hence, the only feasible option is brute-forcing the sealing key, which would take an unreasonable amount of time, even considering one of the most performant supercomputers in the world.

The second approach would be to access enclave memory during request serving, to try and capture users' files or even the private keys (e.g. during folder creation) in the clear, through a memory dump. Enclave memory however is encrypted using an AES-CTR 128bit key and the key exists within the CPU which leads to a situation similar to the previous approach.

The last approach involves targeting the message exchange between clients and the service through a *man-in-the middle attack*. Nonetheless, the message exchange between the client application and the file-sharing service's servers is protected through the use of secure channels. Secure channels employ symmetric cryptography through

the use of a shared key that is computed using the server's public key. The server can generate the secret key only if it can decrypt that data with the correct private key [41]. For client authentication, the server uses the public key in the client certificate to decrypt the data the client sends. The exchange of messages that are encrypted with the secret key confirms that authentication is complete. However, the endpoint of the secure channels terminates inside an SGX enclave which is responsible for decrypting and reading the user data. Thus, user data is encrypted even when received at the provider, since enclaves are encrypted memory regions not visible even to privileged software managed by employees of the provider. Using this encryption scheme, we ensure that user data is not viewed in the clear even by a curious cloud provider, achieving end-to-end security throughout the whole execution of the file-sharing service's protocols.

Chapter 5

Implementation

In this chapter we present the implementation details of the proposed file-sharing service prototype leveraging Intel SGX technology to secure sensitive data, and the MinIO object store as the back-end for storing and managing data. We call our prototype TeeStore. We describe: (1) the data structures that support our secure protocols to enable fine-grained access to data, (2) the functionality we added to the client application forwarding file-sharing requests to TeeStore, (3) the implementation of the secure protocols described in section 4.7 through the aid of SGX enclaves, and (4) the MinIO client operations we implemented to facilitate the interaction between TeeStore and the object store. In the end of this chapter we briefly mention the limitations of our implemented service.

5.1 Overview

Our implementation is based on an open source *WolfSSL TLS server application* written in C and C++ programming language. The difference between this application and other commodity TLS server applications, is that the application integrates the Intel SGX technology to handle incoming TLS connections. More specifically, the server decrypts incoming traffic sent over the TLS protocol inside an SGX enclave, thus never revealing the underneath plaintext data transmitted by the client even to privileged software of the provider. Nevertheless, the SGX technology does not allow the execution of dynamic libraries inside enclaves and permits only a limited subset of trusted functions referenced in the SGX *Software Development Kit (SDK)* [27] to execute inside enclaves. To this end, in a similar fashion with other related work [42], we compiled an SGX-enabled static library version of WolfSSL named *libwolfssl.sgx.static.a* and linked this library to the SGX enclave to access WolfSSL APIs with SGX hardware. Afterwards, we extended the functionality of the enclave managing the incoming TLS data traffic to receive files of scaling size up to the maximum useful EPC limit. At the same time, we extended the functionality of a baseline WolfSSL client application to forward file-sharing requests directly to the SGX enclave. We name our implemented client *TeeStore Client*.

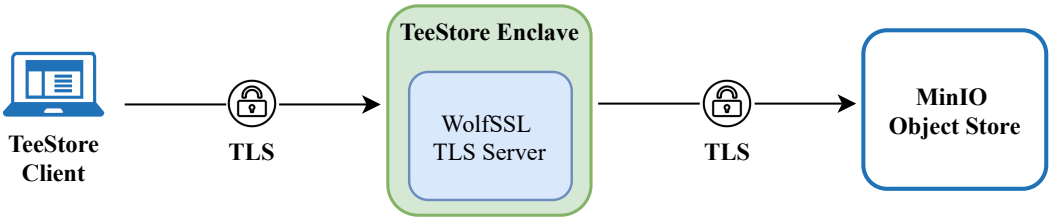


Figure 5.1: Overview on the system's implementation.

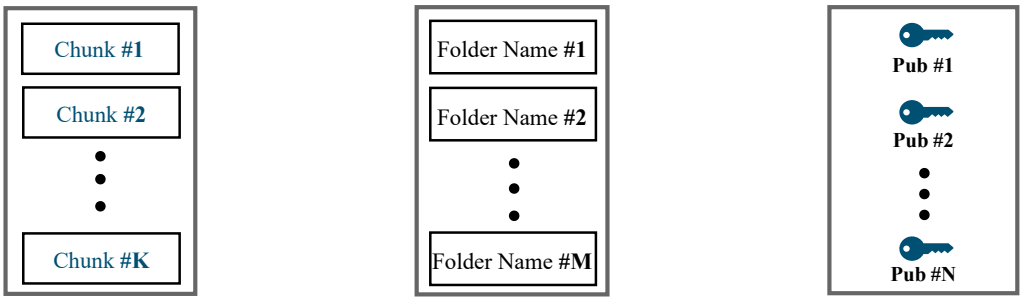
Figure 5.1 illustrates the basic system components of TeeStore. On the left side of the figure, a TeeStore Client sends data to the WolfSSl TLS server that runs inside an SGX enclave, over a secure TLS channel. The enclave receives and decrypts the incoming communication, processes user requests, and encrypts any sensitive user information depending on the request type sent by the client application. Afterwards, the encrypted data leaves the enclave memory region and is sent over TLS to the MinIO object store that serves as permanent storage, for the now encrypted user data.

5.2 Data Structures

The implementation of the protocols described in the design of TeeStore requires the aid of data structures to support the execution of the proposed protocols. The first structure we implemented both at the client application and to the server application is a C structure we name **Request**. Struct Request is populated with all necessary information needed to serve a particular user request. The necessary information the user provides in her request are the following:

1. **Request Type**: A character array indicating the type of the request the user wishes to perform on the file service.
2. **Bucket Name**: A character array specifying the name of the relevant MinIO bucket the user request involves.
3. **File Name**: The name of the file the request is associated with. This also corresponds to the case of updating ACL files.
4. **Payload**: The size of the file to be transferred to the service counted in bytes.

This series of information is bundled together in the form of the Request struct. Consequently, the structure is serialized in the form of a character array *char[]* we refer to as a *request string* using the *memcpy* function call, and sent over TLS to TeeStore by the TeeStore Client. The server application also contains a memory buffer to receive incoming TLS data. Initially, this internal buffer's size was 2048 bytes (2KB) and after proper examination which will be presented in section 6.7, we increased its size to 16384 bytes (16KB) which is the maximum TLS record size [43].



(a) Data buffer for accumulating file chunks.

(b) User folder list file as a series of folder names.

(c) A series of public keys in an ACL file.

Figure 5.2: Data structures supporting TeeStore's protocols.

In Figure 5.2 we illustrate an important class of supporting structures that aid our implementation. First of all, Figure 5.2 (a) shows a data buffer that collects the data transmitted by the TeeStore Client. This buffer stores the data received at the internal TLS memory buffer in the form of chunks (e.g. K chunks) of 16KB size, and is dynamically allocated using the C library function *malloc*, with its size being specified in the *payload* field of the user request.

Figures 5.2 (b) & (c) illustrates another two fundamental structures of our proposed design. The first is the user *folder list file*, and the second one describes *access control lists*. For the purpose of this thesis, their structure is being kept as simple as possible. The folder list file is initially an empty character buffer that lists the names of the data folders a user owns and creates while using the file service, and all folder list files are placed as objects in the *User Folders Bucket*. The names of the folders, which are translated as MinIO buckets, are comma separated and stored in sequential order. Thus, the complexity of parsing this list is approximately *O(M)*, with M being the total number of folder names listed in a user folder list file. The second file structure that is the ACL file, is composed of a list of say N user public keys. Each ACL file corresponds to exactly one data folder, the ACL is stored as an object inside the folder it belongs to, and it must contain at least the folder owner's public key. The keys are stored as a sequence of 1024 bytes, and the cost of parsing a list of N keys is *O(N)* since the keys are also stored in sequential order.

5.3 TeeStore Client

Users interact with TeeStore using the TeeStore client application we implemented. The client is written in the C programming language, and leverages standard C libraries to interact with the local filesystem (e.g., stdlib.h, stdio.h). Following the official documentation of WolfSSL [12] and the developer manual [14], we compiled a baseline WolfSSL client application with basic functionality. The basic functionality the client provide included: (1) setting up an IP socket, (2) initializing a WolfSSL client, (3) making a new SSL context with user provided information (keys, certificates etc.), and (4) sending a "hello message" to a WolfSSL TLS server which in our case operates inside the TeeStore Enclave. Nonetheless, in order to support the secure protocols we explain in our design, we extended the client's functionality with code

functions that implement the interaction with TeeStore, and forward requests to the service.

Function Name	Description
void generate_rsa_key	Generates an RSA key pair.
void store_rsa_key	Stores an RSA key pair in **.der** format.
char* load_rsa_key	Loads an RSA key pair from a **.der** file.
void register_user	Registers a user to the service using her RSA key pair.
void create_folder	Sends a ”create user data folder” request to the service.
void init_ACL	Initializes an ACL file containing a list of public keys.
void update_ACL	Propagates an ACL file to the service for update.
void upload_file	Splits a file into chunks and sends it to the service.
void download_file	Receives data chunks and stores them locally as a file.
void delete_file	Sends a ”delete file” request to the service.
void list_files	Receives a file listing a specific folder’s contents.

