EAI Endorsed Transactions
on Security and Safety
Research Article EAI.EU

# SeFS: A Secure and Efficient File Sharing Framework based on the Trusted Execution Environment

Yun He[1,2], Xiaoqi Jia[1,2*], Shengzhi Zhang[3], Lou Chitkushev[3]

[1]Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
[2]School of Cyber Security, University of Chinese Academy of Sciences,Beijing, China
[3]Computer Science Department, Boston University Metropolitan College, Boston, USA

## Abstract

As the cloud-based file sharing becomes increasingly popular, it is crucial to protect the outsourced data against unauthorized access. Existing cryptography-based approach suffers from expensive re-encryption upon permission revocation. Other solutions that utilize Trusted Execution Environment (TEE) to enforce access control either expose the plaintext keys to users or turn out incapable of handling concurrent requests. In this paper, we propose SeFS, a secure and practical file sharing framework that leverages cooperation of server-side and client-side enclaves to enforce access control, with the former responsible for registration, authentication and access control enforcement and the latter performing file decryption. Such design significantly reduces the computation workload of server-side enclave, thus capable of handling concurrent requests. Meanwhile, it also supports immediate permission revocation, since the file decryption keys inside the client-side enclaves are destroyed immediately after use. We implement a prototype of SeFS and the evaluation demonstrates it enforces access control securely with high throughput and low latency.

Received on 14 November 2022; accepted on 30 June 2025; published on 18 July 2025

**Keywords:** Cloud storage, Trusted Execution Environment, Access Control



doi:10.4108/eetss.v9i1.2854

## 1. Introduction

Cloud-based file sharing has been widely adopted by many companies and individuals during the last few years. Relying on the cloud infrastructure, users gain huge storage capacity, access files conveniently across different geographic regions as well as multiple devices, and share files easily by point-and-click. Many commercial cloud providers, e.g., Google Drive, Dropbox, One Drive, etc., offer such services. In such cloud-based file storage and sharing model, the cloud providers enforce access control policies on behalf of users to ensure only the authorized users can access the data. However, the cloud providers cannot always be fully trusted, and may either peek at the outsourced data deliberately or leak it to unauthorized parties incidentally, which clearly violates the users' access control policy.

Several schemes [1–5] based on cryptographic primitives (e.g., broadcast encryption [6], attribute-based encryption [7], etc.) have been proposed to perform access control faithfully given the cloud providers are untrusted. Both A-SKY [8] and IBBE-SGX [9] utilize Intel SGX [10–12] to address the overhead of cryptographic operations, i.e., anonymous broadcast encryption (ANOBE) [1] and identity-based broadcasting encryption (IBBE) [13]. However, the main drawback of the above cryptographic approach is that users gain plaintext access to the keys used to encrypt/decrypt files. Consequently, the data owner has to re-encrypt (involving expensive cryptographic operations) the corresponding files with a new key to achieve permission revocation, which introduces significant performance overhead [14]. SeGShare [15] and NEXUS [16] are pure TEE-based schemes and support immediate permission revocation without re-encrypting files using new keys. However, SeGShare does not support concurrent requests well, since both the encryption and decryption operations inside

*Corresponding author. Email: jiaxiaoqi@iie.ac.cn

the server-side enclave introduce significant overhead. NEXUS embeds access control policies and keys into a data volume protected by the client-side enclave. Therefore, accessing the whole data volume is needed even if the user requests only one file in the data volume. Furthermore, the data owner has to remain online to transmit the rootkey of the data volume to the authorized user.

In this paper, we propose and implement **SeFS**, a secure and practical file sharing system that ensures the separation of authorization (i.e., access control check) and data access (i.e., data encryption or decryption), as well as immediate permission revocation without re-encryption. In particular, the design of SeFS involves the cooperation of one server-side enclave and multiple client-side enclaves, with the permission check performed by the former and the data decryption performed by the latter. Therefore, the data decryption overhead is distributed to each individual client enclave, allowing the server enclave to focus on the authentication and access control check to support concurrent requests. Since even the authorized users do not have the access to the decryption keys directly (only their enclaves have), there is no way for malicious users to keep the keys locally to evade later permission revocation.

SeFS is also a purely TEE-based solution, not relying on cryptographic access control schemes. Furthermore, SeFS does not depend on the underlying file systems, which makes it flexible and easy to cooperate with existing cloud storage services (with few changes to them). It also supports data sharing with finer granularity securely and easily, e.g., users still share files by point-and-click. We implemented SeFS based on Intel SGX [10–12], a popular TEE solution provided by Intel processors[1], and deployed SeFS with an open-source file storage and sharing solution, ownCloud [17], with a tiny change (i.e., less than 10 lines of PHP code) to it. To achieve high throughput, SeFS implemented an efficient SGX-enabled TLS stack based on mbedtls [18] using switchless ecalls/ocalls for network and file operations.

The main contribution of this paper is summarized as below:

(1) We propose a novel architecture that leverages the cooperation of server-side and client-side enclaves to decouple the access control check and data decryption operations. SeFS transfers the decryption task to the client-side enclaves, so the server-side enclave only performs relatively lightweight authorization operations to improve throughput.

(2) We propose a series of protocols based on TEE's security features, e.g., unforgeable enclave measurement, to enable trust validation between the client-side enclaves and the server-side enclave, thus securing data transmission between them.

(3) We implement a prototype of SeFS using the SGX-enabled TLS stack to support efficient I/O operations. SeFS can be easily deployed with existing cloud-based file storage services, e.g., ownCloud. The implementation only contains 4,489 lines of code (LoC), with 2,984 LoC for the server-side enclave and 1,505 LoC for the client-side enclave. The evaluation results show that SeFS can securely and efficiently share files with high throughput and low latency.

The rest of this paper is organized as follows. Section 2 presents the background of Intel SGX and cloud-based file sharing. Section 3 details the threat model of SeFS and the general design requirements for similar solutions. The design and implementation of SeFS are introduced in Section 4 and Section 5 respectively. We evaluate the performance of SeFS in Section 6, and discuss potential attacks as well as the corresponding countermeasures in Section 7. Finally, we present related work in Section 8 and conclude in Section 9.

## 2. Background

### 2.1. Trusted Execution Environments (TEEs)

Trusted Execution Environments (TEEs) provide hardware-supported isolated environments where sensitive data can be securely processed, code can be verified via measurement and attestation. Existing TEEs[2] designs mainly support three levels of TEE abstractions, process-based (e.g., Intel SGX [10–12]), virtualization-based (e.g., AMD SEV [21], Intel TDX [22] and ARM CCA [23]) and separate worlds (ARM TrustZone [24]). In this section, we first provide a detailed introduction to Intel SGX, as the prototype system of SeFS is based on it. Following that, we will give an overview of other TEE technologies and conduct a brief comparison.

**Intel Software Guard Extensions (SGX).** Intel SGX [10–12] is an instruction set extension that creates an isolated execution environment, allowing applications to maintain data confidentiality and integrity. Even the privileged software, e.g., OS, hypervisor or BIOS, cannot violate its protection.

**Enclave**. Intel SGX creates a trusted execution environment called an *enclave* and an encrypted

[1] Although our implementation of SeFS is based on Intel SGX, the similar idea can work on other hardware-assisted TEE platforms, e.g., ARM TrustZone, AMD SEV, etc.

[2] Due to spatial limitations, this work mainly focuses on the primary industrial platform providers. Others which are lack commercial availability were excluded, such as Keystone [19] and Penglai [20] for RISC-V and so on.

memory region called Enclave Page Cache (EPC) for the enclave to store code and data. SGX uses a hardware Memory Encryption Engine (MEE) [25] to encrypt/decrypt the enclave data, and also provides a hardware access control mechanism to prevent any illegal access to the enclave memory. An Intel SGX application generally consists of two parts: secure code (running in the enclave, also called enclave code) and non-secure code (also called non-enclave code). `ecall`/`ocall` interfaces are used to switch control between them. Since the enclave code runs in the user mode, i.e., ring 3, privileged operations such as system calls cannot be executed inside the enclave. Hence, the secure code in the enclave needs to invoke `ocall` to execute those privileged operations indirectly. Before executing the enclave code, CPU switches to the enclave mode and jumps to the predefined enclave entry-point.

**SGX remote attestation**. Intel SGX remote attestation [26] ensures that the enclave is correctly initialized on a remote SGX enabled platform by evaluating the enclave identity, its structure, and the integrity of the code inside the enclave. Furthermore, remote attestation can also provide a shared secret between the enclave application and its owner to setup a secure communication channel over the untrusted network.

**Sealing**. An enclave can write confidential data to the persistent storage securely by means of *sealing* [26], a mechanism to encrypt and authenticate the enclave data. Each enclave is provided with a sealing key, derived from either the Enclave Identity or the Signing Identity. This sealing key is private to the executing platform and the enclave. Data sealed against the Enclave Identity can only be unsealed by the same enclave, whereas data sealed against the Signing Identify can be unsealed by any enclave signed by the same developer.

