The Lord of the Shares: Combining Attribute-Based Encryption and Searchable Encryption for Flexible Data Sharing*

Antonis Michalas[1] [0000−0002−0189−3520]

Tampere University of Technology,
Tampere, Finland
antonis.michalas@tut.fi

Abstract. Secure cloud storage is considered one of the most important issues that both businesses and end-users are considering before moving their private data to the cloud. Lately, we have seen some interesting approaches that are based either on the promising concept of Symmetric Searchable Encryption (SSE) or on the well-studied field of Attribute-Based Encryption (ABE). In the first case, researchers are trying to design protocols where users' data will be protected from both *internal* and *external* attacks without paying the necessary attention to the problem of user revocation. On the other hand, in the second case existing approaches address the problem of revocation. However, the overall efficiency of these systems is compromised since the proposed protocols are solely based on ABE schemes and the size of the produced ciphertexts and the time required to decrypt grows with the complexity of the access formula. In this paper, we propose a protocol that combines *both* SSE and ABE in a way that the main advantages of each scheme are used. The proposed protocol allows users to directly search over encrypted data by using a SSE scheme while the corresponding symmetric key that is needed for the decryption is protected via a Ciphertext-Policy Attribute-Based Encryption scheme.

Keywords: Cloud Security · Storage Protection · Access Control · Policies · Attribute-Based Encryption · Symmetric Searchable Encryption · Hybrid Encryption

1 Introduction

One of the most well-studied problems of cloud security, is how to protect users' private data. When it comes to protecting sensitive data, different organizations have different needs. Each use-case brings its own level of risk and corresponding risk reduction once it is addressed. However, and despite the different needs of each use-case the main problem always remains the same – *how to prevent*

* This work was funded by the ASCLEPIOS: Advanced Secure Cloud Encrypted Platform for Internationally Orchestrated Solutions in Healthcare Project No. 826093 EU research project.

unauthorized access to the data that are stored in a remote location. Furthermore, it has been observed that failing to provide users with proper security and privacy-preserving mechanisms it can slow down the overall adoption of cloud services.

Having this in mind, researchers have proposed many protocols where users' data are sent to the cloud service provider (CSP) over an encrypted channel and upon reception, the CSP is responsible for storing encrypted versions of the data. However, most of the existing approaches fail to protect users' data against internal attacks. This is due to the fact that the key used for the encryption of the data is known to the CSP. Hence, a malicious CSP is able to reveal the content of all stored data – even without user's permission. To overcome this problem, researchers lately started proposing solutions that are based on the promising concept of symmetric searchable encryption (SSE) [16,**?**]. SSE allows users to symmetrically encrypt data with a key that is not known to the CSP and then search directly over the encrypted data. However, these approaches discourage a user from sharing data with other users. Sharing a symmetrically encrypted file requires the sharing of the secret key. Hence, when a user needs to be revoked, data owner needs to re-encrypt data with a fresh key and distribute the new key to the rest of the legitimate users.

To address the problem of data access and revocation, solutions that are based on Attribute-Based Encryption schemes (ABE) [23,**?**] have been proposed. ABE schemes allow user to encrypt a file based on a certain policy. Then, a unique key is generated for each user that has access to the CSP resources. This key is generated based on a list of attributes. Then a user is able to decrypt a file that is associated with a certain policy only if the attributes of her key satisfy the underlying policy. As a result, revoking a user does not affect the rest of the users. This is due to the fact that only the key for the corresponding revoked user has been blacklisted while the keys for the rest of the users are still functional. While ABE schemes offer great flexibility in terms of access management, the size of the produced ciphertexts and the time required to both encrypt and decrypt them grows with the complexity of the underlying policy [15]. As a result, protocols and cloud-based services that are solely based on such schemes are currently considered as inefficient.

1.1 Contribution

Having identified both the advantages and the disadvantages of Symmetric Searchable Encryption and Attribute-Based Encryption schemes we propose a protocol that allows users of a cloud service to securely and efficiently share files between multiple users. Furthermore, the proposed solution can be thought as an independent contribution to the field of *hybrid encryption* since it combines both SSE and ABE schemes. Moreover, the designed protocol allows users to directly search over encrypted data while the access to the decryption key is protected by a Ciphertext-Policy Attribute-Based Encryption scheme. To the best of our knowledge, *this is the first work that combines these two promising encryption*

techniques and we hope that will inspire protocol designers to conduct further research towards that direction.

Apart from that, in this work we have been looking at the problem of key revocation in ABE schemes. As pointed out in [8], key revocation is traditionally a difficult to solve problem in ABE schemes. The main problem is that in such schemes several different users might satisfy the decryption policy. In this paper, we tried to overcome this problem by looking on alternative ways to revoke users' access. To this end, in our design we *separated the revocation functionality from the actual ABE scheme*. More precisely, we used Intel's Software Guard Extensions (SGX) to host a revocation authority in a trusted execution environment. The proposed technique is considered as practical and provably secure since Intel SGX provides hardware support for isolated program execution environments.

1.2 Organization

The remainder of this paper is organized as follows: In Section 2, we present important works that have been published and address the problem of secure cloud storage and data sharing in the cloud. In Section 3, we present the main entities that participate in our system model and we proceed by defining the problem statement. In Section 4, we describe the cryptographic primitives that are needed for a proper run of the protocol and we introduce the threat model that we will consider throughout the paper. In Section 5 we provide a formal construction of the protocol while in Section 6 we prove the soundness of our protocol against several malicious behaviors. In Section 7 we present extended experimental results that shows the performance of our approach under real life (i.e. realistic) scenarios. Finally, in Section 8 we present our conclusions.

2 Related Work

In this section we present related works that mainly focus on the problem of secure cloud storage with data sharing functionality.

In [20] authors presented a framework for data and operation security in Infrastructure-as-a-Service (IaaS) clouds, consisting of protocols for a trusted launch of virtual machines and domain-based storage protection. Its security guarantees are supported by an extensive theoretical analysis with proofs about protocol resistance against attacks in the defined threat model. The protocols allow trust to be established by remotely attesting host platform configuration prior to launching guest virtual machines and ensure confidentiality of data in remote storage, with encryption keys maintained outside of the IaaS domain. In addition to that, authors provide functionality for sharing data between different domains. To this end, they presented an XML-based language framework [21] which enables clients of IaaS clouds to securely share data and clearly define access rights granted to peers.

Santos et al. [24] proposed Excalibur, a system using trusted computing mechanisms to allow decrypting client data exclusively on nodes that satisfy a

tenant-specified policy. Excalibur introduces a new trusted computing abstraction, *policy-sealed data* to address the fact that TPM abstractions are designed to protect data and secrets on a standalone machine, at the same time over-exposing the cloud infrastructure by revealing the identity and software fingerprint of individual cloud hosts. The core of Excalibur is "*the monitor*", which is a part of the cloud provider, which organises computations across a series of hosts and provides guarantees to tenants. Tenants first decide a policy and receive evidence regarding the status of the monitor along with a public encryption key, and then encrypt their data and policy using ciphertext-policy attribute-based encryption (CP-ABE) [8]. To decrypt, the stored data hosts receive the decryption key from the monitor who ensures that the corresponding host has a valid status and satisfies the policy specified by the client at encryption time. The main drawback of this work is that the protocol does not offer revocation functionality. In addition to that, users' files are stored by using explicitly a CP-ABE scheme. Hence, the overall performance of the framework is considered as low.

In [1] the authors presented a forward-looking design of a cryptographic cloud storage built on an untrusted IaaS infrastructure. The approach aims to provide confidentiality and integrity, while retaining the benefits of cloud storage – availability, reliability, efficient retrieval and data sharing – and ensuring security through cryptographic guarantees rather than administrative controls. The solution requires four client-side components: *data processor, data verifier, credential generator and token generator*. Important building blocks of the solution are: *Symmetric searchable encryption (SSE)*, appropriate in settings where the data consumer is also the one who generates it (efficient for single writer-single reader (SWSR) models) and *Asymmetric searchable encryption (ASE)*, appropriate for many writer single reader (MWSR) models, offers weaker security guarantees as the server can mount a dictionary attack against the token and learn the search terms of the client.

In [17] authors presented a constructive design for secure storage and file sharing in cloud environments. The scheme was based on a SSE scheme that allows patients of an electronic healthcare system to securely store encrypted versions of their medical data and search directly on them without having to decrypt them first. Even though the scheme offers some kind of secure sharing it lacks flexibility since data sharing is not based on policies. Furthermore, even though authors provide a discussion regarding access revocation they do not provide a concrete solution. Hence, the protocol is considered as inefficient for sharing large amount of data between multiple users.