Table 5.1: TeeStore client application implemented function set.

Apart from the functions we implemented, the most important functions calls we leverage from the WolfSSL TLS library API are: (1) **wolfSSL_read** to receive and decrypt incoming TLS data communication, and (2) **wolfSSL_write** to send encrypted data to the service over TLS.

Table 5.1 summarizes the functions we implemented and added to the existing client application, and it also contains a brief sentence describing each function. The client application uses the **RSA** algorithm for asymmetric key management operations. First, we implemented the function *generate_rsa_key* to bestow key generation logic to the client. In our implementation the user defines the size of the RSA key pair to be generated, but we mainly use **1024-bit** keys for user public-private key pairs. RSA keys are then stored in .der format (*store_rsa_key*), and loaded back to the application from the .der files when needed (*load_rsa_key*).

As a next step, we declared the Request struct in the client application to instantiate user requests. For each of our designed system’s protocols, we further implemented a separate function that: (1) instantiates the user request with the necessary information, (2) serializes the request and propagates it as a request string to TeeStore, and (3)

receives and handles incoming data responses by TeeStore corresponding to user requests. Following, we shortly describe the implementation details for each request type on the client side:

- **Register User**: Registering a user (*register_user*) involves sending a registration request to TeeStore. In this case the client specifies only the request type and sends the user's public key to the service, which serves as the user's identity to the system.

- **Create Data Folder**: A registered user to the service wants to create a data folder (*create_folder*) to store and share her data. To begin with, the user needs to create and specify an ACL for the corresponding folder to be created. ACL initialization (*init_ACL*) is handled at the client application. The client loads locally stored RSA derived public keys as byte buffers in the application. Afterwards, the byte arrays are sequentially written to an empty file using a *File** pointer, and the ACL is formed. The client sends a request to TeeStore indicating the *folder creation operation*, the name of the folder to be created, and attaches the user's public-private RSA key pair to the request. Moreover, the client loads the previously generated ACL file, and sends the file to TeeStore as a series of 16KB chunks of bytes.

- **Update ACL**: A user may wish to update the ACL of a folder she owns (*update_ACL*), in order to remove or add other users from the ACL. First, the client makes a call to *init_ACL* function to construct a new ACL file, and then the application propagates the user request to TeeStore. The application also sends the user's RSA key pair and the new ACL file split in data chunks.

- **Upload File**: Uploading a file to a specific folder (*upload_file*) involves constructing a request specifying: (1) the folder name, (2) the file name, and (3) the byte length of the file. The client sends the request, as well as the input file split in chunks.

- **Download File**: To download a shared file (*download_file*) the client application sends a request indicating: (1) the file name, and (2) the bucket name the file is stored in. Afterwards the client waits in a *while-loop* with a *timeout function* of 10 seconds for the service response. The client allocates a memory buffer with call to *malloc* to receive the file to be downloaded, and reads the incoming data

chunks to the buffer. The byte buffer is then stored locally as the original file, with a call to *fwrite* using a *File** pointer.

- **Delete File**: Deleting a file requires the user to populate her request with the file name and the folder name. The client application sends this information to TeeStore, along with the user's public key to issue file deletion.

- **List Files**: To obtain the list of files stored in a MinIO bucket as objects, the client application sends a request specifying the bucket name. If the user is authorized, TeeStore returns the list of objects stored in the bucket, along with the byte length of each object listed in the file. The client application receives the list in data chunks and writes the data to a local text file.

5.4 TeeStore Implementation

In this section we present our prototype implementation of TeeStore that is responsible for serving user requests submitted by the TeeStore Client described in 5.3. First, we present a high level overview of TeeStore's internal architecture, we describe how the individual system components our service consists of collaborate, and then we proceed to detail the functionality of each component.

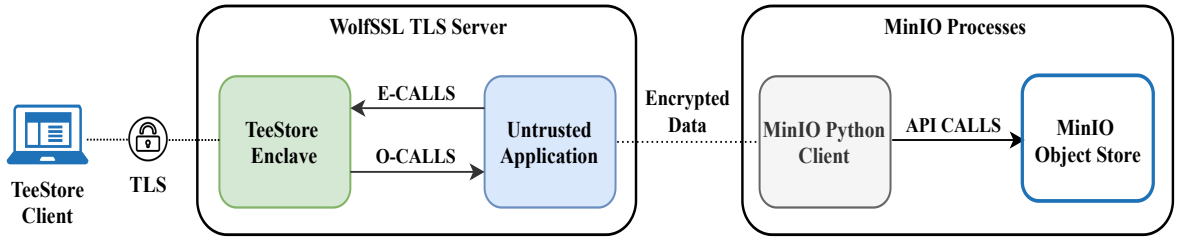


Figure 5.3: Overview on TeeStore's internal components.

Figure 5.3 illustrates the overall implementation of TeeStore. On the left side of the figure we depict the TeeStore Client, and next to it the WolfSSL TLS server that is responsible for receiving and processing user requests. This WolfSSL TLS server is split it up into two sub-components: (1) the *TeeStore Enclave*, and (2) the *Untrusted Application*. The TeeStore Enclave is responsible for processing user requests and data inside the protected memory region of SGX and handles all encryption logic. The Untrusted Application is responsible for creating and managing the TeeStore Enclave.

The Untrusted Application uses E-CALLs to access the enclave's trusted function set to process on user data, whereas the enclave uses O-CALLs to leverage untrusted functions and libraries declared in the code of the Untrusted Application. Data encrypted by the enclave is stored in untrusted memory. The Untrusted Application passes and reads encrypted data to and from MinIO. To be more precise, on the right side of Figure 5.3, we depict the components operating on behalf of the object store. First, there is the *MinIO Python Client* which is launched by the Untrusted Application and forwards requests to MinIO, and second there is the *MinIO Object Store Server* that actually stores the encrypted volumes of data. The client receives requests and data by the Untrusted Application and makes calls to the corresponding APIs of the MinIO service to serve data-sharing requests. The MinIO server ultimately stores encrypted data as objects with metadata attached to them.

5.4.1 TeeStore Enclave

The TeeStore Enclave is the component that operates the WolfSSL TLS server. The enclave is written in C/C++ and uses Intel's SGX SDK. Furthermore, the enclave makes use of the WolfSSL static library to access WolfSSL APIs and services. As a starting guide, we built and executed the baseline WolfSSL TLS server published on GitHub [44] by WolfSSL Inc. Initially, the main functionality of the TLS server operating inside the SGX enclave was: (1) creating an SGX enclave that operates the TLS server, (2) binding to an IP socket, (3) initiating a new SSL context with the corresponding certificates, private key, etc., and (4) wait for incoming client requests. The server establishes connections with clients, reads data transmitted by them inside the enclave region, and sends a string-reply to the request issuer as a response.

In order to facilitate the execution of the file-sharing protocols introduced in the design of this thesis, we extended the functionality of the TeeStore Enclave with 11 additional trusted function calls that support the implementation of our proposed protocols. Table 5.2 summarizes the E-CALLs we added or modified in the implementation of the TeeStore Enclave. Most functions are related to the implementation of exactly one protocol (e.g. *enc_upload_file* function corresponds to the file upload protocol). and process on user data inside the protected memory regions of the enclave. Usage details will be given in the protocols implementation section 5.6. Additionally, the enclave implements the Request structure introduced in the specification of the

TeeStore client application, so as to parse user requests submitted to the service.

E-CALL	Description
void sgx_seal_sek	Seals the SEK using the platform's Root Seal Key.
void sgx_unseal_sek	Unseals SEK using the platform's Root Seal Key.
void encrypt_message	Encrypts data with 128bit input key.
void decrypt_message	Decrypts data with 128bit input key.
int enc_wolfSSL_read	Reads data from incoming TLS messages.
int enc_wolfSSL_write	Writes data to connected clients over TLS.
int enc_register_user	Registers a user to the service.
int enc_create_data_folder	Creates a new data folder.
int enc_update_ACL	Updates the ACL file of a data folder.
int enc_upload_file	Uploads a file to a specific data folder.
int enc_download_file	Downloads a file from a data folder.
int enc_delete_file	Deletes a file stored in a data folder.
int enc_list_files	Returns the list of files managed in a data folder.

Table 5.2: E-CALLs we implemented on the TeeStore Enclave to facilitate secure file sharing.

5.4.2 Untrusted Application

The Untrusted Application is the part of the system that acts as an intermediary between the TeeStore Enclave and the MinIO object store. Sensitive user data and request information is passed directly to the enclave. However, the enclave encrypts sensitive data and passes this data to the untrusted part of the system using O-CALLs. Table 5.3 briefly summarizes the O-CALLs we implemented to support our file-sharing service. The primary objective of our implemented O-CALLs is to: (1) write and load encrypted data to and from the filesystem, (2) establish the communication with the object store. Therefore, for each of our proposed protocols we implemented a corresponding O-CALL that handles request serving through the MinIO Python Client.