**Switchless Calls**. An enclave transition, i.e., switching into or out of an enclave, will introduce performance overhead to SGX applications, due to the execution context save/restore and the security check. Therefore, Intel SGX SDK provides switchless calls [27], a technique to reduce the enclave transition overhead. Specifically, the non-enclave code encapsulates ecalls (the calls into the enclave) into tasks and submits these tasks into an untrusted buffer. Then enclave worker threads asynchronously read tasks from this buffer and perform those tasks. Finally, the non-enclave code reads results of those tasks from the same buffer. Similarly, ocalls (the calls out of the enclave) can be handled by the untrusted threads of the non-enclave code in a similar way.

**Virtualization-based TEEs.** For VM-based TEEs, their threat model assumes that the privileged software, such as hypervisors or host operating systems, may be untrustworthy or adversarial. They typically design a customized, minimal (or lightweight) privileged hypervisor (or monitor) using hardware-assisted virtualization techniques, along with hardware encryption, to implement the TEE mechanism. The VM-based TEEs usually treat the entire virtual machine running inside the isolated domain as the trusted execution environment.

**Intel Trust Domain Extensions (TDX)** [22] is an architectural extension that provides TEE capabilities with conducting the Virtual Machine (VM) as a secure and isolated environment. Intel TDX introduces the Secure-Arbitration Mode (SEAM) to offer cryptographic isolation and protection for Virtual Machines (VMs), which are called Trust Domains (TDs) in the TDX terminology. Intel TDX aims to protect the confidentiality and integrity of CPU state and memory for designated TDs, and also enables TD owners to verify the authenticity of remote platforms. It utilizes the Multi-key Total Memory Encryption (MKTME) [28] to perform transparent encryption/decryption for the cryptographic isolation. It uses the TDX Module [29] to facilitate the construction, execution, and termination of TDs while enforcing the security guarantees.

**AMD Secure Encrypted Virtualization (SEV)** [30] leverages the AMD Secure Memory Encryption (SME) [31] and AMD Virtualization (AMD-V) to enforce cryptographic isolation between VMs and the hypervisor. Each VM is assigned a unique ephemeral Advanced Encryption Standard (AES) key, which is used for runtime memory encryption. The AES engine integrated into the memory controller handles the encryption and decryption of data transparently as it is written to or read from the primary memory. The keys for each VM are managed by the AMD Platform Security Processor (PSP), an independent Arm processor embedded within the AMD System-on-Chip (SoC). SEV also incorporates a remote attestation feature to verify the integrity of a VM's launch measurements and the SEV-enabled platforms. AMD SEV-ES (Encrypted State) [32] extends AMD SEV by securing the CPU register state during the hypervisor transitions, and SEV-SNP (Secure Nested Paging) [21] introduces integrity protection to defend against memory corruption, replay, and remapping attacks. Particularly, SEV-SNP conducts memory integrity protection by using the Reverse Mapping Table (RMP).

**ARM Confidential Compute Architecture (CCA)** [23] introduces the Realm Management Extension (RME) with two additional worlds, the Realm World and the Root World. The Realm World provides secure and isolated execution environments for confidential VMs where workloads run securely. The confidential VMs are isolated from any other domains, including host operating systems, hypervisors, other Realms and the TrustZone. ARM CCA uses a Granule Protection Table (GPT) to enforce isolation of address spaces,

which is an extension to the page table that tracks the ownership of each page with different worlds. The Monitor running inside the Root World is responsible for the creation and management of the GPT, preventing hypervisors or operating systems from directly changing it. The Monitor has the capability to transfer physical memory between worlds by modifying the GPT. ARM CCA aslo provides attestation capabilities to verify both the CCA platform and the initial state of the Realms.

**TrustZone.** ARM TrustZone [24] follows a System-on-Chip (SoC) and CPU system-wide approach to security. This technology revolves around the notion of protection domains known as the secure world and the normal world. The processor runs either in the secure world or non-secure world (i.e., normal world). The secure monitor (a privileged software) is responsible for enforcing secure context switches between the two worlds. Both worlds are entirely hardware isolated and granted uneven privileges, with non-secure world prevented from directly accessing secure world resources. The critical applications can run inside the secure world without depending on the rich OS (running inside normal world) for protection. TrustZone has been widely available on commodity mobile devices and has become quite mature, and there are many critical applications (e.g., facial recognition) that operate within the secure world of TrustZone.

For a brief comparison, Intel SGX is a process-based TEE technology. Compared to VM-based TEEs and TrustZone, its Trusted Computing Base (TCB) is significantly smaller. However, as a user-space TEE, it cannot perform kernel-mode operations. VM-based TEEs treat the entire VM as a TEE, capable of executing both user-space and kernel-mode operations, offering better software compatibility than SGX. TrustZone also supports kernel-mode operations, but its secure world depends on a thin OS with poor compatibility, making it unsuitable for running large-scale software, although it is sufficient for running SeFS Client (SeFS Client enclave is lightweight and it only depends on AES encryption/decryption and TLS connections, detailed in Section 4) . Additionally, the design principle of SeFS is to be independent of specific TEE hardware platform.

### 2.2. Cloud-based File Storage and Sharing

More and more users and companies choose to outsource their files on remote cloud platforms that provide huge storage space, convenient file sharing service, and user-friendly interfaces, e.g., web UI, to customers to manage their files. There are many popular commercial cloud platforms, e.g., Google Drive, OneDrive, Dropbox, MEGA [33], etc., and open-source platforms, e.g., ownCloud [17], Nextcloud [34], etc. The above-mentioned platforms conceal from users the underlying file systems (usually distributed ones), which are responsible for reading/writing/syncing data blocks. For instance, Amazon S3 object storage service [35] and Alibaba OSS [36] are representative commercial cloud object storage platforms, mainly focusing on vast data storage capacity. They can be used as the external storage backends for the above mentioned cloud platforms, e.g., ownCloud [37]. OpenAFS [38], FastDFS [39], and Ceph [40] are open-source distributed file systems for the cloud platforms. These distributed file systems are usually used to sync data among multiple distributed storage servers.

## 3. Problem Statement

### 3.1. Threat Model

We mainly consider the scenario that employees of an organization share files with their colleagues. The files are stored on the Cloud Storage Service (CSS) and the data owner (e.g., Alice) wants to share her files with one of her colleagues (e.g., Bob) securely and efficiently. For example, Alice can send a sharelink to Bob without undermining the correct enforcement of the access control policy or the confidentiality/integrity of her files.

We assume the system administrator of the organization is trusted, who should be responsible for deploying the server enclave on the CSS, and distributing the client enclaves to the users. The CSS provider is supposed to be willing to collaborate with SeFS to initialize the file sharing system. Otherwise, the administrator can switch to another provider to achieve the security guarantees provided by SeFS. Though willing to collaborate with SeFS, the CSS providers cannot be fully trusted, since they may tamper with user-defined access control policies and grant access permissions to unauthorized parties.

Since authorized users will obtain the plaintext file contents decrypted by the client-side enclave, they are trusted not to leak any data intentionally[3]. We assume attackers can by somehow compromise the cloud platform, thus altering any file stored there, but they cannot access the enclave data or code protected by Intel SGX's security primitives. Recent researches show that SGX is vulnerable to side-channel attacks [41–43], which can be mitigated by integrating countermeasures proposed in [44–47]. Finally, denial-of-service attacks and hardware attacks are out of the scope of this paper.

[3]We discuss the data leakage by authorized users in Section 7.

### 3.2. Design Requirements

We summarize the design requirements for a secure and practical cloud-based file sharing system as below:

(**R1**) *Enforcement of authorization.* Any entity, without the owner's explicit access authorization, should be prohibited from accessing the plaintext files and the corresponding access control policy files. Note that some entity may be able to get the encrypted files, but he/she cannot decrypt them to obtain the plaintext contents.

(**R2**) *Access revocation.* The file access permission should be revoked immediately upon the request of the owner. This implicitly indicates that the authorized users should not access the decryption key directly. Otherwise, revoking their permissions typically involves re-encrypting the files, which may take a while to complete depending on the size of the files.

(**R3**) *Performance.* The file sharing system should support concurrent requests and the overhead introduced by the access control enforcement should not significantly downgrade the response time or throughput. Meanwhile, the system should be compatible with existing cloud storage services and independent of the underlying file systems, thus incurring little instrumentation cost.

(**R4**) *Ease-of-use.* The file sharing system should allow users to manage/update/share their files easily and not require users to manage any key by themselves.

## 4. Design

### 4.1. Overview

As shown in Figure 1, SeFS is composed of SeFS server and SeFS client, running on the remote cloud server and the users' local computer respectively. The former is responsible for loading the SeFS's server-side enclave (called *Server enclave*) that provides registration, authorization, authentication and secure I/O operation (i.e., network communication and file reading/writing) services to the administrators. The latter is used by both the file owners (e.g., Alice) and users (e.g., Bob). The file owners utilize it to register with the server, encrypt files and update access control policy, while the users utilize it to authenticate themselves to the server, decrypt and access files securely.

