

Received October 29, 2020, accepted November 12, 2020, date of publication November 17, 2020, date of current version December 7, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3038838

The Cloud we Share: Access Control on Symmetrically Encrypted Data in Untrusted Clouds

ALEXANDROS BAKAS¹, HAI-VAN DANG², ANTONIS MICHALAS¹, AND ALEXANDR ZALITKO¹

¹Department of Computing Sciences, Tampere University, 33720 Tampere, Finland

²School of Computer Science and Engineering, University of Westminster, London W1W 7BY, U.K.

Corresponding author: Alexandros Bakas (alexandros.bakas@tuni.fi)

This work was supported by the ASCLEPIOS: Advanced Secure Cloud Encrypted Platform for Internationally Orchestrated Solutions in Healthcare Project No. 826093, European Union research project.


ABSTRACT Along with the rapid growth of cloud environments, rises the problem of secure data storage—a problem that both businesses and end-users take into consideration before moving their data online. Recently, a lot of solutions have been proposed based either on Symmetric Searchable Encryption (SSE) or Attribute-Based Encryption (ABE). SSE is an encryption technique that offers security against both *internal* and *external* attacks. However, since in an SSE scheme, a single key is used to encrypt everything, revoking a user would imply downloading the entire encrypted database and re-encrypt it with a fresh key. On the other hand, in an ABE scheme, the problem of revocation can be addressed. Unfortunately, though, the proposed solutions are based on the properties of the underlying ABE scheme and hence, the revocation costs grow along with the complexity of the policies. To this end, we use these two cryptographic techniques that squarely fit cloud-based environments to design a hybrid encryption scheme based on ABE and SSE in such a way that we utilize the best out of both of them. Moreover, we exploit the functionalities offered by Intel's SGX to design a revocation mechanism and an access control one, that are agnostic to the cryptographic primitives used in our construction.

INDEX TERMS Access control, attribute-based encryption, cloud, data sharing, scope, secure storage, SGX, symmetric searchable encryption.

I. INTRODUCTION

Over the past few years, cloud computing has grown to an extent that affects the day-to-day life of almost everyone. From big corporations to casual internet users the cloud has become an integral part of our lives. However, many users still feel reluctant about outsourcing their personal files since cloud services are hosted and run by third untrusted parties and thus, the files are vulnerable to internal attacks. To this end, both key industrial players as well as researchers have turned for solutions to the promising technique of Symmetric Searchable Encryption [8], [9] and to the well-studied field of Attribute-Based Encryption [10].

In an SSE scheme, users encrypt their files locally before outsourcing them to the Cloud Service Provider (CSP). Thus, the CSP who does not possess the encryption key cannot

The associate editor coordinating the review of this manuscript and approving it for publication was Muhamamd Aleem .

extract any valuable information about the users' data. However, the most fascinating property about SSE, is that it allows users to search directly on their encrypted data for those that contain specific keywords. Unfortunately, SSE schemes do not support the revocation of users – a problem of paramount importance in cloud-based environments. Hence, revoking a user is equivalent to downloading the entire database and re-encrypt it with a fresh key.

Another technique that fits cloud-based environments is ABE. In ABE schemes, all files are encrypted using a master public key, but in contrast to traditional public key cryptosystems, the resulted ciphertext is bound by a policy. Moreover, each user has a unique secret key associated with the user's attributes (e.g. id, age, organization, etc.). Thus, decrypting a file is possible if and only if the user's attributes satisfy the policy bound to the ciphertext. However, using an asymmetric encryption scheme to encrypt large volumes of data, is rather inefficient.

Contribution: We propose a revocable hybrid encryption scheme combining ABE and SSE. In our construction, the ABE scheme is used as a tool that allows efficient sharing of the SSE key between legitimate users. Having identified both the advantages and disadvantages of SSE and ABE, we propose a solution that uses the best out of both techniques. More specifically, the only access control in searchable encryption occurs only in those schemes that are set in the public-key setting [13]. To solve this, we encrypt the SSE symmetric key, using ABE. Hence, the key can be decrypted if and only if the decryptor satisfies the policy specified by the ABE policy. Moreover, as pointed out in [10], key revocation in ABE schemes can be tricky and inefficient. To overcome this problem, we design a revocation mechanism that is solely based on the functionalities offered by Intel SGX and is agnostic to the aforementioned cryptographic primitives. Our design is accompanied by a detailed and to the depth security analysis, where we prove the security of our construction against various attacks, where an adversary targets different components of our architecture. Finally, we provide a theoretical evaluation of the SSE scheme used in this construction and an extensive experimental control with very promising results.

Organization: The rest of the paper is organized as follows: In Section 2, we describe important works that address the problem of secure cloud storage. In Section 3, we present a detailed description of the system model, while in Section 4, we provide formal definitions of the cryptographic primitives used in our protocol. In Section 5, we present a formal construction of our scheme, followed by its security analysis in Section 6. Section 7 consists of our experimental results and finally, Section 8 concludes the paper.

II. RELATED WORK

In [21] authors present HardIDX, a scheme that supports range queries by utilizing the functionality offered by SGX. Their construction minimizes the leakage by hiding the search pattern but the proposed scheme is static, thus file additions and deletions are not supported. A dynamic SSE with stronger security guarantees is presented in [14], where the authors presented Sophos. Sophos is a forward private SSE scheme in the sense that newly added keywords cannot be linked to previous queries. A more efficient forward private scheme was presented in [19] where the authors improved the search time by presenting a parallelizable scheme. However, all the aforementioned schemes only support the single-client model where the only occurring communication is between a data owner and a cloud service provider. For this work, since we are interested in a multi-client scheme, we chose the scheme we designed and presented in [9], which is an extension of [19], in the multi-client model.

A promising scheme is presented in [20] where authors present IRON, a functional encryption scheme based on SGX. IRON's main functionalities (such as decryption of a file and application of a function on the decrypted file) are executed in

the isolated environment offered by SGX. In our construction, we use the same hardware principles to design our revocable hybrid encryption scheme and we further exploit SGX by designing a revocation mechanism that is solely based on SGX enclaves.

In [28], authors try to tackle the problem of storing data on untrusted clouds, by designing a revocable hybrid encryption scheme, enhanced with a key rotation mechanism to avoid key scraping attacks. Authors use an All-or-Nothing-Transformation (AONT) [16] to prevent revoked users from accessing the stored data. In particular, they use Optimal Asymmetric Encryption Padding (OAEP) as the AONT, since reversing OAEP requires the entire output to be known. Thus, by changing random bits, reversing OAEP becomes infeasible. Naturally, to decrypt a file, the changed bits need to be stored, so that the AONT could be later reversed. However, this implies that with each re-encryption, the size of the ciphertexts grows and, as a result, decrypting a file that has been re-encrypted multiple times, becomes an expensive operation. Moreover, to make the scheme more efficient, the authors suggest that the AONT could be applied by the online storage server. However, this implies the existence of a fully trusted server and hence, the scheme can be vulnerable to internal attacks.

A revocable Ciphertext-Policy Attribute-based Encryption (CP-ABE) presented in [23] proposes to embed the revocation list in the ciphertexts. However, this embedding results in bigger ciphertexts, deeming the decryption and file modification operations much more demanding. To overcome this problem, authors in [12] propose a method based on Hierarchical Identity Based Encryption (HIBE). In their construction, the users' secret keys, expire after a specified period of time. Thus, the revocation list only contains the keys revoked before the expiration time. Similarly, in [11] authors constructed a Key-Policy ABE (KP-ABE) by extending their work on Revocable Identity Based Encryption. In their design, the revocation of users relies on frequent key updates for all the different attributes; hence, their solution does not scale well for practical usage.

Another promising technique is presented in [30], where authors propose a Traceable CP-ABE scheme that supports the revocation of malicious users. In particular, in their construction, they design a mechanism that can trace users that have leaked information about the key from the system. However, on each revocation, a new group key, for a group of users, needs to be generated and then distributed to all eligible users. Moreover, authors place sensitive operations such as the re-encryption and partial decryption of ciphertexts, in untrusted entities. In our construction, even if a malicious adversary cannot be traced, we ensure that no adversary will be able to tamper with a user's access rights to bypass the system's authentication. Moreover, in our scheme, even if we make use of the trusted execution environment offered by Intel's SGX, all sensitive operations occur on the user's side to minimize the leakage from side-channel attacks.

This work is an extension of [7], [24], [26], [27] where authors presented a hybrid encryption scheme based on ABE and SSE. The constructions in [7], [24], lacked a proper implementation as well as an access control mechanism like the one we introduce in this paper. Apart from that, our work differentiates from the schemes presented in [26], [27] since we extend the underlying access control mechanism while at the same time we enable users to *search over multiple datasets in one round*. Moreover, in contrast to all previous works, the entire protocol is redesigned to support symmetric key cryptography instead of asymmetric. Hence, resulting in a much more efficient approach. In addition to that, we used a far more modern SSE scheme, presented in [9] which supports forward privacy [15]. On top of this, to further examine the efficiency of our construction, we used a more efficient CP-ABE scheme presented in [5]. This way we could evaluate the performance of our construction even under the most demanding policies.

III. ARCHITECTURE

The underlying architecture consists of different components which are described as below.

Cloud Service Provider (CSP)

We consider a cloud computing environment similar to the one described in [29]. We assume that the CSP is SGX-enabled, and that core entities will be running in the trusted execution environment offered by SGX.

Master Authority (MS)

Similarly to CSP, MS is SGX-enabled and running in an enclave called the Master Enclave. MS enclave generates and distributes ABE keys to registered users.

Key Tray (KT)

KT is also SGX-enabled and running in an enclave called the KT Enclave. KT enclave is responsible for storing the ciphertexts of the symmetric keys generated by data owners. Such symmetric keys are needed to decrypt the data.

Revocation Authority (REV)

Similarly, REV is also SGX-enabled and running in an enclave called the REV Enclave. REV is responsible for maintaining the valid scopes of users.

User (u_i)

In our scenario, a user interacts with the CSP to manage certain files that has access to. A user can (1) store data in the cloud and (2) share data with other users. A user is referred to as a data owner when she is storing data in the cloud.

Moreover, we assume the existence of a registration authority that is responsible for the registration of users. However, registration is out of the scope of this paper and we assume that all users have been already registered.

SGX: We briefly present the main SGX functionalities which are used for our construction. Further detailed description can be found in [17], [20].

Isolation: Enclaves are located in a hardware guarded area of memory of 128MB in which only 90MB can be used by the software. The processor tracks which parts of memory belong to which enclave, and ensures that *only* enclaves can access their own memory.

Attestation: SGX supports attestation between enclaves of the same (local attestation) and different platforms (remote attestation). In local attestation, an enclave enc_i can verify another enclave enc_j and the program/software running in the latter through a report generated by enc_j . The report contains information about the enclave and the program running in it and is signed with a secret key SK_{rpt} . This key is the same for all enclaves of the same platform. In the case of remote attestation, the verification is performed through a report signed with a special private key provided by Intel. Therefore, it requires contacting Intel's Attestation Server.