The MinIO Python Client is instantiated by the Untrusted Application through the use of the *fork* system call. Each time a request is successfully processed by

O-CALL	Description
void ocall_write_file	Writes (encrypted) data to untrusted storage.
void ocall_load_file	Loads (encrypted) data from untrusted storage.
void ocall_register_user	Uploads a user's folder list.
void ocall_get_folder_list	Returns a user's folder list.
void ocall_create_folder	Creates a new user bucket at MinIO.
void ocall_update_folder_ACL	Updates the MinIO bucket's ACL object.
void ocall_get_folder_ACL	Returns the corresponding folder's ACL.
void ocall_upload_file	Uploads a (encrypted) file as a MinIO object.
void ocall_download_file	Downloads a MinIO object to local storage.
void ocall_delete_file	Deletes the specified MinIO object.
void ocall_list_files	Returns a list of the objects stored in a bucket.

Table 5.3: O-CALLs we implemented on the TeeStore Enclave to support secure file sharing.

the enclave, the enclave propagates the resulting encrypted data to the Untrusted Application. Thenceforth, the untrusted part uses the fork call to create a separate *child process* to pass on the execution flow. In turn, the child process makes a call to *execlp* to instantiate a MinIO Python Client and passes all necessary arguments needed by the client. The execlp function is most commonly used to overlay a process image that has been created by a call to the fork function. The call to execlp replaces the calling child process with a new process image that eventually executes the MinIO client. Thus, the client process takes over the request serving, and interacts with the object-store server.

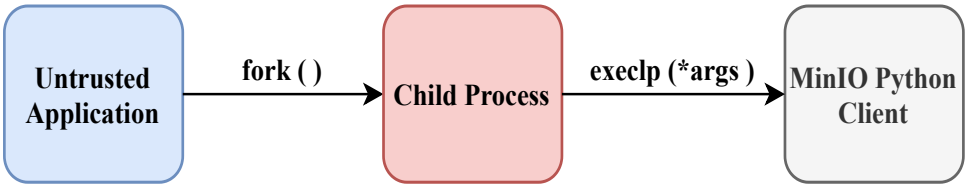


Figure 5.4: MinIO client process execution using system calls *fork* and *execlp*.

5.4.3 MinIO Client & Server

The MinIO object store serves as the storage back-end for our data-sharing service. Requests to the MinIO server are passed through the MinIO client launched by the Untrusted Application. Encrypted data is stored as MinIO objects, and the corresponding encryption key is also stored in an encrypted form as part of the object metadata. MinIO limits user-defined metadata for each object to **2 KB**. The client leverages the *Python Client API Reference* [45] to pass requests onto the object-store server, and the communication between the client and the object store is protected via TLS. Table 5.4 lists the set of APIs used by our client module to interact with the object store, along with a short description for each API call.

API CALL	Description
MinIO	Initializes a new MinIO client.
bucket_exists	Check if a bucket exists.
make_bucket	Creates a new bucket.
set_bucket_tags	Set tags configuration to a bucket.
get_bucket_tags	Get tags configuration of a bucket.
list_objects	Lists information of all objects in a bucket.
fput_object	Uploads data from a file to an object in a bucket.
fget_object	Downloads data of an object to file.
remove_object	Remove an object from a bucket.
stat_object	Get object information and metadata of an object.

Table 5.4: Python API calls used for the interaction between the service and the MinIO object store.

5.5 Enclave Encryption Scheme

The TeeStore Enclave manages a series of secrets related to TeeStore and user data committed to the service. Next, we briefly mention the encryption schemes leveraged to encrypt the distinct pieces of information managed by TeeStore.

First of all, the Service Encryption Key is sealed using the platform's Root Seal

Key. Essentially, SEK is encrypted using a platform specific key using the *sgx_seal_data* function referenced in the SGX SDK [27], and the *MRSIGNER* policy which binds the identity of the enclave author to the sealing key derived within the CPU.

Data passed to the enclave is encrypted using the *sgx_rijndael128GCM_encrypt* function which leverages Rijndael Galois Counter-Mode as the encryption algorithm with 128-bit input keys. Encryption keys (OEKs and FEKs) are derived within the enclave using the *sgx_read_rand* random function to generate 16-byte random byte sequences that serve as encryption keys. An object encryption key is generated for every new file sent to the service, and a new folder encryption key is derived for each instantiated bucket. All FEKs are encrypted with SEK, whereas OEKs and ACL files are encrypted using the specified folder's encryption key.

Nevertheless, specific types of user requests rely on the use of asymmetric encryption. For asymmetric encryption and hash generation, we rely on the APIs the WolfSSL static library provides. Precisely, we leverage WolfSSL RSA API to encrypt (*wc_RsaPublicEncrypt*) and decrypt (*wc_RsaPrivateDecrypt*) specific types of information (e.g. OEK of User Folder List File), and SHA256 (*wc_Sha256Update*) to generate hash signatures of user public keys stored in the metadata of user folder list file objects.

5.6 Protocol Implementation Details

In the following section, we present the encryption schemes applied on user data, and give implementation details for each of our proposed protocols. All data processing logic and request serving is executed inside the TeeStore Enclave, and data exchange between the enclave and MinIO is facilitated by the MinIO Python Client launched by the Untrusted Application.

Access Control List Check

Each user bucket is associated with an ACL file that determines the set of users authorized to access the bucket's contents. All user requests except user registration and folder creation, involve performing an *ACL check*. ACL's are encrypted using the corresponding folder's encryption key, and are parsed in plaintext only inside enclave memory. To determine if a user public key is listed in the ACL we parse the ACL

file in sequential order. We compare the user provided public key against each key listed in the ACL using the *memcmp* C function that compares two blocks of memory. Upon successfully passing the ACL check, the enclave proceeds to serving the user request. Otherwise, it rejects requests submitted by unauthorized users.

5.6.1 User Registration

User registration requires a series of steps executed inside the TeeStore Enclave and by the Untrusted Application. The enclave receives the registration request by the client application which contains the requesting user's public key and proceeds to serving the request (*enc_register_user*). Figure 5.5 depicts the process of serving a user registration request.

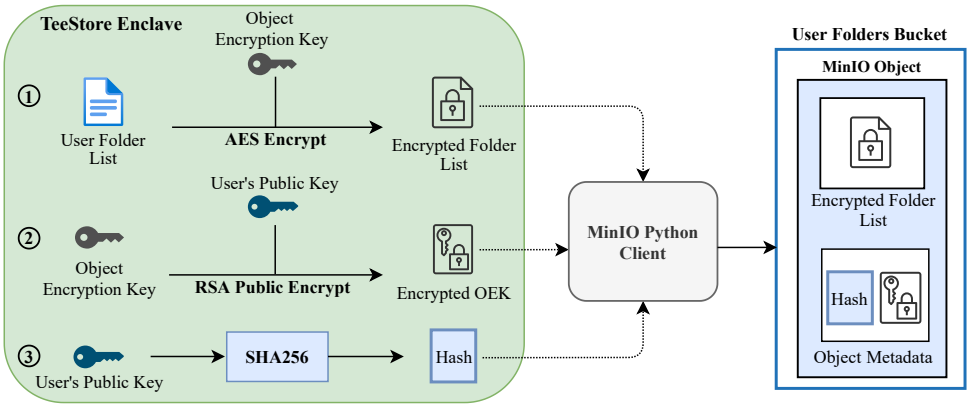


Figure 5.5: User registration request implementation.

Initially, the enclave allocates an empty character buffer that is the user's folder list file. Inside the enclave: (1) a new object encryption key is randomly generated, and the buffer is encrypted (*encrypt_message*) using OEK, (2) OEK is encrypted using RSA encryption with user's public key, and (3) the system generates the SHA256 hash of the public key. The encrypted folder list file, the encrypted OEK, and the hash of the user's public key are written to untrusted storage. Then the Untrusted Application uses the MinIO Python Client to pass on the three pieces of information to the object store. The folder list file is uploaded (*fput_object*) as a MinIO object in the User Folders Bucket, and the encrypted OEK and public key hash are stored as part of the object's metadata fields.

5.6.2 Bucket Creation

To create a new bucket at MinIO, the user attaches her RSA key pair to the request and an initial ACL file for the folder. Upon receiving and parsing the information the enclave serves the request (*enc_create_data_folder)*, by reading the user folder list file and the encryption key inside the enclave region as we show in Figure 5.6.

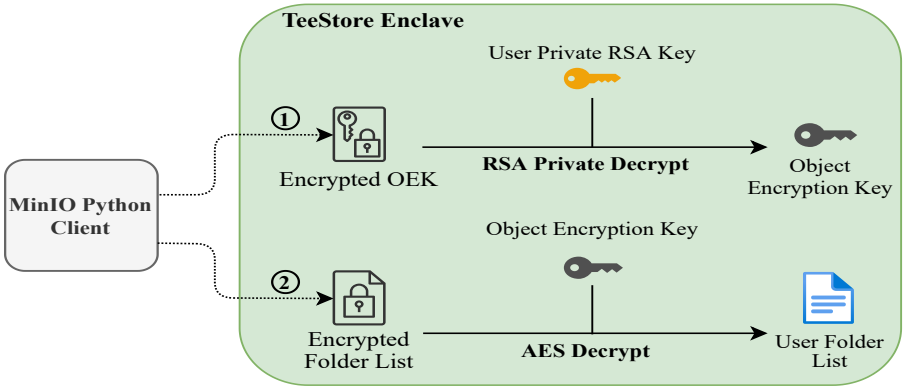


Figure 5.6: Accessing folder list files inside enclave memory for bucket creation.