All the approaches described above have the same limitation. More precisely, revoking access to misbehaving users is considered as inefficient since it requires to decrypt the encrypted data and then re-encrypt them with a fresh key. This also implies that data owner will have to share again the fresh key with the legitimate users. Our approach overcomes this limitation by using a CP-ABE encryption scheme combined with SSE while at the same time allows efficient revocation of misbehaving users.

Authors in [26] proposed an efficient access control scheme that allows users to dynamically update a policy. The policy update is outsourced to the cloud server while at the same time the server does not learn any private information regarding the processed data. In addition to that, the scheme is based on ABE and assumes a semi-trusted and not fully trusted cloud service provider. Furthermore, the proposed scheme supports different types of access policies (e.g., Boolean Formulas, LSSS Structure and Access Tree).

In [18], author showed how to construct a framework for secure file sharing by using the benefits of Revocable Attribute-Based Encryption. More precisely, the protocol is using a Key-Policy Attribute-Based technique through which access revocation is optimized. In addition to that, author showed how to securely remove access to a file, for a certain user that is misbehaving or is no longer part of a user group, without having to decrypt and re-encrypt the original data with a new key or a new policy.

Boldyreva et al. proposed a revocation scheme [9] that supports only a limited set of ABE functionality. Sahai et al.'s revocation scheme [22], which is based on the notion of re-encrypting from one policy to another more restrictive policy by utilizing the delegation capabilities of the underlying ABE construct, requires maintaining additional attributes and relatively expensive operations even though the complexity is reduced to the polylogarithm number of group members. Furthermore, in applications involving stateless members where it is not possible to update the initially given private keys and the only way to revoke a member is to exclude it from the public information, an ABE based approach does not readily work.

While the last schemes provide an interesting solution for the problem of revocation, they solely rely on the use of ABE schemes. As a result, the produced ciphertexts are rather long and the decryption requires lot of computational resources [15]. In our approach, we mainly rely on the use of an SSE scheme for the decryption of users' data. Thus, making the overall protocol more efficient. Additionally, our approach can pave the way for key-industrial players to build novel cloud-based services based on these two promising encryption techniques.

3 System Model and Problem Statement

In this section, we introduce the system model that we consider by explicitly describing the main entities that participate in our protocol as well as their capabilities. Furthermore, we strictly define the problem statement.

Cloud Service Provider (CSP): One of the common models of a cloud computing platform is Infrastructure-as-a-Service (IaaS). In its simplest form, such a platform consists of cloud hosts which operate virtual machine guests and communicate through a network. Often a cloud middleware manages the cloud hosts, virtual machine guests, network communication, storage resources, a public key infrastructure and other resources. Cloud middleware creates the *cloud infrastructure* abstraction by weaving the available resources into a single platform. In our system model we consider a cloud computing environment based on a trusted

IaaS provider similar to the one described in [20]. The IaaS platform consists of cloud hosts which operate virtual machine guests and communicate through a network. In addition to that, we assume a Platform-as-a-Service (PaaS) provider, like the one described in [25], that is built on top of the IaaS platform and can host multiple outsourced databases. Furthermore, the cloud service provider is responsible for storing users' data. Finally, the CSP must support the Intel Software Guard Extensions (SGX) [12] since core entities of the protocol will be running in a trusted execution environment offered by SGX.

Master Authority (MS): MS is responsible for setting up all the necessary public parameters that are needed for the proper run of the underlying protocols. Furthermore, MS is responsible for generating and distributing ABE keys to the registered users. Finally, MS is considered as a single trusted authority. Thus, we assume that MS is SGX-enabled and is running in an enclave called the Master Enclave.

Key Tray (KeyTray): KeyTray is a key storage that exists in the CSP and stores ciphertexts of all the symmetric keys that have been generated by various data owners and are needed in order to decrypt data. Every registered user can contact the KeyTray directly and request access to the stored ciphertexts. Furthermore, the symmetric keys are encrypted with a CP-ABE scheme. Thus, a single symmetric key is encrypted only once and users with certain access rights and different keys are able to access it (i.e. decrypt it). Moreover, similar to MS, KeyTray is also SGX-enabled and is running in an enclave called the KeyTray Enclave.

Revocation Authority (REV): REV is responsible for maintaining a revocation list (rl) with the unique identifier of the users that have been revoked. At this point it is worth mentioning that a single user might own more than one CP-ABE secret key. Therefore, rl maintains a mapping of users with the CP-ABE keys they own. Every time that a key of a user is revoked, REV needs to update rl. This, as we will see later, will prevent revoked users from accessing ciphertexts that are not authorized anymore. Similar to MS and KeyTray, REV is also SGX-enabled and is running in an enclave called the Revocation Enclave.

Registration Authority (RA): RA is responsible for the registration of users in the CSP. Additionally, RA has a public/private key pair denoted as $\mathsf{pk}_{\mathsf{RA}}/\mathsf{sk}_{\mathsf{RA}}$. RA can run as a separate third party but can be also implemented as part of the CSP. The registration process is out of the scope of this paper. Thus, we will not describe how the registration of a new user takes place. Instead, we will assume that a user has been already registered and has access to the remote storage and the services offered by the CSP.

User (u): In our scenario a user interacts with the CSP to manage certain files that has access to. The operations that a user can perform are: *a)* register to the service, *b)* generate encryption keys to safely protect her data, *c)* store data in the cloud, *d)* share data with other users by creating certain policies using a Ciphertext-Policy Attribute-Based Encryption scheme. Furthermore, each user has a unique identifier i. A user u_i might be also referred as data owner when she is the one who generates a certain file. Each u_i has a public/private key pair

($\mathsf{pk_i}/\mathsf{sk_i}$). The private key is kept secret, while the public key is shared with the rest of the community. These keys will be used to secure message exchanges. Hence, the communication lines between parties are assumed to be secure. It is also assumed that the public keys of all entities in the system model are known to all registered users.

Problem Statement: *Let $\mathcal{U} = \{u_1, \ldots, u_n\}$ be the set of all users that are registered through a registration authority (RA) and CSP the cloud service provider that stores users' data. Lets assume that a user u_i stores a set of m different files to the CSP. We denote this set of files as $\mathcal{D}_i = \{d_1^i, \ldots d_m^i\}$. The problem here is to find a way to achieve the following:*

1. *Keep the content of each $d_j^i \in \mathcal{D}_i$ private against both internal and external attacks;*
2. *User u_i should be able to securely share a file d_j^i with another user based on a certain policy;*
3. *A data owner u_i should be able to efficiently revoke access to a user u_c for a file that has shared with her. This should not require the data owner to decrypt and re-encrypt the file with a fresh key and it should not affect the access to the rest of the legitimate users.*

4 Cryptographic Primitives and Threat Model

In this section, we introduce the notations that we use throughout the paper while we also give a formal definition for the two main encryption schemes that the paper is based on. Finally, we present the threat model that we consider.

4.1 Cryptographic Primitives

In order to provide a concrete and reliable solution for the problem described in Section 3, we need to build a protocol through which newly encrypted data will not be decryptable by a user if her access has been revoked. Additionally, we want to allow users with certain access rights to be able to search directly over encrypted data. To this end, we will be using a CP-ABE scheme. In a CP-ABE scheme every secret key (e.g. user key) is generated based on a public and a private key as well as on a concrete list of attributes $\mathcal{A}$. Then, every ciphertext is associated with a policy P. Then, decryption is only possible if $P(\mathcal{A}) = \mathsf{True}$ – if the attributes on a key satisfy the policy on the ciphertext. From now on we will refer to the space of attributes as $\Omega = \{a_1, \ldots, a_n\}$, while the space of policies will be denoted as $\mathcal{P} = \{P_1, \ldots, P_m\}$.

We now proceed with the definition of a CP-ABE scheme as described in [8].

Definition 1 (Ciphertext-Policy ABE). *A CP-ABE scheme is a tuple of the following four algorithms:*

1. $\mathsf{CPABE.Setup}$ *is a probabilistic algorithm that takes as input a security parameter λ and outputs a master public key* MPK *and a master secret key* MSK. *We denote this by* $(\mathsf{MPK}, \mathsf{MSK}) \leftarrow \mathsf{Setup}(1^\lambda)$.