Sealing: As being stored in untrusted memory, data is encrypted with a Root Seal Key provided with every SGX processor. The sealed data can be recovered even after an enclave is destroyed and rebooted on the same platform.

IV. BACKGROUND

A. NOTATION

The set of all users is $\mathcal{U} = \{u_1, \dots, u_n\}$. The public/private key pair of a user u_i is denoted by (pk_i, sk_i) and the signature of u_i on a message m is $\sigma_i(m)$. The Symmetric key of u_i is denoted by K_i . The access rights of a u_i are denoted by a list of valid scopes \mathcal{SC}_i such that $\mathcal{SC}_i = \{(j, s_j^i), (z, s_z^i), \dots, (k, s_k^i)\}$, in which j, k, \dots, z represent a collection of files encrypted under the symmetric keys K_j, K_k, \dots, K_z and s_j^i is a one dimensional bit array of length five that represents the scopes (i.e. view, add, delete, manager, owner) assigned to u_i . For instance, in case u_i has access rights *view* and *delete* for data encrypted under the symmetric key K_j , then $s_j^i = [10100]$. The output y of an algorithm A is denoted by $y \leftarrow A$ if A is probabilistic, and by $A \rightarrow y$ if A is deterministic. A function $negl(\cdot)$ is called negligible, if $\forall n > 0, \exists N_n$ such that $\forall x > N_n: |negl(x)| < 1/poly(x)$. A *probabilistic polynomial time* (PPT) adversary \mathcal{ADV} is a randomized algorithm for which there exists a polynomial $poly(\cdot)$ such that for all input x , the running time of $\mathcal{ADV}(x)$ is bounded by $poly(|x|)$.

B. CRYPTOGRAPHIC PRIMITIVES

We now present the cryptographic primitives used in our construction. As already mentioned, we make use of an ABE scheme and a dynamic SSE scheme. We now proceed with the corresponding definitions as described in [10] and [9] respectively.

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

- **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 $(\text{MPK}, \text{MSK}) \leftarrow \text{Setup}(1^\lambda)$.

- **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 bound both to the corresponding list of attributes and the user. We denote this by $(\text{sk}_{\mathcal{A}, u_i}) \leftarrow \text{Gen}(\text{MSK}, \mathcal{A}, u_i)$.
- **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 $c_P \leftarrow \text{Enc}(\text{MPK}, m, P)$.
- **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 $\text{Dec}(\text{sk}_{\mathcal{A}, u_i}, c_P) \rightarrow m$.

Definition 2 (Dynamic Index-based SSE (DSSE)): A dynamic index-based symmetric searchable encryption scheme is a tuple of five polynomial algorithms

$\text{DSSE} = (\text{KeyGen}, \text{InGen}, \text{AddFile}, \text{Search}, \text{Delete})$ such that:

- **DSSE.KeyGen** is probabilistic key-generation algorithm that takes as input a security parameter λ and outputs a secret key \mathbf{K} . It is used by the client to generate her secret key.
- **DSSE.InGen** is a probabilistic algorithm that takes as input a secret key \mathbf{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.
- **DSSE.AddFile** is a probabilistic algorithm that takes as input a secret key \mathbf{K} and a file f and outputs an add token $\tau_\alpha(f)$ and a ciphertext c_f . The token and the ciphertext are then sent to the storage server, where c_f will be added to the collection of ciphertexts and the index γ will be updated accordingly.
- **DSSE.Search** is a deterministic algorithm that takes as input a secret key \mathbf{K} and a keyword w and outputs a search token $\tau_s(w)$. The token is then sent to the storage server who will output a sequence of file identifiers $\mathbf{I}_w \subset \mathbf{c}$.
- **DSSE.Delete** is a deterministic algorithm that takes as input a secret key \mathbf{K} and a file identifier $\text{id}(f)$ and outputs a delete token $\tau_d(f)$ for f . The token will be sent to the storage server, who will delete c_f and update the index γ accordingly.

The security of a DSSE scheme is based on the existence of a simulator that is given as input information leaked during the execution of the protocol. In particular to define the security of SSE we make use of the leakage functions $\mathcal{L}_{in}, \mathcal{L}_s, \mathcal{L}_a, \mathcal{L}_d$ associated to index creation, search, add and delete operations [9].

Definition 3 (Dynamic CKA 2-Security): Let

$\text{DSSE} = (\text{KeyGen}, \text{InGen}, \text{AddFile}, \text{Search}, \text{Delete})$ be a dynamic index based symmetric searchable encryption scheme and $\mathcal{L}_{in}, \mathcal{L}_s, \mathcal{L}_a, \mathcal{L}_d$ be leakage functions associated to index creation, search, add and delete operations. We consider the following experiments between an adversary \mathcal{ADV} and a challenger \mathcal{C} :

Real $_{\mathcal{ADV}}(\lambda)$

\mathcal{C} runs $\text{Gen}(1^\lambda)$ to generate a key \mathbf{K} . \mathcal{ADV} outputs a file \mathbf{f} and receives $(\gamma, \mathbf{c}) \leftarrow \text{Enc}(\mathbf{K}, \mathbf{f})$ from \mathcal{C} . \mathcal{ADV} makes a polynomial time of adaptive queries $q = \{w, f_1, f_2\}$ and for each q he receives back either a search token for w , $\tau_s(w)$, an add token and a ciphertext for f_1 , $(\tau_\alpha(f_1), c_1)$ or a delete token for f_2 , $\tau_d(f_2)$. Finally, \mathcal{ADV} outputs a bit b .

Ideal $_{\mathcal{ADV}, \mathcal{S}}(\lambda)$

\mathcal{ADV} outputs a file \mathbf{f} . \mathcal{S} is given \mathcal{L}_{in} and generates (γ, \mathbf{c}) which is sent back to \mathcal{ADV} . \mathcal{ADV} makes a polynomial time of adaptive queries $q = \{w, f_1, f_2\}$ and for each q , \mathcal{S} is given either $\mathcal{L}_s(\mathbf{f}, w)$, $\mathcal{L}_a(\mathbf{f}, f_1)$ or $\mathcal{L}_d(\mathbf{f}, f_2)$. \mathcal{S} then returns a token and, in the case of addition, a ciphertext c . Finally, \mathcal{ADV} outputs a bit b .

We say that the SSE scheme is \mathcal{L} -i secure if for all probabilistic polynomial adversaries \mathcal{ADV} , there exists a probabilistic simulator \mathcal{S} such that:

$$|\Pr[(\text{Real}) = 1] - \Pr[(\text{Ideal}) = 1]| \leq \text{negl}(\lambda)$$

In the cases of file addition and deletion, the simulator must also generate ciphertexts and update the current indexes.

In addition to ABE and SSE, we rely on SGX functionalities to attest among the components.

During the execution of the protocol, all parties have access to the secure hardware as defined in [20]. In the beginning, **HW.Setup** runs to produce the secret key needed to verify reports. Each enclave is then initialized by loading a program P and producing a handle hdl which is used as an identification for the enclave running P . This is done by running the **HW.Load** interface. After the initialization of the enclave, **HW.Run** is executed with different inputs. For simplicity, we assume that all enclaves run on the same host, so they only perform local attestations with each other. To do so, an enclave (enc_i) first runs **HW.RunReport** which produces a report (rpt_i) that is sent to enc_j . Upon reception, enc_j executes **HW.ReportVerify** and verifies the validity of rpt_i . A more detailed description of the hardware algorithms used by the enclaves is given below:

- **HW.Load(Q)**: Takes as input a program Q . An enclave enc_i is created in which Q will be loaded. Moreover a handle hdl_{enc} is created that will be used as an identifier for the enclave.
- **HW.Run(hdl, in)**: Takes as input a handle hdl and some input in . It runs the program in the enclave specified by hdl with in as input.
- **HW.RunReport(hdl, in)**: Takes as input a handle hdl and some input in . It will output a report that is verifiable by any other enclave on the same platform.

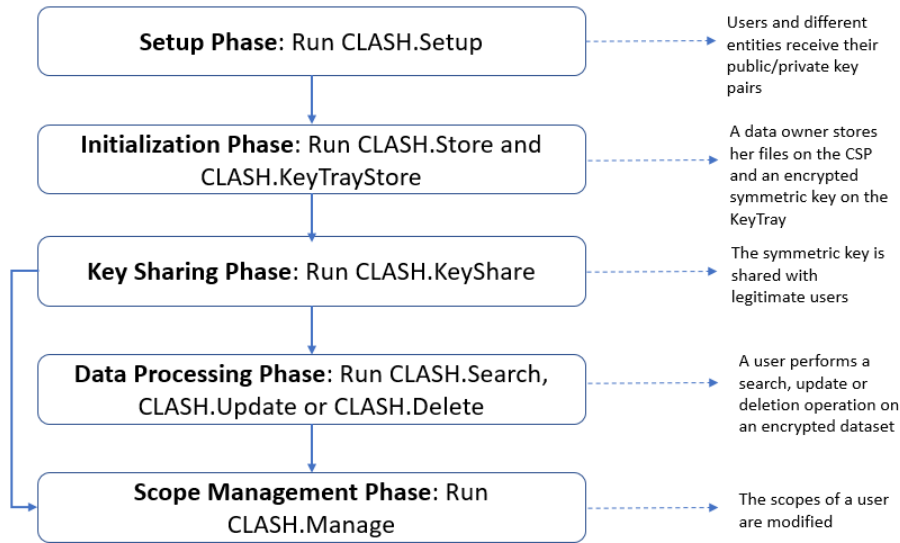


FIGURE 1. High-Level Approach.

The report contains information about the underlying enclave signed with sk_{rpt} .

- **HW.ReportVerify(hdl', rpt)**: Takes as input a handle hdl' and a report rpt . Uses sk_{rpt} generated by HW.Setup to verify the MAC of the report.

V. THE CLOUD WE SHARE (CLASH)

In this section, we present The Cloud we Share (CLASH) – the core of this paper’s contribution.

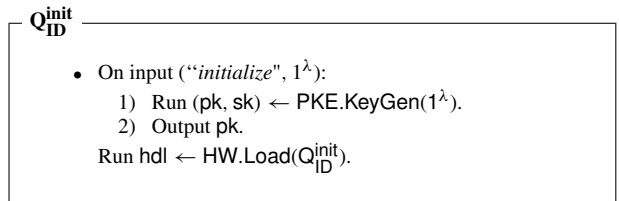
A. HIGH-LEVEL OVERVIEW

Before we proceed to the formal description of our construction, we present a high-level overview. CLASH is divided in a *Setup* phase and four main phases: *Initialization*, *Key Sharing*, *Data Processing* and *Scope Management*. In the Setup phase, all entities receive a public/private key pair that will be used to establish secure communication channels. During the Initialization phase, a data owner encrypts her data using the SSE scheme, uploads the encrypted files to the CSP, and encrypts the SSE key using an ABE key. The ciphertext of the key is bound by a policy specified by the user and it is stored on the Key Tray. In the Key Sharing phase, different users contact the Key Tray and request for the ciphertext of the symmetric key. Upon receiving the ciphertext, they can decrypt it if and only if their attributes satisfy the policy bound to the key. If the decryption of the key is successful, then the Data Processing phase commences where the users can search for different files, add new ones or delete existing ones, according to their access right (scopes). Finally, in the Scope Management phase, a data owner can modify the scopes of the users and even fully revoke their right to access the encrypted dataset. This high-level approach of our construction is depicted in Figure 1.