The system parses the request and issues the MinIO client to get the user folder list file object from MinIO (*fget_object*). The MinIO client searches for the user's public key hash in the metadata fields (*stat_object*) of the objects stored in the User Folders Bucket, and returns the encrypted folder list file along with its encrypted OEK to the enclave. To access the folder list file the enclave: (1) decrypts the encrypted OEK using the user's RSA private key, and (2) decrypts the file with the decrypted OEK. The file's contents are loaded to a buffer, the buffer is re-allocated to append the bucket name for the bucket to be created, and the bucket name is written to the end of the file buffer. The file buffer and the key are re-encrypted, and then re-uploaded to MinIO as an object. As a last step, the enclave generates a folder encryption key with which it encrypts the initial ACL file. Thereafter, the MinIO client issues a create bucket request (*make_bucket*) to MinIO and uploads the ACL file to the newly created bucket.

5.6.3 Uploading Data

Uploading a file to TeeStore involves sending the original file in chunks via TLS to the service's enclave. The file is accumulated inside the data buffer structure and the enclave proceeds to serving the upload request (*enc_upload_file*).

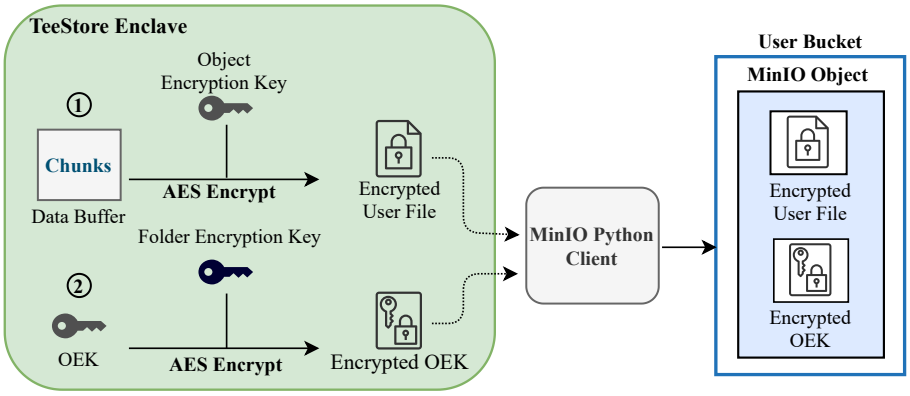


Figure 5.7: Encrypting user file in the TeeStore enclave and uploading it to a MinIO bucket.

Figure 5.7 depicts the process of encrypting and eventually uploading a file to TeeStore. The data buffer containing the file chunks is encrypted using a randomly generated object encryption key (OEK) for AES encryption. Inside the enclave: (1) the file is encrypted using OEK, and (2) OEK is encrypted using the destination folder's encryption key (FEK). The encrypted file and the encrypted OEK are passed to the MinIO client and uploaded to MinIO as an object with the encrypted key in its metadata field.

5.6.4 Downloading Data

Downloading an object from MinIO through TeeStore involves a series of steps (*enc_download_file*). First the MinIO client fetches the specified object and its encryption key and loads them inside the enclave. The encryption key and the data are decrypted (*decrypt_message*) within the enclave, and the file is streamed to the requesting user in data chunks of 16KB. Files that are less than 16KB require sending only a single data chunk to the recipient.

5.6.5 Access Control List Update

The process of updating a data folder's ACL is essentially an upload request, with extra security measures included. The user specifies in her request: (1) RSA key pair, (2) the relevant bucket name, and (3) the new ACL file. The system parses the request (*enc_update_ACL*) and issues the MinIO client to fetch the user's folder list file. The client searches the User Folders Bucket for the object that contains the user's public

key SHA256 hash in its metadata, along with its encryption key. Following the same process that is described in section 5.6.2, the folder list file's OEK is decrypted inside the enclave using the user's RSA private key, and thereafter the file is unsealed using the decrypted key and loaded into a buffer. We parse the buffer contents in search for the folder name declared in the user request, and if the name is included in the list we replace the previous ACL file with the fresh one. The new ACL file is encrypted with AES by re-using the folder's encryption key and put back to the user bucket.

5.6.6 Removing an Object

Removing an object from TeeStore requires specifying the bucket name and the object name in the TeeStore Client request. The system parses the request (*enc_delete_file*) performs an ACL check, and launches the MinIO client to perform the delete action. The object is then removed from the service by the MinIO client (*remove_object*).

5.6.7 Listing Bucket Contents

To list the contents of a MinIO bucket (*enc_list_files*), the enclave launches the MinIO client to construct the list of objects contained in the bucket. The client lists the bucket's contents (*list_objects*) and writes the object names to a text file along with the size of each object next to each name. The client passes the file to the enclave, and the enclave transmits the file to the requesting user in chunks of 16KB via TLS.

5.7 Limitations

Our implementation guarantees the secure execution of TeeStore's file-sharing protocols. The implementation leverages Intel's SGX technology to process on user files and encryption keys inside trusted execution environments namely enclaves, and MinIO as the storage back-end managing data as objects. However, a limitation introduced in this implementation is that enclave memory is restricted to EPC limit which is 128MB total, but only around 90MB is usable by the file-sharing service. Moreover, the WolfSSL TLS server does not leverage multi-threading, thus a single request is served at a time. This limitation can be overcome by increasing the number of serving nodes in the system and by extending the EPC limit in future releases of Intel SGX.

Chapter 6

Performance Evaluation

In this chapter we experimentally evaluate our prototype implementation of TeeStore through benchmarks that aim to quantify the performance of our system. The main questions we seek to answer are the following: (a) how much overhead does the interference of the SGX hardware module incur to the execution of TeeStore's file-sharing protocols when compared to their initial execution time on MinIO, (b) how do long ACL files impact our system's performance, and (c) how well does our TeeStore Client perform when executed on different hosts.

6.1 Experimental System Setup

In order to evaluate the performance of TeeStore, we set up an experimental environment consisting of two physical hosts. We denote the first one as the *Server Machine*,

and the second one as the *Client Machine*. The machines are connected to the local area network through a 8-port 1 Gigabit Ethernet Switch.

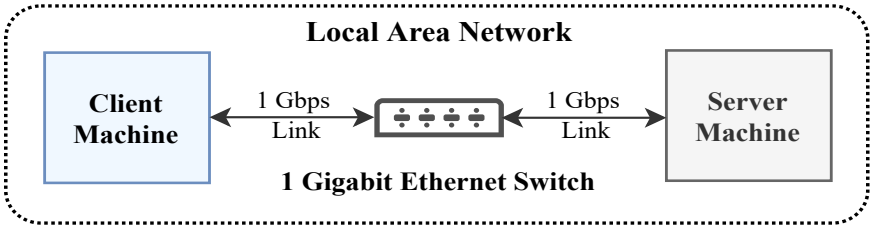


Figure 6.1: Experimentation environment setup.

Figure 6.1 illustrates the system's connection setup. The machines are directly connected to the switch, thus we achieve high data transfer rates between the two hosts to conduct our experiments, with latency being bounded to the networking hardware installed on each individual host. Table 6.1 summarizes the hardware and software installed on each machine.

	Client	**Server**
Operating System	Debian 10 (buster)	Ubuntu 20.04
Linux Kernel	4.19.0-17-amd64	5.8.0-53-generic
Processor Unit	Intel(R) Core(TM) i5-4590	Intel(R) Core(TM) i7-8700K
Intel SGX	Not Supported	Enabled
System Memory	8GiB DDR3 @ 1600 MHz	16GiB DDR4 @ 2666 MHz
Storage Capacity	ATA Disk 1TB (HDD)	2xATA Disk 1TB (HDD)
Ethernet Controller	Intel(R) I217-LM 1GbE	Intel(R) I219-LM 1GbE

Table 6.1: Client and Server hardware specifications.

The server machine is a Dell Precision 3060 Tower equipped with an Intel i7-8700 processor at 3.70GHz with 6 physical cores (12 logical threads), 16GB DDR4 RAM, two SATA HDD disks of 1TB storage capacity each, and an Intel(R) I219-LM 1GbE ethernet controller. The machine is SGX enabled with 128MB maximum enclave size, running the SGX driver, SDK, and platform software version 2.1.2 [27]. Furthermore, the machine uses the Debian-based Ubuntu 20.04.2 LTS 64bit operating system with Linux kernel 5.8.0-53 generic. The Server Machine hosts TeeStore and deploys a standalone version of the MinIO object store built from source. Additionally, we deploy the TeeStore Client we implemented on the server so as to test the performance

of our data-sharing protocols, when both the client application and the service are running on the same host.

