Modern Family: A Revocable Hybrid Encryption Scheme Based on Attribute-Based Encryption, Symmetric Searchable Encryption and SGX

Alexandros Bakas and Antonis Michalas

Tampere University,
Tampere, Finland
{alexandros.bakas,antonios.michalas}@tuni.fi

Abstract. Secure cloud storage is considered as one of the most important issues that both businesses and end-users take into account 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 this paper, we propose a hybrid encryption scheme that combines both SSE and ABE by utilizing the advantages of both these techniques. In contrast to many approaches, we design a revocation mechanism that is completely separated from the ABE scheme and solely based on the functionality offered by SGX.

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

1 Introduction

Cloud computing plays a significant role in our daily routine. From casual internet users, to big corporations, the cloud has become an integral part of our lives. However, using services that are hosted and controlled by third parties raises several security and privacy concerns. For example, in [12] it is stated that there has been a 300% increase in Microsoft cloud-based user's account attacks over the past couple of years. However, when considering a cloud-based environment, cyber-attacks performed by remote adversaries is only a part of the problem. More precisely, when we design cloud services we also need to take into consideration cases where the actual cloud service provider (CSP) acts maliciously.

To overcome this, both academia and big industrial players have started looking on how to build cloud-based services that will utilize Symmetric Searchable Encryption (SSE) [7, 4]. In such a scheme, whenever a user wishes to access her files, she can search directly over the encrypted data for specific keywords. Unfortunately, revocation cannot be implemented efficiently since sharing an encrypted file implies sharing the encryption key. As a result, if a data owner wishes to revoke a user, then all files that are encrypted with the same key

must be decrypted and then re-encrypted under a fresh key. Another promising technique that fits cloud-based services is Attribute-Based Encryption (ABE). In ABE schemes, all files are encrypted under a master public key but in contrast to traditional public key encryption, the generated ciphertext is bounded by a policy. Each user has a distinct secret key which is associated with specific attributes. This way a user's secret key can decrypt a ciphertext if and only if the her attributes satisfy the policy bound to the ciphertext. However, using an asymmetric encryption scheme to store data is rather inefficient.
Contribution: We propose a hybrid encryption scheme that combines SSE and ABE in a way that reduces the problem of multi-user data sharing to that of a single-user. We use the ABE scheme as a sharing mechanism and not as a revocation one to achieve better efficiency. To deal with the problem of revocation, we utilize the functionality offered by SGX. Furthermore, this work extends the protocol presented in [11].
Organization: In Section 2, we present important works that have been published and address the problem of secure cloud storage, data sharing and revocation. In Section 3, we define our system model while in Section 4, we present the cryptographic tools needed for the construction of our scheme. In Section 5, we give a formal construction of our scheme which is followed by the security analysis in Section 6. Finally, Section 7 concludes the paper.

2 Related Work

In [13] authors present a revocable hybrid encryption scheme while at the same time a key-rotation mechanism is used to prevent key-scrapping attacks. The authors use Optimal Asymmetric Encryption Padding (OAEP) as an All-or-Nothing-Transformation (AONT) [2] to prevent revoked users from accessing stored data. This is due to the fact that reversing OAEP, requires to the entire output. Thus, changing random bits, renders the reversion infeasible. Hence, to decrypt a file, the changed bits need to be stored. However, this implies that with each re-encryption, the size of the ciphertext grows. Thus, decrypting a file that has been re-encrypted multiple times is an expensive operation. Moreover, to achieve better efficiency, authors suggest that the AONT could be applied by the server. However, this implies the existence of a fully trusted server.

A promising idea is presented in [5], where the authors present a protocol based on functional encryption, with the main functionalities running in isolated environments. The decryption of a file, and the application of a function f on the decrypted file both occur in SGX enclaves. Moreover, all enclaves can attest to each other and exchange data over secure communication channels. In our construction, even though we use the same hardware principles, we build a hybrid encryption scheme by combining SSE and ABE.

In [9] authors present a revocable ciphertext-policy attribute-based encryption scheme. The revocation mechanism is offered by a revocation list that is attached to the resulted ciphertexts. To avoid maintaining long revocation lists, a policy through which users' keys expire after a certain period of time is enforced.

As a result, the revocation list only includes keys that have been revoked before the expiration date. Another Hybrid encryption scheme is presented in [6], in which authors propose a scheme based on SSE and ABE. In the proposed scheme, data owners encrypt their files using SSE, but the resulted indexes are encrypted under ABE. This way, users can locally generate search tokens based on their attributes, that are then sent to the cloud. However promising, their scheme is static and as a result can only have very limited applications in real-life scenarios. Moreover, authors do not provide a revocation mechanism – a problem of paramount importance in cloud-based services.

In our construction, we overcome these issues by designing an efficient revocation mechanism that is utilizing the SGX functionality and it is separated from the ABE scheme.

3 Architecture

In this section, we introduce the system model by explicitly describing the main entities that participate in our protocol as well as their capabilities. The system model of our work is built on top of the model presented in [10] and it is enhanced with some important additions.

Cloud Service Provider (CSP): We consider a cloud computing environment similar to the one described in [14, 15]. Moreover, the CSP must support SGX since core entities 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 for the proper run of the involved protocols. MS is responsible for generating and distributing ABE keys to the registered users. Finally, MS is SGX-enabled and is running in an enclave called the Master Enclave.

Key Tray (KT): KT is a key storage that stores ciphertexts of the symmetric keys that have been generated by various users and are needed to decrypt data. Registered users can directly contact KT and request access to the stored ciphertexts. KT is also SGX-enabled and runs in an enclave called the KT Enclave.

Revocation Authority (REV): REV is responsible for maintaining a revocation list (rl) with the unique identifiers of the revoked users. Similar to MS and KT, REV is also SGX-enabled and is running in an enclave called the Revocation Enclave. Finally, for the security of the stored revocation list, it is important to mention that rl is generated by the enclave (i.e. in an isolated environment) and never leaves its perimeter. Therefore, there is no need to encrypt rl.