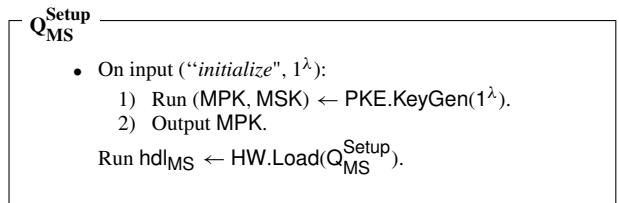
B. FORMAL CONSTRUCTION

Setup Phase: In the Setup phase each enclave is initialized and generates a public/private key pair (pk, sk) for a CCA2 secure public key cryptosystem and signing/verification key pair for a EUF-CMA secure signature scheme. An enclave is initialized as follows:

$CLASH.Setup(“initialize”, 1^\lambda)$: Each enclave is initialized by loading the program Q_{ID}^{init} :



Moreover, the MS enclave loads a program Q_{MS}^{Setup} that outputs the master public/private key pair (MPK, MSK).



Finally, MS is responsible for generating secret ABE keys for registered users. To do so, MS retrieves MSK and a list of attributes \mathcal{A} associated with each user.

$CLASH.ABEUserKey(“KeyRequest”, MSK, u, \mathcal{A}, 1^\lambda)$: The program Q_{MS}^{skKey} , which is responsible for generating secret ABE keys to registered users, is defined as follows:

Finally, the different entities, including the users, use their public key pairs to establish secure channels between them. This way, all the exchanged messages will be encrypted

symmetrically. Clearly, all the symmetric keys will be different. For example, $K_{i,REV}$ denotes the symmetric key shared between the user u_i and the REV enclave.

Q_{MS}^{sKey}
 On input ("KeyRequest", MSK, u , \mathcal{A} , 1^λ):
 1) Verify that u is registered. If not, output \perp .
 2) Run $sk_{\mathcal{A},u} \leftarrow \text{CPABE.Gen}(\text{MSK}, \mathcal{A}, u)$.
 3) Compute and output $c = \text{PKE.Enc}(pk_i, sk_{\mathcal{A},u})$.
 Run $c \leftarrow \text{HW.Run}(\text{hdl}_{\text{MS}}, (\text{"KeyRequest"}, \text{MSK}, u, \mathcal{A}))$.

Initialization Phase: During the initialization phase, the data owner, u_i stores her encrypted data on the cloud and stores the secret key K_i to the KT so that it can be shared with other users. To do so, she first runs CLASH.Store and then CLASH.KeyTrayStore.

CLASH.Store("store", $cred_i$): Assuming that all the enclaves are already initialized and that all registered users have received their secret ABE keys, a data owner, u_i , can start interacting with the CSP to store her files. To this end she contacts CSP by sending $m_{req} = \langle r_1, \text{E}(K_{i,\text{CSP}}, cred_i), \text{StoreReq}, \text{HMAC}(K_{i,\text{CSP}}, r_1 || cred_i || \text{StoreReq}) \rangle$ where r_1 is a random number generated by u_i . Upon reception, CSP verifies u_i as a registered user and sends $m_{ver} = \langle r_2, \text{E}(K_{i,\text{CSP}}, Auth), \text{HMAC}(K_{i,\text{CSP}}, r_2 || u_i || Auth) \rangle$ to u_i . After u_i gets the authorization message from the CSP, she generates a DSSE key K_i and its unique index idx_{K_i} , encrypts her files, f_i , with the key and sends them to the CSP via $m_{store} = \langle r_3, \text{E}(K_{i,\text{CSP}}, idx_{K_i}), \gamma_i, c_i, \text{HMAC}(K_{i,\text{CSP}}, r_3 || \gamma_i || c_i || idx_{K_i}) \rangle$ where γ_i is the encrypted DSSE index. CSP will finally store $\{idx_{K_i}, \gamma_i, c_i\}$.

Q_{CSP}^{Store}
 • On input ("StoreReq", m_{req}):
 1) Open m_{req} ; verify the message^a; if the verification fails, output \perp .
 2) Compute and output m_{ver} .
 Run $m_{ver} \leftarrow \text{HW.Run}(\text{hdl}_{\text{CSP}}, (\text{"StoreReq"}, m_{req}))$.
 • On input ("store", m_{store}):
 1) Open m_{store} ; verify the message; if the verification fails, output \perp .
 2) Store $(idx_{K_i}, c_i, \gamma_i)$.
 Run $\text{HW.Run}(\text{hdl}_{\text{CSP}}, (\text{"store"}, m_{store}))$.
^aBy this, we mean that the entity receiving the message verifies the freshness and the integrity of the message and it can also authenticate the sender.

CLASH.KeyTrayStore("store", K_i , p): To enable efficient sharing of K_i between registered users, the data owner u_i encrypts K_i under the ABE master public key MPK and binds it with a policy P , resulting to a ciphertext $c_p^{K_i}$ of K_i . As a next step, u_i sends $c_p^{K_i}$ to KT via $m_{keystore} = \langle r_4, \text{E}(K_{i,\text{KT}}, u_i || idx_{K_i}), c_p^{K_i}, \text{HMAC}(K_{i,\text{KT}}, r_4 || u_i || c_p^{K_i} || idx_{K_i}) \rangle$. Upon reception, KT runs **Q_{KT}^{Store}** to store $(u_i, c_p^{K_i}, idx_{K_i})$. Finally, u_i sends the list of valid scopes, L_{VS} for every registered user to REV via $m_{scope} = \langle r_5, \text{E}(K_{i,\text{REV}}, (idx_{K_i} || u_1 || s_1^i || u_2 || s_2^i || \dots)), \text{HMAC}(K_{i,\text{REV}}, r_5 || idx_{K_i} || u_1 || s_1^i || u_2 || s_2^i || \dots) \rangle$ to REV, where

$\{u_1, u_2, \dots\}$ are identifiers of the registered users, and $\{s_1^i, s_2^i, \dots\}$ are arrays specifying each user's access rights. The programs **Q_{KT}^{Store}** and **Q_{REV}^{Scope}** are defined as follows:

Q_{KT}^{Store}
 • On input ("store", $m_{keystore}$):
 1) Open $m_{keystore}$; verify the message. If the verification fails, output \perp .
 2) Store $(u_i, c_p^{K_i}, idx_{K_i})$.
 Run $(u_i, c_p^{K_i}, idx_{K_i}) \leftarrow \text{HW.Run}(\text{hdl}_{\text{KT}}, (\text{"store"}, m_{keystore}))$.

Q_{REV}^{Scope}
 • On input ("scope", m_{scope}):
 1) Open m_{scope} ; verify the message. If the verification fails, output \perp .
 2) Store $(idx_{K_i}, \{(u_1, s_1^i), (u_2, s_2^i), \dots\})$ into the list of valid scopes L_{VS} .
 Run $(idx_{K_i}, \{(u_1, s_1^i), (u_2, s_2^i), \dots\}) \leftarrow \text{HW.Run}(\text{hdl}_{\text{REV}}, (\text{"scope"}, m_{scope}))$.

Key Sharing Phase: The goal of this phase is to share data between legitimate users. This is done by running CLASH.KeyShare. For a registered user u_j to access files encrypted by u_i , she first needs to acquire the symmetric key K_i . With K_i in her possession, u_j will be able to both generate the DSSE tokens required to access the encrypted database and to decrypt the files she receives back from the CSP. To this end, u_j sends a request to KT via $m_{verReq} = \langle r_6, \text{E}(K_{j,\text{KT}}, u_j || u_i), \text{HMAC}(K_{j,\text{KT}}, r_6 || u_j || u_i) \rangle$. Upon receiving the message, the Key Tray will reply with $m_{idxkey} = \langle r_7, \text{E}(K_{\text{KT},\text{REV}}, u_j || idx_{K_i}), c_p^{K_i}, \text{HMAC}(K_{\text{KT},\text{REV}}, u_j || idx_{K_i} || c_p^{K_i}) \rangle$ to the user, who then forwards this message to REV. REV then locates s_j^i and will create a report rpt_{REV} containing $m_{rev} = \langle r_8, \text{E}_{\text{pk}_{\text{KT}}}(s_j^i), \sigma_{\text{REV}}(H(r_8 || s_j^i)) \rangle$ that will be sent to KT. At this point, KT will verify rpt_{REV} , retrieve $c_p^{K_i}$ and send $m_{key} = \langle r_9, \text{E}_{\text{pk}_{\text{CSP}}}(u_j, t, s_j^i, idx_{K_i}), c_p^{K_i}, \sigma_{\text{KT}}(H(r_9 || u_j || t || s_j^i || idx_{K_i} || c_p^{K_i})) \rangle$ to u_j . Finally, u_j uses her private CP-ABE key to recover K_i .

CLASH.KeyShare("share", m_{verReq}): The KT and REV programs, **Q_{KT}^{Share}** and **Q_{REV}^{Share}** are defined as follows:

Data Processing Phase:

In the *Data Processing Phase* a user u_j that already received and successfully decrypted $c_p^{K_i}$ can start interacting with the CSP to access files encrypted under K_i . To this end, u_j can either run CLASH.Search, CLASH.Update and CLASH.Delete, depending on her access rights. CLASH.Search, allows users to search directly on the encrypted files, for those that contain a specific keyword w . User u_j first needs to create a search token for a keyword, $\tau_s(w)$. After the search token is created, u_j sends $m_{search} = \langle \tau_s(w), m_{key}, \text{HMAC}(K_{j,\text{CSP}}, \tau_s(w) || m_{key}) \rangle$, where

m_{key} is received in the *Key Sharing Phase*, to CSP. Upon reception, CSP opens m_{key} to check the freshness of the timestamp and to verify that $s_j^i[0] = 1$. Finally, the CSP will search for the files containing w and it will send back to u_j a sequence of file identifiers. This is done by loading the program Q_{CSP}^{Search} to the CSP enclave:

Q_{KT}^{Share}

- On input (“KeyRequest”, m_{keyReq}):
 - Open m_{keyReq} ; verify the message; if the verification fails, output \perp .
 - Compute and output m_{idxkey} .
 Run $m_{idxkey} \leftarrow HW.Run(hdl_{KT}, (“verify”, m_{keyReq})).$
- On input (“token”, rpt_{REV}):
 - Verify rpt_{REV} . If the verification fails, output \perp .
 - Open m_{rev} ; verify the message; if the verification fails, output \perp .
 - Check access rights based on s_j^i
 - Compute and output m_{key}
 Run $m_{key} \leftarrow HW.Run(hdl_{KT}, (“token”, rpt_{REV})) who will internally run $HW.ReportVerify(hdl_{KT}, rpt_{rev})$$