On the other hand, the Client Machine is a Dell Precision T1700 Tower with an Intel i5-4590 processor at 3.30GHz with 4 cores and one hardware thread running per core. Moreover, it is equipped with 8GB DDR3 RAM and one SATA HDD disk of 1TB size. The machine's CPU does not support Intel SGX, and it runs the Debian 10 (Buster) operating system with Linux kernel 4.19.0-17-amd64. Finally, we deploy the same version of the WolfSSL client application that we use on the server machine to forward data-sharing requests to the service. Even though the server machine has more advanced hardware components installed than the client machine, we assume that this will not affect much the results we gain from our experiments, since the TeeStore Client's hardware requirements are limited.

6.2 Methodology

Next, we discuss the approach we followed to evaluate the performance of TeeStore. The main goal of our benchmarks is to quantify the performance of our implemented protocols executing on our SGX-enabled file-sharing system. In our benchmarks, we do not examine the behavior of our system when executing the User Registration, and Create Data Folder requests. This is due to the fact that the User Registration request is executed only once for each user interacting with the service, and the Create Data Folder request does not impose a performance demanding process.

Each of our experiments is repeated 5 times, and we plot the computed average value of our collected experiment samples. Our dataset for testing the system's performance is composed of binary files ranging from 1KB up to 90MB which is the maximum usable EPC limit. To enable a 90MB input file, a 64MB stack and 256MB heap were configured for the enclave in all scenarios, and the maximum stack size was set 128MB for the Untrusted Application.

Finally, we deployed a MinIO server in standalone mode to compare the performance of a subset of our protocols against: (1) MinIO without encryption (Baseline), and (2) MinIO with server-side encryption (SSE). MinIO with SSE enabled uses AES with Galois/Counter Mode with 256-bit encryption keys, whereas the TeeStore Enclave utilizes the same algorithm but with 128-bit keys.

6.3 Protocol Benchmarks

In this section we present the results obtained from conducting our experiments on TeeStore. In the following experiments, we use a constant ACL size comprised of 10 user public keys. Then, we proceed to benchmark the performance of TeeStore by deploying the TeeStore Client on the same machine that hosts the service.

6.3.1 File Upload Time

First, we analyze the cost of uploading a file to TeeStore via the TeeStore Client. The files we upload to the service are of scaling size up to 90MB. The cost of uploading a file to the service includes: (1) transmitting the file in data chunks to the service's enclave, (2) encrypting the collected data inside the TeeStore Enclave, and (3) uploading the encrypted file to the object store through the MinIO client.

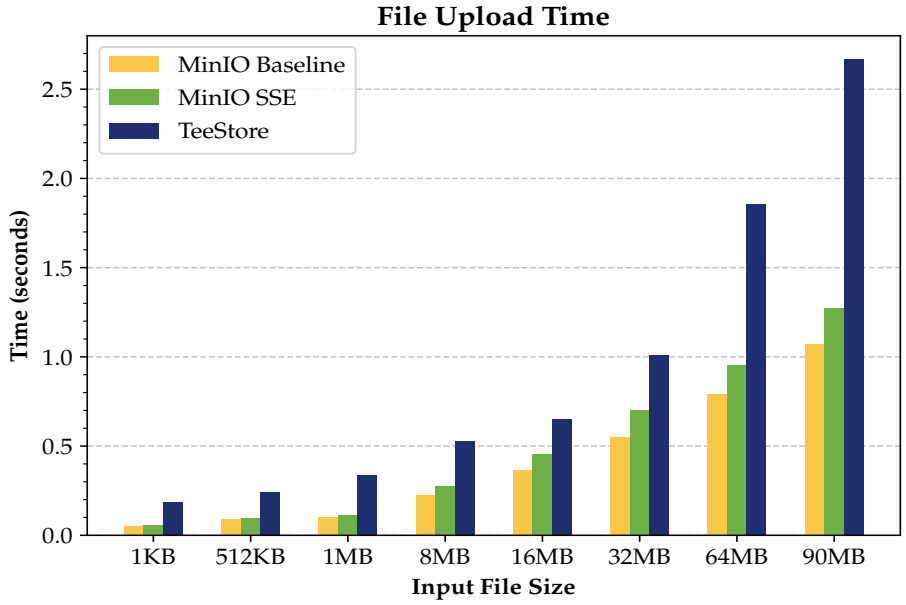


Figure 6.2: File upload time comparison between MinIO Baseline, MinIO SSE, and TeeStore.

In Figure 6.2 we measure the time needed to upload the input files to the service and to MinIO. For files up to 32MB size, TeeStore introduces reasonable overhead of less than 33%, when compared to MinIO Baseline and MinIO SSE, and the system presents consistent upload time of roughly one second for a 32MB file. The overhead observed during upload is due to the presence of ACL's in our implementation, and

the cost of transmitting the data to the enclave via TLS. Files of 64MB and 90MB size require approximately twice the time needed by MinIO using SSE encryption, which seems to be in compliance with what we initially expected.

6.3.2 File Download

The next aspect of our service's performance, is the time needed to reclaim a user file from TeeStore. Downloading files contributed to a cloud service apparently constitutes one of the most commonly used features by users. Thus, it is of major importance to present the download speed of our service. The main operations involved to download a file from TeeStore are: (1) fetching the encrypted file from MinIO, (2) decrypting the file inside the TeeStore Enclave, and (3) sending the original file to the requesting client.

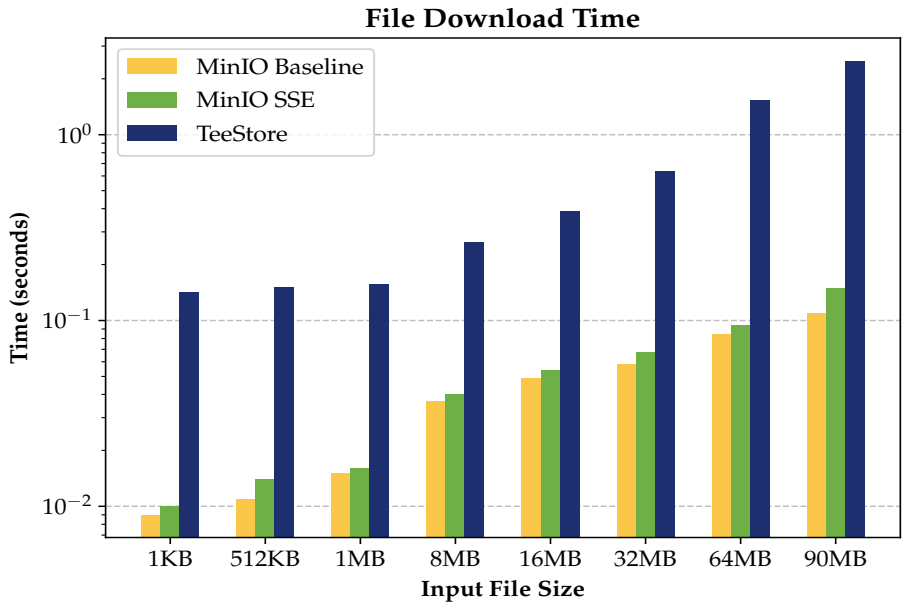


Figure 6.3: File download time comparison between MinIO Baseline, MinIO SSE, and TeeStore.

As shown in Figure 6.3, the y-axis representing the time needed to download a file is in logarithmic scale because we observed a significant deviation between the times required in our three schemes. In a high level overview, TeeStore seems to present significant performance overheads compared to the download time needed by MinIO. This is due to the fact that our implementation of the download request involves using the same WolfSSL library functionality (*wolfSSL_read & wolfSSL_write*)

as in the upload process. Thus, the data transmission time from the server to the TeeStore client is expected to produce similar results to the upload time. Moreover, MinIO supports disk caching to store data closer to tenants, which has an impact on its download speed, whereas our service does not support any form of caching. However, TeeStore displays an almost linear download time, needing approximately 1.6 seconds to download a 64MB file, and 2.5 seconds for 90MB.

6.3.3 List Files

Next, we evaluate the behavior of TeeStore when users request to list the contents of a folder they are authorized to access. Essentially, the system constructs a list of the object names stored in a particular folder, along with each object's size. The cost of obtaining the list involves: (1) constructing the list via the MinIO client, and (2) sending the list to the requesting client through the TeeStore Enclave. Our testbest includes MinIO buckets populated with up to 10^5 objects.