SGX: Below we provide a brief presentation of the main SGX functionalities needed for our construction. A more detailed description can be found in [5, 3]

Isolation: Enclaves are located in a hardware guarded area of memory and they compromise a total memory of 128MB (only 90MB can be used by software). Intel SGX is based on memory isolation built into the processor itself along with strong cryptography. The processor tracks which parts of memory belong to which enclave, and ensures that *only* enclaves can access their own memory.

Attestation: One of the core contributions of SGX is the support for attestation between enclaves of the same (local attestation) and different platforms (remote attestation). In the case of local attestation, an enclave enc_i can verify another enclave enc_j as well as the program/software running in the latter. This is achieved through a report generated by enc_j containing information about the enclave itself and the program running in it. This report is signed with a secret key $\mathsf{sk_{rpt}}$ which is the same for all enclaves of the same platform. In remote attestation, enclaves of different platforms can attest each other through a signed quote. This is a report similar to the one used in local attestation. The difference is that instead of using $\mathsf{sk_{rpt}}$ to sign it, a special private key provided by Intel is used. Thus, verifying these quotes requires contacting Intel's Attestation Server.

Sealing: Every SGX processor comes with a Root Seal Key with which, data is encrypted when stored in untrusted memory. Sealed data can be recovered even after an enclave is destroyed and rebooted on the same platform.

4 Cryptographic Primitives

In this section, we give a formal definition for the two main encryption schemes that the paper is based on. We proceed with the definition of a CP-ABE and SSE schemes as described in [1] and [7] respectively.

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

- $\mathsf{CPABE.Setup}$ *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* $(\mathsf{MPK}, \mathsf{MSK}) \leftarrow \mathsf{Setup}(1^\lambda)$.
- $\mathsf{CPABE.Gen}$ *is a probabilistic algorithm that takes as input a master secret key, a set of attributes* $\mathcal{A}$ *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* $(\mathsf{sk}_{\mathcal{A},\mathsf{u_i}}) \leftarrow \mathsf{Gen}(\mathsf{MSK}, \mathcal{A}, u_i)$.
- $\mathsf{CPABE.Enc}$ *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* $\mathsf{c_P} \leftarrow \mathsf{Enc}(\mathsf{MPK}, m, P)$.
- $\mathsf{CPABE.Dec}$ *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* $\mathsf{Dec}(\mathsf{sk}_{\mathcal{A},\mathsf{u_i}}, c_P) \rightarrow m$.

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

- $\mathsf{SSE.Gen}$ *is a probabilistic key-generation algorithm that takes as input a security parameter* λ *and outputs a secret key* K.

- $\mathsf{SSE.Enc}$ *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* $\mathbf{c}$*.*
- $\mathsf{SSE.SearchToken}$ *is a (possibly probabilistic) algorithm that takes as input a secret key* K *and a keyword* w *and outputs a search token* $\tau_s(w)$*.*
- $\mathsf{SSE.AddToken}$ *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*.*
- $\mathsf{SSE.DeleteToken}$ *is a (possibly probabilistic) algorithm that takes as input a secret key* K *and a file* f *and outputs a delete token* $\tau_d(f)$*.*
- $\mathsf{SSE.Search}$ *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* $\mathbf{I}_w \subset \mathbf{c}$*.*
- $\mathsf{SSE.Add}$ *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* $\mathbf{c}'$*.*
- $\mathsf{SSE.Delete}$ *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* $\mathbf{c}'$*.*
- $\mathsf{SSE.Dec}$ *is a deterministic algorithm that takes as input a secret key* K *and a ciphertext* c *and outputs a file* f*.*

The security of an SSE 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 [4].

5 Modern Family (MF)

In this section, we present Modern Family (MF) – the core of this paper's contribution. We start by giving an overview of the SGX hardware functionalities used by the communicating parties as defined in [5]. and we continue with a formal construction.

Hardware:

- $\mathbf{HW}.\mathbf{Setup}(\mathbf{1}^\lambda)$: Takes as input a security parameter λ and produces the secret key $\mathsf{sk}_{\mathsf{rpt}}$[1] used to MAC the reports.
- $\mathbf{HW}.\mathbf{Load}(\mathrm{Q})$: Takes as input a program Q. An enclave enc_i is created in which Q will be loaded. Moreover a handle $\mathsf{hdl}_{\mathsf{enc}}$ is created that will be used as an identifier for the enclave.
- $\mathbf{HW}.\mathbf{Run}(\mathbf{hdl}, in)$: Takes as input a handle hdl and some input in. It runs the program in the enclave specified by hdl with in as input.

[1] $\mathsf{sk}_{\mathsf{rpt}}$ is shared with every enclave on the same platform

– $\mathbf{HW}.\mathbf{Run}\&\mathbf{Report}(\mathbf{hdl}, in)$: 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. The report contains information about the underlying enclave signed with $\mathsf{sk}_{\mathsf{rpt}}$.
– $\mathbf{HW}.\mathbf{ReportVerifiy}(\mathbf{hdl}', \mathsf{rpt})$: Takes as input a handle hdl' and a report rpt. Uses $\mathsf{sk}_{\mathsf{rpt}}$ generated by $\mathsf{HW.Setup}$ to verify the MAC of the report.