Q_{REV}^{Share}

- On input (“share”, m_{idxkey}):
 - Open m_{idxkey} ; verify the message; if the verification fails, output \perp .
 - Identify s_j^i based on idx_{K_i}
 - Compute m_{rev} .
 - Generate and output rpt_{REV} containing m_{rev}
 Run $rpt_{REV} \leftarrow HW.RunReport(hdl_{REV}, (“share”, m_{idxkey})).$

Q_{CSP}^{Search}

- On input (“search”, m_{search}):
 - Open m_{search} ; verify the message; if the verification fails, output \perp .
 - Identify (γ_i, c_i) based on idx_{K_i} .
 - Run $I_w \leftarrow DSSE.Search(\gamma_i, c_i, \tau_s(w))$
 - Output I_w .
 Run $HW.Run(hdl_{CSP}, (“search”, m_{search})).$

By running $CLASH.Update(“update”, f)$ a user u_j can add new files to u_i 's encrypted databases. For u_j to successfully run $CLASH.Update$ she first needs to create an add token. To this end, she generates $(\tau_\alpha(f), c_f)$ for a file f and sends it to the CSP via $m_{add} = \langle m_{key}, \tau_\alpha(f), c_f, HMAC(K_{j_{CSP}}, \tau_\alpha(f) || c_f || m_{key}) \rangle$. Upon reception, the CSP will check the freshness of the message and that $s_j^i[1] = 1$. If the verifications are successful, the new file will be added to the database. This is done by loading the program Q_{CSP}^{Up} program in the CSP enclave:

$CLASH.Delete(“delete”, f)$: A user u_j can also delete files from the database (provided that $s_j^i[2] = 1$) by running $CLASH.Delete$. To do so, u_j first generates a delete token $\tau_d(d)$ for a file f and sends it to the CSP via $m_{del} = \langle m_{key}, \tau_d(f), HMAC(K_{j_{CSP}}, \tau_d(f) || m_{key}) \rangle$. CSP then verifies

the timestamp and u_j 's access rights and proceeds with deleting the specified file. This is done by loading the program Q_{CSP}^{Del} to the CSP enclave.

Q_{CSP}^{Up}

- On input (“update”, m_{add}):
 - Open m_{add} to retrieve m_{key} and the add token $(\tau_\alpha(f), c_f)$
 - Verify m_{key} . If the verification fails, output \perp .
 - Run $(\gamma_i', c_i') \leftarrow DSSE.AddFile(\gamma_i, c_i, \tau_\alpha(f), c_f)$.
 Run $HW.Run(hdl_{CSP}, (“update”, m_{add})).$

Q_{CSP}^{Del}

- On input (“delete”, m_{del}):
 - Open m_{del} to retrieve m_{key} and the delete token $\tau_d(f)$
 - Verify m_{key} . If the verification fails, output \perp .
 - Run $(\gamma_i'', c_i'') \leftarrow DSSE.Delete(\gamma_i, c_i, \tau_d(f))$.
 Run $HW.Run(hdl_{CSP}, (“delete”, m_{del})).$

Scope Management Phase: The last phase of our construction focuses on the revocation and assignment of users' scopes through $CLASH.Manage$.

$CLASH.Manage(“assign/revoke”, u_\ell, n)$: A user u_j can revoke and assign scopes such as **search**, **update** and **delete** to another user u_ℓ if and only if $s_j^i[3] = 1$ and $s_\ell^i \neq [11111]$ (i.e. u_ℓ does not have owner's rights over the encrypted files). To revoke the scope **manager**, u_j must have owner's rights. Finally, ownership rights are assigned and revoked only by the data owner (i.e. u_i). In particular, for u_j to revoke a scope from a user u_ℓ she first contacts REV by sending $m_{manage} = \langle r_{10}, E(K_{j_{REV}}, u_j || u_\ell || n || “assign/revoke”), c_p^{K_i}, HMAC(K_{j_{REV}}, r_{10} || u_j || u_\ell || n || c_p^{K_i} || “assign/revoke”) \rangle$, where $n \in [0, 4]$ is an index of the one dimensional bit array s_ℓ^i and specifies which bit of the array will be flipped. Upon reception, REV will verify the message and it will generate a report rpt_{REV} containing $m_{idx.req} = \langle r_{11}, E(K_{KT_{REV}}, u_\ell), c_p^{K_i}, HMAC(K_{KT_{REV}}, r_{11} || c_p^{K_i} || u_\ell) \rangle$ that will be sent to KT. After KT verifies rpt_{REV} , it will send a report rpt_{KT} containing idx_{K_i} back to REV. REV then verifies rpt_{KT} and uses idx_{K_i} to identify the bit arrays s_j^i and s_ℓ^i , and checks whether u_j has the right to revoke or assign scopes to other users. If so, REV revokes/ assigns the requested scope from u_ℓ by setting $s_\ell^i[n] = 0 / s_\ell^i[n] = 1$. In case of assigning the scope **owner** ($n = 4$), REV further sets $s_\ell^i = [11111]$. The programs Q_{REV}^{Rev} and Q_{KT}^{idx} responsible for handling this procedure are the following:

C. SEARCHING ON MULTIPLE DATASETS

In a realistic scenario, a user would want to perform a search operation on multiple data sets at once. However, our construction focuses on the problem of searching on a single dataset per search query. The problem that arises is that each data owner is using a different symmetric key

to encrypt her data and thus, to perform a *global* search the CSP would require all the indexes $\text{id}x_{K_i}$. To solve this problem, we slightly modify the key sharing protocol as follows: u_j sends a request to KT via $m_{\text{verReq}} = \langle r_6, E(K_{\text{KT}}, u_j || L_u), \text{HMAC}(K_{\text{KT}}, r_6 || u_j || L_u) \rangle$, where L_u is a list containing unique identifiers of data owners that have granted u_j with at least one scope. Upon reception, KT replies with $m_{\text{id}x\text{key}} = \langle r_7, E(K_{\text{KTREV}}, u_j || L_{\text{id}xK}), L_{c_p}^K, \text{HMAC}(K_{\text{KTREV}}, u_j || L_{\text{id}xK} || L_{c_p}^K) \rangle$, where $L_{c_p}^K$ is the list of all the corresponding ciphers of the symmetric keys. The user then forwards this message to REV, who will locate $s_j^i, \forall i$ such that $c_p^{K_i} \in L_{c_p}^K$ and store them in a list $L_{s_j^i}$. Finally, after the KT and REV enclaves execute the local attestation protocol just like in the original construction, u_j will receive $m_{\text{key}} = \langle r_9, E_{\text{pk}_{\text{CSP}}}(u_j, t, L_{s_j^i}, L_{\text{id}xK_i}, L_{c_p}^{K_i}, \sigma_{\text{KT}}(H(r_9 || u_j || t || L_{s_j^i} || L_{\text{id}xK_i} || L_{c_p}^{K_i}))) \rangle$. At this point, u_j decrypts all $c_p^{K_i} \in L_{c_p}^{K_i}$ to recover the different symmetric keys. To perform a search operation on multiple datasets, u_j can now send the new m_{key} to the CSP as part of m_{search} , and the CSP will proceed with searching on every dataset specified by $L_{\text{id}xK_i}$.

Q_{REV}

- On input (“idx”, m_{manage}):
 - Verify the message. If the verification fails, output \perp .
 - Generate rpt_{REV} containing $m_{\text{id}x.req}$
Run $\text{HW.Run}(\text{hdl}_{\text{REV}}, (“idx”, m_{manage})), and $\text{rpt}_{\text{REV}} \leftarrow \text{HW.RunReport}(\text{hdl}_{\text{REV}}, (“idx”, m_{manage})).$$
- On input (“revoke/assign”, rpt_{KT}):
 - Verify rpt_{KT} . If the verification fails, output \perp .
 - Check the scope of u_j and u_ℓ . If u_j is not eligible to revoke/ assign u_ℓ , output \perp .
 - Set $s_j^\ell[n] = 0$ in case of revocation; otherwise, set $s_j^\ell[n] = 1$. If $n = 4$ and in case of assignment, set $s_j^\ell = [11111]$.
Run $\text{HW.Run}(\text{hdl}_{\text{REV}}, (“revoke/assign”, rpt_{KT})) who will internally run $\text{HW.ReportVerify}(\text{hdl}_{\text{REV}}, \text{rpt}_{\text{KT}})$.$

Q_{KT}

- On input (“idx.request”, rpt_{REV}):
 - Verify rpt_{REV} . If the verification fails, output \perp .
 - Get $c_p^{K_i}$ and identify $\text{id}x_{K_i}, u_i$.
 - Check whether $u_\ell \neq u_i$. If the verification fails, output \perp .
 - Generate and output a report rpt_{KT} containing $\text{id}x_{K_i}$.
Run $\text{HW.ReportVerify}(\text{hdl}_{\text{KT}}, \text{rpt}_{\text{REV}})$, then $\text{rpt}_{\text{KT}} \leftarrow \text{HW.RunReport}(\text{hdl}_{\text{KT}}, (“idx.request”, rpt_{REV})).$

VI. SECURITY ANALYSIS

A. SIMULATION-BASED SECURITY

To prove the security of our construction, we assume the existence of a simulator \mathcal{S} . The main purpose of \mathcal{S} is to simulate the algorithms of the real protocol in such a way that any polynomial time adversary \mathcal{ADV} will not be able to

distinguish between the real protocol and \mathcal{S} . We assume that \mathcal{S} intercepts \mathcal{ADV} 's communication with the real protocol and replies with simulated outputs. Before we proceed with the proof, we define the capabilities of \mathcal{S} and \mathcal{ADV} .

- 1) Everything \mathcal{ADV} 's observes in the real experiment can be simulated by \mathcal{S} .
- 2) \mathcal{ADV} intercepts all communication between different entities. Since we use an IND-CCA2 public key encryption scheme, if \mathcal{ADV} can distinguish between real and simulated answers, then she can also break the IND-CCA2 security.
- 3) \mathcal{ADV} can load different programs in the enclaves and record the output. This assumption significantly strengthens \mathcal{ADV} since we need to ensure that only honest attested programs will be executed in the enclaves.

Definition 4 (Sim-Security): We consider the following experiments. In the real experiment, all algorithms run as defined in our construction. In the ideal experiment, a simulator \mathcal{S} intercepts \mathcal{ADV} 's queries and replies with simulated responses.