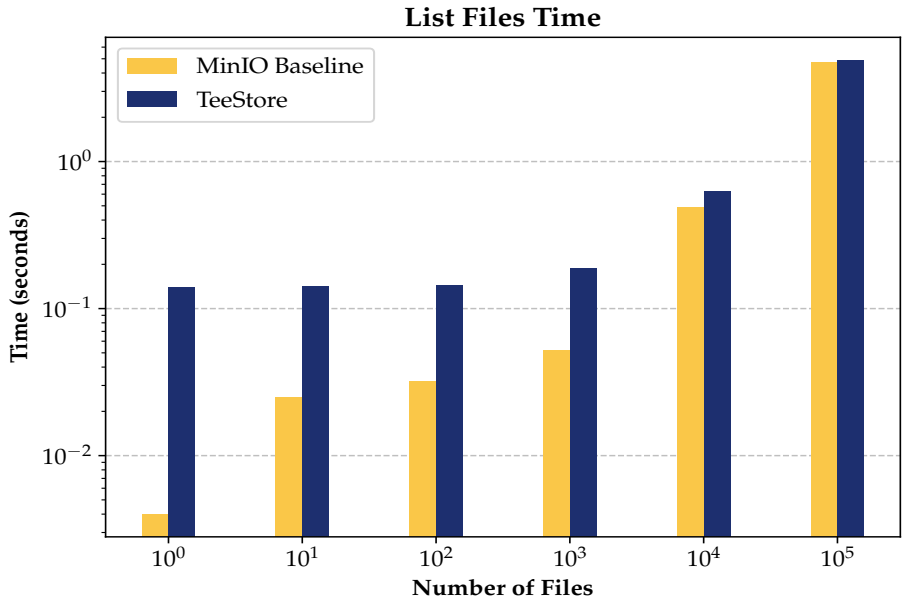


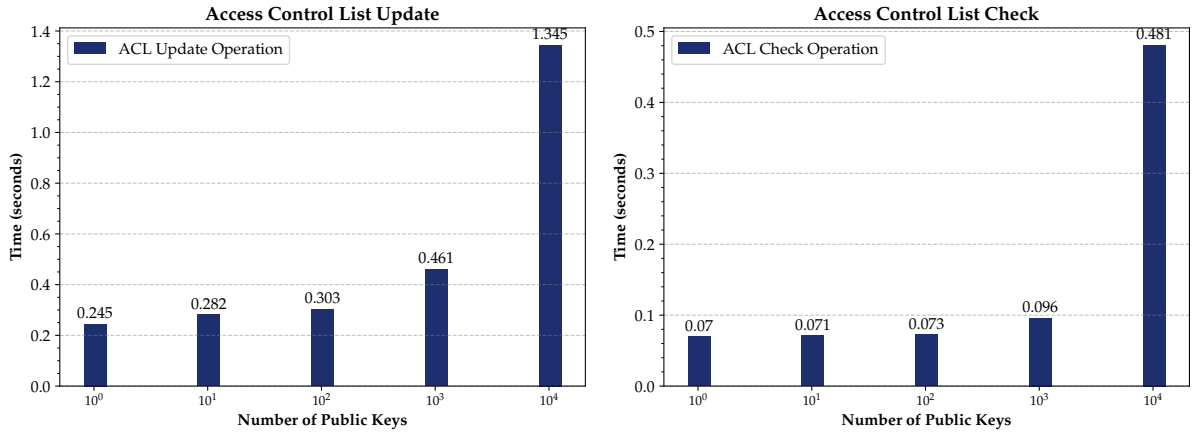
Figure 6.4: Time to list bucket contents with scaling number of objects.

Figure 6.4 shows the results. The time needed to return the list of files in a bucket is in logarithmic scale. Listing the contents of buckets with up to 10^3 objects takes less than 0.2 seconds, with MinIO Baseline performing better as we anticipated. Nevertheless, the construct of lists with orders of magnitude higher object count (that is 10^5) is almost identical in both cases, with TeeStore incurring only a 5% performance

overhead in the time needed to execute the request.

6.4 Access Control List Benchmarks

Next, our evaluation focuses on quantifying the influence of access control lists on TeeStore's performance. In a real-world scenario, there are situations where a large group of users is granted access to files residing in the same data folder in a cloud service. Access to the contents is restricted by the use of ACLs that are populated with the public keys of authorized users. Our benchmarks include the use of ACL files consisting of up to 10^4 public keys in logarithmic scale. Our goal is to highlight the time needed: (1) to update a data folder's ACL, (2) to search for the presence of a user's public key in the ACL, (3) to execute the file-sharing protocols with respect to ACLs of scaling size.



(a) Access control list update time. (b) Access control list public key search time.

Figure 6.5: ACL update and search times with scaling number of user public keys.

The results are shown in Figures 6.5(a) and 6.5(b). In each case we constructed ACL files populated with increasing number of user public keys. Figure 6.5(a) measures the time needed to update the ACL of a specific data folder. The time required to update an ACL involves: (1) transmitting a new ACL file to the service's enclave, (2) encrypting the updated ACL, and (3) replacing the old ACL file with the new ACL in the corresponding MinIO bucket. ACL files with up to 10^3 public keys require an almost constant update time of 0.3 seconds, while ACL's listing 10^4 users need less than 1.4 seconds which is an acceptable amount of time. On the other

hand, Figure 6.5(b) illustrates the time needed to perform an *ACL Check*. An ACL check determines if a specific user public key is listed in an ACL file by decrypting a data folder's ACL, and searching for the public key in it. ACLs containing up to 10^3 keys incur only a minimal overhead of less than 0.1 seconds to perform the ACL check, while for ACLs listing 10^4 keys the system needs approximately less than half a second.

In the next experiment, we measure the impact of long ACLs to the execution time of our implemented protocols. To run our experiment, we use a constant file size of 1MB and examine the performance of the file upload, download and delete operations of TeeStore.

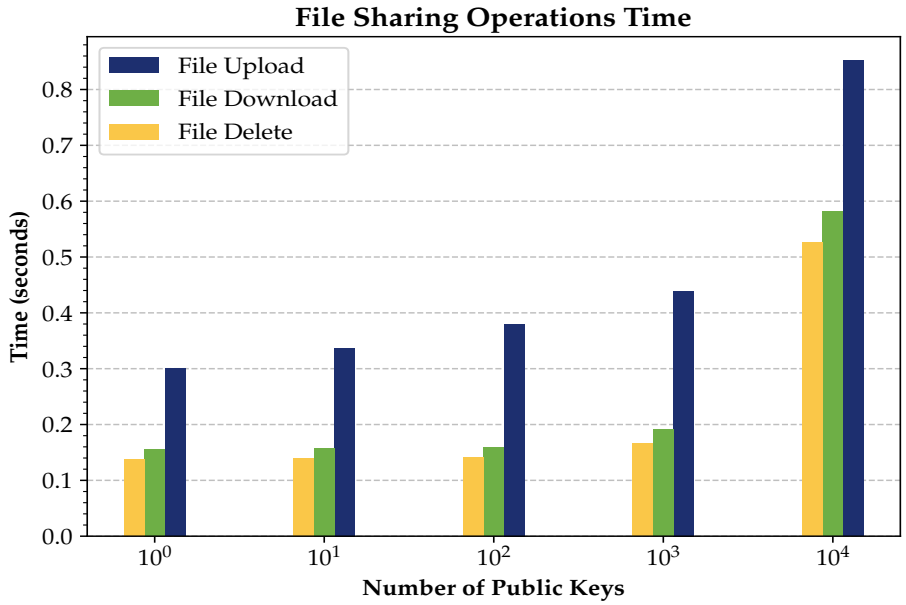


Figure 6.6: Impact of access control lists on TeeStore's performance.

Figure 6.6 summarizes the obtained results. By increasing the ACL size up to 10^3 user public keys, the execution of the download and delete operations are almost not affected, while the data upload presents a small increase of approximately 0.1 seconds in its execution time. For an ACL file comprised of 10^4 public keys, the execution time of all three protocols presents an increase of 0.4 seconds, which is in compliance with the results we observed in Figure 6.5(b).

6.5 Networked Evaluation

This section evaluates the performance of TeeStore with a two machine setup. As in the execution of our previous experiments TeeStore operates on the server machine, however we now deploy the TeeStore Client on the client machine, and compare its performance to the results we obtained while running the client application on the same host as the file-sharing service.

Prior to comparing the performance of TeeStore with client's running on different hosts, we examine the importance of defining the correct TLS data chunk size for our experiments. Our goal is determine which is the optimal TLS chunk size that benefits the execution of our service's protocols. In this experiment, we quantify the performance of the primary WolfSSL functions we leverage in our implementation with TLS data chunks of 1KB, 2KB, 4KB, 8KB, and 16KB. For this experiment, instead of 1MB file we used a 10MB in order to tenfold the number of TLS chunks needed to be transferred for serving the requests.

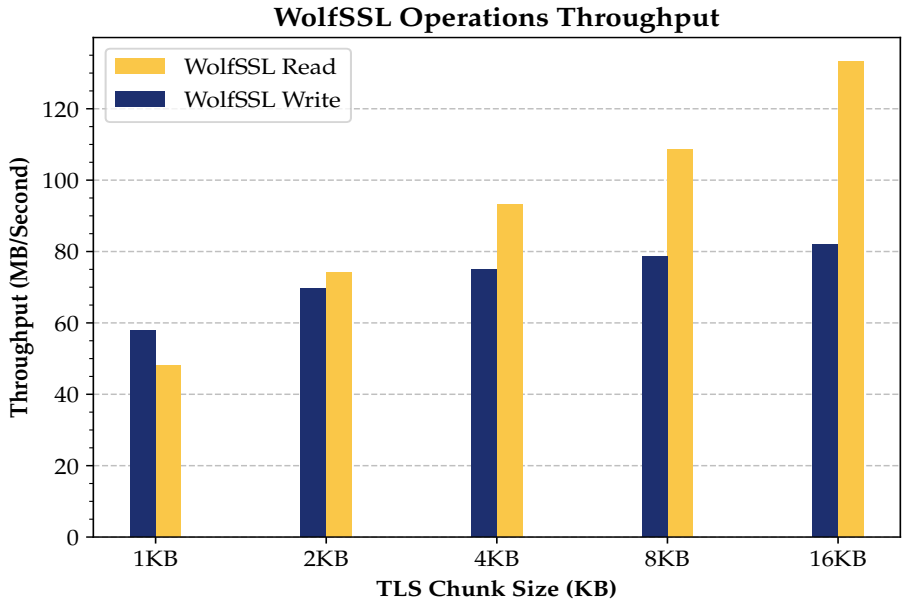


Figure 6.7: WolfSSL operations data throughput for increasing TLS chunk size.