5.1 Formal Construction

MF is divided into a *Setup phase* and four main phases; *Initialization, Key Sharing, Editing* and *Revocation*. During the *Setup phase*, all the necessary enclaves are initialized by running the $\mathsf{MF.Setup}$ algorithm. In the rest of the phases, the user is interacting with the enclaves by running one of the following algorithms: $\mathsf{MF.ABEUserKey}$, $\mathsf{MF.Store}$, $\mathsf{MF.KTStore}$, $\mathsf{MF.KeyShare}$, $\mathsf{MF.Search}$, $\mathsf{MF.Update}$, $\mathsf{MF.Delete}$ and $\mathsf{MF.Revoke}$ as described below.
Setup Phase: In this phase $\mathsf{MF.Setup}$ runs. Each entity receives a public/private key pair $(\mathsf{pk}, \mathsf{sk})$ for a CCA2 secure public cryptosystem PKE. In addition to that, the entities running in enclaves generate a signing and a verification key pair. Finally, MS runs $\mathsf{CPABE.Setup}$ to acquire the master public/private key pair $(\mathsf{MPK}, \mathsf{MSK})$. An enclave is initialized as follows:
$\mathsf{MF.Setup}(\text{“}initialize\text{”}, 1^\lambda)$: Each enclave is initialized by generating a public/private and signing/verification key pairs. To do so, the program $\mathbf{Q}_{\mathbf{ID}}^{\mathbf{init}}$ is loaded:

$\mathbf{Q}_{\mathbf{ID}}^{\mathbf{init}}$
```
– On input ($“initialize”, 1^\lambda$):
    1. Run $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{PKE.KeyGen}(1^\lambda)$.
    2. Output $\mathsf{pk}$.
  Run $\mathsf{hdl} \leftarrow \mathsf{HW.Load}(\mathsf{Q}_{\mathsf{ID}}^{\mathsf{init}})$.
```

Additionally, during the setup phase, the MS enclave loads a program $\mathbf{Q}_{\mathbf{MS}}^{\mathbf{Setup}}$ that outputs the master public/private key pair $(\mathsf{MPK}, \mathsf{MSK})$:

$\mathbf{Q}_{\mathbf{MS}}^{\mathbf{Setup}}$
```
– On input ($“initialize”, 1^\lambda$):
    1. Run $(\mathsf{MPK}, \mathsf{MSK}) \leftarrow \mathsf{PKE.KeyGen}(1^\lambda)$.
    2. Output $\mathsf{MPK}$.
  Run $\mathsf{hdl}_{\mathsf{MS}} \leftarrow \mathsf{HW.Load}(\mathsf{Q}_{\mathsf{MS}}^{\mathsf{Setup}})$.
```

Initialization Phase: As a first step , a user u_i contacts the MS enclave and requests a secret CP-ABE key. Upon reception, MS authenticates u_i and checks if the user is eligible for receiving such a key. If so, MS generates a CP-ABE key $\mathsf{sk}_{\mathcal{A},\mathsf{u_i}}$, encrypts it under $\mathsf{pk_i}$ and sends it back to u_i. This is done by running the program $\mathbf{Q}_{\mathbf{MS}}^{\mathbf{SKey}}$ in the MS enclave as shown below:
$\mathsf{MF.ABEUserKey}(\text{”}KeyRequest\text{”}, \mathsf{MSK}, u_i, cred_i, \mathcal{A})$: The master enclave program $\mathbf{Q}_{\mathbf{MS}}^{\mathbf{SKey}}$ for generating users' ABE keys is defined as follows:

$\mathbf{Q}_{\mathbf{MS}}^{\mathbf{SKey}}$

```
– On input (“$KeyRequest$”, $\mathsf{MSK}, u_i, cred_i, \mathcal{A}$):
    1. Verify that $u_i$ is registered. If not, output $\perp$.
    2. Use $\mathsf{MSK}$ and compute $\mathsf{sk}_{\mathcal{A},\mathsf{u_i}}$.
    3. Compute and output $c = \mathsf{PKE.Enc}(\mathsf{pk_i}, \mathsf{sk}_{\mathcal{A},\mathsf{u_i}})$.
  Run $c \leftarrow \mathsf{HW.Run}(\mathsf{hdl_{MS}}, (\text{“KeyRequest”}, \mathsf{MSK}, \mathsf{i}, \mathsf{cred_i}, \mathcal{A}))$.
```

After u_i successfully received $\mathsf{sk}_{\mathcal{A},\mathsf{u_i}}$ she can start using the CSP to store files remotely. To do so, she first sends a store request $StoreReq$ to the CSP. Specifically, u_i sends $m_{req} = \langle r_1, \mathsf{E}_{\mathsf{pk_{CSP}}}(cred_i), StoreReq, H(r_1||cred_i||StoreReq)\rangle$ where r_i is a random number. The CSP authenticates u_i as legitimate and sends back an authorization $Auth$ as $m_{ver} = \langle r_2, (Auth), \sigma_{CSP}(H(r_2||u_i||Auth))\rangle$. At this point, u_i generates a symmetric key $\mathsf{K_i}$ to encrypt her files and sends $m_{store} = \langle r_3, \mathsf{E}_{\mathsf{pk_{CSP}}}(\gamma_i), \mathbf{c_i}, H(r_3||\gamma_i||c_i)\rangle$ to the CSP.
$\mathsf{MF.Store}$(”$Store$”, m_{req}) : The CSP enclave program $\mathbf{Q}_{\mathbf{CSP}}^{\mathbf{Store}}$ that is responsible for storing encrypted files is defined as follows:

$\mathbf{Q}_{\mathbf{CSP}}^{\mathbf{Store}}$

```
– On input (“$StoreReq$”, $m_{req}$):
    1. Open $m_{req}$; verify the message[a]; if the verification fails, output $\perp$.
    2. Compute and output $m_{ver} = \langle r_2, (Auth), \sigma_{CSP}(H(r_2||u_i||Auth))\rangle$.
  Run $m_{ver} \leftarrow \mathsf{HW.Run}(\mathsf{hdl_{CSP}}, (\text{“}StoreReq\text{”}, m_4))$.
– On input (”$store$”, $m_{store}$):
    1. Open $m_{store}$; verify the message; if the verification fails, output $\perp$.
    2. Store $(c_i, \gamma_i)$.
  Run $\mathsf{HW.Run}(\mathsf{hdl_{CSP}}, (\text{“}store\text{”}, m_{store}))$.
```

[a] By 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.