In particular, any user first needs to register himself/herself by submitting his/her identity information, e.g., email, to the server enclave. The server enclave then generates a master token for the user, which should never be exposed to other users or untrusted parties. Based on the master token, the server enclave also generates a share token for the user, which is used by the file owner to enforce access control policy. For each file access, users need to verify the server enclave's identity, and then authenticate themselves to the server enclave using their master tokens to establish a secure channel between them and the server enclave. The server enclave checks the access control policy based on the shared tokens of the users to enforce read or write access control on files. If the users have the corresponding permissions, the file key will be sent to the client-side enclave (called *Client enclave*), which will decrypt the encrypted files using the key and then destroy the key immediately after use. Hence, the plaintext of the key will never be exposed to the users directly, since it is only kept inside the client enclave temporarily.

### 4.2. SeFS Server

As shown in Figure 1, SeFS server consists of three major components as below.

**TLS Handler**. The TLS handler is partitioned into an untrusted part (*uTLS Handler*) and a trusted part (*tTLS Handler*), running outside and inside the server enclave respectively. Since the I/O operations cannot be performed inside the enclave directly, the *uTLS Handler* is to create/terminate the network connection, and forward all TLS records to the *tTLS Handler*. All incoming/outgoing TLS data is decrypted/encrypted inside the enclave by the *tTLS Handler*. SeFS uses the switchless call technique to exchange data between the *tTLS Handler* and the *uTLS Handler* efficiently.

**Request Handler**. The request handler consists of three sub-handlers, which parse and process the incoming requests. For example, the registration request is forwarded to the *tRegister* to generate the master token and the share token. The policy update or file access request is dispatched to the *Access Control*, which is responsible for access control check and access control policy update. The *Authentication* is responsible for establishing trust with SeFS Client, e.g., generating signatures for the *tRegister*, encrypting the challenge received from the client enclave.

**File Interface**. Similar as the TLS handler, the file interface is also divided into two parts, *tFile Interface* running inside the enclave and *uFile Interface* running outside the enclave. The former invokes switchless ocalls exposed by the latter to read/write files.

### 4.3. SeFS Client

The SeFS Client provides support for both the file owner and regular users as shown in the left part of Figure 1. The data owner (i.e., Alice) does not require TEE-supported hardware, and uses the non-enclave application to easily register and update policy entries. However, to obtain the file shared by Alice, the other regular user, e.g., Bob, demands TEE-supported hardware (e.g., Intel SGX processors), which ensures the file decryption key only resides inside the TEE.

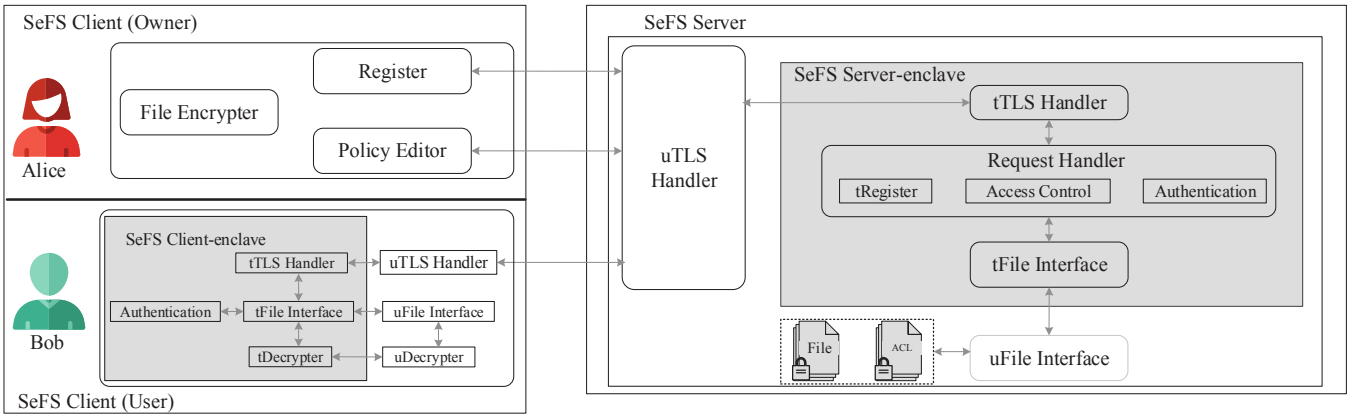


**Figure 1.** SeFS architecture

**For the file owner**. The SeFS client provides three components for the file owner: *Register*, *Policy Editor*, and *File Encrypter*. The Register establishes a secure channel with the server enclave, and validates the authenticity of the server enclave on behalf of the file owner. If the server enclave is trusted, the Register sends the owner's identity formation and password to the server enclave to generate the master token and share token. The Policy Editor interacts with the server enclave to insert/delete/update ACL entries remotely and securely. The File Encrypter either uses a key provided by the file owner or randomly generates a key to encrypt the file to be shared. For each file that has not been uploaded, the File Encrypter also automatically creates an initial ACL file, where the file encryption/decryption key will be embedded. Thus, the file owners do not need to maintain and manage the keys by themselves.

**For other users**. The SeFS client is also responsible for loading the client enclave. Before issuing file access requests to the server enclave, the client enclave needs to validate the server enclave's authenticity using *Authentication*. After establishing trust between the client enclave and the server enclave, the client enclave submits a file access request to the server enclave via the *tTLS handler*. If the user has the corresponding permission, the server enclave will send the file decryption key and the download URL to the client enclave via the TLS channel. The client enclave downloads the file using the *uFile Interface* that has file downloading functionality (e.g., wget [48]) and runs outside the enclave. The *uDecrypter* loads the encrypted file into the non-enclave memory, and invokes the switchless ecalls provided by *tDecrypter* to decrypt the file in order to reduce the overhead introduced by the enclave transitions.

Since the file decryption is performed by the client enclave on the users' computers, the server enclave only performs relatively lightweight access control check, thus achieving high concurrency and throughput. As Intel SGX has limited enclave memory space, *tDecrypter* uses a fixed-size enclave memory to exchange data with *uDecrypter*. In particular, the *uDecrypter* loads the encrypted files into the non-enclave memory, and then the *tDecrypter* each time loads the fixed-size chunk of the encrypted file to decrypt. Moreover, SeFS requires that the users exclusively occupy the client-side enclave resources, so the decryption of files can be optimized. SeFS utilizes the switchless call mechanism to reduce the enclave transitions overhead.

## 5. Implementation

### 5.1. Setup

The trusted administrator runs the client enclave and the server enclave locally to generate their public keys before assigning them to users and the cloud providers. Note that since all the users share exactly the same client enclave, the administrator only needs to generate one public key for all the client enclaves and then distributes the client enclave as well as its public key to each user. The public key is generated inside the enclave using the ECDSA [49] signature scheme based on the private key derived from a combination of the enclave identity, i.e., enclave measurement, and the *HC_key* (Hard-Coded Key). The enclave identity is constructed inside the enclave and can be obtained by invoking the EREPORT instruction. The *HC_key* is a string containing a certain number (e.g., 256) of random numbers, which is hard-coded into each enclave by the administrator and never exposed to any untrusted entity. To prevent the adversaries from stealing the *HC_key* by reverse-engineering the enclave binary, the administrator needs to encrypt the enclave binary at the build time using Intel SGX PCL [50] technique. Hence, the adversaries can never get either the enclave measurement or the *HC_key*, so the private key cannot be forged by the adversaries. Eventually, the server

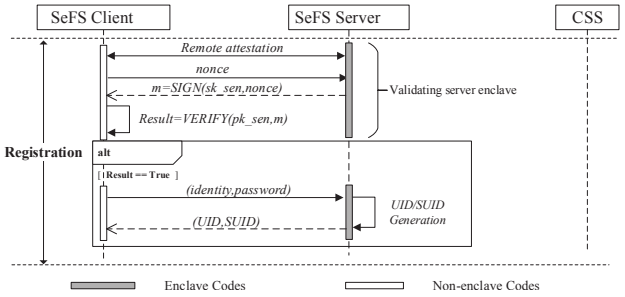


**Figure 2.** The protocol for token registration in SeFS

public key *pk_sen* can be generated based on its private key *sk_sen* as below:

$$pk\_sen = ECDSA\_get\_key(sk\_sen)$$

The client enclave's public key can be generated in the same way. After generating all the keys, the administrator can deploy the SeFS server on the cloud platform and distribute the SeFS client as well as the server enclave's public key (*pk_sen*) to users.

## 5.2. Registration

Figure 2 shows token registration in SeFS. At the very beginning, any user should validate the server enclave's authenticity, e.g., check if the server is running on a genuine hardware platform via the SGX remote attestation mechanism, and its code integrity is not compromised via the ECDSA signature primitive using the server enclave's public key. Briefly, the user randomly generates a *nonce* and sends it to the server enclave via a secure channel. The server enclave signs this *nonce* using its private key *sk_sen*, i.e., $m = SIGN(sk\_sen, nonce)$. The user can validate the signature using the server enclave's public key *pk_sen*, i.e., $Result = VERIFY(pk\_sen, m)$. If *Result* is correct, the server enclave is validated and can be trusted.

