Nowhere to Leak: Forward and Backward Private Symmetric Searchable Encryption in the Multi-Client Setting (Extended Version)

Alexandros Bakas and Antonis Michalas

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

Abstract. Symmetric Searchable Encryption (SSE) allows users to outsource encrypted data to a possibly untrusted remote location while simultaneously being able to perform keyword search directly through the stored ciphertexts. An ideal SSE scheme should reveal no information about the content of the encrypted information nor about the searched keywords and their mapping to the stored files. However, most of the existing SSE schemes fail to fulfil this property since in every search query, some information potentially valuable to a malicious adversary is leaked. The leakage becomes even bigger if the underlying SSE scheme is dynamic. In this paper, we *minimize the leaked information* by proposing a *forward and backward private* SSE scheme in a multi-client setting. Our construction achieves optimal search and update costs. In contrast to many recent works, each search query only requires one round of interaction between a user and the cloud service provider. In order to guarantee the security and privacy of the scheme and support the multi-client model (i.e. synchronization between users), we exploit the functionality offered by AMD's Secure Encrypted Virtualization (SEV).

Keywords: Backward Privacy · Cloud Security · Forward Privacy · Multi-Client · Symmetric Searchable Encryption

1 Introduction

Symmetric Searchable Encryption (SSE) is an encryption technique with remarkable capabilities. More precisely, using an SSE scheme users can encrypt their data in such a way that the encrypted data can still be searched. The most straight-forward application of an SSE scheme is the design of secure cloud storage services, as this encryption method helps users protecting their data from both external and *internal* attacks (e.g. a malicious administrator). In particular, in an SSE scheme the encryption algorithm takes as input a sequence of files and outputs a sequence of ciphertexts and an encrypted index. At a later point, a user can generate a search token for a specific keyword w. Given the search token, the encrypted index and the collection of ciphertexts, the CSP can then locate all files that contain the requested keyword. What is fascinating about

SSE schemes, is that the CSP can do so without knowing neither the underlying keyword, nor the content of the files.

An ideal SSE scheme should reveal no information about the content of the encrypted information nor about the searched keywords and their mapping to the stored files. However, most SSE schemes fail to fulfill this property since in every search or update query, some information potentially valuable to a malicious adversary is leaked. In the early years of SSE, researchers utilized techniques such as oblivious RAM (ORAM). However, according to [24], adopting such a technique is even less efficient than downloading and decrypting the entire database locally. As a result, researchers have come to a silent agreement that *"nothing should be revealed beyond some well defined and "reasonable" leakage"* [4].

Leaked information in SSE schemes has become a problem of paramount importance since it is the main factor in defining the overall level of security. In works such as [12] and [18] it is pointed out that even a small leakage can lead to several privacy attacks. These works were further extended in [28] where the authors assumed that an active adversary can perform file-injection attacks and record the output. This "new" ability allowed the adversary to recover information about past queries only after ten file insertions. This result led researchers to design *forward private* SSE schemes [7, 9, 15]. Forward privacy is a notion introduced in [26] and guarantees that that newly added files cannot be related to past search queries. While forward privacy is a very important property, unfortunately it has been shown to also be vulnerable to certain file-injection attacks [28].

While forward privacy secures the content of a past query, its binary property, *backward privacy*, ensures the privacy of future queries. Backward privacy was formalized in [10] where three different flavors were defined. A backward private SSE scheme ensures that queries do not reveal their association with deleted documents. To the best of our knowledge, there are only a handful of backward private schemes per flavor where *none* of them supports the multi-client model.

Our Contribution: We extend the work proposed in [15] by constructing a forward and backward private Dynamic SSE scheme that supports a multi-client model. We deal with the problem of synchronization between multiple clients by utilizing the functionality offered by AMD's SEV [2]. In contrast to many works in the area, we assume that the database consists of files with multiple keywords and not entries of the form $\{id(f_i), w_j\}$. Naturally, this results in higher computational cost whenever a new file is added to the database, but it simulates a more realistic scenario. In particular, our construction:

– **Provides Forward Privacy:** The CSP stores a dictionary Dict that contains mappings between keywords and filenames ($id(f)$). Each address in the dictionary is computed using a keyword key $\mathsf{K_w}$ for a keyword w that is *deleted after every search for that specific address*. As a result, in every round, we are required to compute new addresses for each affected row of the dictionary. This is achieved with the generation of a new keyword key $\mathsf{K'_w}$. By doing so, we minimize the information the CSP gains during new file

additions, as the CSP will not be able to tell if a newly added file contains a keyword w that was searched for in the past.
- **Provides Backward Privacy:** To delete a file f_i, a user sends the a deletion request to an SEV trusted Virtual Machine. Later, and when a search query for a keyword $w_j \in f_i$ is executed, the result is filtered through the SEV-enabled VM (SVM). In particular, the SVM decrypts the result, which consists of encryptions of file identifiers, removes the ones that have been marked as deleted from the previous round, re-encrypts the rest and updates Dict accordingly. Hence, even if the nature of the operation is leaked, the CSP cannot know which entries have been deleted. *Thus, the scheme satisfies the property of backward privacy.*
- **Is Asymptotically Optimal:** The update cost is $O(m)$ and the search time is $O(\ell)$, where m is the number of unique keywords in a file and ℓ is the number of the resulted files. To delete an entry from Dict, the update cost is $O(1)$ since deletion is executed for each unique keyword separately, whenever a keyword is searched for.
- **Is Parallelizable:** The CSP maintains encryptions of the file names in a dictionary. The address of each $c_{id(f_i)}$ is calculated by the data owner, before inserting the files, and is then hashed. This results to $O(\ell)$ independent hashes for each search. Hence, if the load is distributed to p processors, we achieve optimal search cost $O(\ell/p)$. Similarly, the update cost is $O(m/p)$.

2 Related Work

Our work is based on [15] where the authors presented a Symmetric Searchable Encryption scheme with Forward Privacy, a notion first introduced in [26]. Their construction however is only forward private and limited to a single client model. In this work, we make use of the functionality offered by AMD's SEV VMs to both extend the original scheme to be backward private and to support the multi-client setting.

Another single-client forward private SSE scheme is presented in [9], where the authors designed *Sophos*. While *Sophos* achieves asymptotically optimal search and update costs, $O(\ell)$ and $O(m)$ respectively, a file addition requires $O(m)$ asymmetric cryptographic operations on the user's side. An improvement in the search time of *Sophos* is presented in [19]. Authors of [9] extended their work in [10] by designing a number of SSE schemes that are both forward and backward private. Out of those schemes, Diana_{del} and *Janus* are the most efficient but at the same time they satisfy the weakest notion of backward privacy. In particular, *Janus* achieves its security by using public puncturable encryption. *Fides* is among the first efficient backward private SSE schemes with stronger security guarantees. However, it only satisfies the single-client model. Moreover, the search operation requires two rounds of interaction while our scheme only requires one. An improvement of *Janus* is presented in [27] where authors design *Janus++*. While *Janus++* is more efficient than Janus as it is based on symmetric puncturable encryption, Janus++ can only achieve the same security level as Janus.

An SGX-based forward/backward private scheme called *Bunker-B* is presented in [4]. Our construction is similar to that in the sense that we use a trusted execution environment (TEE) to reduce the number of required rounds to one. However, as in the case of *Sophos* and *Fides*, *Bunker-B* only supports the single-client model. Moreover, we believe that SGX in not a suitable TEE for a cloud-based service due to its limitations.

Comparison						
Scheme	MC	FP	BP	Search Time	Update Time	Client Storage
Etemad et al. [15]	✗	✓	✗	$O(\ell/p)$	$O(m/p)$	$O(m+n)$
HardIDX	✗	✗	✗	$O(\log k)$	-	None
Sophos	✗	✓	✗	$O(\ell)$	$O(m)$	$O(m)$
Kim et al. [19]	✗	✓	✗	$O(a_w)$	$O(m)$	$O(m)$
Bunker-B	✗	✓	Type-II	$O(\ell)$	$O(1)$	$O(mlogn)$
Fides	✗	✓	Type-II	$O(\ell)$	$O(1)$	$O(mlogn)$
Diana$_{del}$	✗	✓	Type-III	$O(\ell)$	$O(logN)$	$O(mlogn)$
Janus	✗	✓	Type-III	$O(\ell d)t_{PE.Dec}$	$O(\ell)(t_{PE.Enc} \vee t_{PE.Dec})$	$O(mlogn)$
Janus++	✗	✓	Type-III	$O(\ell d)t_{SPE.Dec}$	$O(\ell)(t_{SPE.Enc} \vee t_{SPE.Dec})$	$O(mlogn)$
Moneta	✗	✓	Type-I	$\widetilde{O}(a_w \log N + \log^3 N)$	$\widetilde{O}(\log^2 N)$	$O(1)$
Orion	✗	✓	Type-I	$O(\ell \log N^2)$	$O(logN^2)$	$O(1)$
Ours	✓	✓	Type-II	$O(\ell/p)$	$O(m/p)$	None

Table 1: N: number of (w, id) pairs, n: total number of files, m: total number of keywords, p: number of processors, k: number of keys, a_w: number of updates matching w, d_w: number of deleted entries matching w, ℓ: result size ($\ell = a_w - d_w$), $(t_{PE.Enc}, t_{PE.Dec})$: encryption and decryption times for a public pancturable encryption scheme, $(t_{SPE.Enc}, t_{SPE.Dec})$: encryption and decryption times for a symmetric pancturable encryption scheme MC: Multi-Client, FP: Forward Privacy, BP: Backward Privacy. $\widetilde{O}$ notation hides polylogarithmic factors.