2. $\mathsf{CPABE.Gen}$ *is a probabilistic algorithm that takes as input a master secret key, a set of attributes* $\mathcal{A} \in \Omega$ *and the unique identifier of a user and outputs a secret key which is bind both to the corresponding list of attributes and the user. We denote this by* $(\mathsf{sk}_{\mathcal{A},\mathsf{u_i}}) \leftarrow \mathsf{Gen}(\mathsf{MSK}, \mathcal{A}, u_i)$.
3. $\mathsf{CPABE.Enc}$ *is a probabilistic algorithm that takes as input a master public key, a message* m *and a policy* $P \in \mathcal{P}$*. After a proper run, the algorithm outputs a ciphertext* c_P *which is associated to the policy* P*. We denote this by* $\mathsf{c_P} \leftarrow \mathsf{Enc}(\mathsf{MPK}, m, P)$.
4. $\mathsf{CPABE.Dec}$ *is a deterministic algorithm that takes as input a user's secret key and a ciphertext and outputs the original message* m *iff the set of attributes* $\mathcal{A}$ *that are associated with the underlying secret key satisfies the policy* P *that is associated with* c_p*. We denote this by* $\mathsf{Dec}(\mathsf{sk}_{\mathcal{A},\mathsf{u_i}}, c_P) \rightarrow m$.

Apart from the CP-ABE scheme, one of the core components of our solution is the SSE component which will allow users to encrypt their data using a symmetric secret key and later search directly over the encrypted data. In the rest of the paper, we will be assuming the existence of the following SSE scheme:

Definition 2 (Dynamic Index-based SSE). *A dynamic index-based symmetric searchable encryption scheme is a tuple of nine polynomial algorithms* $\mathsf{SSE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{SearchToken}, \mathsf{AddToken}, \mathsf{DeleteToken}, \mathsf{Search}, \mathsf{Add}, \mathsf{Delete}, \mathsf{Dec})$ *such that:*

- $\mathsf{SSE.Gen}$ *is probabilistic key-generation algorithm that takes as input a security parameter* λ *and outputs a secret key* K*. It is used by the client to generate her secret-key.*
- $\mathsf{SSE.Enc}$ *is a probabilistic algorithm that takes as input a secret key* K *and a collection of files* $\mathbf{f}$ *and outputs an encrypted index* γ *and a sequence of ciphertexts* $\mathbf{c}$*. It is used by the client to get ciphertexts corresponding to her files as well as an encrypted index which are then sent to the storage server.*
- $\mathsf{SSE.SearchToken}$ *is a (possibly probabilistic) algorithm that takes as input a secret key* K *and a keyword* w *and outputs a search token* $\tau_s(w)$*. It is used by the client in order to create a search token for some specific keyword. The token is then sent to the storage server.*
- $\mathsf{SSE.AddToken}$ *is a (possibly probabilistic) algorithm that takes as input a secret key* K *and a file* f *and outputs an add token* $\tau_a(f)$ *and a ciphertext* c_f*. It is used by the client in order to create an add token for a new file as well as the encryption of the file which are then sent to the storage server.*
- $\mathsf{SSE.DeleteToken}$ *is a (possibly probabilistic) algorithm that takes as input a secret key* K *and a file* f *and outputs a delete token* $\tau_d(f)$*. It is used by the client in order to create a delete token for some file which is then sent to the storage server.*
- $\mathsf{SSE.Search}$ *is a deterministic algorithm that takes as input an encrypted index* γ*, a sequence of ciphertexts* $\mathbf{c}$ *and a search token* $\tau_s(w)$ *and outputs a sequence of file identifiers* $\mathbf{I}_w \subset \mathbf{c}$*. This algorithm is used by the storage server upon receive of a search token in order to perform the search over the*

encrypted data and determine which ciphertexts correspond to the searched keyword and thus should be sent to the client.

- SSE.Add *is a deterministic algorithm that takes as input an encrypted index γ, a sequence of ciphertexts $\mathbf{c}$, an add token $\tau_a(f)$ and a ciphertext c_f and outputs a new encrypted index γ' and a new sequence of ciphertexts $\mathbf{c}'$. This algorithm is used by the storage server upon receive of an add token in order to update the encrypted index and the ciphertext vector to include the data corresponding to the new file.*
- SSE.Delete *is a deterministic algorithm that takes as input an encrypted index γ, a sequence of ciphertexts $\mathbf{c}$ and a delete token $\tau_d(f)$ and outputs a new encrypted index γ' and a new sequence of ciphertexts $\mathbf{c}'$. This algorithm is used by the storage server upon receive of a delete token in order to update the encrypted index and the ciphertext vector to delete the data corresponding to the deleted file.*
- SSE.Dec *is a deterministic algorithm that takes as input a secret key* K *and a ciphertext c and outputs a file f. It is used by the client to decrypt the ciphertexts that she gets from the storage server.*

4.2 Threat Model

Our threat model is similar to the one described in [20], which is based on the Dolev-Yao adversarial model [13] and further assumes that privileged access rights can be used by a remote adversary $\mathcal{ADV}$ to leak confidential information. $\mathcal{ADV}$, e.g. a corrupted system administrator, can obtain remote access to any host maintained by the IaaS provider, but cannot access the volatile memory of guest VMs residing on the compute hosts of the IaaS provider.

Hardware Integrity: We assume that the cloud provider has taken all the necessary technical and non-technical measures in order to protect the underling hardware from tampering.

Physical Security: We assume physical security of the data centres where the IaaS is deployed. This assumption holds both when the IaaS provider owns and manages the data center (as in the case of Amazon Web Services, Google Compute Engine, Microsoft Azure, etc.) and when the provider utilizes third party capacity, since physical security can be observed, enforced and verified through known best practices by audit organizations. This assumption is important to build higher-level hardware and software security guarantees for the components of the IaaS. We assume the record is kept on protected storage with read-only access and the adversary cannot tamper with it.

Network Infrastructure: The IaaS provider has physical and administrative control of the network. $\mathcal{ADV}$ is in full control of the network configuration, can overhear, create, replay and destroy all the exchanged messages between the CSP and their resources (virtual machines, database components etc.) as well as with other entities in our system model (e.g. the Master Authority).

Cryptographic Security: We assume encryption schemes are semantically secure and the $\mathcal{ADV}$ cannot obtain the plaintext of encrypted messages. Moreover, we explicitly assume that $\mathcal{ADV}$ cannot forge a revocation list and cannot

decrypt a ciphertext without knowing the corresponding secret key. Furthermore, we assume that the probability of $\mathcal{ADV}$ guessing a generated random number is negligible. Finally, we explicitly exclude denial-of-service attacks from our adversarial model and we focus on $\mathcal{ADV}$ that aims to compromise the confidentiality of data by forging existing access policies generated by the corresponding data owners.

5 The Lord of the Shares (LotS)

In this section, we present the Lord of the Shares (LotS) that constitutes the core of this paper's contribution. To this end, we continue with providing the construction of the main protocols that are involved. As we described earlier, LotS runs three secure enclaves (Master, KeyTray and the Revocation enclave) and it contains the following nine main protocols: Setup, ABEUserKey, Store, KeyTrayStore, KeyShare, Search, Update, Delete and Revoke. Due to space constraints, we omit the description of trivial functions such as the registration of a new user. LotS mainly comprises a public-key encryption scheme, a ciphertext-policy attribute-based encryption scheme and a symmetric searchable encryption scheme.

LotS.Setup : Each entity from the described system model obtains a public/private key pair $(\mathsf{pk}, \mathsf{sk})$ for a CCA2 secure public cryptosystem and publishes its public key while it keeps the private key secret. Apart from that, all three entities that are running in an enclave they generate a signing and a verification key. Furthermore, MS runs CPABE.Setup and generates a master public and private key. Below we provide the list of the generated key pairs:

- $(\mathsf{pk}_{\mathsf{CSP}}, \mathsf{sk}_{\mathsf{CSP}})$ – public/private key pair for the cloud service provider;
- $(\mathsf{pk}_{\mathsf{MS}}, \mathsf{sk}_{\mathsf{MS}})$, $(\mathsf{sig}_{\mathsf{MS}}, \mathsf{ver}_{\mathsf{MS}})$, $(\mathsf{MPK}, \mathsf{MSK})$ – public/private, verification/signing and master key pairs for the Master Authority;
- $(\mathsf{pk}_{\mathsf{KT}}, \mathsf{sk}_{\mathsf{KT}}, \mathsf{sig}_{\mathsf{KT}}, \mathsf{ver}_{\mathsf{KT}})$ – public/private and verification/signing key pairs for the KeyTray;
- $(\mathsf{pk}_{\mathsf{REV}}, \mathsf{sk}_{\mathsf{REV}}, \mathsf{sig}_{\mathsf{REV}}, \mathsf{ver}_{\mathsf{REV}})$ – public/private key and verification/signing key pairs for the Revocation Authority.

LotS.ABEUserKey : This phase is taking place between a registered user u_i that wishes to obtain a CP-ABE key and MS who is responsible for generating such keys. This is a probabilistic key-generation algorithm that runs in the master enclave and takes as input MSK, the identity of the user that is requesting a key and a list of attributes $\mathcal{A}$ that is derived from user's registered information. More precisely, u_i contacts MS and proves that she is a registered user. Then, attests MS and requests a new CP-ABE key. MS then runs CPABE.Gen and generates $\mathsf{sk}_{\mathcal{A},\mathsf{u_i}}$. This is then sent back to the user over a secure channel.