Real Experiment

- 1) $\text{EXP}_{\text{CLASH}}^{\text{real}}(1^\lambda)$:
- 2) $(\text{MPK}, \text{MSK}) \leftarrow \text{CLASH.Setup}(1^\lambda)$
- 3) $(\gamma, c) \leftarrow \mathcal{ADV}^{\text{DSSE}}.\text{InGen}(\text{K}, \mathfrak{f})$
- 4) $\text{CLASH.Search}(\text{“search”}, m_s) \rightarrow \mathbf{I}_w$
- 5) $\text{CLASH.Update}(\text{“update”}, m_{\text{add}}) \rightarrow (\gamma', c')$
- 6) $\text{CLASH.Delete}(\text{“delete”}, m_{\text{delete}}) \rightarrow (\gamma', c')$
- 7) Output b

Ideal Experiment

- 1) $\text{EXP}_{\text{CLASH}}^{\text{ideal}}(1^\lambda)$:
- 2) $(\text{MPK}) \leftarrow \mathcal{S}(1^\lambda)$
- 3) $(\gamma, c) \leftarrow \mathcal{ADV}^{\mathcal{S}}(\mathcal{L}_{\text{in}}(\mathfrak{f}))$
- 4) $\mathcal{S}(\text{“search”}, m_s) \rightarrow \mathbf{I}_w$
- 5) $\mathcal{S}(\text{“update”}, m_{\text{add}}) \rightarrow (\gamma', c')$
- 6) $\mathcal{S}(\text{“delete”}, m_{\text{delete}}) \rightarrow (\gamma', c')$
- 7) Output b'

We say that CLASH is sim-secure if for all PPT adversaries \mathcal{ADV} :

$$|\Pr[(\text{Real}) = 1] - \Pr[(\text{Ideal}) = 1]| \leq \text{negl}(\lambda) \quad (1)$$

At a high-level, we construct a simulator \mathcal{S} that will replace the CLASH algorithms. In particular, in the real experiment, the adversary \mathcal{ADV} observes the algorithms being executed honestly, while in the ideal experiment \mathcal{S} responds with simulated answers. The idea is the following: \mathcal{ADV} has full control of the client. Thus, she can trigger Setup, Search, Update and Delete operations for the DSSE scheme. For each of these operations, \mathcal{S} gets as input the corresponding leakage function \mathcal{L}_i and simulates the CLASH.Search, CLASH.Update and CLASH.Delete oracles. Finally, we exclude the KeyShare and Manage oracles from the security game as they do not require to produce any simulated output for \mathcal{ADV} . However, for purposes of completeness, we include them in the proof of the theorem provided below.

Theorem 1: Assuming that PKE is an IND-CCA2 secure public key cryptosystem, SSE is an IND-CPA secure symmetric key cryptosystem and Sign is an EUF-CMA secure signature scheme then CLASH is a sim-secure protocol according to Definition 4.

Proof: We start by defining the algorithms used by the simulator. Then, we will replace them the real algorithms with the ones executed by the simulator. The algorithms not mentioned below, work just like in the real experiment. This does not affect the security of our construction as we are mainly focusing on the access control mechanism and the SSE scheme. For example, to further strengthen the threat model, we assume that the adversary has a real SSE key. Thus, there is no point in providing her with a simulated ABE secret key. However, in the security proof, we show that \mathcal{ADV} cannot tamper with any of the messages that are being exchanged during a run of the protocol. With the help of a Hybrid Argument, we will prove that the two distributions are indistinguishable.

- **CLASH.Setup***: Will only generate MPK that will be given to \mathcal{ADV} .
- **CLASH.Store***: \mathcal{S} generates a dictionary that will enable it to consistently reply to search queries even after file additions and deletions. In particular when \mathcal{ADV} triggers CLASH.Store, she actually triggers the InGen algorithm of the SSE scheme. Thus, \mathcal{S} gets as input the corresponding leakage function and simulates the SSE indexes.
- **CLASH.KeyShare***: \mathcal{S} encrypts $K_{\mathcal{ADV}}$ under MPK and sends it back to \mathcal{ADV} . Moreover, \mathcal{S} simulates and sends to \mathcal{ADV} m_{idxkey} and m_{key} . Finally m_{key} and m_{idxkey} are stored in a list L in order to prevent an attack in which \mathcal{ADV} would try to use a different set of valid scopes than the one she received.
- **CLASH.Search***: When \mathcal{ADV} performs a search operation for the files containing a keyword w , \mathcal{S} gets as input the leakage function \mathcal{L}_s and outputs a simulated token $\tau_s(w)$. Based on the simulated $\tau_s(w)$ can retrieve the files \mathcal{ADV} is looking for without performing the real search operation.
- **CLASH.Update***: When \mathcal{ADV} generates an add token $\tau_\alpha(f)$, \mathcal{S} gets as input the leakage function \mathcal{L}_a and outputs a simulated response. \mathcal{S} will simulate the add token, the ciphertext to be added to the database, and will also update the encrypted index.
- **CLASH.Delete***: When \mathcal{S} generates a delete token, \mathcal{S} gets as input the leakage function \mathcal{L}_d and outputs a simulated response. Apart from $\tau_d(f)$, \mathcal{S} will also update the encrypted index so that if \mathcal{ADV} performs a search operation in the future, for a keyword that is contained in the deleted file, the file will not be included in the result.
- **CLASH.Manage***: \mathcal{S} gets as input the list L_{VS} . By getting this list, an attack in which \mathcal{ADV} would try to assign/revoke scopes from a legitimate user can be avoided. In contrast to the real algorithm,

CLASH.Manage* does not assign/revoke any scopes from other users.

In a pre-processing phase, \mathcal{S} runs HW.Setup(1^λ), just as in the real experiment, in order to acquire sk_{rpt} . \mathcal{ADV} outputs a file collection \mathbf{f} and it encrypts it using SSE. Finally, she receives a set of scopes $\mathcal{SC}_{\mathcal{ADV}}$, that she can use during the run of the game. We will now use a hybrid argument to prove that \mathcal{ADV} cannot distinguish between the real and the ideal experiments.

Hybrid 0
CLASH runs normally.

Hybrid 1
Everything runs like in Hybrid 0, but we replace CLASH.Setup with CLASH.Setup*.

The difference between CLASH.Setup and CLASH.Setup* is that in CLASH.Setup*, \mathcal{S} only generates a key MPK instead of a (MPK, MSK) pair. Since in the real experiment, MSK is not given to \mathcal{ADV} anyway, MPK \mathcal{ADV} cannot distinguish between the two hybrids. Hence:

$$|Pr[(Hybrid 1) = 1] - Pr[(Hybrid 0) = 1]| \leq negl(\lambda) \quad (2)$$

Hybrid 2
Like Hybrid 1, but CLASH.KeyShare* runs instead of CLASH.KeyShare. Also, the algorithm outputs \perp if HW.ReportVerify is queried with $(hd|_{KT}, ("share", rpt_{REV}))$ but \mathcal{ADV} never contacts REV.

Lemma 1: Hybrid 2 is indistinguishable from Hybrid 1.

Proof: The simulator encrypts $K_{\mathcal{ADV}}$ with MPK and sends it to \mathcal{ADV} . Moreover since \mathcal{ADV} does not possess $K_{KT_{REV}}$ then she can only generate m_{idxkey} with negligible probability. Finally, \mathcal{ADV} can only generate a valid MAC of the report sent from KT to REV with negligible probability. Hence:

$$|Pr[(Hybrid 2) = 1] - Pr[(Hybrid 1) = 1]| \leq negl(\lambda) \quad (3)$$

□

At this point, \mathcal{ADV} can start making search, add and delete queries. The simulator now gets access to all leakage functions \mathcal{L} from the SSE scheme.

Hybrid 3
Like Hybrid 2, but when HW.Run is queried with $(hd|_{CSP}, ("search", m_{search}))$, \mathcal{S} is given the leakage function \mathcal{L}_s and generates a simulated search token. Moreover, the algorithm outputs \perp if the m_{key} message it receives is different than the one stored in L .

Lemma 2: Hybrid 3 is indistinguishable from Hybrid 2.

Proof: The algorithm already outputs \perp if m_{key} is different than the one stored in L since the verifications would fail. Assuming the \mathcal{L}_i - security of the SSE scheme, the token sent by \mathcal{ADV} to the CSP, as part of m_{search} , is generated by \mathcal{S}

with \mathcal{L}_S as input. As a result, when the CSP receives m_{search} , it will send back to \mathcal{ADV} the correct files without running DSSE.Search. \mathcal{ADV} cannot distinguish between the real and the ideal experiment since she receives a sequence of files corresponding to a search token that was simulated by \mathcal{S} given \mathcal{L}_S as input. Moreover, \mathcal{ADV} can only generate m_{search} without having contacted KT earlier with negligible probability, since she does not possess the secret key used to mac this message, and as a result \mathcal{ADV} can only distinguish between hybrids 3 and 2 with negligible probability. Thus:

$$|Pr[(Hybrid 3) = 1] - Pr[(Hybrid 2) = 1]| \leq \text{negl}(\lambda) \quad (4)$$

□

Hybrid 4

Like Hybrid 3, but when HW.Run is queried with $(\text{hdl}_{\text{CSP}}, ("update", m_{add}))$, \mathcal{S} is given the leakage function \mathcal{L}_a and tricks \mathcal{ADV} into thinking that she updated the database. Moreover, the algorithm outputs \perp is the m_{key} message it receives is different than the one stored in L .

Lemma 3: Hybrid 4 is indistinguishable from Hybrid 3.

Proof: The proof is similar to the previous one but simpler since \mathcal{ADV} does not expect an output from this algorithm. So, by assuming the \mathcal{L}_i -security of the SSE scheme, we know that \mathcal{ADV} will not be able to distinguish between the real add token and the simulated one. Additionally, the CPA-security of the symmetric encryption scheme, ensures that \mathcal{ADV} cannot distinguish between the encryption of an actual file and that of zeros. Moreover, if \mathcal{ADV} can generate m_{add} without having contacted KT, then she can also forge KT's MAC – which can only happen with negligible probability. Finally, the ciphertext sent along with the add token is stored in a list L , so that the simulator will answer consistently future search queries. Hence:

$$|Pr[(Hybrid 4) = 1] - Pr[(Hybrid 3) = 1]| \leq \text{negl}(\lambda) \quad (5)$$

□

Hybrid 5

Like Hybrid 4, but when HW.Run is queried with $(\text{hdl}_{\text{CSP}}, ("delete", m_{del}))$, \mathcal{S} is given the leakage function \mathcal{L}_d and simulates the delete token.

Lemma 4: Hybrid 5 is indistinguishable from Hybrid 4.

Proof: Just like before, the algorithm already outputs \perp if m_{key} is different than the one stored in L . By assuming the \mathcal{L}_i -security of the SSE scheme, we know that \mathcal{ADV} will not be able to distinguish between the real delete token and the simulated one. Thus, \mathcal{ADV} can only distinguish between Hybrids 5 and 6 with negligible probability. Thus:

$$|Pr[(Hybrid 5) = 1] - Pr[(Hybrid 4) = 1]| \leq \text{negl}(\lambda) \quad (6)$$

□

Hybrid 6

Like Hybrid 5 but instead of CLASH.Manage, \mathcal{S} executes CLASH.Manage*.

Lemma 5: Hybrid 6 is indistinguishable from Hybrid 5.