In [11] authors presented *HardIDX*, a scheme that also supports range queries with the use of SGX [13] based on B^+ trees. *HardIDX* minimizes the leakage by hiding the search pattern but at the same time, their construction is *static*. As a result, it does *not* support file insertions after the generation of the initial index. Therefore, even though the scheme achieves logarithmic search cost, a direct comparison to our scheme is *not* possible.

ORAM-based approaches: The first forward private SSE scheme was proposed in [26], where the authors presented an ORAM-based construction. More recently, in [10], authors proposed *Moneta*, an SSE scheme that achieves the strongest level of backward privacy, but at the cost of efficiency. *Moneta* is based on the TWORAM construction presented in [16]. However, as argued in [17], the use of TWORAM renders *Moneta* impractical for realistic scenarios and the scheme can serve mostly as a theoretical result for the feasibility of stronger backward private schemes schemes. Finally, in [17], another ORAM-bsed scheme, *Orion*, is proposed. While *Orion* outperforms *Moneta*, the number of interactions between the user and the CSP depends on the size of the encrypted database.

In table 1 we see a comparison of the aforementioned schemes to our construction.

3 Background

In this section, we introduce our notation, and we provide a formal definition of a dynamic SSE scheme along with the necessary security definitions.

Notation Let $\mathcal{X}$ be a set. We use $x \leftarrow \mathcal{X}$ if x is sampled uniformly from $\mathcal{X}$ and $x \overset{\$}{\leftarrow} \mathcal{X}$, if x is chosen uniformly at random. If $\mathcal{X}$ and $\mathcal{Y}$ are two sets, we denote by $[\mathcal{X}, \mathcal{Y}]$ all the functions from $\mathcal{X}$ to $\mathcal{Y}$ and by $\overline{[\mathcal{X}, \mathcal{Y}]}$ all the injective functions from $\mathcal{X}$ to $\mathcal{Y}$. $R(\cdot)$ is a truly random function and $R^{-1}(\cdot)$ its inverse. A function $negl(\cdot)$ is called negligible, iff $\forall c \in \mathbb{N}, \exists n_0 \in \mathbb{N} : \forall n \geq n_0, negl(n) < n^{-c}$. If $s(n)$ is a string of length n, we denote by $\overline{s}(l)$ its prefix of length l and by $\underline{s}(l)$, its suffix of length l, where $l < n$. A file collection is represented as $\mathbf{f} = (f_1, \ldots, f_z)$ while the corresponding collection of ciphertexts is $\mathbf{c} = (c_{f_1}, \ldots, c_{f_z})$. The universe of keywords is $\mathcal{W} = (w_1, \ldots, w_k)$ and the distinct keywords in a file f_i are $w_i = (w_{i_1}, \ldots, w_{i_\ell})$. An invertible pseudorandom function [8] is defined as follows:

Definition 1 (Invertible Pseudorandom Function (IPRF)). *An IRPF with key-space $\mathcal{K}$, domain of definition $\mathcal{X}$ and range $\mathcal{Y}$ consists of two functions $G : (\mathcal{K} \times \mathcal{X}) \rightarrow \mathcal{Y}$ and $G^{-1} : (\mathcal{K} \times \mathcal{Y}) \rightarrow \mathcal{X} \cup \{\perp\}$. Moreover, let $\mathsf{G.Gen}(1^\lambda)$ be an algorithm that given the security parameter λ, outputs $k \in \mathcal{K}$. The functions G and G^{-1} satisfy the following properties:*

1. *$G^{-1}(k, G(k, x)) = x, \forall x \in \mathcal{X}$.*
2. *$G^{-1}(k, y) = \perp$ if y is not an image of G.*
3. *G and G^{-1} can be efficiently computed by deterministic polynomial algorithms.*
4. *$G(k, \cdot) \in \overline{[\mathcal{X}, \mathcal{Y}]}, G^{-1}(k, \cdot) \in \overline{[\mathcal{Y}, \mathcal{X}]}$*