After validating the authenticity of the server enclave, users should authenticate themselves to the server enclave to get their master token (i.e., UID) and share token (i.e., SUID) generated by the server enclave. The UID is a unique identifier of the user in SeFS. All the users need to protect their UIDs private and never expose them to others. The UID is generated by the server enclave using the user's identity information, e.g., email, username, etc. The server enclave uses the UID as the key to generate the SUID (share-UID) that is public to other users. Hence, the server enclave can associate a certain UID with SUIDs to examine if they belong to the same user. Both the UID and SUID are generated inside the server enclave based on its identity, i.e., enclave measurement and *HC_key*, to make sure they cannot be forged. The UID and SUID are stored and managed by the user, and can be used on multiple devices of the user.

**Algorithm 1:** UID/SUID Generation

**Input:** identity,password
**Output:** UID,SUID
1 $en\_measurement \leftarrow gen\_self\_measurement()$
2 HMAC_$set_type$(SHA256)
3 HMAC_$set_key(en_measurement|HC_key)$
4 UID←HMAC_$gen_digest(identity|password)$
5 HMAC_$set_key$(UID)
6 SUID←HMAC_$gen_digest(identity)$

Algorithm 1 outlines the UID/SUID generation scheme. After checking the format of the identity information, the server enclave firstly gets its enclave measurement (Line 1) by invoking a private function, i.e., gen_self_measurement(), which can only be executed inside the enclave. Setting its measurement and *HC_key* as the generation key, the server enclave utilizes the HMAC [51] scheme to generate a UID based on the combination of the identity and password of the user (Lines 2-4). A SUID then can be generated by setting the UID as the generation key based on the identity of the user (Lines 5-6).

## 5.3. Policy Creation and Updating

Whenever the file owner wants to upload a new file to the CSS, he/she first needs to encrypt the file using a symmetric key generated randomly, and then upload the encrypted file ($f$). The CSS generates a sharelink for this encrypted file, which can be used by the owner to share the file with other users. Note that the sharelink is not a complete download URL, so users cannot use it to download the file directly. Instead, it is used by the CSS to easily locate the path of the file on the cloud platform and construct a download URL.

**Creating Access Control Policy**. SeFS uses the access control list (ACL) to maintain the user-defined access control policy. Therefore, for each encrypted file $f$, the file owner also needs to create an initial ACL file ($f.acl$) that contains his/her UID, the key used to decrypt $f$ and a number of ACL entries. Each ACL entry contains one SUID (indicating a specific user) and the corresponding permissions granted to the SUID. The file owner should upload the initial ACL file to the server enclave via a secure channel, and the ACL file will be encrypted by the server enclave using a key generated in Algorithm 2. In particular, the server enclave uses the SHA-256 cryptographic hash algorithm to generate a digest of a random vector $IV$ (Lines 1-2). The server enclave updates the digest iteratively based on its enclave measurement and *HC_key* (Lines 3-5), and finally derives the key at the end (Line 6). The random vector $IV$ is stored in the ACL file in plaintext, so the server enclave can re-generate the same key to decrypt the ACL entries. Note that the server enclave encrypts each entry of the ACL file separately,

rather than encrypting the entire file. Hence, updating any access control entry does not need to decrypt the entire ACL file. Finally, the server enclave exports the encrypted ACL file (i.e., $f.acl$) on the cloud storage platform under the same directory as the file it is used to protect (i.e., $f$).

**Algorithm 2:** Key Generation

**Input:** enclave measurement, $en\_measurement$, $HC\_key$, initial vector $IV$
**Output:** key

```
1  MAC_set_type(SHA256)
2  digest ←MAC_update(IV)
3  for i ← 1 to 8192 do
4  |  digest ←MAC_update(en_measurement|HC_key|digest)
5  end
6  key ←MAC_finish(digest)
```

**Updating Access Control Policy**.

Before any policy update, the file owner also needs to validate the server enclave's authenticity (the same as in Section 5.2), and then submit the new policy entry containing the file's sharelink, another user's SUID and the corresponding permission to the server enclave. Receiving the policy update request, the server enclave retrieves the corresponding encrypted ACL file stored on the CSS (Line 1 in Algorithm 3) and derives the key from its enclave identity and $HC\_key$ (Lines 2-6). Then, the server enclave checks whether the UID of this request belongs to the owner of the ACL file (Line 7). If so, it encrypts the new ACL entry, inserts it into the ACL file (Lines 8-9), and uploads the ACL file to the cloud storage platform (Lines 10-11). All the ACL entries are sorted based on the plaintext SUIDs of each entry.

When removing/updating an existing ACL entry in the ACL file, the server enclave first locates the offset of the target ACL entry to be removed/updated. Since all the ACL entries are sorted based on the plaintext SUIDs, the server enclave only needs to perform a binary search to locate any ACL entry. Therefore, it suffices to decrypt the corresponding ACL entries involved in the search path, rather than iterating and decrypting each ACL entry from the beginning. After removing/updating the target ACL entry, the server enclave uploads the updated ACL file to the CSS.

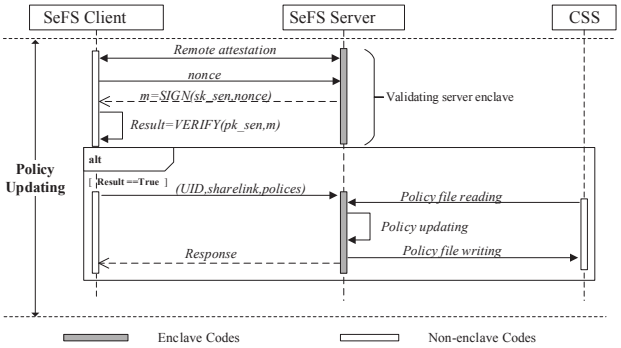


**Figure 3.** The protocol for policy updating in SeFS

**Algorithm 3:** Granting permission

**Input:** UID of data owner, SUID of data user, file's sharelink, permission

```
1  ACL_file_enc ← get_file_from_CSS(sharelink)
2  if ACL_file_enc exist then
3  |  en_measurement ← gen_self_measurement()
4  |  IV ← extract_IV(ACL_file_enc)
5  |  key ← gen_key(en_measurement, HC_key, IV)
6  |  ACL_file ← DEC(ACL_file_enc, key)
7  |  if UID owns ACL_file then
8  |  |  policy_entry ← (SUID, permission)
9  |  |  insert_policy_entry(ACL_file, policy_entry)
10 |  |  ACL_file_enc ← ENC(ACL_file, key)
11 |  |  upload_file_to_CSS(ACL_file_enc, sharelink)
12 |  end
13 end
```

## 5.4. Access Control

Figure 4 shows the enforcement of access control in SeFS. At the very beginning, the client enclave and the server enclave need to validate the authenticity of each other, to ensure the UID of the owner and the decryption key from the server enclave will not be leaked to adversaries.

**Reading files**. After the trust established between the client enclave and the server enclave successfully, the client enclave submits the file read request, containing his/her UID, SUID, identity, sharelink and read request, to the server enclave. Since the UID is private to its owner only, it can be used to authenticate the identity of the client. For instance, an attacker may want to impersonate as Bob using Bob's SUID (everyone's SUID is public) to read a file that has been granted to Bob for read but not to himself/herself. However, the attacker cannot present the UID of Bob, which fails the association check (Lines 1-2 in Algorithm 4). Then, the server enclave checks if Bob has been granted read permission to the associated file based on his SUID (lines 3-10). If so, the server enclave will extract the file decryption key from the ACL file and send it to

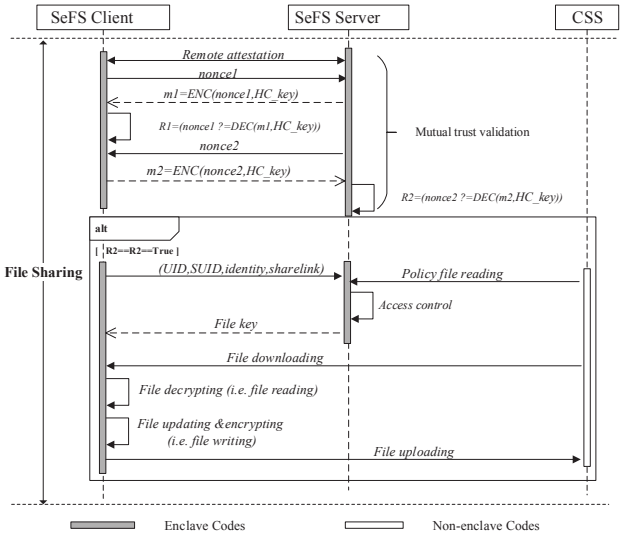


**Figure 4.** The protocol for file reading/writing in SeFS

Bob's client enclave via the TLS channel. Moreover, the server enclave will send the sharelink to the CSS to obtain a download link, which will be forwarded to the client enclave. Following the link, the client enclave can download the associated encrypted file, and decrypt it using the decryption key for read inside the enclave. After decryption, the client enclave destroys the file decryption key immediately.