Proof: Since the valid scope list is not retrievable during the execution of the protocol, \mathcal{ADV} can never tell if she really revoked any scope from a specific user. \mathcal{ADV} could try to bypass KT's authentication by generating and sending rpt directly to REV. However, since \mathcal{ADV} does not possess sk_{rpt} , she can only do that with negligible probability. Hence, \mathcal{ADV} can only distinguish between Hybrids 6 and 7 with negligible probability and as a result:

$$|Pr[(Hybrid 6) = 1] - Pr[(Hybrid 5) = 1]| \leq \text{negl}(\lambda) \quad (7)$$

□

By combining inequalities 2 - 7 and using the triangle inequality property, we get:

$$|Pr[(Hybrid 6) = 1] - Pr[(Hybrid 0) = 1]| \leq 5 \cdot \text{negl}(\lambda) \quad (8)$$

However, it is a standard result in analysis that the finite sum of negligible functions, is still negligible. And thus:

$$|Pr[(Hybrid 6) = 1] - Pr[(Hybrid 0) = 1]| \leq \text{negl}(\lambda) \quad (9)$$

which implies:

$$|Pr[(Real) = 1] - Pr[(Ideal) = 1]| \leq \text{negl}(\lambda)$$

And hence, our proof is complete. We managed to replace the expected outputs with simulated responses, in a way that no PPT \mathcal{ADV} cannot distinguish between the real and ideal experiments. □

B. SGX SECURITY

Recent works [17], [22], [31], [32] have shown that SGX is vulnerable to software attacks. However, according to [20], these attacks can be prevented if the programs running in the enclaves are data-oblivious. Thus, leakage can be avoided if the programs do not have memory access patterns or control flow branches that depend on the values of sensitive data. In our construction, no sensitive data (such as decryption keys) are used by the enclaves. KT acts as a storage space for the symmetric keys and does not perform any computation on them. Hence, all the $c_p^{K_i}$ are data-oblivious. Moreover, L_{VS} is stored in plaintext and every entry in the list is padded to achieve same length. Moreover, we can prevent timing attacks on L_{VS} by ensuring that every time REV accesses the list, it goes through the whole list. Finally, as also mentioned in [6], Encryption and Decryption using AES-NI hardware instruction ensure there is no leakage of the encryption key during search and update operations. This is because since AES encryption and decryption using these instructions have data-independent timing and involve only data-independent memory access. Thus, by assuming a constant time implementation, our construction is not vulnerable to side-channel attacks.

TABLE 1. ℓ : Number of resulted files after a search query, m : Number of unique keywords in a file, MC: Multi-Client, FP: Forward Privacy.

Comparison						
Scheme	MC	FP	Search Time	Update Time	Client Storage	SGX
Etamad et al. [19]	✗	✓	$O(\ell/p)$	$O(m/p)$	$O(m+n)$	✗
HardIDX	✗	✗	$O(\log k)$	-	None	✓
Sophos	✗	✓	$O(\ell)$	$O(m)$	$O(m)$	✗
Ours	✓	✓	$O(\ell/p)$	$O(m/p)$	None	✓

VII. EVALUATION AND EXPERIMENTAL RESULTS

In this section, we present our experimental results that aimed at measuring the processing time of our construction. For the implementation of the SSE scheme, we used the forward private scheme presented in [9], while for the ABE scheme we used the library provided by [5]. Finally, to construct the cryptographic parts of the protocol within SGX secure containers (i.e. enclaves), we used the SGX-OpenSSL library in [3].

As we aimed to evaluate the performance of CLASH under realistic conditions, we used different machines – depending on the process to be measured. The setup of the SSE scheme was measured on a Microsoft Surface Book laptop with a 2.1GHz Intel Core i7 processor and 16GB RAM running Windows 10 64-bit. The reason being that in a practical scenario, this process would take place on a user’s machine. Conducting the experiments on a powerful server would result in a set of non-realistic results. The parts running in an enclave were measured in a powerful desktop PC with Intel Core i7-8700 at 3.20GHz (6 cores), 32GB of RAM running Ubuntu 64-bit, and Intel SGX Hardware Debug mode build configurations. The reason for running these parts on such a computer is based on the assumption that these processes will be running on the CSP.

A. SYMMETRIC SEARCHABLE ENCRYPTION

1) THEORETICAL EVALUATION AND COMPARISON

While our construction can work with any dynamic SSE scheme, we chose to use the scheme we developed and presented in [9]. Our SSE scheme is amongst the most efficient schemes that also support the crucial notion of forward privacy in the multi-client model. Informally, an SSE scheme is said to be forward private when the adversary cannot link newly added keyword to previous search queries. More information on forward privacy can be found in [9]. More precisely, our scheme achieves optimal search and update costs $O(\ell)$ and $O(m)$ respectively, where ℓ is the number of the resulted files on each search operation and m is the number of unique keywords in a file. Additionally, the scheme is parallelizable and hence, distributing the load to p processors, would further improve the search and update operations by a factor of $1/p$, resulting in a search cost of $O(\ell/p)$ and an update cost of $O(m/p)$ respectively. Finally, our scheme in [9] supports the multi-client model and is SGX-assisted. Hence, it shares a very similar architecture with the one presented in this work. In Table 1, we compare the SSE scheme used in this work with the SSE schemes presented in Section II.

TABLE 2. Size of Datasets and Keywords.

TXT Files	Dataset Size	Unique Keywords
425	184MB	1,370,023
815	357MB	1,999,520
1,694	670MB	2,688,552
1,883	1GB	7,453,612
2,808	1.7GB	12,124,904

2) EXPERIMENTAL RESULTS

For this part of our experiments, we mainly focused on (1) *Indexing* and (2) *Searching for a keyword w* . The SSE scheme was implemented in Python 2.7 using the PyCrypto library [2]. To extensively test the performance of the SSE scheme, we extracted various datasets, illustrated in Table 2, from the Gutenberg project [1]. Finally, the dictionaries were stored in a MySQL database.

a: Indexing & Encryption

This is the setup phase of the SSE scheme. This phase includes (1) *reading plaintext files, extracting the keywords and creating the necessary dictionaries*, (2) *encrypting the files* and (3) *building the encrypted indexes*. We run each process ten times, for each dataset in Table 2 and measured the average time. The results are illustrated in Figure 2. To index and encrypt 1,370,023 keywords, the average time was measured at 22.48min, while for 12,124,904 keywords, the corresponding time was 203.28min. Note here, that this is the most demanding phase of the protocol and that it only occurs once, on the data owner’s side. As a result, it does not affect the overall efficiency of our construction. Moreover, based on the results of other SSE schemes that are not forward private [18], the times measured are acceptable. Finally, to recreate a realistic scenario, this phase of the experiments was measured at a commodity laptop.

Additionally to the index of unique keywords, the SSE schemes makes use of one more index containing a mapping between keywords and file identifiers. The total number of these mappings can be seen in Table 3.

TABLE 3. Keywords and Filenames pairs.

Unique Keywords	(w, id) Pairs
1,370,023	5,387,216
1,999,520	10,036,252
2,688,552	19,258,625
7,453,612	28,781,567
12,124,904	39,747,904

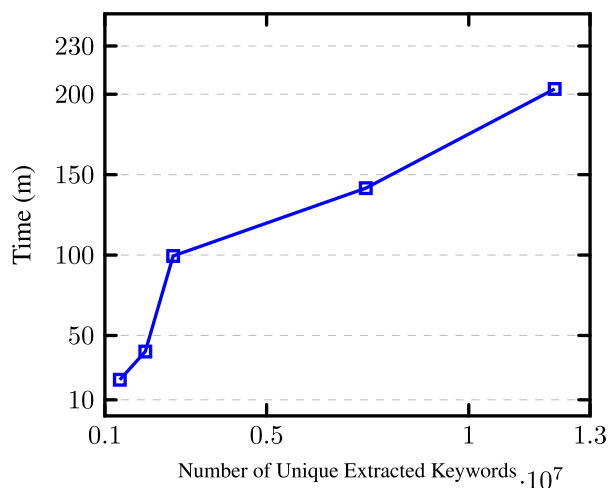


FIGURE 2. Indexing and Encrypting Files.

b: Search

To measure the exact time needed to perform a search operation we need to take into account (1) The time required to generate a search token and (2) The time needed by the CSP to find and return the file identifiers of those files that contain the specified keyword. On average, the creation time for a search token was measured at 9μs, while searching for a specific keyword over a set of 12.124.904 distinct keywords and 39.747.904 addresses required 3.2sec.

B. CIPHERTEXT-POLICY ATTRIBUTE-BASED ENCRYPTION

For the implementation of CP-ABE, we used the scheme presented in [5], offered by Charm-Crypto Framework version 50.0 in a Docker container. The experiments were implemented in Python 3.6 and conducted on a Desktop machine with Intel Core i7-8700 at 3.20GHz (6 cores), 32GB RAM.

c: Setup Phase

The first phase of our experiments was devoted to measuring the time required to generate a master public/private key pair for a master entity. In our setup, we considered the existence of a single master entity responsible for the generation of CP-ABE keys. The time to generate a single pair was less than a second, while the total time for the generation of 200 master key pairs was measured at almost 6 seconds. These results are illustrated in Figure 3.

d: Users Key Generation

In the second phase of the experiments, we measured the average time needed to generate secret users' keys. In particular, we measured the time to generate a user's key while increasing the number of attributes associated with it. As can be seen in Figure 4, the average time to generate a user key with 1.000 attributes took almost 6.41sec, while a key with 500 attributes required approximately 3.23sec. These results are suitable for covering even more complex cases where big companies are required to generate large keys based on a wide variety of information. Thus, it can be stated

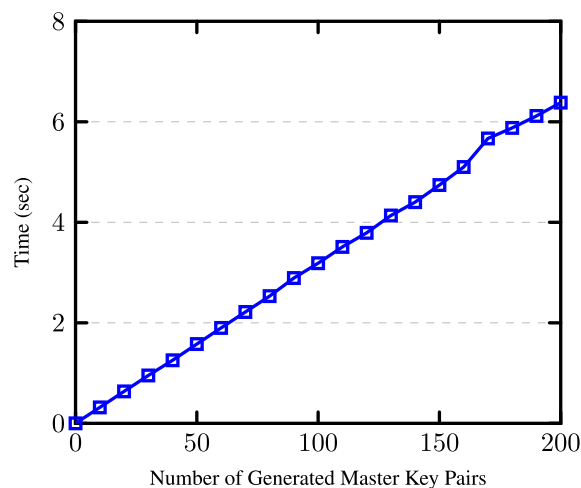


FIGURE 3. Generation of master public/private key pairs.

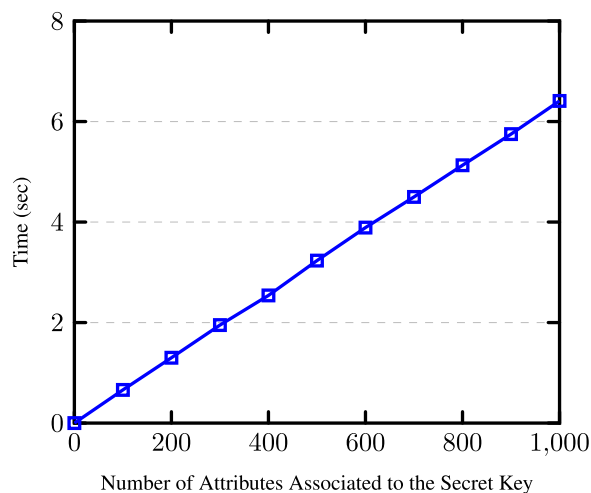


FIGURE 4. Generation of user key with up to 1000 attributes.

that covering a long list of attributes is realistic and should not prevent an organization from adopting such an approach.