Figure 6.7 presents the throughput of the TeeStore client application for different sizes of TLS data chunks. Decrypting and reading data received by a TLS communication channel requires a call to *wolfSSL_read* function, while data transmission is handled using *wolfSSL_write*. As shown in Figure 6.7, increasing the size of the TLS data chunk generally improves the throughput of our system. Increasing the TLS

chunk size linearly enhances the performance of both read and write operations. By setting the chunk size to 16KB that is the *maximum TLS record size* [43], read operations achieve an average throughput of 133MB per second, while write operations display approximately 82MB data throughput per second.

Following, we use the optimal TLS chunk size of 16KB to measure the execution time of the upload and download operations, by running the TeeStore client application at: (1) the same host that deploys TeeStore, (2) a separate machine (Client) with slightly less performant hardware installed on it.

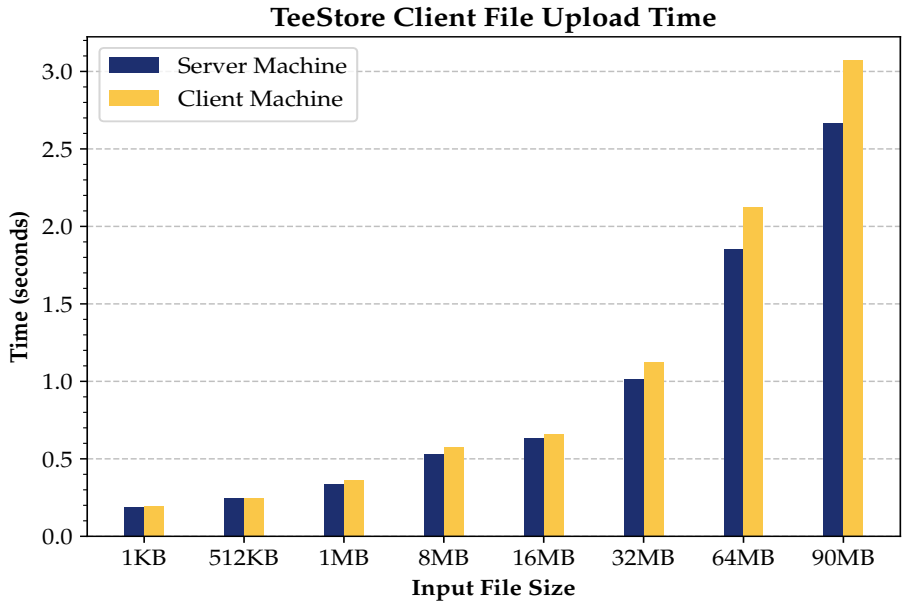


Figure 6.8: File upload time with TeeStore Client running on different hosts.

Figure 6.8 reports the time needed to upload a file to TeeStore. In all scenarios, the server machine outperforms the client machine needing less time to upload the input files. For files up to 16MB, both schemes demonstrate similar trends with minimal deviations of less than 0.1 seconds. By increasing the size of the input files, we observe that the client machine demonstrates a 15% overhead in the upload time for 90MB file, which is acceptable considering the individual hosts hardware specifications.

Next, we study the download speed the client and server machines present in Figure 6.9. As in the previous experiment, for files up to 16MB we observe an identical behavior in the time needed to download the file in different hosts, with the server machine performing slightly better. By measuring the download time with an 90MB input file, the client machine scheme demonstrates only a minor overhead of 1%

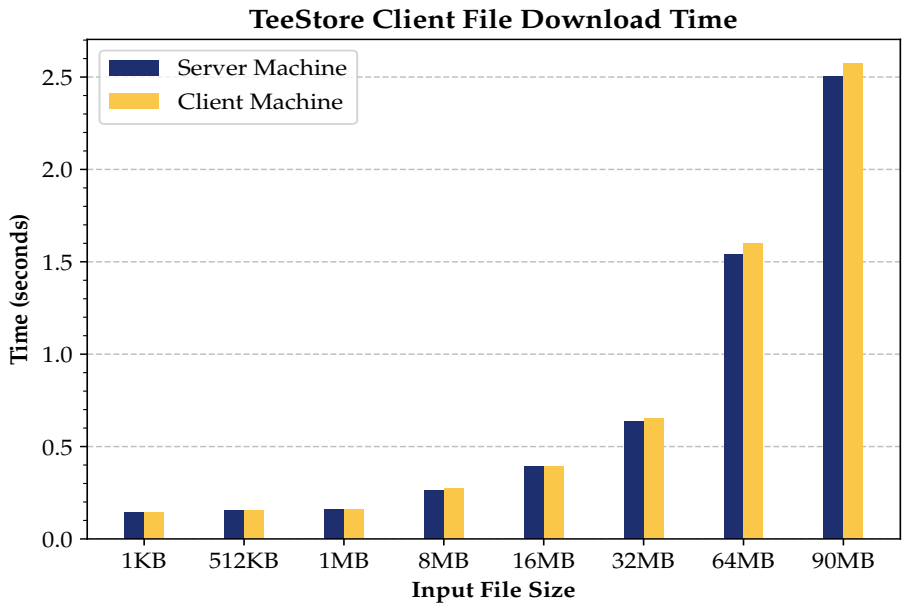


Figure 6.9: File download time with TeeStore Client running on different hosts.

which does not affect the efficiency of download requests executed on different hosts.

6.6 Summary

In this chapter we experimentally evaluated the performance of TeeStore. Initially, we quantified the performance of our proposed protocols and compared the results to the performance of MinIO without encryption and with SSE encryption. Our results indicate that with reasonable performance overheads we provide stronger security guarantees leveraging the use of enclaves. Furthermore, we explored the impact of ACLs in our implementation, and our experiments demonstrate that long ACLs incur acceptable latency to the execution of our protocols. Finally, we reasoned about the performance of our protocols by deploying the TeeStore client application to a separate host, and conclude that this scheme imposes minimal decline in the throughout of TeeStore's services.

Chapter 7

Conclusions & Future Work

In the present chapter, we discuss the main findings of our research and suggest potential future improvements of our work.

7.1 Conclusions

Storing and sharing data in the cloud poses a challenging problem. Individuals and organizations contribute their privately held data in the cloud in order to share it with others, and collaboratively edit on the data. The privacy and confidentiality of user data relies on the security mechanisms employed by cloud providers. However, even though cloud providers provide strong security guarantees to their customers against external attackers, user data is still at risk by privileged employees of the provider. Thus, our adversary is a curious-but-honest cloud storage provider. Solutions that depend on software to secure user data are considered to be incompetent, and do not protect against privileged software operating on the provider's systems.

To solve this problem, we designed secure protocols that manage user data leveraging trusted execution environments. We used specialized hardware with Intel's SGX technology to support our protocols, and facilitate secure file sharing in the cloud.

Our proposed protocols process on user files inside trusted memory regions called enclaves that are protected with the use encryption. Furthermore, we employ secure channels to encrypt user data transmitted by client applications to SGX enclaves, thus providing end-to-end security guarantees. Enclaves encrypt user files committed to the service, and also handle secret keys generation. In order to support file sharing among group of users, we introduce access control lists (ACLs) populated with users' public keys, enabling users to control with whom they share their files with.

We implemented a prototype of our secure file-sharing service, integrating our system with a production-level object store that stores encrypted user files as objects, managed in collections of objects called buckets. Furthermore, we implemented a client application that establishes secure communication channels with the service via the use of TLS, and is responsible for propagating file-sharing requests to the service.

To conclude, we evaluated the performance of our file-sharing service against the initial performance of the baseline object store we integrated to our system without the use of secure protocols. We demonstrated that our system: (1) adds reasonable performance overheads to the execution of file-sharing requests when compared to the baseline object storage system, (2) scales well with long ACLs, and (3) secures user data against the considered adversary.

7.2 Future Work

There are several directions for future work regarding the implementation and evaluation of our proposed file-sharing service. In the future, our aim is to extend the number of protocols we support in order to grant users the ability to leverage more complex operations on the file-sharing service. Moreover, our system would benefit if we employ hash-based data structures (e.g., hashtables, hashsets) to minimize the overhead introduced by ACL check operations, for long ACL files. Additionally, it would be of great interest to evaluate the performance of our service with a scaling number of serving nodes participating in it. Furthermore, another important advance would be to develop a responsive web application that integrates the functionality of our implemented client application, and will be responsible for interacting with the service. Last but not least, we expect to overcome EPC memory limitations [46] in future releases of the Intel SGX technology in order to manage larger files.