Initialization phase concludes with $\mathsf{MF.KTStore}$ where u_i encrypts $\mathsf{K_i}$ under MPK to get $c_P^{\mathsf{K_i}}$ and sends $m_{keystore} = \left\langle \mathsf{E}_{\mathsf{pk_{KT}}}(r_4), c_P^{\mathsf{K_i}}, \sigma_i\left(H\left(r_4||c_P^{\mathsf{K_i}}\right)\right)\right\rangle$ to KT. Upon reception, KT generates a random number $r_{\mathsf{K_i}}$ that is stored next to $c_P^{\mathsf{K_i}}$.
$\mathsf{MF.KTStore}$(”$store$”, $m_{keystore}$) : The KT enclave program $\mathbf{Q}_{\mathbf{KT}}^{\mathbf{Store}}$ that stores a symmetric key $\mathsf{K_i}$ encrypted with MPK is defined as follows:

$\mathbf{Q}_{\mathbf{KT}}^{\mathbf{Store}}$

```
– On input (“$store$”, $m_{keystore}$):
    1. Open $m_{keystore}$; verify the message. If the verification fails, output $\perp$.
    2. Generate a random number $r_{\mathsf{K_i}}$.
    3. Compute $c = \mathsf{PKE.Enc}(\mathsf{pk_{u_i}}, \mathsf{r_{K_i}})$.
    4. Store $\left(\mathsf{c_p^{K_i}}, c\right)$.
  Run $\left(\mathsf{c_p^{K_i}}, c\right) \leftarrow \mathsf{HW.Run}(\mathsf{hdl_{KT}}, (\text{”}store\text{”}, m_{keystore}))$.
```

Key Sharing Phase: This phase begins with u_j executing $\mathsf{MF.KeyShare}$ to prove that is not revoked. To this end, u_j sends $m_{verReq} = \langle r_5, \mathsf{E}_{\mathsf{pk_{REV}}}(u_j), \sigma_j(r_5||u_j)\rangle$ to REV. Upon reception, REV verifies the message and checks whether $u_j \in rl$ or not. Assuming that $u_j \notin rl$ (i.e. she has not been revoked), REV replies with $m_{token} = \langle r_6, \mathsf{E}_{\mathsf{pk}_{\mathsf{KT}}}(u_j), \mathsf{E}_{\mathsf{pk}_{\mathsf{KT}}}(\tau_{KS}), \sigma_{REV}\left(H(r_6||u_j||\tau_{KS})\right)\rangle$. The user then simply forwards m_{token} to KT who verifies it. After the verification is complete KT sends $m_{key} = \left\langle (\mathsf{E}_{\mathsf{pk}_{\mathsf{CSP}}}(u_j, t)), c_p^{\mathsf{K_i}}, \sigma_{KT}\left(H(u_j||t)\right)\right\rangle$ back to u_j, where t is a timestamp declaring the time that u_j accessed $c_p^{\mathsf{K_i}}$. If u_j already received $\mathsf{K_i}$ in the past, KT will only send back the first and last components of m_{key}.

$\mathsf{MF.KeyShare}(\text{“}share\text{”}, m_4)$: REV and KT enclave programs ($\mathbf{Q_{REV}^{Ver}}$, $\mathbf{Q_{KT}^{Share}}$) that are responsible for sharing $c_p^{\mathsf{K_i}}$ are defined as follows:

$\mathbf{Q_{REV}^{Ver}}$

```
– On input (“share”, $m_{verReq}$):
    1. Open $m_{verReq}$; verify the message; if the verification fails, output $\perp$.
    2. Check if $u_j \in rl$; if so, output $\perp$.
    3. Generate $\tau_{KS}$.
    4. Compute and output $m_{token}$.
  Run $m_{token} \leftarrow \mathsf{HW.Run}(\mathsf{hdl_{REV}}, (\text{“}share\text{”}, m_{verReq}))$.
```

$\mathbf{Q_{KT}^{share}}$

```
– On input (“share”, $m_{token}$):
    1. Open $m_{token}$; verify the message; if the verification fails, output $\perp$.
    2. Decrypt $\mathsf{PKE.Enc}(\mathsf{pk_{KT}}, u_j)$ and $\mathsf{PKE.Enc}(\mathsf{pk_{KT}}, \tau_{KS})$.
    3. Compute and output $m_{key}$.
  Run $m_{key} \leftarrow \mathsf{HW.Run}(\mathsf{hdl_{KT}}, (\text{“}share\text{”}, m_{ver}))$.
```

User u_j can now run $\mathsf{MF.Search}$ to access certain files that are stored in the CSP. To do so, she locally runs $\mathsf{SSE.SearchToken}$ to generate $\tau_s(w)$ and then sends $m_{search} = \langle \mathsf{E}_{\mathsf{pk}_{\mathsf{CSP}}}(u_j, t, \tau_s(w)), \sigma_i(H(u_j||t||\tau_s(w))), \sigma_{KT}\left(H(u_j||t)\right)\rangle$ to the CSP[2]. Upon reception, CSP runs $\mathsf{SSE.Search}$.