LotS.Store : After a successful registration, we assume that u_i has received a valid credential ($cred_i$) that can be used to login to a cloud service offered by the CSP. Additionally, u_i is now able to store data to the cloud storage. During

this phase the communication takes place between the user and the CSP. First, u_i contacts the CSP by sending the following: $m_1 = \langle r_1, \mathsf{E}_{\mathsf{pk_{CSP}}}(Auth), StoreReq, H_1 \rangle$, where r_1 is a random number generated by u_1, $Auth$ is an authenticator that allows u_i to prove to the CSP that is a legitimate/registered user and H_1 is the following hash $H(r_1||Auth||StoreReq)$. Upon reception, CSP verifies the freshness of the message, the identity of the user and starts processing the store request. To do so, CSP creates the message $m_2 = \langle r_2, \sigma_{CSP}(H_2) \rangle$, where H_2 is the following hash $H(r_2||u_i)$ and $\sigma_{CSP}(H_2)$ is a signature of CSP on H_2. Then, m_2 is sent back to u_i. Upon reception, u_1 verifies both the freshness as well as the integrity of the message. Now, u_i simply generates a symmetric key $\mathsf{K_i}$ by running $\mathsf{SSE.Gen}$. This key will be used to protect the data that will be stored in the cloud. The final step of this phase is the storage of encrypted files by u_i to a storage resource offered by the CSP. User u_i runs $\mathsf{StoreFile}$ – a deterministic algorithm that takes as input the symmetric secret key $\mathsf{K_i}$ that generated earlier and a collection of files $\mathbf{f_i}$ and outputs a collection of ciphertexts $\mathbf{c_i}$ as well as an encrypted index γ_i. Both γ_i and $\mathbf{c_i}$ are then send to the CSP via a secure channel. More precisely, u_i sends the following message to the CSP: $m_3 = \langle r_3, \mathsf{E}_{\mathsf{pk_{CSP}}}(\gamma_i), \mathbf{c_i}, H_3 \rangle$, where $H_3 = H(r_3||\gamma_i||\mathbf{c_i})$. Upon reception, CSP verifies both the integrity and the freshness of m_3 and stores $\mathbf{c_i}$ along with the encrypted index γ_i in a local database.

LotS.KeyTrayStore : A key storage algorithm allows an already logged-in user to safely store a symmetric secret key $\mathsf{K_i}$, that generated earlier, in the KeyTray. This is a probabilistic algorithm that takes as input a symmetric key $\mathsf{K_i}$, MPK and a policy P and outputs an encrypted version of $\mathsf{K_i}$ which is associated with P. This is done by running $c_P^{\mathsf{K_i}} \leftarrow \mathsf{CPABE.Enc}(\mathsf{MPK}, \mathsf{K_i}, P)$. The generated ciphertext, is sent by u_i to the KeyTray who stores it locally. More precisely, u_i first attests the KeyTray and then sends the following message: $m_4 = \left\langle \mathsf{E}_{\mathsf{pk_{KT}}}(r_4), c_P^{\mathsf{K_i}}, \sigma_i \left(H \left(r_4 || c_P^{\mathsf{K_i}} \right) \right) \right\rangle$. Additionally, the KeyTray generates a random number $r_{\mathsf{K_i}}$ encrypts it with $\mathsf{pk_i}$ and stores it next to $c_P^{\mathsf{K_i}}$. As we will see later, this number will be used during the revocation phase to prove that u_i is the owner of $\mathsf{K_i}$.

LotS.KeyShare : Now that u_i has stored an encrypted version of $\mathsf{K_i}$ to the KeyTray, other users should be able to access it. Hence, u_i must have a way to share the encrypted data $\mathbf{c_i}$ that stored earlier. Lets assume that there is another registered user u_j, $j \neq i$ that wishes to access $\mathbf{c_i}$. To do so, u_j needs to get access to $\mathsf{K_i}$ that is stored in the KeyTray. The important thing to notice here is that the data sharing will be done without the involvement of u_i. Therefore, after u_i stores $c_P^{\mathsf{K_i}}$ to the KeyTray, she can be offline. In order for u_j to access $c_P^{\mathsf{K_i}}$ she first needs to get a special token from REV that will prove that u_j's access has not been revoked. To this end, u_j first attests REV and then sends the following message to obtain the token: $m_5 = \langle r_5, \mathsf{E}_{\mathsf{pk_{REV}}}(u_j), \sigma_j(r_5||u_j) \rangle$. Upon reception, REV verifies the integrity and the freshness of the message and checks if $u_j \in rl$. In such case, REV drops the connection since u_j has been revoked. Otherwise, REV generates a token τ_{KS} and sends the following to u_j: $m_6 =$

$\langle r_6, \mathsf{E}_{\mathsf{pk_{KT}}}(u_j), \mathsf{E}_{\mathsf{pk_{KT}}}(\tau_{KS}), \sigma_{REV}\,(H(r_6||u_j||\tau_{KS}))\rangle$. Upon reception, u_j forwards m_6 to the KeyTray who verifies the signature as well as the freshness and user's id and sends $c_P^{\mathsf{K_i}}$ to u_j. At this point, u_j uses her private CP-ABE key to recover $\mathsf{K_i}$. The decryption will only work if the attributes that are associated with u_j's key satisfy the policy that is associated with $c_P^{\mathsf{K_i}}$. Apart from that, the KeyTray sends also the following to u_j: $m_7 = \langle \mathsf{E}_{\mathsf{pk_{CSP}}}(u_j, t), \sigma_{KT}\,(H(u_j||t))\rangle$, where t is the time that u_j accessed $c_P^{\mathsf{K_i}}$. As we will see in the next step, t plays a crucial role in the access control.

LotS.Search : Now that u_j has gained access to $\mathsf{K_i}$, she can start searching directly over encrypted data. Lets assume that u_j wishes to search over ciphertexts that have been encrypted with $\mathsf{K_i}$, for a specific keyword w. To do so, she first forwards to the CSP m_7 that received in the previous step. Upon reception, CSP decrypts $\mathsf{E}_{\mathsf{pk_{CSP}}}(u_j, t)$, verifies the signature and then checks if t is valid. We assume that there is a time interval since u_j got access to $\mathsf{K_i}$, where she is eligible to access files that are stored in the CSP. After that time, u_j will have to run again the previous step in order to receive a fresh timestamp. This will guarantee that u_j has not been revoked since the last time that got access to $\mathsf{K_i}$. Then, if all the verifications are successful, u_j runs $\mathsf{SSE.SearchToken}\,(\mathsf{K_i}, w) \rightarrow \tau_s(w)$ and obtains a token $\tau_s(w)$. Then, she sends the generated token to the CSP who runs $\mathsf{SSE.Search}(\gamma_i, \mathbf{c_i}, \tau_s(w)) \rightarrow \mathbf{I}_w$ that outputs a sequence of file identifiers $\mathbf{I}_w$, such that $\mathbf{I}_w \subseteq \mathbf{c_i}$. In addition to that, all files in $\mathbf{I}_w$ contain the keyword w that u_j searched for. The resulted $\mathbf{I}_w$ is sent back to the user. Upon reception, u_j executes the $\mathsf{SSE.Dec}$ algorithm by giving as input $\mathsf{K_i}$ and the sequence of encrypted files that corresponds to the list of identifiers that received from the CSP. By doing this, u_j recovers the files that contain keyword w.

LotS.Update : Apart from storing data and searching over the encrypted data, users also need to be able to update stored data. Here, we consider the scenario where u_i wishes to add a new file f to the cloud storage. A naive approach that u_i could follow would be to run $\mathsf{LotS.Store}$ again, generate the ciphertext of f and send it to the CSP. However, this would mean that u_i would also create a new encrypted index that would correspond to the encryption of file f. Such an approach is not efficient since the user would end-up with a long list of encrypted indexes that are not related to each other and every time that wishes to perform a search over her data would require from the CSP to search over all the encrypted indexes. To avoid this, u_i needs to store f but instead of creating a separate encrypted index she needs to update the current one in order to also include the newly added file. To achieve that, u_i first generates an add token by executing $(\tau_\alpha(f), c_f) \leftarrow \mathsf{AddToken}(\mathsf{K_i}, f)$ and sends it to the CSP. Upon reception, CSP executes $\mathsf{SSE.Add}\,(\gamma_i, \mathbf{c_i}, \tau_\alpha(f), c_f) \rightarrow (\gamma', \mathbf{c_i'})$ and outputs an updated encrypted index γ_i' and an updated sequence of ciphertexts $\mathbf{c_i'}$ that corresponds to the data stored by u_i. Thus, by running $\mathsf{SSE.Add}$, CSP stores the ciphertext of f and updates the existing encrypted index and ciphertext list of u_i.