Bibliography

[1] J. Bulao. How Much Data Is Created Every Day in 2021? [Online]. Available: https://techjury.net/blog/how-much-data-is-created-every-day/#gref

[2] Amazon Simple Storage Service (S3). [Online]. Available: https://aws.amazon.com/s3/

[3] Dropbox. [Online]. Available: https://www.dropbox.com/

[4] Google Drive. [Online]. Available: https://www.google.com/intl/en_en/drive/

[5] D. Lohrmann. 2020 Data Breaches Point to Cybersecurity Trends for 2021. [Online]. Available: https://www.govtech.com/blogs/lohrmann-on-cybersecurity/2020-data-breaches-point-to-cybersecurity-trends-for-2021.html

[6] M. J. Dworkin, "SP 800-38D. Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC," Tech. Rep., 2007.

[7] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 21, no. 2, p. 120–126, Feb. 1978. [Online]. Available: https://doi.org/10.1145/359340.359342

[8] W. Diffie and M. Hellman, "New directions in cryptography," *IEEE Transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, 1976.

[9] C. Gentry, "A fully homomorphic encryption scheme," PhD Thesis, Stanford University, 2009, crypto.stanford.edu/craig.

[10] Y. Liu, W. Gong, and W. Fan, "Application of AES and RSA Hybrid Algorithm in E-mail," in *2018 IEEE/ACIS 17th International Conference on Computer and Information Science (ICIS)*, 2018, pp. 701–703.

[11] C. Inc. What happens in a TLS handshake? [Online]. Available: https://www.cloudflare.com/learning/ssl/what-happens-in-a-tls-handshake/

[12] wolfSSL Inc. Wolfssl documentation. [Online]. Available: https://www.wolfssl.com/docs/

[13] OpenSSL, Cryptography and SSL/TLS Toolkit. [Online]. Available: https://github.com/openssl/openssl

[14] wolfSSL Inc. *wolfSSL User Manual*, version 4.1.0, august 2019. [Online]. Available: https://www.yassl.com/documentation/wolfSSL-Manual.pdf

[15] Infineon Technologies AG. (2018, Sep.) Securing the connected world with wolfSSL seamless TPM 2.0 integration Whitepaper.

[16] N. Ekker, *File & Object Storage For Dummies*, ibm limited edition ed. John Wiley & Sons, Inc., 2016.

[17] IBM Cloud Education. Object storage. [Online]. Available: https://www.ibm.com/cloud/learn/object-storage

[18] MinIO Quickstart Guide. [Online]. Available: https://docs.min.io/docs/minio-quickstart-guide.html

[19] The Go Programming Language. [Online]. Available: https://golang.org/

[20] B. Parno, J. M. McCune, and A. Perrig, "Bootstrapping trust in commodity computers," in *2010 IEEE Symposium on Security and Privacy*, 2010, pp. 414–429.

[21] W. Arthur and D. Challener, *A Practical Guide to TPM 2.0: Using the Trusted Platform Module in the New Age of Security*, 1st ed. USA: Apress, 2015.

[22] M. Sabt, M. Achemlal, and A. Bouabdallah, "Trusted Execution Environment: What It is, and What It is Not," in *2015 IEEE Trustcom/BigDataSE/ISPA*, vol. 1, 2015, pp. 57–64.

[23] D. Kaplan, J. Powell, and T. Woller. (2016) AMD Memory Whitepaper. [Online]. Available: https://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf

[24] T. Alves and D. Felton, "Trustzone: Integrated hardware and software security," 01 2004.

[25] V. Costan, I. Lebedev, and S. Devadas, "Sanctum: Minimal hardware extensions for strong software isolation," in *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, Aug. 2016, pp. 857–874. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/costan

[26] V. Costan and S. Devadas, "Intel SGX Explained," Cryptology ePrint Archive, Report 2016/086, 2016, https://eprint.iacr.org/2016/086.

[27] Intel. *Intel softare guard extensions SDK developer reference for Linux OS*, version 2.1.2, march 2018. [Online]. Available: https://download.01.org/intel-sgx/linux-2.1.2/docs/Intel_SGX_Developer_Reference_Linux_2.1.2_Open_Source.pdf

[28] S. Johnson, V. Scarlata, C. Rozas, E. Brickell, and F. X. McKeen, "Intel® Software Guard Extensions: EPID Provisioning and Attestation Services," 2016.

[29] C. Priebe, D. Muthukumaran, J. Lind, H. Zhu, S. Cui, V. A. Sartakov, and P. R. Pietzuch, "SGX-LKL: Securing the Host OS Interface for Trusted Execution," *CoRR*, vol. abs/1908.11143, 2019. [Online]. Available: http://arxiv.org/abs/1908.11143

[30] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, p. 991–1008. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/bulck

[31] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, "SgxPectre: Stealing Intel Secrets from SGX Enclaves Via Speculative Execution," in *2019 IEEE European Symposium on Security and Privacy (EuroS P)*, 2019, pp. 142–157.

[32] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu, "Plutus: Scalable Secure File Sharing on Untrusted Storage," in *2nd USENIX Conference on File and Storage Technologies (FAST 03)*. San Francisco, CA: USENIX

Association, Mar. 2003. [Online]. Available: https://www.usenix.org/conference/fast-03/plutus-scalable-secure-file-sharing-untrusted-storage

[33] G. Kappes, A. Hatzieleftheriou, and S. V. Anastasiadis, "Multitenant Access Control for Cloud-Aware Distributed Filesystems," *IEEE Transactions on Dependable and Secure Computing*, vol. 16, no. 6, pp. 1070–1085, 2019.

[34] F. Wang, J. Mickens, N. Zeldovich, and V. Vaikuntanathan, "Sieve: Cryptographically Enforced Access Control for User Data in Untrusted Clouds," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. Santa Clara, CA: USENIX Association, Mar. 2016, pp. 611–626. [Online]. Available: https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/wang-frank

[35] T. Kim, J. Park, J. Woo, S. Jeon, and J. Huh, "ShieldStore: Shielded In-Memory Key-Value Storage with SGX," in *Proceedings of the Fourteenth EuroSys Conference 2019*, ser. EuroSys '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3302424.3303951

[36] B. Fisch, D. Vinayagamurthy, D. Boneh, and S. Gorbunov, "IRON: Functional Encryption Using Intel SGX," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 765–782. [Online]. Available: https://doi.org/10.1145/3133956.3134106

[37] S. Contiu, R. Pires, S. Vaucher, M. Pasin, P. Felber, and L. Réveillère, "IBBE-SGX: Cryptographic Group Access Control using Trusted Execution Environments," *CoRR*, vol. abs/1805.01563, 2018. [Online]. Available: http://arxiv.org/abs/1805.01563

[38] Microsoft. Always Encrypted with secure enclaves. [Online]. Available: https://docs.microsoft.com/en-us/sql/relational-databases/security/encryption/always-encrypted-enclaves?view=sql-server-ver15

[39] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest We Remember: Cold-Boot Attacks on Encryption Keys," *Commun. ACM*, vol. 52, no. 5, p. 91–98, May 2009. [Online]. Available: https://doi.org/10.1145/1506409.1506429

[40] Intel. Introduction to Intel® SGX Sealing. [Online]. Available: https://software.intel.com/content/www/us/en/develop/blogs/introduction-to-intel-sgx-sealing.html

[41] IBM Coorporation. How SSL and TLS provide identification, authentication, confidentiality, and integrity. [Online]. Available: https://www.ibm.com/docs/en/ibm-mq/7.5?topic=ssl-how-tls-provide-authentication-confidentiality-integrity

[42] R. C. R. Condé, C. A. Maziero, and N. C. Will, "Using Intel SGX to Protect Authentication Credentials in an Untrusted Operating System," in *2018 IEEE Symposium on Computers and Communications (ISCC)*, 2018, pp. 00 158–00 163.

[43] I. Grigorik, *High Performance Browser Networking: What every web developer should know about networking and browser performance*. O'Reilly Media, Incorporated, 2013. [Online]. Available: https://books.google.gr/books?id=X_SsMQEACAAJ

[44] wolfSSL Inc. wolfssl Example Applications. [Online]. Available: https://github.com/wolfSSL/wolfssl-examples

[45] Python Client API Reference. [Online]. Available: https://docs.min.io/docs/python-client-api-reference

[46] S. Johnson. Keynote: Scaling Towards Confidential Computing. [Online]. Available: https://systex.ibr.cs.tu-bs.de/systex19/slides/systex19-keynote-simon.pdf

Short Biography

Evangelos A. Dimoulis was born in Corfu, Greece in 1995. He earned his Bachelor of Engineering from the Department of Computer Science and Engineering of the University of Ioannina in 2019. His B.Sc. thesis was entitled "Development of a Web Application for the Management of Interbank Transactions". Currently, he is a MSc candidate at the same Department and a member of the Systems Research Group of the University of Ioannina. His research interests include systems security, blockchain, distributed systems, data storage, as well as data mining.