**Writing files**. Enforcing access control on file writing is quite similar as that of file reading. The only difference is that after decrypting the downloaded file, the client enclave does not destroy the decryption key immediately. Instead, it waits for the completion of the update to the file from Bob, encrypts it using the key, uploads the updated file to the CSS, and finally destroy the key.

**Ownership delegation**. SeFS also supports the file owner to delegate the ownership to other users. In particular, if Alice wants to delegate the ownership of the file $f$ to Bob, she just needs to grant the write-only permission of the ACL file of $f$, i.e., $f.acl$, to Bob. Since $f.acl$ contains Alice's UID that can never be exposed to others, Bob can write-only $f.acl$ without read. Alice creates a new file $f.acl.acl$ associated with $f.acl$ and inserts a new ACL entry with Bob's SUID and write-only permission to $f.acl$. Note that the owner of $f.acl.acl$ is still Alice. Hence, Bob can insert/update/delete the ACL entries of $f.acl$ to grant read/write permissions of $f$ to others.

## 5.5. Integration with Cloud Storage Service

We implemented a prototype of SeFS based on Intel SGX and integrated the SeFS prototype with ownCloud [17], an open-source file storage and sharing system. Since ownCloud does not depend on the underlying file systems, deploying SeFS on ownCloud cloud platforms makes our solution more lightweight, scalable and flexible. In our implementation, we extended the file sharing module of ownCloud to ensure it shares files using the share links. In particular, when receiving a share link from the server enclave, ownCloud returns the file location of the share link to the server enclave. Moreover, the server enclave can read/write the ACL file associated with the share link. Note that the server enclave runs as a daemon on the CSS, co-located with ownCloud on the same physical server and waiting for file access or policy update requests from users. The extension to ownCloud is less than 10 lines of PHP code. The Amazon S3 [35] or Alibaba OSS [36] can be used as the external storage backends for ownCloud, without any change to the Amazon S3 or Alibaba OSS platforms.

**Algorithm 4:** Access control check on file reading/writing

**Input:** UID of data user, SUID of data user, identity of data user, file's sharelink, permission
**Output:** file decryption key

```
tmp_SUID ← generate_SUID(UID, identity)
if tmp_SUID==SUID then
    ACL_file_enc ← get_file_from_CSS(sharelink)
    if ACL_file_enc exist then
        IV ← extract_IV(ACL_file_enc)
        key ← gen_key(en_measurement, HC_key, IV)
        ACL_file ← DEC(ACL_file_enc, key)
        policy_entry ← (SUID, permission)
        if policy_entry in ACL_file then
            file_decryption_key ← extract_key(ACL_file)
        end
    end
end
```

## 6. Evaluation

We implemented SeFS's prototype based on Intel SGX SDK 2.10 and Intel SGX Driver 2.11. We also made extension and optimization on a public SGX-enabled TLS stack, mbedtls-SGX [52], i.e., mbedtls-2.6.0, to provide efficient SGX-supported I/O operations. The secure channel protocol is TLSv1.2 with ECDHE-ECDSA-AES256-GCM-SHA384 cipher suite. We deployed the prototype of SeFS on the ownCloud 10.0.2 [17], which only requires a tiny change (less than 10 lines of PHP code).

## 6.1. Security Analysis

**Security of Keys**. One of SeFS's security objectives is to guarantee the secrecy of the file encryption/decryption keys. In the setup phase, the file owner generates a file encryption key randomly to encrypt the content of a file. The file encryption key will be embedded into the ACL file that will be uploaded to the server enclave via a secure channel. The enclave encrypts this ACL file using a key derived from its enclave identity and *HC_key*. When the client enclave issues a file access request to the server enclave on behalf of the user who has been granted access to this file, the server enclave decrypts the file encryption key from the ACL file and sends it to the client enclave via a TLS channel. Note that the client enclave must also be validated by the server enclave before sending the file decryption key to it. The client enclave will clean this file decryption key after use. Hence, SeFS guarantees that the file decryption key can only be accessed within the enclaves (i.e., Client enclave, Server enclave). In addition, other keys derived from the enclave identity and its *HC_key* are also only accessible to the enclaves, since the enclave identity can only be generated within the enclaves and the *HC_key* can never be exposed to any untrusted entity.

**Confidentiality and Integrity**. In the setup phase, the files are encrypted by the file owner on the client-side. These file encryption/decryption keys belong to the file owner, who should protect them securely. On the server-side, all sensitive data is placed within the enclave, including file decryption keys and ACL files. After performing the corresponding permission check, i.e., validating the client enclave's authenticity and examining the user's permission, the decryption of the encrypted files can be performed inside the client enclave. Hence, SeFS can guarantee the confidentiality of the content of the files, file decryption keys and ACL files. Moreover, the encryption of files is performed using the AES cryptography primitive with GCM mode, meaning that the data integrity is provided alongside confidentiality. Thus, any malicious modification of the ciphertext will be detected during the decryption process executed in the enclaves.

**Access Control Enforcement**. SeFS can enforce authorization securely inside the enclave according to user-defined access control policies. Moreover, SeFS uses the share token, i.e., SUID, to generate policy entries, instead of directly using users' identity information, which can defeat impersonation attack. In the impersonation attack, if an attacker gets Alice's identity information, he/she can use it to maliciously register a new UID/SUID, and even gain the access to Alice's files. Furthermore, even the attacker gets Bob's SUID, he/she cannot impersonate Bob to access files, since the server enclave will check the ownership of the SUID before performing the access control check.

**Table 1.** Latency of trust validation/registration/permission check

| Mutual verification | Registration | Permission_Check | |
|---|---|---|---|
| | | SeFS | ownCloud |
| 67.04 ms | 34.76 ms | 77.57 ms | 81.86 ms |

**TCB Size**. SeFS's enclaves only comprise 4,489 LoC, with 2,984 LoC for the server enclave and 1,505 LoC for the client enclave, in contrast to SeGShare [15] with 8,441 LoC. This significantly reduces the potential programming errors and attack surfaces.

## 6.2. Performance Evaluation

**Experimental Setup**. Our performance evaluation was performed on three virtual machines (VMs) hosted at Microsoft Azure Confidential Computing (ACC) [53]. The SeFS server runs on a VM with 16 GB RAM and 4 vCPU cores on an Intel Xeon(R) E-2288G CPU @ 3.70GHz platform, and the ownCloud shares the same VM as the SeFS server. The other two VMs with 8 GB RAM and 2 vCPU cores on an Intel Xeon(R) E-2288G CPU @ 3.70GHz platform located in the east US region are used by the file owner (Alice) and the file consumer (Bob), respectively. The latency of all the following experiments are measured from the start of the request to the end of the corresponding response at the client-side applications, i.e., beginning when the *Register* issuing a registration request to the *tRegister* and ending at the *Register* receiving the response from the *tRegister*. Besides, the latency is averaged over 10 runs, each of which contains 100 request-to-response operations.

Table 1 shows the average time of establishing mutual trust between the client enclave and server enclave as well as registering tokens (i.e., UID and SUID) is 67.04 ms and 34.76 ms, respectively. We also measure the time spent on performing access control check on the file access request, which involves policy entry search and extracting file decryption key from the ACL file, but excluding the time to download the file from the CSS and the time spent on decrypting the file inside the client enclave. The experimental result indicates that the client enclave only takes 77.57 ms to get the file decryption key from the server enclave. Besides, ownCloud also supports sharing a file via a URL with a password or verification code. The users can download the file only when they submit the correct password. We also measured the time spent on the password verification of ownCloud. On average, it takes 81.86 ms to perform the password check, which means that the permission check time of SeFS and ownCloud is in the same order of magnitude. In addition, we measure the latency of adding/removing an ACL entry to/from an ACL file. For an ACL file containing 2,000 ACL entries, the average latency of permission addition and

**Table 2.** The latency of accessing file

| File size (MB) | 10 | 50 | 100 | 150 | 200 |
|---|---|---|---|---|---|
| Mutual verification (s) | 0.066 | 0.066 | .064 | 0.064 | 0.065 |
| Permission check (s) | 0.087 | 0.087 | 0.086 | 0.088 | 0.086 |
| Download file (s) | 0.502 | 1.024 | 1.664 | 2.533 | 4.637 |
| Decrypt file (s) | 0.379 | 1.892 | 4.059 | 5.854 | 8.242 |

permission revocation is around 43.36 ms and 43.86 ms respectively. Moreover, in the current implementation of SeFS, each ACL entry is fixed-size, i.e., 16 bytes, so the size of the ACL file is independent of the size of the file it protects. An ACL file containing 100k entries only requires about 1.5 KB storage space.