Moreover, as can be seen from Figure 5, we observe that the size of the key is almost linear to the number of attributes associated with it. In particular, the size of a key associated with 1.000 attributes is around 420KB, while for a key associated with 500 attributes the disk size was measured at almost 215KB. Finally, a key associated with 100 attributes has a size of approximately 45KB on the disk.

e: Encryption & Decryption

CLASH only use CP-ABE to encrypt a symmetric key and not large volumes of data. Hence, we measured the time needed to encrypt and decrypt a symmetric key under policies of different sizes. We used access policies of type {1 AND 2 AND ... AND n} similar to [5]. Such policies are the most demanding since all attributes are required for the successful decryption. The experiment can be divided into two stages. In the first stage, we measured the encryption process. In particular, we ran an encryption algorithm on a message

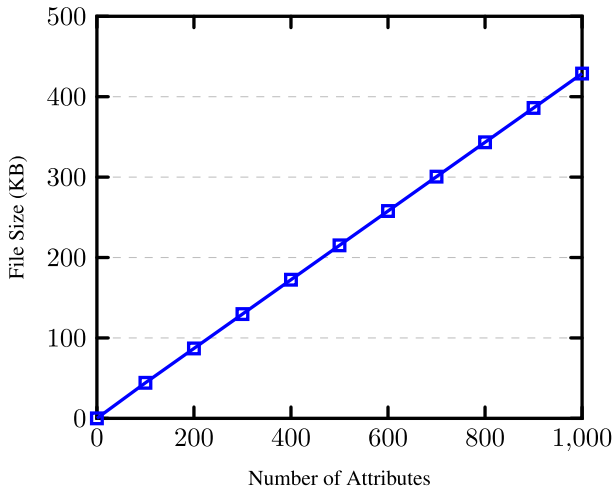


FIGURE 5. Disk size of the key as the number of attributes is increasing.

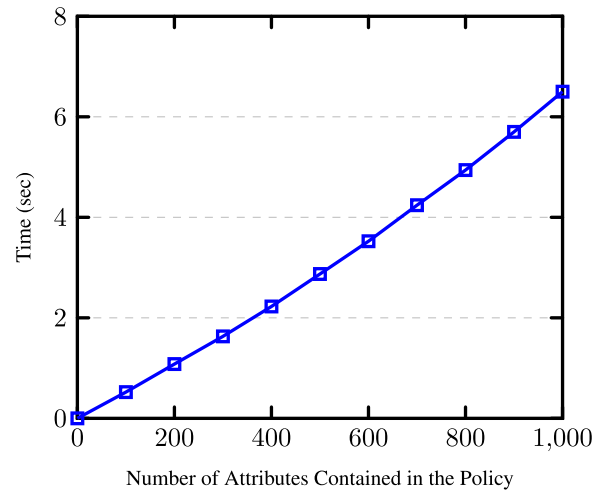


FIGURE 6. Encryption.

with different policies. In the second stage, we decrypted the freshly generated ciphertexts with keys that are associated with a different number of attributes. In addition to that, we were adding access policies of a different structure to record the performance of the decryption not only when all conditions needed to be fulfilled (most demanding case), but also when a random number of attributes is needed to satisfy the underlying policy. Figure 6, demonstrates the time required to encrypt a symmetric key with a random policy of size up to 1,000 attributes. Similarly, Figure 7 illustrates the time needed to decrypt a ciphertext by using a key with up to 1,000 attributes. As can be seen from the figures, the time to encrypt and decrypt a message depends on the particular attributes available and the size of the policy. In particular, the encryption of a key with a policy of 1,000 attributes took approximately 6.5 seconds while the decryption time was measured at almost 0.068 seconds. However, for more realistic scenarios where policies contain around 200 attributes, the encryption time was around a second and the decryption time was almost 0.028 seconds. It is evident that the underlying CP-ABE scheme does not add any real computational burden to the overall performance of the protocol.

In the second stage of the experiment, we focused on analyzing the behavior of the underlying ABE scheme. More precisely, we created an algorithm that randomly generates a policy that contains numerical attributes as well as conditions such as $\{(1 \text{ AND } 2) \text{ OR } (3 \text{ AND } 4)\}$. This condition required that at least one of two parenthesis are satisfied by the attributes of a user's key. Figure 8 shows the time needed to decrypt a ciphertext bound with a policy of up to 1,000 attributes. From the result shown in the graph, we can observe that the decryption time is linear regardless the randomness of the policy.

C. IMPLEMENTATION AND EVALUATION THE OF Cloud we Share

We used the SGX OpenSSL cryptographic library [3] to implement RSA with 4096 bit keys. The reason for select-

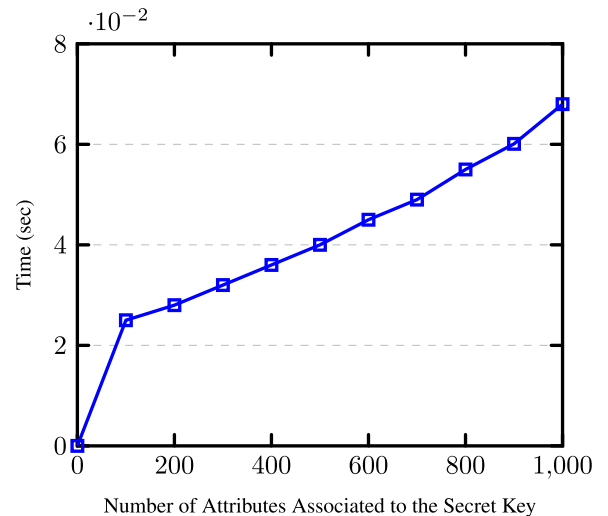


FIGURE 7. Decryption.

ing such a long key size was that we wanted to test the performance of our construction under the most demanding circumstances. Additionally, the development was done in C/C++ using Intel(R) SGX SDK 2.6 for Linux [4].

An SGX application is divided into two different parts; the trusted part (i.e. the enclave) and the untrusted part (i.e. application). To make a call to the enclave, the untrusted application is using the SGX's `ECALL` function, which allows the application to enter the enclave. Similarly, SGX's function `OCALL` is used to exit the enclave back to the untrusted space.

f: Setup Phase

The setup phase consists of launching the enclaves and generating all the necessary keys. Each enclave contains multiple functions that correspond to different parts of our construction. Therefore, we measured the time taken to launch each enclave separately. To acquire more accurate results, each enclave was launched 10,000 times and we

TABLE 4. Processing time of primitive blocks of The Cloud we Share.

Primitive Blocks	Functions	Details	Processing Time
CP-ABE	Encryption	200 attributes	1.08sec
		1000 attributes	6.5sec
	Decryption	200 attributes	28ms
		1000 attributes	68ms
SSE	Indexing	184MB; 1,370,023 keywords	22.48min
		357MB; 1,999,520 keywords	40.00min
		670MB; 2,688,552 keywords	86.43min
		1GB; 7,453,612 keywords	141.60min
		1.7GB; 12,124,904 keywords	203.28min
	Token creation	1 search token	9μs
	Search	12,124,904 keywords	3.2sec
SGX	Enclave Creation	MS Enclave	25.4ms
		REV Enclave	28ms
		KT Enclave	27.6ms
		CSP Enclave	28ms
	Key generation	4096 bit RSA key	840ms
	Local attestation	REV and KT	1.1ms

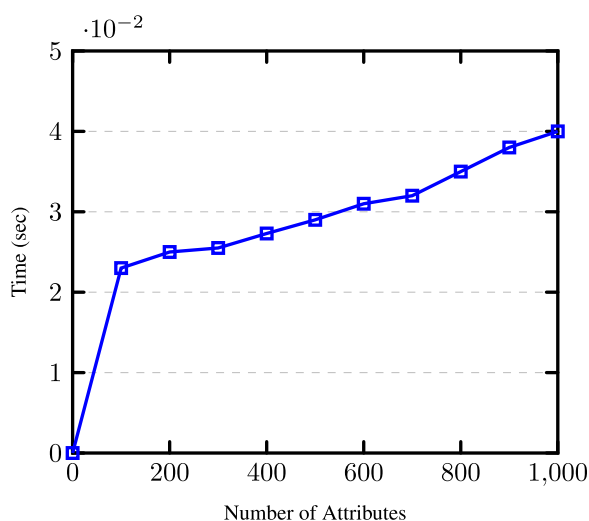


FIGURE 8. Decryption of ciphertext associated with a random policy.

measured the average completion time. The average time to launch the MS enclave, containing all the functions required for the generation of the RSA keys was 25.4ms while the average time to launch the REV enclave was 28ms. Similarly, the time needed to launch the KT enclave was measured at 27.6ms, and finally, the launching of the CSP enclave, required 28ms. Note here that the enclaves can be launched in parallel. Therefore, the time required to launch all four enclaves is 28ms. The final step of this setup phase is the generation of the RSA keys. In our construction, each enclave generates an 4096-bit RSA key pair. The time required for the generation of such a pair was measured at 840ms. However, this procedure can also be run in parallel, since each enclave generates its own key pair independently from the other enclaves. These results are illustrated in Table 4, along with the functions contained in each enclave.

g: Enclave Attestation

Different enclaves can attest to each other to demonstrate the integrity of their software. SGX offers two different kinds of attestation, local and remote. Local attestation

occurs between two or more enclaves running on the same platform, while Remote attestation enables a third party to attest an enclave. Currently, verifying a quote from a third party involves contacting Intel’s attestation server - a process that requires a license. Thus, for our experiments, we consider the case of local attestation. We measured the time needed between the KT enclave and the REV one to attest to each other, as part of CLASH.Manage. We run the experiment 10.000 times and the average time was measured at 1.1ms.

h: Execution Time

In the last part of our experiments, we determined the running time of CLASH’s functions by measuring the time required to (1) generate, (2) exchange and (3) verify all messages of our protocol. Each experiment was run 100.000 times to achieve a better estimation of the average time. Our focus was to measure the application execution time while it was running in secure containers (i.e. enclaves). Namely, we measured ECall functions at the moment of entering and exiting enclaves from the untrusted part of the application.