LotS.Delete : Users must also be able to delete a file. Assume that u_i wishes to delete a file f. To do so, u_i runs $\mathsf{SSE.DeleteToken}$ which takes as input the sym-

metric key $\mathsf{K_i}$ and the file that needs to be deleted and outputs a delete token: $\tau_d(f) \leftarrow \mathsf{DeleteToken}\,(K_i, f)$. The generated token is sent to the CSP with the following message: $m_8 = \langle m_7, \mathsf{E}_{\mathsf{pk_{CSP}}}(\tau_d(f), \gamma_i'), \mathbf{c_i'}, H(\tau_d(f)||\gamma_i'||\mathbf{c_i'})\rangle$ Upon reception, the CSP first checks that u_i is eligible to delete a file and she has not been revoked (this is done by opening m_7 and looking at the timestamp provided by the KeyTray). Then, the CSP runs $\mathsf{SSE.Delete}\,(\gamma_i', \mathbf{c_i'}, \tau_d(f)) \rightarrow (\gamma_i'', \mathbf{c_i''})$ which removes the requested file f and updates both the corresponding encrypted index and the sequence of ciphertexts.

LotS.Revoke : The last phase of our protocol allows a data owner to revoke access to a user. We assume that u_i wishes to revoke access to u_j. To do so, u_i contacts the revocation authority (REV) by sending $m_9 = \left\langle r_9, \mathsf{E}_{\mathsf{pk_{REV}}}(u_i, u_j, c_P^{\mathsf{K_i}}), H(u_i||u_j||c_P^{\mathsf{K_i}})\right\rangle$. Upon reception, REV checks the integrity and the freshness of the message and recovers the identity of data owner (u_i) as well as the user that needs to be revoked (u_j). Then, REV contacts the KeyTray by requesting the ciphertext of $r_{\mathsf{K_i}}$ that was stored next to $c_P^{\mathsf{K_i}}$ during the run of $\mathsf{LotS.KeyTrayStore}$. So, KeyTray sends the following message to REV: $m_{10} = \left\langle \mathsf{E}_{\mathsf{pk_{u_i}}}(r_{\mathsf{K_i}}), \sigma_{KT}(H(r_{\mathsf{K_i}}||r_9))\right\rangle$. Upon reception, REV forwards m_{10} to u_i who recovers $r_{\mathsf{K_i}}$ and verifies that the message has been generated by the KeyTray (verifying the signature). Then, u_i signs $r_{\mathsf{K_i}}$ and sends it to the KeyTray through REV. KeyTray verifies the signature and is also convinced that u_i is the owner of $\mathsf{K_i}$. Hence, KeyTray generates a fresh random number $r'_{\mathsf{K_i}}$ that replaces $r_{\mathsf{K_i}}$ and also sends an acknowledgement to REV that u_i has the right to revoke access to u_j for all files that are encrypted with $\mathsf{K_i}$. Finally, REV adds the identity of u_j in rl. As a result, the next time that u_j will try to access any of the files that are encrypted with $\mathsf{K_i}$ access will be denied.

6 Security Analysis

We now analyze LotS behavior in the presence of a malicious adversary. We prove the security of the scheme through a theoretical analysis, showing its resistance to a list of malicious behaviours. Our security analysis explicitly focuses on the described protocol and not on the underlying cryptographic schemes. The security of all utilized cryptographic schemes (including CP-ABE and SSE) has been proved in other works [8,**?**]. Hence, we assume that all the involved cryptographic schemes are semantically secure.

Realistic Assumption. We assume that all user ids have the same length (or at least they are not a prefix of each other). By ensuring this property is satisfied, we can avoid a prefix attack, such as the following:

Assume two users with ids $u_0 = 001$ and $u_1 = 0011$ respectively. Then if an adversary $\mathcal{ADV}$ gets a valid signature $\sigma_{KT}\,(H(u_j||t))$ (from a valid m_7) this will be the same as a valid signature on u_0 with a much larger time. However, by setting all users' ids to have the same length we avoid such an attack.

Proposition 1 (Compromise Revoked Users). *Let $\mathcal{U}_{\mathsf{K_i}}$ be the set of all users that have been given access to $\mathsf{K_i}$ and $\mathcal{R}_{\mathsf{K_i}}$ the set of all users that their access to $\mathsf{K_i}$ has been revoked. Assume an adversary $\mathcal{ADV}$ corrupts n, $n \leq |\mathcal{R}_{\mathsf{K_i}}|$ users out of those in the set $\mathcal{R}_{\mathsf{K_i}}$. Then $\mathcal{ADV}$ cannot infer any information about the files that have been encrypted with $\mathsf{K_i}$.*

Proof. Here, we consider the case where $\mathcal{ADV}$ corrupts at least one user $u_c \in \mathcal{R}_{\mathsf{K_i}}$. In other words, $\mathcal{ADV}$ corrupts at least a user who in the past was eligible to use $\mathsf{K_i}$ and therefore she was able to decrypt all files from $\mathbf{c_i}$ that were encrypted with that key. $\mathcal{ADV}$ will try to use u_c in order to obtain the collection of ciphertexts $\mathbf{c_i}$ and access the contents of the files.

$\mathcal{ADV}$ trying to access the content of any file in $\mathbf{c_i}$ can succeed if all the following conditions hold:

a. Access the symmetric key $\mathsf{K_i}$ that used to encrypt the files;
b. Successfully bypass the authentication of CSP during the $\mathsf{LotS.Search}$ phase;
c. Access the latest ciphertexts list $\mathbf{c_i}^{fresh}$.

- *Condition a* is always true. We know that $u_c \in \mathcal{R}_{\mathsf{K_i}}$. Therefore, at some point in the past u_c was member of $\mathcal{U}_{\mathsf{K_i}}$. Hence, we can safely assume that u_c was able to decrypt $c_P^{\mathsf{K_i}}$ and recover $\mathsf{K_i}$.
- *Condition b* can only be true if the adversary convince the CSP that $u_c \notin \mathcal{R}_{\mathsf{K_i}}$. To do so, the adversary needs to generate a valid m_8 messages that will also contain a fresh timestamp, say t_c, that will prove that u_c received access to $\mathsf{K_i}$ recently and it is still active. Generating a valid m_8 message can be done with the following two options:

- *Replay Old Message:* First, we consider the case were $\mathcal{ADV}$ replays an older message m_7 in order to generate a valid m_8 that will allow her to bypass the checks of the CSP. To this end, $\mathcal{ADV}$ uses the following message that was received in the past: $m_7 = \langle \mathsf{E}_{\mathsf{pk_{CSP}}}(u_c, t), \sigma_{KT}(u_c||t) \rangle$. This is a valid message that contains the identity of the corrupted user as well as a valid signature from the KeyTray. Then $\mathcal{ADV}$ generates a fresh random number $r_8^{'}$ and creates the message $m_8 = \left\langle r_8^{'}, m_7^{'}, \sigma_j(m_7||r_8^{'}) \right\rangle$ that is sent to the CSP. Even though the structure of the generated message is correct, the CSP will drop the connection since it will identify it as an old message. This is due to the fact that the timestamp t contained in m_7 has expired. Therefore, the CSP cannot be sure if u_c's access right is still active. To bypass that, $\mathcal{ADV}$ will try to replace t with the current time t_c. To do so, the adversary will use $\mathsf{pk_{CSP}}$ to generate $\mathsf{E}_{\mathsf{pk_{CSP}}}(u_c, t_c)$, and replace the first part of m_7. However, the second part of the message has a signature from the KeyTray that contains the initial timestamp t. Replacing this with a valid signature on t_c fails due to the assumption of soundness of the signature scheme. Therefore, $\mathcal{ADV}$ will fail to bypass CSP's authentication.
- *Impersonate a Legitimate User*: The only remaining alternative for the adversary is to impersonate a legitimate user u_l from the set $\mathcal{U}_{\mathsf{K_i}}$. To do so, $\mathcal{ADV}$

overhears the communication between u_l and the CSP. By doing this, $\mathcal{ADV}$ intercepts the message m_8 that u_l sent to the CSP. This message is fresh and contains an acceptable timestamp t. However, it also contains (in m_7) the identity of u_l. This will be used at the end of the LotS.Search phase where the CSP will use $\mathsf{pk_{u_l}}$ to encrypt the data that will be sent to the user. Therefore, $\mathcal{ADV}$ will use $\mathsf{pk_{CSP}}$ and will replace $\mathsf{E_{pk_{CSP}}}(u_l, t)$ with $\mathsf{E_{pk_{CSP}}}(u_c, t)$ we denote the new message as m_7^c. In addition to that, she will calculate $\sigma_j(m_7^c||r_8$ and will contact the CSP by sending the following: $m_8 = \langle r_8, m_7^c, \sigma_j(m_7^c||r_8)\rangle$. Upon reception, CSP will verify the first signature but will fail to verify the one that is included in m_7^c. This is due to the fact that $\mathcal{ADV}$ had to change the identity of the legitimate user to u_c but she could not generate a valid signature on the new message. Hence, the attack will fail.