$\mathsf{MF.Search}(\text{”}search\text{”}, m_{search},)$: The CSP enclave program $\mathbf{Q_{CSP}^{Search}}$ that is responsible for searching over the encrypted data is defined as follows:

$\mathbf{Q_{CSP}^{Search}}$

```
– On input (“search”, $m_{search}$):
    1. Open $m_{search}$; verify the message; if the verification fails, output $\perp$.
    2. Run $\mathsf{SSE.Search}(\gamma_i, c_i, \tau_s(w)) \rightarrow \mathbf{I_w}$
    3. Output $\mathbf{I}_w$.
  Run $\mathsf{HW.Run}(\mathsf{hdl_{CSP}}, (\text{“}search\text{”}, m_{search})$, which internally runs $\mathsf{SSE.Search} \rightarrow \mathbf{I}_w$.
```

Editing Phase: In this phase[3], registered users can add files to the database and data owners can also delete files. To do so, u_i executes $\mathsf{MF.Update}$ and $\mathsf{MF.Delete}$. To update the database, u_i first generates an add token by running $(\tau_a(f), c_f) \leftarrow \mathsf{SSE.AddToken}(\mathsf{K_i}, f)$. This token is sent to the CSP via $m_{add} = \langle \mathsf{E}_{\mathsf{pk}_{\mathsf{CSP}}}(u_i, t, \tau_\alpha(f), c_i, \gamma_i), \sigma_i\left(H(u_i||t||\tau_{\alpha(f)}||c_i||\gamma_i)\right), \sigma_{KT}(u_i||t)\rangle$. Finally, the CSP verifies the message and its freshness and executes $\mathsf{SSE.Add}(\gamma_i, c_i, \tau_\alpha(f), c_f) \rightarrow (\gamma_i', c_i')$.

[2] The user simply forwards the components of m_{key} to the CSP along with a search token $\tau_s(w)$.

[3] One could completely ignore the *Editing Phase* and the result would be a static MF.

$\mathsf{MF.Update}(\text{“}update\text{”}, m_{add})$: The CSP enclave program $\mathbf{Q^{Up}_{CSP}}$ for adding files to the database is defined as follows:

$\mathbf{Q^{Up}_{CSP}}$

– On input ($\text{“}update\text{”}, m_{add}$):
  1. Verify the message. If the verification fails, output $\perp$.
  2. Run $\mathsf{SSE.Add}(\gamma_i, c_i, \tau_\alpha(f), c_f) \rightarrow (\gamma'_i, c'_i)$.

  Run $\mathsf{HW.Run}(\mathsf{hld_{CSP}}, (\text{“}update\text{”}, m_{add})$, which internally runs $\mathsf{SSE.Add}(\gamma_i, c_i, \tau_\alpha(f), c_f) \rightarrow (\gamma'_i, c'_i)$.

Deletion of a file is a more complicated task. This is due to the fact that we only allow the data owner to delete files. To achieve this, u_i needs to to prove her ownership over $\mathsf{K_i}$. This can be done by requesting the random number $r_{\mathsf{K_i}}$ from KT. After u_i receives $r_{\mathsf{K_i}}$, she signs it, runs $\tau_d \leftarrow \mathsf{SSE.DeleteToken}(\mathsf{K_i}, \mathsf{f})$ and replies to KT with: $m_{delete} = \langle \mathsf{E}_{\mathsf{pk_{CSP}}}(u_i, t, \tau_d(f), \gamma'_i), \sigma_i(H(u_i||\tau_d(f)||\gamma'_i||r_{\mathsf{K_i}}\rangle$. KT verifies the message and is convinced that u_i is the owner of $\mathsf{K_i}$. Finally, KT generates a report (rpt) containing the delete token. This is sent to the CSP who proceeds with the deletion of the specified files.

$\mathsf{MF.Delete}(\text{“}request\text{”}, \sigma_i(u_i||t), c_p^{\mathsf{K_i}})$: The enclave programs $\mathbf{Q^{Del}_{CSP}}, \mathbf{Q^{Del}_{KT}}$ that are responsible for deleting files from the database are defined as follows:

$\mathbf{Q^{Del}_{KT}}$

– On input ($\text{“}request\text{”}, \sigma_i(u_i||t), c_p^{\mathsf{K_i}}$):
  1. Verify the signature. If the verification fails, output $\perp$.
  2. Get $r_{\mathsf{K_i}}$ and compute $c = \mathsf{PKE.Enc}(\mathsf{pk_{u_i}}, r_{\mathsf{K_i}})$.
  3. Output c.

  Run $c \leftarrow \mathsf{HW.Run}(\mathsf{hdl_{KT}}, (\text{“}request\text{”}, \sigma_i(u_i||t), c_p^{\mathsf{K_i}})$.

– On input ($\text{“}delete\text{”}, m_{delete}$):
  1. Open m_{delete}; verify the message and authenticate u_i as the owner of $\mathsf{K_i}$. If the verification or the authentication fail, output $\perp$.
  2. Generate and output rpt.

  Run $\mathsf{HW.Run}(\mathsf{hdl_{KT}}, (\text{“}delete\text{”}, m_{delete})$ and then $\mathsf{rpt} \leftarrow \mathsf{HW.RunReport}(\mathsf{hdl_{KT}}, (\text{“}delete\text{”}, m_{delete}))$.

$\mathbf{Q^{Del}_{CSP}}$

– On input ($\text{“}delete\text{”}, \mathsf{rpt}$):
  1. Verify rpt. If the verification fails, output $\perp$.
  2. Run $\mathsf{SSE.Delete}(\gamma'_i, c'_i, \tau_d(f)) \rightarrow (\gamma''_i, c''_i)$.

  Run $\mathsf{HW.Run}(\mathsf{hdl_{CSP}}, (\text{“}delete\text{”}, \mathsf{rpt})$ who will internally run $\mathsf{HW.ReportVerify}$ $(\mathsf{hdl_{CSP}}, \mathsf{rpt})$ and $\mathsf{SSE.Delete}(\gamma_i, c_i, \tau_d(f)) \rightarrow (\gamma''_i, c''_i)$.