**File Access**. We also evaluate the end-to-end time of the file access, including the trust validation between the client enclave and the server enclave, the access control check inside the server enclave, downloading a file from the CSS, and decrypting the file inside the client enclave. Based on the results in Table 2, as the file size increases, the time of downloading and decrypting the file also increases. However, the time of file decryption is still in the same order of magnitude as file downloading, e.g., downloading a file of 200 MB takes 4.637 s and decrypting the same file takes 8.242 s. Moreover, the time spent on the trust validation and permission check accounts for a quite small proportion of the end-to-end time, since the computation workload of the server enclave is lightweight and paralleled.

## 6.3. Concurrency

We evaluate the overhead of uploading and downloading files using SeFS, and compare it with the state-of-the-art SeGShare [15]. Since SeGShare did not include the total latency of uploading and downloading files inside SeGShare's server-side enclave, we implemented a server-side enclave based on the mbedtls-SGX [52] following the idea proposed by SeGShare[4]. The server-side enclave (called SeGShare_enclave) is responsible for accessing files via a TLS channel and encrypting/decrypting them in real time, excluding the access control check of SeGShare, which is not part of our evaluation.

**Uploading Files**. We start multiple threads (from 1 to 10) to simulate multiple users uploading the files to SeFS's server-side storage backend, i.e., ownCloud. As shown in Figure 5, as the concurrency level (i.e., the number of users) increases, the latency of uploading files is relatively constant, which indicates that SeFS is capable of handling concurrent requests with high throughput and low latency. This can be explained by the fact that the expensive file encryption computation is done by individual client enclave, rather than the server enclave. In contrast, since SeGShare handles all the file encryption inside the SeGShare_enclave, the time of uploading files directly to the SeGShare_enclave (i.e., reading and encrypting files inside SeGShare_enclave) significantly increases as the concurrency level increases.

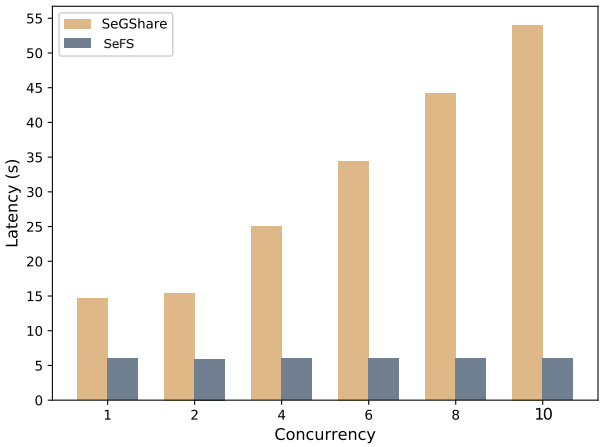


**Figure 5.** The latency of uploading files concurrently

**Downloading Files**. To evaluate the performance of SeFS when handling the concurrent requests of downloading files, we created 10 Microsoft Azure Confidential Computing (ACC) VMs, each of which runs a client enclave respectively. These 10 client enclaves can simultaneously issue file access requests for a 200 MB file to SeFS's server enclave. We still measure the end-to-end time of the file access process at the client-side. As shown in Figure 6, as the concurrency level increases, the time of the file access is relatively constant when using SeFS. This can be explained by the fact that the server enclave only needs to perform lightweight access control enforcement, instead of expensive file decryption computation, which indicates SeFS is capable of handling concurrent requests with very little impact on the throughput. However, as the concurrency level increases, the time of downloading files from the SeGShare_enclave (i.e., reading and decrypting files inside the server-side enclave) significantly increases.

# 7. Discussion and Limitation

## 7.1. SeFS's implementation on Other Platforms

In the design of SeFS, we abstract the trusted execution environment as an 'enclave'. Although the prototype system of SeFS is based on Intel SGX, it can be ported to

[4]We contacted the authors of SeGShare and they do not have the plan to open-source SeGShare's implementation.

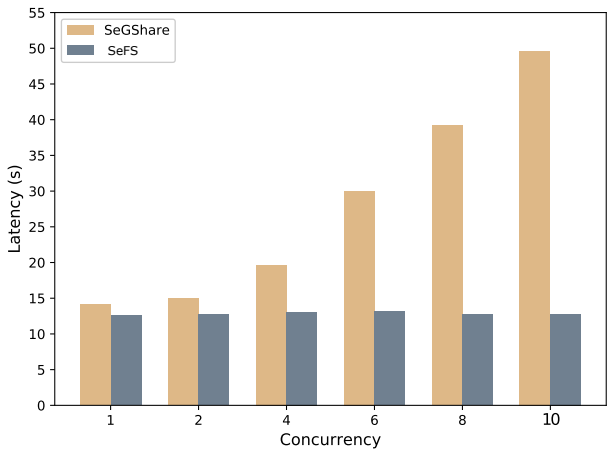


**Figure 6.** The latency of downloading files concurrently

other hardware-assisted TEE platforms with moderate efforts as the following: (1) As described in the previous algorithms 1 to 4, the platform-specific function is *gen_self_measurement()* and its internal implementation will invoke different TEE platforms' interfaces (e.g., Intel SGX's EREPORT instruction, the *calc_measurement* provided by the AMD sev-tool [54]). Therefore, when porting the SeFS into other TEE platforms, this function can be abstracted as a high-level interface and just adjust its internal implementation to invoke the specific platform's interfaces. Except for *gen_self_measurement()*, other functions in those algorithm are general and independent of the specific TEE platform. (2) Another critical platform-specific operation is the remote attestation. Fortunately, there exist unified attestation techniques (e.g., [55]) that abstract away the underlying TEE platform differences, reducing the effort required for SeFS to implement remote attestation across platforms.

Furthermore, the SeFS server can be relatively easily ported to a VM-based TEE. A simple approach is to run it within a confidential container [56] (a lightweight container designed based on VM-TEE). Given that TrustZone is primarily employed on mobile devices, the portion of the SeFS server to TrustZone need not be considered. As for the SeFS client, it can also be ported to the VM-based TEE, following the same method as the SeFS server. When porting the SeFS client to the TrustZone, a Trusted Application (TA) will be developed to implement the SeFS client functionality. This effort should also be acceptable because of that most of the SeFS client enclave's dependencies (e.g., AES encryption/decryption) have been provided by the TrustZone TA SDK (e.g., incubator-teaclave-trustzone-sdk [57]). It is worth noting that porting the SeFS client to the TrustZone is meaningful future work. On one hand, nearly all current mobile devices (such as smartphones) support the TrustZone mechanism, which means concerns about hardware limitations can be disregarded. On the other hand, for scenarios where a large number of users are utilizing SeFS, deploying the SeFS client on a distributed multitude of mobile endpoints is an optimal deployment model. This deployment can also leverage the advantage of the high-concurrency response capabilities of the SeFS server.

## 7.2. Rollback Attacks

An attacker may perform a *rollback attack* against the encrypted file, in which the attacker uses an outdated encrypted file to replace the latest one. In particular, the attacker can launch DoS attack by replacing an existing file with an outdated version. Then, even the authorized user cannot read the file content any more, since the newer version and the outdated version are encrypted using different keys. By rolling back the ACL file to an outdated version, a user whose permission is recently revoked, may re-obtain the access to a certain file. SeGShare [15] uses a Merkle hash tree [58] variant to defeat the rollback attack of individual files with the assumption that the root hash of the Merkle hash tree is secure. However, SeGShare did not provide rollback protection for the entire file system. The key to mitigating the rollback attack is to provide a trusted storage entity that can store the state or version information persistently. Previously, Intel SGX provided a monotonic counter that can only be accessed by a specific enclave, which is non-volatile across restarts with the help of Intel ME [59]. However, Intel SGX SDK removed this feature since the version 2.8 [60]. Existing methods, such as [61] that leverage Trusted Platform Modules (TPMs) to provide persistent monotonic counters or ROTE [62] that stores enclave-specific counters in a distributed system of collaborative enclaves, can be used together with SeFS against the rollback attack. The LCM (lightweight collective memory) [63] protocol can also be used to strengthen SeFS as well. We leave the protection against the rollback attack as our future work.

## 7.3. Side-channel and Hardware Attacks

Recent research has shown that SGX is vulnerable to various side-channel attacks, such as timing attack [41], cache-based attack [64], and page fault attack [43]. Accordingly, some methods (e.g., [44, 46, 47]) are proposed to mitigate side-channel attacks, detect side-channel attacks [65], or design data oblivious file system [45]. Although side-channel attacks and the corresponding mitigation are orthogonal to our work, SeFS can be improved using data oblivious methods (e.g., [45]) to conceal file access patterns. Recent research [66] presents a hardware-based voltage glitching attack against Intel SGX, which requires the attackers to monitor and control the CPU core

voltage using professional devices. Such hardware-based attacks are out of scope of our paper.