- *Condition c* cannot be true. This implies immediately from the exculpability of the previous attack. More precisely, in order for $\mathcal{ADV}$ to access $\mathbf{c_i}^{fresh}$, she needs to first bypass CSP's authentication. However, we showed that this is not possible. □

Proposition 2 (Revoke Legitimate Users). *Let u_i be the owner of data that has been encrypted with $\mathsf{K_i}$. Additionally, let $\mathcal{U}$ the set of all users that have been given registered with the CSP and $\mathcal{U}_{\mathsf{K_i}}$ be the set of all users that have been given access to $\mathsf{K_i}$. Assume an adversary $\mathcal{ADV}$ corrupts a user u_c, $u_c \in \mathcal{U} \setminus \{u_i\}$. Then $\mathcal{ADV}$ cannot successfully revoke access to any $u_l \in \mathcal{U}_{\mathsf{K_i}}$.*

Proof. Here, we consider the case where $\mathcal{ADV}$ corrupts a user u_c such that $u_c \in \mathcal{U} \setminus \{u_i\}$. The attack will be successful if $\mathcal{ADV}$ manages to revoke access to data that has been encrypted with $\mathsf{K_i}$ for a legitimate user $u_l \in \mathcal{U}_{\mathsf{K_i}}$. To do so, $\mathcal{ADV}$ needs to run LotS.Revoke and convince REV that she is the data owner. Hence, $\mathcal{ADV}$ generates $m_9^c = \left\langle r_9^c, \mathsf{E_{pk_{REV}}}(u_c, u_l, c_P^{\mathsf{K_i}}), H(u_c||u_j||c_P^{\mathsf{K_i}})\right\rangle$ and sends it to REV. Upon reception, REV checks the integrity and the freshness of the message and recovers the identity of u_c, who is pretending to act as data owner, as well as the id of user that needs to be revoked u_l. Then, REV contacts the KeyTray by requesting the ciphertext of $\mathsf{E_{pk_i}}(r_{\mathsf{K_i}})$ that was stored next to $c_P^{\mathsf{K_i}})$ during the run of LotS.KeyTrayStore. So, REV will receive the following: $m_{10} = \left\langle \mathsf{E_{pk_{u_i}}}(r_{\mathsf{K_i}}), \sigma_{KT}(H(r_{\mathsf{K_i}}||r_9^c))\right\rangle$. REV forwards m_{10} to u_c who fails to recover $r_{\mathsf{K_i}}$ since it is encrypted with u_i's public key. Therefore, $\mathcal{ADV}$ will fail to prove that is the data owner and the attack will not succeed.

An alternative, would be that $\mathcal{ADV}$ uses u_i's id in m_9. More precisely, $\mathcal{ADV}$ would send $m_9^c = \left\langle r_9^c, \mathsf{E_{pk_{REV}}}(u_i, u_l, c_P^{\mathsf{K_i}}), H(u_i||u_j||c_P^{\mathsf{K_i}})\right\rangle$ to REV. However, this case is identical to the previous one since $c_P^{\mathsf{K_i}}$ remains the same. Hence, KeyTray mapping will again identify u_i as the data owner.

Finally, u_c could bypass this step is either if she gains access to $\mathsf{sk_i}$ or by replacing $c_P^{\mathsf{K_i}}$ with $c_P^{\mathsf{K_c}}$, where $\mathsf{K_c}$ is a key that has been generated by u_c. However, the first case is not possible since we have assumed that a private key is only known to its owner. Regarding the second case, this will only allow u_c to revoke

u_l's access to data that u_c has generates. As a result, a corrupted user cannot revoke access to data that has not been encrypted and generated by her. □

6.1 SGX Security

The security of LotS heavily depends on the use of Intel's SGX functionality. More precisely, we are using Intel SGX to create isolated environments and launch trusted entities that any third party would be able to attest and verify their integrity. In the next paragraphs, we provide some basic information regarding Intel SGX as well as the functionality that we are using in LotS .

Isolation. Intel Software Guard Extensions (SGX) is an extension to Intel's x86 design and allows the creation of isolated execution environments, called enclaves, in which small pieces of code can be executed in isolation from the rest of the system. Software developers can use Intel SGX enclaves to create trusted execution environments (TEEs) during OS execution. Such enclaves, rely for their security on the platform's trusted computing base (TCB) code and data loaded at initialization creation time, processor firmware and processor hardware. *Program execution within an enclave is transparent to both the underlying operating system and other enclaves.* Multiple mutually distrusting enclaves can operate on the platform. Enclaves operate in a dedicated memory area called the Enclave Page Cache (EPC) which in turn is a subset of Processor Reserved Memory (PRM). The PRM is a range of DRAM reserved by BIOS that cannot be accessed by system software or peripherals. Furthermore, EPCM is a data structure that contains a mapping between the enclave identities and the EPC pages that belong to them. This mapping is used by the CPU to verify enclave access to memory pages and *prevent unauthorized access.* The CPU firmware and hardware are the Root of Trust of an enclave. It prevents access to the memory segment of the enclave either by the platform operating system, other enclaves, or other external agents.

Attestation. Simply assuming that trustworthy software is executing on the target device is (in most cases) insufficient for a remote user to establish a trust relationship with the target platform. That is because remote users cannot know whether they are communicating with the intended software or a maliciously modified instance. Therefore, attestation is of central importance for trust establishment in a remote system. Attestation of a target can be performed either by a dedicated appraiser, or directly by the remote user. Users delegate trust (either directly or transitively) to an appraiser, which is an entity – generally a computer or a network – making a decision about one or more other targets. A target is a party (for example a computer system) about which an appraiser needs to make such a decision. Alternatively, a remote user can itself assume the appraiser role and attest the target using components that are part of it or under its control. As described in [11] *"Attestation is the activity of making a claim to an appraiser about the properties of a target by supplying evidence which supports that claim. An attester is a party performing this activity. An appraiser's decision-making process based on attested information is appraisal."* The goal of an appraisal is to take a decision regarding the expected behavior

of the target prior to establishing a trust relationship. This is done by collecting enough information about the target – such as hardware, software, and configuration data – in order to establish that the target is in an acceptable state or will not transfer to an unacceptable state after a trust relationship is established.

Attestation can be done either *remotely* or *locally*. The main idea of the remote attestation, is that any remote party can verify the integrity and the trustworthiness of a an entity. For a detailed description of a remote attestation protocol we refer reader to [20]. On the other hand, local attestation is taking place between two enclaves that are part of the same platform. In our case, we assume that each time two enclaves communicate with each other, they first run a local attestation protocol. For more information on how local attestation works, we refer reader to [14].

7 Experimental Results

This section presents the implementation of the main parts of our protocol. More precisely, in order to prove the effectiveness of the protocol we conducted extensive experimental results even under conditions that can be considered as unrealistic (e.g. creating a key that is associated with a list of 1,000 attributes). These experiments allowed us to examine the behavior of the utilized encryption schemes under various scenarios (demanding, less demanding etc.) and argue about the overall applicability of our approach. As it is evident from the conducted experiments, the overall performance of the protocol proves that it can be used in real life scenarios and has the potential to considerably expand the cryptographic methods that we currently use in most of the existing online services.

Our experiments mainly aimed at analyzing the processing time of both utilized encryption schemes (ABE and SSE). In order to provide a well-rounded analysis that would also allow us to identify possible incongruities, we considered several scenarios to give a concrete picture of the performance of our approach. Our test bed for the cloud environment included a Dell PowerEdge R320 host connected on a Cisco Catalyst 2960 switch . Furthermore, we used Linux CentOS, kernel version 2.6.32-358.123.2.openstack.el6.x86_64 and the OpenStack IaaS platform[1] (version Icehouse) using KVM virtualization support. For the client, all experiments were implemented on a Macbook Pro laptop with a 2.9 GHz Intel Core i5 processor and 8GB RAM.

7.1 Ciphertext-Policy Attribute-Based Encryption

For the implementation of the ciphertext-policy attribute-based encryption, we used the library provided by Bethencourt et al. [8]. Furthermore, our experiments were implemented in Python 2.7.10.