Revocation Phase: To successfully run $\mathsf{MF.Revoke}$, u_i first needs to prove ownership over $\mathsf{K_i}$ by following the same steps as in $\mathsf{MF.Delete}$. When u_i signs $r_{\mathsf{K_i}}$, she sends $m_{revoke} = \left\langle r_{10}, \mathsf{E}_{\mathsf{pk_{KT}}}\left(u_i, u_j, c_P^{\mathsf{K_i}}\right), \sigma_i\left(H(u_i||u_j||c_P^{\mathsf{K_i}}||r_{\mathsf{K_i}}\right)\right\rangle$ to KT. Now that KT is convinced that u_i is the owner of $\mathsf{K_i}$, it generates rpt containing u_j's identity, which is then sent to REV, who adds u_j to rl.

$\mathsf{MF.Revoke}(\text{“}request\text{”}, \sigma_i(u_i||t), c_p^{\mathsf{K_i}})$: The enclave programs $\mathbf{Q^{Rev}_{KT}}, \mathbf{Q^{Rev}_{REV}}$ that are responsible for revoking users are defined as follows:

$\mathbf{Q^{Rev}_{KT}}$

- On input ("*request*", $\sigma_i(u_i||t), c_p^{\mathsf{K_i}}$):
  1. Verify the signature. If the verification fails, output $\perp$.
  2. Get $r_{\mathsf{K_i}}$ and compute $c = \mathsf{PKE.Enc}(\mathsf{pk_{u_i}}, r_{\mathsf{K_i}})$.
  3. Output c.

  Run $c \leftarrow \mathsf{HW.Run}(\mathsf{hdl_{KT}}, (\text{"}request\text{"}, u_i, c_p^{\mathsf{K_i}}))$.
- On input ("*revoke*", m_{revoke}):
  1. Open m_{revoke}; verify the message and authenticate u_i as the owner of $\mathsf{K_i}$. If the verification or the authentication fails, output $\perp$.
  2. Generate $r_{\mathsf{K_i}'}$ and replace it with $r_{\mathsf{K_i}}$.
  3. Generate and output rpt.

  Run $\mathsf{HW.Run}(\mathsf{hdl_{KT}}, (\text{"}revoke\text{"}, m_{revoke})$ and then $\mathsf{rpt} \leftarrow \mathsf{HW.RunReport}(\mathsf{hdl_{KT}}, (\text{"}revoke\text{"}, m_{report}))$.

$\mathbf{Q^{Rev}_{REV}}$

- On input ("*revoke*", rpt):
  1. Veirfy rpt. If the verification fails, output $\perp$.
  2. Add u_j to the revocation list rl.

  Run $\mathsf{HW.Run}(\mathsf{hdl_{REV}}, (\text{"}revoke\text{"}, \mathsf{rpt})$ who will internally run $\mathsf{HW.Report\ \ Verify}$ $(\mathsf{hdl_{REV}}, report)$.

6 Security Analysis

We construct a simulator $\mathcal{S}$ that simulates 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 $\mathcal{S}$. $\mathcal{S}$ intercepts $\mathcal{ADV}$'s communication with the real protocol and replies with simulated outputs.

Definition 3. *(Sim-Security). We consider the following experiments. In the real experiment, all algorithms run as defined in our construction while in the ideal one, $\mathcal{S}$ intercepts $\mathcal{ADV}$'s queries and replies with simulated responses.*

Real Experiment	*Ideal Experiment*
1. $\mathbf{EXP}^{real}_{MF}(1^\lambda)$:	*1.* $\mathbf{EXP}^{ideal}_{MF}(1^\lambda)$:
2. $(\mathsf{MPK}, \mathsf{MSK}) \leftarrow \mathsf{MF.Setup}(1^\lambda)$	*2.* $(\mathsf{MPK}) \leftarrow \mathcal{S}(1^\lambda)$
3. $\mathsf{sk}_{\mathcal{A},\mathsf{u_i}} \leftarrow \mathcal{ADV}^{\mathsf{MF.ABEUserKey}(\mathsf{MSK},\mathcal{A})}$	*3.* $\mathsf{sk}_{\mathcal{A},\mathsf{u_i}} \leftarrow \mathcal{ADV}^{\mathcal{S}(1^\lambda)}$
4. $ct \leftarrow \mathsf{CPABE.Enc}(\mathsf{mpk}, m)$	*4.* $ct \leftarrow \mathcal{S}(1^\lambda, 1^{\lvert m \rvert})$
5. $(\gamma, c) \leftarrow \mathcal{ADV}^{\mathsf{SSE.Enc}(\mathsf{K},\mathbf{f})}$	*5.* $(\gamma, c) \leftarrow \mathcal{ADV}^{\mathcal{S}(\mathcal{L}_{in}(\mathbf{f}))}$
6. $\mathsf{MF.Search}(\text{"}search\text{"}, m_s) \rightarrow \mathbf{I_w}$	*6.* $\mathcal{S}(\text{"}search\text{"}, m_s) \rightarrow \mathbf{I_w}$
7. $\mathsf{MF.Update}(\text{"}update\text{"}, m_{add}) \rightarrow (\gamma', c')$	*7.* $\mathcal{S}(\text{"}update\text{"}, m_{add}) \rightarrow (\gamma', c')$
8. $\mathsf{MF.Delete}(\text{"}delete\text{"}, m_{delete}) \rightarrow (\gamma', c')$	*8.* $\mathcal{S}(\text{"}delete\text{"}, m_{delete}) \rightarrow (\gamma', c')$
9. Output b	*9. Output* b'

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

$$\mathbf{EXP}^{real}_{MF}(1^\lambda) \approx \mathbf{EXP}^{ideal}_{MF}(1^\lambda)$$

Everything $\mathcal{ADV}$ observes in the real experiment can be simulated by $\mathcal{S}$. Moreover, we use an IND-CCA2 public key encryption scheme. If $\mathcal{ADV}$ can distinguish between real and ideal answers, she can also break the IND-CCA2 security. Finally, we let $\mathcal{ADV}$ 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.