### 7.4. Data Leakage from Client-side

When the file owner shares his/her file with the other user, the user can access the content of this file via the decryption functionality of the client enclave. However, the user may intentionally leak this file to other users who have not been granted the read permission. To the best of our knowledge, none of the existing works, e.g., [8, 9, 15, 16] and SeFS can address this problem. However, SeFS supports immediate permission revocation, which can be used to prevent users from accessing updated files if they are detected malicious. SeFS aims to prevent the file decryption keys from being leaked out to any user even if he/she is authorized to access the file. If any user can get the plaintext of the file decryption key, he/she may perform collusion attacks with the cloud service provider to decrypt files on which his/her permission has been revoked. From another perspective, the data access can be bound to a specific physical device by prohibiting file copying and file transmission via the network. This usually requires the support from the underlying platforms. For instance, the ARM TrustZone [67] maybe a promising exploration, since it can control IO operations inside the secure world. Specifically, Trusted UI (user interfaces) [68] based on the TrustZone can prevent the users from capturing screens when show privacy-sensitive data.

## 8. Related Works

### 8.1. SGX-assisted Cloud Storage

During the past years, many approaches use TEE to design secure file sharing systems [8, 9, 15, 16, 69]. In particular, A-SKY [8] proposed a cryptographic access control extension based on anonymous broadcast encryption (ANOBE) [1], which utilizes Intel SGX to address the impracticality of ANOBE scheme and provides confidentiality and anonymity guarantees. IBBE-SGX [9] utilizes SGX to derive cuts in the computational complexity of the identity-based broadcasting encryption (IBBE) [13] cryptographic scheme, and also proposes an efficient group partitioning mechanism that updates membership with a constant computational cost. However, both of A-SKY [8] and IBBE-SGX [9] suffer from the problem that users can access to the plaintext of file decryption keys.

SeGShare [15] and NEXUS [16] are purely TEE-based schemes without complicated cryptographic operations (only involving symmetric encryption), so both of them support to revoke permission immediately without re-encrypting files using new keys. However, SeGShare [15] encrypts/decrypts files in real time inside the server-side enclave running on the remote cloud servers to respond to users' requests, i.e., uploading or downloading files. It suffers from the concurrency or throughput problem, since that encrypting or decrypting big files inside the enclave introduces significant performance overhead. SeGShare [15] also depends on the third-party certificate authority (CA) to establish trust between users and the server-side enclave. NEXUS [16] embedded access control policies and file keys into the data volume protected by the client-side enclave running on the client's computer. Data is shared as the data volumes, which is complicated and not flexible for finer granularity access. For example, NEXUS [16] has to access the entire data volume even if the user only needs one file of this data volume.

Pesos [69] leverages a combination of Intel SGX and Kinetic Object Storage [70] solution to enforce access control policies on untrusted commodity platforms and also provides a guarantee of confidentiality and integrity. However, the implementation of Pesos [69] depends on LibOS [71] that severely increases the TCB size and requires a special Kinetic hard disk, which hurts the flexibility and applicability. OBLIVIATE [45] and ZeroTrace [72] protect file access patterns and files by using the ORAM protocol, which is immune to side-channel attacks, but they did not support file sharing.

### 8.2. Cryptography-based Access Control

Several schemes [1–5] based on cryptography primitives have also been proposed to perform rich access control policies for untrusted cloud storage services. The primary goal of ANOBE [1] is to achieve anonymity among all file consumers by extending the broadcast encryption primitive [6], but it has server impact on its practicality. REED [3] uses attribute-based encryption [7] to envelope the symmetric keys for performing deduplication on encrypted files, but it is also not efficient on highly dynamic access control. CloudProof [4] still uses broadcast encryption to envelope read access keys and write keys for conducting read/write access control. Unfortunately, all of the purely cryptographic approaches suffer from the problem that users gain plaintext access to the file decryption key. Consequently, owners have to re-encrypt the corresponding file with a new key to achieve immediate permission revocation, which can involve expensive cryptographic operations. Meanwhile, the new key has to be distributed to the users who already obtain the access permission. As shown in [14], the overhead introduced by this permission revocation can become a critical problem if members are removed and added frequently.

## 9. Conclusion

To enable users to securely share their outsourced files with other users, we present SeFS, a secure and practical file sharing system that leverages a combination of the server-side enclave and the client-side enclave to enforce access control, to provide the confidentiality and integrity of the outsourced files. The server-side enclave is only responsible for registration, authentication, and access control check, while the data decryption is performed on the client-side enclaves, which significantly reduces the server-side's computations burden. SeFS supports immediate permission revocations without file re-encryption, since the file decryption keys never leave the enclave memory and will be cleaned after use. SeFS dose not depend on the underlying file systems, which makes it flexible to cooperate with existing cloud storage services. We deployed SeFS on a real-world cloud storage service, i.e., ownCloud and evaluate its practicality and security. The experimental results show that SeFS can perform access control enforcement securely with high throughput and low latency.

## References


[1] Barth, A., Boneh, D. and Waters, B. (2006) Privacy in encrypted content distribution using private broadcast encryption. In Di Crescenzo, G. and Rubin, A. [eds.] *Financial Cryptography and Data Security* (Berlin, Heidelberg: Springer Berlin Heidelberg): 52–64.

[2] Castiglione, A., Catuogno, L., Del Sorbo, A., Fiore, U. and Palmieri, F. (2014) A secure file sharing service for distributed computing environments. *The Journal of Supercomputing* **67**: 691–710. doi:10.1007/s11227-013-0975-y.

[3] Li, J., Qin, C., Lee, P.P.C. and Li, J. (2016) Rekeying for encrypted deduplication storage. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*: 618–629. doi:10.1109/DSN.2016.62.

[4] Popa, R.A., Lorch, J.R., Molnar, D., Wang, H.J. and Zhuang, L. (2011) Enabling security in cloud storage slas with cloudproof. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'11 (USA: USENIX Association): 31.

[5] Goh, E.J., Shacham, H., Modadugu, N. and Boneh, D. (2003) Sirius: Securing remote untrusted storage.

[6] Boneh, D., Gentry, C. and Waters, B. (2005) Collusion resistant broadcast encryption with short ciphertexts and private keys. In *Proceedings of the 25th Annual International Conference on Advances in Cryptology*, CRYPTO'05 (Berlin, Heidelberg: Springer-Verlag): 258–275.

[7] Bethencourt, J., Sahai, A. and Waters, B. (2007) Ciphertext-policy attribute-based encryption. In *2007 IEEE Symposium on Security and Privacy (SP '07)*: 321–334.

[8] Contiu, S., Vaucher, S., Pires, R., Pasin, M., Felber, P. and Réveillère, L. (2019) Anonymous and confidential file sharing over untrusted clouds. In *2019 38th Symposium on Reliable Distributed Systems (SRDS)*: 21–2110. doi:10.1109/SRDS47363.2019.00013.

[9] Contiu, S., Pires, R., Vaucher, S., Pasin, M., Felber, P. and Réveillère, L. (2018) Ibbe-sgx: Cryptographic group access control using trusted execution environments. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*: 207–218. doi:10.1109/DSN.2018.00032.

[10] (2014), Intel® software guard extensions programming reference, https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf.

[11] Costan, V. and Devadas, S. (2016) Intel sgx explained. *IACR Cryptol. ePrint Arch.* **2016**: 86.

[12] Hoekstra, M., Lal, R., Pappachan, P., Phegade, V. and Del Cuvillo, J. (2013) Using innovative instructions to create trustworthy software solutions. HASP '13 (New York, NY, USA: Association for Computing Machinery). doi:10.1145/2487726.2488370.

[13] Delerablée, C. (2007) Identity-based broadcast encryption with constant size ciphertexts and private keys. In Kurosawa, K. [ed.] *Advances in Cryptology – ASIACRYPT 2007* (Berlin, Heidelberg: Springer Berlin Heidelberg): 200–215.

[14] Garrison, W.C., Shull, A., Myers, S. and Lee, A.J. (2016) On the practicality of cryptographically enforcing dynamic access control policies in the cloud. In *2016 IEEE Symposium on Security and Privacy (SP)*: 819–838. doi:10.1109/SP.2016.54.

[15] Fuhry, B., Hirschoff, L., Koesnadi, S. and Kerschbaum, F. (2020) Segshare: Secure group file sharing in the cloud using enclaves. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*: 476–488. doi:10.1109/DSN48063.2020.00061.

[16] Djoko, J.B., Lange, J. and Lee, A.J. (2019) Nexus: Practical and secure access control on untrusted storage platforms using client-side sgx. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*: 401–413. doi:10.1109/DSN.2019.00049.

[17] (2020), owncloud, https://owncloud.com/product/.

[18] (2020), mbedtls, https://github.com/ARMmbed/mbedtls.

[19] Lee, D., Kohlbrenner, D., Shinde, S., Asanović, K. and Song, D. (2020) Keystone. In *Proceedings of the Fifteenth European Conference on Computer Systems*. doi:10.1145/3342195.3387532, URL http://dx.doi.org/10.1145/3342195.3387532.

[20] Feng, E., Lu, X., Du, D., Yang, B., Jiang, X., Xia, Y., Zang, B. *et al.* (2021) Scalable memory protection in the PENGLAI enclave. *Operating Systems Design and Implementation,Operating Systems Design and Implementation* .