[1] OpenStack project website: `https://www.openstack.org/`

Setup Phase The first phase of the experiments, was dedicated to measuring the time that is needed to generate a pair of keys for a master entity. This is part of the setup phase for our framework. Even though applications that will be based on our protocol or in general, applications that will be using an ABE scheme can run with the existence of a single master entity, it is common that more than one master entities will exist. Especially, in multi-cloud environments where several CSPs are connected to each other, one master entity can be generated for each CSP. Apart from that, depending on the security level that needs to be achieved, a master key pair might be generated for each data owner. Such an approach can also lead to a multi-authority ABE model as described in [10]. By doing this, someone could argue that the overall security of the system is increased since there is no single point of failure. However, it has been observed that in a multi-authority setting malicious adversaries may collude to successfully launch attacks based on information received by key components from different authorities. To cover all these cases, we ran an algorithm where up to 200 master key pairs were generated simultaneously. The result of the experiment is illustrated in Figure 1. As can be seen, the time needed to generate 200 master key pairs at the same time (e.g. during the setup phase of a large cloud environment) is less than 12 sec. Furthermore, the average time to generate just one pair of master keys (i.e. one single master entity) is 0.061 sec – which is considered as acceptable. Apart from that, as can be seen from Figure 1, the processing time is increasing almost linearly with the number of pairs that are generated.

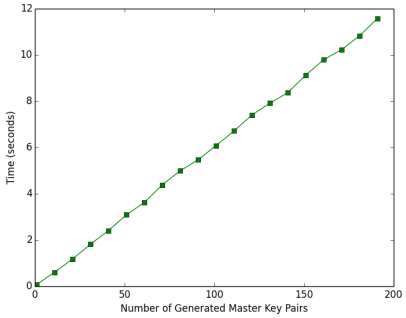

Fig. 1: Generation of master public/private key pairs.

Users Key Generation The next phase of our experiments, focused on the generation of the users' keys. Since in ABE users' keys are associated with attributes, we first developed an algorithm for generating a list of attributes with different length. The generated attributes were of the form `Attribute_1, ..., Attribute_n`. Furthermore, we wanted to use a mix of regular and numerical attributes. This would allow us to later on create more complex policies (e.g.

check if a user is at least 21 years old) and thus give results that are closer to real world scenarios. Hence, we also used a list of numerical attributes. Then, we measured the processing time needed for the generation of a user key. Each of the generated keys was associated with a list of attributes of different length. More precisely, for every generated key we were increasing the number of associated attributes by one. Since we wanted to measure the overall processing time even under demanding circumstances, we created keys with up to 1000 attributes. This is considered enough even for very large organizations and/or publicly available online services where typically an abundance of information about users is stored. The results of this experiment are illustrated in Figure 2. As can be seen from the figure, the time for generating a key associated with 1000 attributes takes around 11 sec. However, this is considered as an extreme and not very likely case. However, generating a key that is associated with 100 attributes takes around 1.2 sec. This number of attributes is considered suitable for covering even complex cases where companies need to generate keys based on a wide variety of information. Hence, it is clear that the time needed in order to generate keys for a large number of users, while at the same time cover a long list of options (i.e. attributes), is realistic and does not prevent an organization from adopting such an approach.

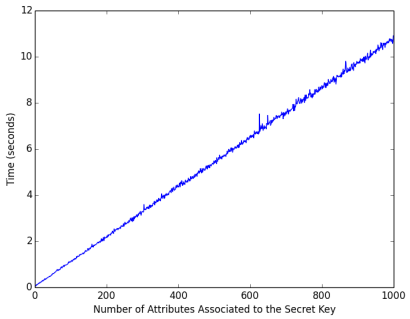

Fig. 2: Generation of user key with up to 1000 attributes

Apart from measuring the processing time needed to generate keys, it was interesting to see how the size of the generated file is changing as the list of attributes that are associated with a key are increasing. Figure 3 illustrates the results of this experiment. As we can see, the size of the generated file is increasing precisely linear with the number of attributes. A key that is associated with only one attribute has a size of 412 bytes on the disk, while the size of a key with 10 attributes is increasing to 3KB. Furthermore, a key associated with 100 attributes has a size of 28KB on the disk while in the extreme case where a key is associated with 1000 attributes the size is increasing to 278KB.

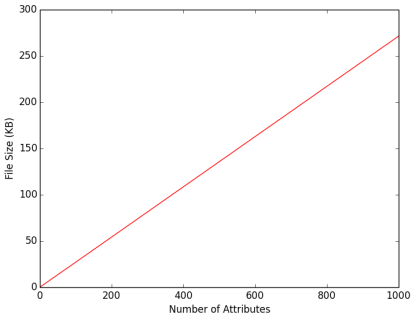

Fig. 3: Disk size of the key as the number of attributes is increasing

Encryption & Decryption The last part of the ABE experiments, was to measure both the encryption and decryption time. Recall that our protocol is only using ABE to encrypt a symmetric key and not other files that are stored in the cloud. Thus, we only ran experiments where we were encrypting a symmetric key with different policies and decrypting the generated ciphertext with keys that are associated with a different number of attributes. Moreover, we used access policies of different structure in order to measure the performance of the decryption not only when all conditions need to be fulfilled (most demanding case) but also when a random number of attributes is needed to satisfy the underlying policy.

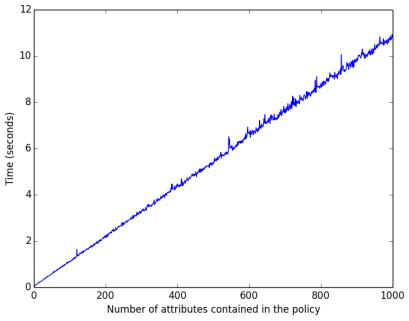

Fig. 4: Encryption

During our first experiment we used access policies of the type '`Attribute_1 AND Attribute_2 AND ...AND Attribute_n`' as in [15] and [6]. This policy is considered as the most demanding case since all n attributes are required for successful decryption. We say that such a policy is of size n. Figure 4 shows the encryption time needed to encrypt a file with a policy of size up to 1000.

Figure 5, shows the time needed to decrypt that file by using a key with up to 1000 number of attributes where all were required to satisfy the policy. As can be seen from Figure **??** the time to encrypt and decrypt a file also depends on the particular attributes available and the size of the policy. More precisely, the time needed to encrypt a file with a policy of size 1000 is around 11 sec while the time needed to decrypt the same file is 4.5 sec. However, having such a long attribute list is not considered as a realistic scenario. Hence, the results show that using the proposed scheme to encrypt and decrypt a file of small size (in our case only a symmetric key) does not put any real burden to the overall performance of the protocol.

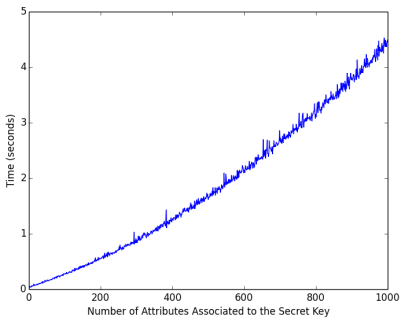

Fig. 5: Decryption

The second experiment of this phase aimed at analyzing the behavior of the underlying CP-ABE scheme under a randomly generated policy that contained a mix of regular and numerical attributes as well as conditions like the following: `2 of (age > 35, Attribute_x, sysadmin)`. This condition, requires that at least two of the attributes in the parenthesis will be satisfied by the attributes associated with the key of the user that is trying to decrypt a file. Figure 6 shows the time needed to decrypt a file with a policy size up to 1000 but generated in a random way (i.e. not all attributes were needed to satisfy the policy). As it is evident from the graph, the decryption time varies a lot and it is not linear. This was to be expected since in most cases not all attributes of a key would be needed to satisfy the policy associated with the ciphertext. Additionally, there were many cases where a ciphertext with a policy size i needed less time for decryption than a ciphertext with policy size j, where $i > j$. This was something that we were expecting since in contrast to the previous case where all attributes of the key were needed to satisfy the policy, here only a subset of the total attributes was required. Thus, during the decryption process the policy was satisfied earlier. Finally, from this experiment we can see that on average, decryption time is much less compared to the previous case where all attributes needed to satisfy

the policy. Hence, we can safely conclude that the decryption time even when large policies are used maintains reasonable running times.