Theorem 1. *Assuming that* PKE *is an IND-CCA2 secure public key cryptosystem and* Sign *is an EUF-CMA secure signature scheme then MF is a sim-secure protocol according to Definition 3.*

Proof. We start by defining the algorithms used by the simulator. Then, we will replace them with the real algorithms. Finally, the help of a Hybrid Argument we will prove that the two distributions are indistinguishable.

- $\mathsf{MF.Setup}^*$: Will only generate MPK that will be given to $\mathcal{ADV}$.
- $\mathsf{MF.ABEUserKey}^*$: Will generate a random key to be sent to the adversary. That is, when $\mathcal{ADV}$ makes a key generation query, $\mathcal{S}$ will simulate $\mathsf{CPABE.KeyGen}$ and it will output $\mathsf{sk}^*_{\mathsf{A},\mathsf{u_i}}$. This key is a random string that has the same length as the output of the real $\mathsf{MF.ABEUserKey}^*$. The key will be given to $\mathcal{ADV}$.
- $\mathsf{MF.KeyShare}^*$: In the ideal experiment, after $\mathcal{ADV}$ requests a secret key, $\mathcal{S}$ will encrypt a sequence of bits based on $\mathcal{L}_{in}$, under MPK. The ciphertext will be returned to $\mathcal{ADV}$.
- $\mathsf{MF.Search}^*$: When $\mathcal{ADV}$ generates a search token $\tau_s(w)$, $\mathcal{S}$ gets as input the leakage function $\mathcal{L}_s$ and outputs a simulated response. When $\mathcal{ADV}$ makes a search query, $\mathcal{S}$ will once again generate a simulated $\mathbf{I}^*_{\mathbf{w}}$ which will be sent back to her.
- $\mathsf{MF.Update}^*$: 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.
- $\mathsf{MF.Delete}^*$: 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.
- $\mathsf{MF.Revoke}^*$: The system does not revoke any user.

In the pre-processing phase, $\mathcal{S}$ runs $\mathsf{HW.Setup}(1^\lambda)$, just as in the real experiment, in order to acquire $\mathsf{sk_{rpt}}$. Moreover, the challenger $\mathcal{C}$ generates a symmetric key $\mathsf{K_i}$, that will be needed in order to reply to search, add and delete queries. We will now use a hybrid argument to prove that $\mathcal{ADV}$ cannot distinguish between the real and the ideal experiments.

$\boxed{\textbf{Hybrid 0}}$ MF runs normally.

$\boxed{\textbf{Hybrid 1}}$ Everything runs like in Hybrid 0, but we replace $\mathsf{MF.Setup}$ with $\mathsf{MF.Setup}^*$.

These algorithms are identical from $\mathcal{ADV}$'s perspective and as a result the hybrids are indistinguishable.

$\boxed{\textbf{Hybrid 2}}$ Everything runs like in Hybrid 1, but $\mathsf{MF.ABEUserKey}^*$ runs instead of $\mathsf{MF.ABEUserKey}$.

Hybrid 2 is indistinguishable from Hybrid 1 because nothing changes from $\mathcal{ADV}$'s point of view.

After Hybrid 2, we have ensured that $\mathcal{ADV}$ has followed all the required steps in order to ask for $\mathsf{K_i}$. We are now ready to replace $\mathsf{MF.KeyShare}$ with $\mathsf{MF.KeyShare}^*$.

$\boxed{\textbf{Hybrid 3}}$ Like Hybrid 2, but $\mathsf{MF.KeyShare}^*$ runs instead of $\mathsf{MF.KeyShare}$. Also, the algorithm outputs $\perp$ if $\mathsf{HW.Run}$ is queried with $(\mathsf{hdl_{KT}}, (\text{"}share\text{"},\ m_{token}))$ but $\mathcal{ADV}$ never contacts REV.

Lemma 1. *Hybrid 3 is indistinguishable from Hybrid 2.*

Proof. Replacing the two algorithms, does not change from $\mathcal{ADV}$'s perspective. If $\mathcal{ADV}$ can generate m_{token}, then she can forge REV's signature. Given the security of the signature scheme, this can only happen with negligible probability. So $\mathcal{ADV}$ can distinguish between the Hybrids with negligible probability. □

At this point, $\mathcal{ADV}$ has received what she thinks is a valid $\mathsf{K_i}$. The simulator now gets access to all leakage functions $\mathcal{L}$ from the SSE scheme.

$\boxed{\textbf{Hybrid 4}}$ Like Hybrid 3, but when $\mathsf{HW.Run}$ is queried with $(\mathsf{hdl_{CSP}}, (\text{"}search\text{"}, m_{search}))$, $\mathcal{S}$ is given the leakage function $\mathcal{L}_\mathcal{S}$ and generates $\mathbf{I}^*_\mathbf{w}$ which is then sent to the user.

Lemma 2. *Hybrid 4 is indistinguishable from Hybrid 3.*

Proof. 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 $\mathcal{S}$ receives m_{search}, it will generate a sequence of file identifiers $\mathbf{I}^*_\mathbf{w}$ that will be send back to $\mathcal{ADV}$. $\mathcal{ADV}$ cannot distinguish between the real and the ideal experiment since she receives a sequence of files corresponding to a search token that was also simulated by $\mathcal{S}$. Moreover, if $\mathcal{ADV}$ manages to generate m_{search} without having contacted KT earlier, then she can also forge KT's signature. However, this can only happen with negligible probability, and as a result $\mathcal{ADV}$ can only distinguish between hybrids 4 and 3 with negligible probability. □

$\boxed{\textbf{Hybrid 5}}$ Like Hybrid 4, but when $\mathsf{HW.Run}$ is queried with $(\mathsf{hdl_{CSP}}, (\text{“}update\text{”}, m_{add}))$, $\mathcal{S}$ is given the leakage function $\mathcal{L}_a$ and tricks $\mathcal{ADV}$ into thinking that she updated the database.