[21] (2023), Amd sev-snp: Strengthening vm isolation with integrity protection and more, https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/solution-briefs/amd-secure-encrypted-virtualization-solution-brief.pdf.

[22] Cheng, P.C., Ozga, W., Valdez, E., Ahmed, S., Gu, Z., Jamjoom, H., Franke, H. *et al.* (2024) Intel tdx demystified: A top-down approach. *ACM Comput. Surv.*

doi:10.1145/3652597, URL https://doi.org/10.1145/3652597. Just Accepted.

[23] Li, X., Li, X., Dall, C., Gu, R., Nieh, J., Sait, Y. and Stockwell, G. (2022) Design and verification of the arm confidential compute architecture. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)* (Carlsbad, CA: USENIX Association): 465–484. URL https://www.usenix.org/conference/osdi22/presentation/li.

[24] Pinto, S. and Santos, N. (2019) Demystifying arm trustzone: A comprehensive survey. *ACM Computing Surveys* : 1–36doi:10.1145/3291047, URL http://dx.doi.org/10.1145/3291047.

[25] Gueron, S. (2016) Memory encryption for general-purpose processors. *IEEE Security Privacy* **14**(6): 54–62. doi:10.1109/MSP.2016.124.

[26] (2013), Innovative technology for cpu based attestation and sealing, https://software.intel.com/en-us/articles/innovative-technology-for-cpu-based-attestation-and-sealing.

[27] (2020), Intel® software guard extensions (intel® sgx) sdk for linux* os, https://download.01.org/intel-sgx/sgx-linux/2.12/docs/Intel_SGX_Developer_Reference_Linux_2.12_Open_Source.pdf.

[28] (2023), Runtime encryption of memory with intel® total memory encryption–multi-key (intel® tme-mk), https://www.intel.com/content/www/us/en/developer/articles/news/runtime-encryption-of-memory-with-intel-tme-mk.html.

[29] (2023), Intel® trust domain extension (intel® tdx) module, https://www.intel.com/content/www/us/en/download/738875/intel-trust-domain-extension-intel-tdx-module.html.

[30] (2023), Amd memory encryption, https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/memory-encryption-white-paper.pdf.

[31] Kaplan, D. (2016) AMD x86 memory encryption technologies (Austin, TX: USENIX Association).

[32] (2023), Protecting vm register state with sev-es, https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/Protecting-VM-Register-State-with-SEV-ES.pdf.

[33] (2020), Mega, https://mega.nz/.

[34] (2020), Nextcloud, https://nextcloud.com/.

[35] (2020), Amazon s3, https://aws.amazon.com/cn/s3/.

[36] (2020), aliyun oss, https://cn.aliyun.com/product/oss.

[37] (2020), Storage backends, https://doc.owncloud.com/server/developer_manual/app/advanced/storage-backend.html.

[38] (2020), Openafs, http://www.openafs.org/.

[39] (2020), Fastdfs, https://github.com/happyfish100/fastdfs.

[40] (2020), Ceph, https://ceph.io/.

[41] Weichbrodt, N., Kurmus, A., Pietzuch, P. and Kapitza, R. (2016) Asyncshock: Exploiting synchronisation bugs in intel sgx enclaves. In Askoxylakis, I., Ioannidis, S., Katsikas, S. and Meadows, C. [eds.] *Computer Security – ESORICS 2016* (Cham: Springer International Publishing): 440–457.

[42] Murdock, K., Oswald, D., Garcia, F.D., Van Bulck, J., Gruss, D. and Piessens, F. (2020) Plundervolt: Software-based fault injection attacks against intel sgx. In *2020 IEEE Symposium on Security and Privacy (SP)*: 1466–1482. doi:10.1109/SP40000.2020.00057.

[43] Xu, Y., Cui, W. and Peinado, M. (2015) Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *2015 IEEE Symposium on Security and Privacy*: 640–656. doi:10.1109/SP.2015.45.

[44] Shih, M.W., Lee, S., Kim, T. and Peinado, M. (2017) T-sgx: Eradicating controlled-channel attacks against enclave programs. In *Proc. of NDSS*.

[45] Ahmad, A., Kim, K., Sarfaraz, M.I. and Lee, B. (2018) OBLIVIATE: A data oblivious filesystem for intel SGX. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018* (The Internet Society).

[46] Gruss, D., Lettner, J., Schuster, F., Ohrimenko, O., Haller, I. and Costa, M. (2017) Strong and efficient cache side-channel protection using hardware transactional memory. In *26th USENIX Security Symposium (USENIX Security 17)* (Vancouver, BC: USENIX Association): 217–233.

[47] Crane, S., Homescu, A., Brunthaler, S., Larsen, P. and Franz, M. (2015) Thwarting cache side-channel attacks through dynamic software diversity. In *In Network and Distributed System Security Symposium*.

[48] (2020), Gnu wget, https://www.gnu.org/software/wget/.

[49] (2020), Elliptic curve digital signature algorithm, https://en.wikipedia.org/wiki/Elliptic_Curve_Digital_Signature_Algorithm.

[50] (2021), Intel(r) software guard extensions (sgx) protected code loader (pcl) for linux* os, https://github.com/intel/linux-sgx-pcl.

[51] (2020), Hmac, https://en.wikipedia.org/wiki/HMAC.

[52] (2020), mbedtls-sgx: a tls stack in sgx, https://github.com/bl4ck5un/mbedtls-SGX.

[53] (2020), Azure confidential computing, https://azure.microsoft.com/en-us/solutions/confidential-compute/.

[54] (2023), Amd sev tool, https://github.com/AMDESE/sev-tool.

[55] (2023), Jinzhao attest, https://github.com/asterinas/jinzhao-attest.

[56] (2023), confidential-containers, https://github.com/confidential-containers/confidential-containers.

[57] (2023), incubator-teaclave-trustzone-sdk, https://github.com/apache/incubator-teaclave-trustzone-sdk/wiki/.

[58] Merkle, R.C. (1987) A digital signature based on a conventional encryption function. In *A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology*, CRYPTO '87 (Berlin, Heidelberg: Springer-Verlag): 369–378.

[59] (2017), Some notes on the monotonic counter in intel sgx and me, https://davejingtian.org/2017/11/10/some-notes-on-the-monotonic-counter-in-intel-sgx-and-me/.

[60] (2020), Sealeddata example missing platform service capability, https://github.com/intel/linux-sgx/issues/541.

[61] Strackx, R. and Piessens, F. (2016) Ariadne: A minimal approach to state continuity. In *25th USENIX Security Symposium (USENIX Security 16)* (Austin, TX: USENIX Association): 875–892.

[62] Matetic, S., Ahmed, M., Kostiainen, K., Dhar, A., Sommer, D., Gervais, A., Juels, A. *et al.* (2017) ROTE: Rollback protection for trusted execution. In *26th USENIX Security Symposium (USENIX Security 17)* (Vancouver, BC: USENIX Association): 1289–1306.

[63] Brandenburger, M., Cachin, C., Lorenz, M. and Kapitza, R. (2017), Rollback and forking detection for trusted execution environments using lightweight collective memory. 1701.00981.

[64] Brasser, F., Müller, U., Dmitrienko, A., Kostiainen, K., Capkun, S. and Sadeghi, A.R. (2017) Software grand exposure: SGX cache attacks are practical. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)* (Vancouver, BC: USENIX Association).

[65] Chen, S., Zhang, X., Reiter, M.K. and Zhang, Y. (2017) Detecting privileged side-channel attacks in shielded execution with déjà vu. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '17 (New York, NY, USA: Association for Computing Machinery): 7–18. doi:10.1145/3052973.3053007.

[66] (2021) Voltpillager: Hardware-based fault injection attacks against intel SGX enclaves using the SVID voltage scaling interface. In *30th USENIX Security Symposium (USENIX Security 21)* (Vancouver, B.C.: USENIX Association).

[67] Alves, T. and Felton, D. (2004) Trustzone: Integrated hardware and software security .

[68] Cai, Y., Wang, Y., Lei, L., Zhou, Q. and Li, J. (2019) Suit: Secure user interface based on trustzone. In *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)*: 1–7. doi:10.1109/ICC.2019.8761616.

[69] Krahn, R., Trach, B., Vahldiek-Oberwagner, A., Knauth, T., Bhatotia, P. and Fetzer, C. (2018) Pesos: Policy enhanced secure object store. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18 (New York, NY, USA: Association for Computing Machinery). doi:10.1145/3190508.3190518.

[70] (2020), Kinetic, https://www.seagate.com/cn/zh/support/enterprise-servers-storage/nearline-storage/kinetic-hdd/.

[71] Arnautov, S., Trach, B., Gregor, F., Knauth, T., Martin, A., Priebe, C., Lind, J. *et al.* (2016) Scone: Secure linux containers with intel sgx. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16 (USA: USENIX Association): 689–703.

[72] Sasy, S., Gorbunov, S. and Fletcher, C.W. (2018) Zerotrace : Oblivious memory primitives from intel SGX. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018* (The Internet Society).