Our results are presented in Figure 9 by showing the average time needed for each one of the functions. As can be seen, CLASH.Manage, CLASH.KeyTrayStore and CLASH.KeyShare are the most demanding functions as they were measured at 474μs, 245μs and 81μs respectively. This result was expected due to the big number of exchanged messages. Moreover, the CLASH.Store took 23μs. Finally, the total execution time of CLASH.Search, CLASH.Update and CLASH.Delete were measured at 17μs, 17μs and 20μs.

i: Open Science and Reproducible Research

As a way to support open science and reproducible research and allow other researchers to use, test, and hopefully extend/enhance our protocol. Our CLASH prototype, as well as the ABE experiments, have been already uploaded to GitLab and are publicly available online.¹ In addition to that,

¹<https://gitlab.com/qdalza/clash>

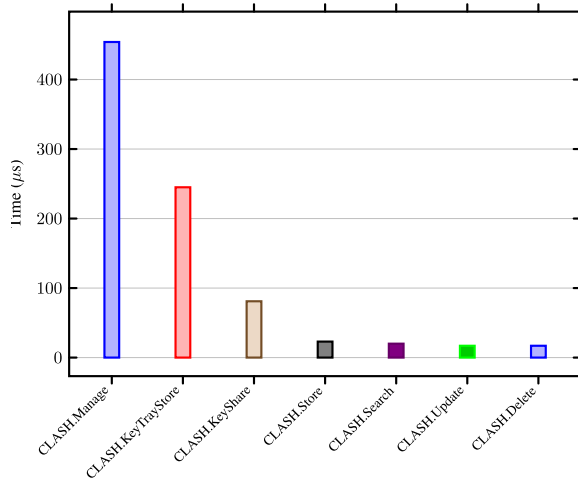


FIGURE 9. Message Creation and Verification.

the dataset that we used to perform the SSE experiments has been uploaded as a research artifact (Open Access) on Zenodo [25].

VIII. CONCLUSION

In this paper, we proposed The Cloud we Share, a hybrid encryption scheme based on SSE and ABE. Our construction allows a data owner to share her data in a privacy-preserving way and manage the access rights of the rest of the users. Moreover, we show that we can rely on the functionalities offered by Intel SGX, to design an access control mechanism that is agnostic to the underlying cryptographic primitives. In addition to that, we strongly believe that cloud-based services will rely less on traditional decryption of information, and more on computations over encrypted data. We hope that this work will kick-start a period of greater research in the area of privacy-preserving computations in untrusted clouds.

REFERENCES

- [1] *Project Gutenberg*, 1971.
- [2] *PyCrypto—The Python Cryptography Toolkit*, 2013.
- [3] *SGX-Openssl*, 2017.
- [4] *SGX-SDK*, 2019.
- [5] S. Agrawal and M. Chase, “FAME: Fast attribute-based message encryption,” in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2017, pp. 665–682.
- [6] G. Amjad, S. Kamara, and T. Moataz, “Forward and backward private searchable encryption with SGX,” in *Proc. 12th Eur. Workshop Syst. Secur. (EuroSec)*. New York, NY, USA: Association for Computing Machinery, 2019.
- [7] A. Bakas and A. Michalas, “Modern family: A revocable hybrid encryption scheme based on attribute-based encryption, symmetric searchable encryption and SGX,” in *Security and Privacy in Communication Networks*, S. Chen, K.-K. R. Choo, X. Fu, W. Lou, and A. Mohaisen, Eds. Cham, Switzerland: Springer, 2019, pp. 472–486.
- [8] A. Bakas and A. Michalas, “Multi-client symmetric searchable encryption with forward privacy,” *Cryptol. ePrint Arch.*, Tampere Univ., Tampere, Finland, Tech. Rep. 2019/813, 2019. [Online]. Available: <https://eprint.iacr.org/2019/813>
- [9] A. Bakas and A. Michalas, “Power range: Forward private multi-client symmetric searchable encryption with range queries support,” in *Proc. IEEE Symp. Comput. Commun. (ISCC)*, Jul. 2020, pp. 1–7.
- [10] J. Bethencourt, A. Sahai, and B. Waters, “Ciphertext-policy attribute-based encryption,” in *Proc. IEEE Symp. Secur. Privacy (SP)*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 321–334.
- [11] A. Boldyreva, V. Goyal, and V. Kumar, “Identity-based encryption with efficient revocation,” in *Proc. 15th ACM Conf. Comput. Commun. Secur. (CCS)*, New York, NY, USA, 2008, pp. 417–426.
- [12] D. Boneh, X. Boyen, and E.-J. Goh, “Hierarchical identity based encryption with constant size ciphertext,” in *Advances in Cryptology—EUROCRYPT*, R. Cramer, Ed. Berlin, Germany: Springer, 2005, pp. 440–456.
- [13] D. Boneh, G. D. Crescenzo, R. Ostrovsky, and G. Persiano, “Public key encryption with keyword search,” in *Proc. Int. Conf. Appl. Cryptograph. Techn.* Berlin, Germany: Springer, 2004, pp. 506–522.
- [14] R. Bost, “ σ olog: Forward secure searchable encryption,” in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2016, pp. 1143–1154.
- [15] R. Bost, B. Minaud, and O. Ohrimenko, “Forward and backward private searchable encryption from constrained cryptographic primitives,” in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2017, pp. 1465–1482.
- [16] V. Boyko, “On the security properties of OAEP as an all-or-nothing transform,” in *Advances Cryptology—CRYPTO*, M. Wiener, Ed. Berlin, Germany: Springer, 1999, pp. 503–518.
- [17] V. Costan and S. Devadas, “Intel SGX explained,” *Cryptol. ePrint Arch.*, Intel, Mountain View, CA, USA, Tech. Rep. 2016/086, 2016.
- [18] R. Dowsley, A. Michalas, M. Nagel, and N. Paladi, “A survey on design and implementation of protected searchable data in the cloud,” *Comput. Sci. Rev.*, vol. 26, pp. 17–30, Nov. 2017.
- [19] M. Etemad, A. Küpçü, C. Papamanthou, and D. Evans, “Efficient dynamic searchable encryption with forward privacy,” *Proc. Privacy Enhancing Technol.*, vol. 2018, no. 1, pp. 5–20, Jan. 2018.
- [20] B. Fisch, D. Vinayagamurthy, D. Boneh, and S. Gorbunov, “IRON: Functional encryption using Intel SGX,” in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*, Oct. 2017, pp. 765–782.
- [21] B. Fuhry, R. Bahmani, F. Brasser, F. Hahn, F. Kerschbaum, and A.-R. Sadeghi, “HardIDX: Practical and secure index with SGX,” in *Data and Applications Security and Privacy*, G. Livraga and S. Zhu, Eds. Cham, Switzerland: Springer, 2017, pp. 386–408.
- [22] S. Lee, M. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, “Inferring fine-grained control flow inside SGX enclaves with branch shadowing,” in *Proc. 26th USENIX Secur. Symp.*, Victoria, BC, Canada, Aug. 2017, pp. 557–574.
- [23] J. K. Liu, T. H. Yuen, P. Zhang, and K. Liang, “Time-based direct revocable ciphertext-policy attribute-based encryption with short revocation list,” in *Proc. Int. Conf. Appl. Cryptogr. Netw. Secur.* Cham, Switzerland: Springer, Jul. 2018, pp. 516–534.
- [24] A. Michalas, “The lord of the shares: Combining attribute-based encryption and searchable encryption for flexible data sharing,” in *Proc. 34th ACM/SIGAPP Symp. Appl. Comput.*, New York, NY, USA, Apr. 2019, pp. 146–155.
- [25] A. Michalas, “Text files from Gutenberg database,” Tampere Univ., Tampere, Finland, Tech. Rep., Aug. 2019. [Online]. Available: <https://zenodo.org/record/3360392#.X7fuasOzaUk>
- [26] A. Michalas, A. Bakas, H.-V. Dang, and A. Zaitko, “Access control in searchable encryption with the use of attribute-based encryption and SGX,” in *Proc. ACM SIGSAC Conf. Cloud Comput. Secur. Workshop*, 2019, p. 183.
- [27] A. Michalas, A. Bakas, H.-V. Dang, and A. Zaitko, “MicroSCOPE: Enabling access control in searchable encryption with the use of attribute-based encryption and SGX,” in *Nordic Conference on Secure IT Systems*. Cham, Switzerland: Springer, 2019, pp. 254–270.
- [28] S. Myers and A. Shull, “Practical revocation and key rotation,” in *Topics in Cryptology—CT-RSA*, P. Nigel, Ed. Cham, Switzerland: Springer, 2018, pp. 157–178.
- [29] N. Paladi, C. Gehrman, and A. Michalas, “Providing user security guarantees in public infrastructure clouds,” *IEEE Trans. Cloud Comput.*, vol. 5, no. 3, pp. 405–419, Jul. 2017.
- [30] S. Wang, K. Guo, and Y. Zhang, “Traceable ciphertext-policy attribute-based encryption scheme with attribute level user revocation for cloud storage,” *PLoS ONE*, vol. 13, no. 9, Sep. 2018, Art. no. e0203225.
- [31] N. Weichbrodt, A. Kurmus, R. Peter Pietzuch, and R. Kapitza, “Asynshock: Exploiting synchronisation bugs in intel SGX enclaves,” in *Proc. 21st Eur. Symp. Res. Comput. Secur. Comput. Secur. (ESORICS)*, Heraklion, Greece, Sep. 2016, pp. 440–457.
- [32] Y. Xu, W. Cui, and M. Peinado, “Controlled-channel attacks: Deterministic side channels for untrusted operating systems,” in *Proc. IEEE Symp. Secur. Privacy (Oakland)*, May 2015, pp. 640–656.



ALEXANDROS BAKAS received the master’s degree in applied mathematics from the National Technical University of Athens, Greece, and the master’s degree in cryptography from the University of Rennes 1, France. He is currently pursuing the Ph.D. degree in applied cryptography, as a member of the Network and Information Security Group (NISEC) with the Department of Computing Sciences, Tampere University, Finland. His research interests include searchable encryption, functional encryption, differential privacy, and cloud security.



ANTONIS MICHALAS received the Ph.D. degree in network security from Aalborg University, Denmark. He is currently working as an Assistant Professor with the Department Computing Sciences, Tampere University, Finland, where he also co-leads the Network and Information Security Group (NISEC). The group comprises Ph.D. students, professors, and researchers. Group members conduct research in areas spanning from the theoretical foundations of cryptography to the design and implementation of leading edge efficient and secure communication protocols. Apart from his research work at NISEC, as an Assistant Professor he is actively involved in the teaching activities of the University. Finally, his role expands to student supervision and research projects coordination. Furthermore, he has published a significant number of articles in field-related journals and conferences and has participated as a Speaker in various conferences and workshops. His research interests include private and secure e-voting systems, reputation systems, privacy in decentralized environments, cloud computing, trusted computing and privacy preserving protocols in eHealth, and participatory sensing applications.



HAI-VAN DANG received the Ph.D. degree with the University of Science, Vietnam National University of HCM City (VNU-HCMC), in 2017. Besides, she gained experiences in teaching assistant in courses, such as Graph Theory, Introduction to Cryptography, and teaching courses, such as Graph Theory and Applied Statistics. She also experienced a part-time job at GlobeDr, a healthcare social network company, as a Technology Researcher. She is currently a Research Associate

with the University of Westminster, London, U.K. Her research interests include privacy-preserving protocols, searchable encryption, authentication, and authorization in cloud computing.



ALEXANDR ZALITKO received the M.Sc. degree in cyber security and forensics from the University of Westminster, London. He is currently working as a Ph.D. Researcher with Tampere University, Finland. His research interests include primarily in the areas of cloud security, malware detection, and implementation of security protocols in widely deployed communication networks.

...