Lemma 3. *Hybrid 5 is indistinguishable form Hybrid 4.*

Proof. 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.

Moreover, similar to the previous Hybrid, if $\mathcal{ADV}$ can generate m_{add} without having contacted KT, then she can also forge KT's signature – which can only happen with negligible probability. Hence, $\mathcal{ADV}$ can only distinguish between hybrids 5 and 4 with negligible probability. $\square$

$\boxed{\textbf{Hybrid 6}}$ Like Hybrid 5, but when $\mathsf{HW.Run}$ is queried with $(\mathsf{hdl}_{\mathsf{KT}}, (\text{"}delete\text{"}, m_{del}))$, $\mathcal{S}$ is given the leakage function $\mathcal{L}_d$ and tricks $\mathcal{ADV}$ into thinking that she deleted a certain file from the database. Moreover, $\mathcal{S}$ outputs $\perp$, if $\mathsf{ReportVerify}$ is queried with $(\mathsf{hdl}_{\mathsf{CSP}}, \mathsf{rpt})$ for a report that was not generated by executing $\mathsf{HW.RunReport}(\mathsf{hdl}_{\mathsf{KT}}, (\text{“}delete\text{”}, m_{delete}))$.

Proof. 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. Moreover, if $\mathcal{ADV}$ can query $\mathsf{HW.ReportVerify}$ with $(\mathsf{hdl}_{\mathsf{CSP}}, \mathsf{rpt})$, for a rpt that was not generated by KT, then $\mathcal{ADV}$ can produce a valid MAC which can only happen with negligible probability since she does not know $\mathsf{sk}_{\mathsf{rpt}}$. Thus, $\mathcal{ADV}$ can only distinguish between Hybrids 5 and 6 with negligible probability. $\square$

$\boxed{\textbf{Hybrid 7}}$ Like Hybrid 6 but instead of $\mathsf{MF.Revoke}$, $\mathcal{S}$ executes $\mathsf{MF.Revoke}^*$.

The hybrids are indistinguishable since no one can access the content of the revocation list and as a result nothing changes from $\mathcal{ADV}$'s point of view.

With this Hybrid our proof is complete. We managed to replace the expected outputs with simulated responses in a way that $\mathcal{ADV}$ cannot distinguish between the real and the ideal experiment. $\square$

6.1 SGX Security

Recent works [3, 17, 8, 16] have shown that SGX is vulnerable to software attacks. However, according to [5], these attacks can be prevented if the programs running in the enclaves are data-obvious. 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 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 $c_p^{\mathsf{K}_i}$ are data-obvious. Moreover, rl is stored in plaintext and every entry in the list is padded to achieve same length.

7 Conclusion

In this paper, we proposed MF, a hybrid encryption scheme that combines *both* SSE and ABE in a way that the main advantages of each encryption technique are used. The proposed scheme enables clients to search over encrypted data by using an SSE scheme, while the symmetric key required for the decryption is protected via a Ciphertext-Policy Attribute-Based Encryption scheme. Moreover, our construction supports the revocation of users by utilizing the functionality provided by SGX. In contrast to recent works, the revocation mechanism has been separated from the actual ABE scheme and is exclusively based on the utilization of trusted SGX enclaves.

References

1. 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)
2. Boyko, V.: On the security properties of oaep as an all-or-nothing transform. In: Wiener, M. (ed.) Advances in Cryptology — CRYPTO' 99. pp. 503–518. Springer Berlin Heidelberg, Berlin, Heidelberg (1999)
3. Costan, V., Devadas, S.: Intel sgx explained. Cryptology ePrint Archive, Report 2016/086 (2016), `https://eprint.iacr.org/2016/086`
4. Dowsley, R., Michalas, A., Nagel, M., Paladi, N.: A survey on design and implementation of protected searchable data in the cloud. Computer Science Review (2017)
5. 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 (2017)
6. Guo, W., Dong, X., Cao, Z., Shen, J.: Efficient attribute-based searchable encryption on cloud storage. Journal of Physics: Conference Series (2018)
7. Kamara, S., Papamanthou, C., Roeder, T.: Dynamic searchable symmetric encryption. pp. 965–976 (2012)
8. Lee, S., Shih, M., Gera, P., Kim, T., Kim, H., Peinado, M.: Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In: 26th USENIX Security Symposium, BC, Canada, August 16-18, 2017. pp. 557–574 (2017)
9. Liu, J.K., Yuen, T.H., Zhang, P., Liang, K.: Time-based direct revocable ciphertext-policy attribute-based encryption with short revocation list. Cryptology ePrint Archive, Report 2018/330 (2018), `https://eprint.iacr.org/2018/330`
10. 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)
11. Michalas, A.: The lord of the shares: Combining attribute-based encryption and searchable encryption for flexible data sharing. In: Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing. pp. 146–155. SAC '19, ACM, New York, NY, USA (2019). https://doi.org/10.1145/3297280.3297297, `http://doi.acm.org/10.1145/3297280.3297297`
12. Microsoft: Microsoft Security Intelligence Report (2017)
13. Myers, S., Shull, A.: Practical revocation and key rotation. In: Smart, N.P. (ed.) Topics in Cryptology – CT-RSA 2018. pp. 157–178. Springer, Cham (2018)
14. 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
15. 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)
16. Weichbrodt, N., Kurmus, A., Pietzuch, P.R., Kapitza, R.: Asyncshock: Exploiting synchronisation bugs in intel SGX enclaves. In: Computer Security - ESORICS 2016 - 21st European Symposium on Research in Computer Security, Heraklion, Greece, September 26-30, 2016, Proceedings, Part I. pp. 440–457 (2016)
17. Xu, Y., Cui, W., Peinado, M.: Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In: Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland). IEEE (May 2015)