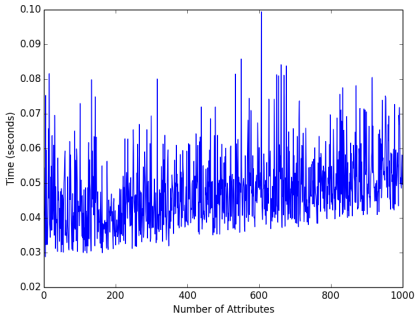

Fig. 6: Decryption of ciphertext associated with a random policy

As a conclusion, it is clear that both encryption and decryption run in a predictable amount of time based on the number of attributes in a key or leaves in a policy tree. Moreover, the performance of these functions depends on the specific policy of the ciphertext as well as on the attributes available in user's private key. Furthermore, the overall performance of these operations can be improved by optimizing the way we generate policies. However, we leave this as a future study. Most importantly, from these experiments it is evident that the overhead remains at acceptable levels while at the same time large private keys and policies are possible in practice while maintaining reasonable running times.

7.2 Symmetric Searchable Encryption

For the needs of our protocol, we also implemented a Searchable Encryption Scheme on top of a Blind Storage System as proposed in [19]. This part of the experiments was implemented in standard C++ using the Boost [2], Crypto++ [3] and CurlPP [7,4] libraries. Moreover, to test the overall performance of the underlying SSE scheme, we used files of different size and structure. To this end, we selected random data from the Enron dataset [5]. Our experiments on the SSE scheme were focused on two main aspects: *(1)* Indexing and uploading files and *(2)* Searching for a specific keyword.

Index & Upload The Enron test set consists of a large number of ASCII files with each file only consisting of a few bytes but each file occupies an integral multiple of the block size. In our experiments, we indexed and uploaded files of a total size up to 4 MB. This, resulted to a set of almost 1400 files from the selected dataset. The *total time* needed for both indexing and uploading was 1054 sec. However, this result corresponds to a rather large number of files.

Therefore, we also ran the same experiment with 250 files resulting in a net total size of approximately 1 MB. In that case, our experiment indicated a the total execution time of 350 sec.

Keyword Search During this phase, we measured the execution time for finding all files that contained the word "the" in our dataset. Searching for a specific keyword is merely comparable with downloading the corresponding index file. We ran the search algorithm for the files that we uploaded earlier. The results, indicated that the total execution time for finding all the files that contained the specified keyword was less than 0.6 sec.

8 Conclusion

In this paper, we proposed LotS – a protocol for secure and privacy-preserving data sharing in a cloud environment. LotS is based on Ciphertext-Policy Attribute-Encryption and Symmetric Searchable Encryption. The described protocol used the advantages of both techniques and can be seen as an independent contribution to the field of *hybrid encryption*. The offered solution allows cloud users to encrypt files based on an SSE scheme while using a CP-ABE scheme to control their access rights and securely remove access to a file without having to decrypt and re-encrypt the original data. Finally, we believe that this work can pave the way for privacy-preserving data sharing between different organizations that are using separate and completely distinct cloud platforms. Hence, one of our main future goals is to test LotS in a multi-cloud environment. We hope that this has the potential to solve important problems, such as how a patient that is for example travelling and is in a critical health condition, can share her medical history with (authorized) doctors of a different country.

References

1. In: Financial Cryptography and Data Security, Lecture Notes in Computer Science, vol. 6054 (2010)
2. Boost c++ libraries (8 2014), http://www.boost.org/
3. Crypto++ library (11 2015), https://www.cryptopp.com/
4. curl and libcurl (12 2015), http://curl.haxx.se/
5. Enron email dataset (5 2015), https://www.cs.cmu.edu/~enron/
6. Agrawal, S., Chase, M.: Fame: Fast attribute-based message encryption. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 665–682. CCS '17, ACM, New York, NY, USA (2017). https://doi.org/10.1145/3133956.3134014, http://doi.acm.org/10.1145/3133956.3134014
7. Barrette-LaPierre, J.P.: curlpp (5 2015), http://www.curlpp.org/
8. Bethencourt, J., Sahai, A., Waters, B.: Ciphertext-policy attribute-based encryption. In: Proceedings of the 2007 IEEE Symposium on Security and Privacy. pp. 321–334. SP '07, IEEE Computer Society, Washington, DC, USA (2007). https://doi.org/10.1109/SP.2007.11, http://dx.doi.org/10.1109/SP.2007.11

9. Boldyreva, A., Goyal, V., Kumar, V.: Identity-based encryption with efficient revocation. In: Proceedings of the 15th ACM Conference on Computer and Communications Security. pp. 417–426. CCS '08, ACM, New York, NY, USA (2008). https://doi.org/10.1145/1455770.1455823, http://doi.acm.org/10.1145/1455770.1455823
10. Chase, M.: Multi-authority attribute based encryption. In: Proceedings of the 4th Conference on Theory of Cryptography. pp. 515–534. TCC'07, Springer-Verlag, Berlin, Heidelberg (2007), http://dl.acm.org/citation.cfm?id=1760749.1760787
11. Coker, G., Guttman, J., Loscocco, P., Herzog, A., Millen, J., O’Hanlon, B., Ramsdell, J., Segall, A., Sheehy, J., Sniffen, B.: Principles of remote attestation. Int. J. Inf. Secur. **10**(2), 63–81 (Jun 2011). https://doi.org/10.1007/s10207-011-0124-7, http://dx.doi.org/10.1007/s10207-011-0124-7
12. Costan, V., Devadas, S.: Intel sgx explained. Cryptology ePrint Archive, Report 2016/086 (2016), https://eprint.iacr.org/2016/086
13. Dolev, D., Yao, A.C.: On the security of public key protocols. Information Theory, IEEE Transactions on **29**(2) (1983)
14. Fisch, B., Vinayagamurthy, D., Boneh, D., Gorbunov, S.: Iron: Functional encryption using intel sgx. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 765–782. CCS '17, ACM, New York, NY, USA (2017). https://doi.org/10.1145/3133956.3134106, http://doi.acm.org/10.1145/3133956.3134106
15. Green, M., Hohenberger, S., Waters, B.: Outsourcing the decryption of abe ciphertexts. In: Proceedings of the 20th USENIX Conference on Security. pp. 34–34. SEC'11 (2011)
16. Kamara, S., Papamanthou, C., Roeder, T.: Dynamic searchable symmetric encryption. pp. 965–976 (2012)
17. Michalas, A., Dowsley, R.: Towards trusted ehealth services in the cloud. In: 2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC). pp. 618–623 (Dec 2015). https://doi.org/10.1109/UCC.2015.108
18. Michalas, A.: Sharing in the rain: Secure and efficient data sharing for the cloud. In: Proceedings of the 11th IEEE International Conference for Internet Technology and Secured Transactions (ICITST-2016). IEEE (2016)
19. Naveed, M., Prabhakaran, M., Gunter, C.A.: Dynamic searchable encryption via blind storage. pp. 639–654 (2014). https://doi.org/10.1109/SP.2014.47
20. Paladi, N., Gehrmann, C., Michalas, A.: Providing user security guarantees in public infrastructure clouds. IEEE Transactions on Cloud Computing **5**(3), 405–419 (July 2017). https://doi.org/10.1109/TCC.2016.2525991
21. Paladi, N., Michalas, A., Gehrmann, C.: Domain based storage protection with secure access control for the cloud. In: Proceedings of the 2014 International Workshop on Security in Cloud Computing. ASIACCS '14, ACM, New York, NY, USA (2014)
22. Sahai, A., Seyalioglu, H., Waters, B.: Dynamic credentials and ciphertext delegation for attribute-based encryption. In: Proceedings of the 32Nd Annual Cryptology Conference on Advances in Cryptology — CRYPTO 2012 - Volume 7417. pp. 199–217 (2012)
23. Sahai, A., Waters, B.: Fuzzy identity-based encryption. In: Proceedings of the 24th Annual International Conference on Theory and Applications of Cryptographic Techniques. pp. 457–473. EUROCRYPT'05 (2005)

24. Santos, N., Rodrigues, R., Gummadi, K.P., Saroiu, S.: Policy-Sealed Data: A New Abstraction for Building Trusted Cloud Services. In: Presented as part of the 21st USENIX Security Symposium (USENIX Security 12). pp. 175–188. USENIX, Bellevue, WA (2012)
25. Verginadis, Y., Michalas, A., Gouvas, P., Schiefer, G., Hübsch, G., Paraskakis, I.: Paasword: A holistic data privacy and security by design framework for cloud services. In: Proceedings of the 5th International Conference on Cloud Computing and Services Science. pp. 206–213 (2015). https://doi.org/10.5220/0005489302060213
26. Yang, K., Jia, X., Ren, K., Xie, R., Huang, L.: Enabling efficient access control with dynamic policy updating for big data in the cloud. In: IEEE INFOCOM 2014 - IEEE Conference on Computer Communications. pp. 2013–2021 (April 2014). https://doi.org/10.1109/INFOCOM.2014.6848142