The function $G : (\mathcal{K} \times \mathcal{X}) \rightarrow \mathcal{Y}$ is an IPRF if $\forall$ PPT adversary $\mathcal{A}$:

$$\begin{aligned} &|Pr[k \leftarrow \mathsf{G.Gen}(1^\lambda) : \mathcal{A}^{G(k,\cdot), G^{-1}(k,\cdot)}(1^\lambda) = 1] - \\ &\quad Pr[k' \overset{\$}{\leftarrow} \overline{[\mathcal{X}, \mathcal{Y}]} : \mathcal{A}^{R(\cdot), R^{-1}(\cdot)}(1^\lambda) = 1| = negl(\lambda) \end{aligned} \tag{1}$$

Definition 2 (DSSE Scheme). *A Dynamic Symmetric Searchable Encryption (DSSE) scheme consists of the following PPT algorithms:*

- $(\mathsf{In}_{\mathsf{CSP}}, \mathbf{c})(\mathsf{In}_{\mathsf{TA}})(\mathsf{K}) \leftarrow \mathsf{Setup}(\lambda, \mathbf{f})$*: The data owners runs this algorithm to generate the key* K *as well as the CSP index* $\mathsf{In}_{\mathsf{CSP}}$ *and a collection of ciphertexts* $\mathbf{c}$ *that will be sent to the CSP. Additionally, the index* $\mathsf{In}_{\mathsf{TA}}$ *that is stored on a remote location is generated.*
- $(\mathsf{In}'_{\mathsf{CSP}}, R_{w_{i_j}})(\mathsf{In}'_{\mathsf{TA}}) \leftarrow \mathsf{Search}(\mathsf{K}, w_{i_j}, \mathsf{In}_{\mathsf{TA}})(\mathsf{In}_{\mathsf{CSP}}, \mathbf{c})$*. This algorithm is executed by a user in order to search for all files f_i containing a specific keyword w_{i_j}. The indexes are updated and the CSP also returns to the user the ciphertexts of the files that contain w_{i_j}.*

– $(\mathsf{In}'_{\mathsf{CSP}}, \mathbf{c}')(\mathsf{In}'_{\mathsf{TA}}) \leftarrow \mathsf{Update}(\mathsf{K}, f_i, \mathsf{In}_{\mathsf{TA}})(\mathsf{In}_{\mathsf{CSP}}, \mathbf{c}, op), op \in \{\text{add}, \text{delete}\}$*: A user is running this algorithm to update the collection of ciphertexts* $\mathbf{c}$*. Based on the value of op, a new file is either added to the collection or an existing one is deleted.*

The Setup algorithms do not require any interaction. However, the rest of the algorithms require synchronization between the different entities.

Security Definitions To capture the notion of security in a searchable encryption scheme we make use of the real experiment against ideal experiment formalization. In particular, in the real experiment the adversary $\mathcal{ADV}$ observes the algorithms being executed honestly, while in the ideal experiment a simulator $\mathcal{S}$ simulates all the functionalities of the SSE scheme based on explicit leakage. The leakage is formalized by a function $\mathcal{L}$ such that $\mathcal{L} = (\mathcal{L}_{stp}, \mathcal{L}_{search}, \mathcal{L}_{update})$ where each component corresponds to the leakage associated with the setup, search and update operations. The idea is the following: $\mathcal{ADV}$ has full control of the client. Thus, she can trigger any operation. $\mathcal{ADV}$ issues a polynomial number of queries, and for each query she records the output. The scheme is said to be $\mathcal{L}$-adaptively secure if $\mathcal{ADV}$ cannot distinguish between the real and the ideal experiments.

Definition 3. *($\mathcal{L}$-Adaptive Security of DSSE) Let DSSE =* (Setup, Search, Update) *be a dynamic symmetric searchable encryption scheme and $\mathcal{L} = (\mathcal{L}_{stp}, \mathcal{L}_{search}, \mathcal{L}_{update})$ be the leakage function of the DSSE scheme. We consider the following experiments between an adversary $\mathcal{ADV}$ and a challenger $\mathcal{C}$:*

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

$\mathcal{ADV}$ outputs a set of files $\mathbf{f}$. $\mathcal{C}$ generates a key K*, and runs* Setup*. $\mathcal{ADV}$ then makes a polynomial number of adaptive queries $q = \{w, f_1, f_2\}$ such that $f_1 \notin \mathbf{f}$ and $f_2 \in \mathbf{f}$. For each q, she receives back either a search token for w, $\tau_s(w)$, an add token, τ_α, and a ciphertext for f_1 or a delete token τ_d for $\{w, f_2\}$. Finally, $\mathcal{ADV}$ outputs a bit b.*

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

$\mathcal{ADV}$ outputs a set of files $\mathbf{f}$. $\mathcal{S}$ gets $\mathcal{L}_{\mathsf{setup}}(\mathbf{f})$ to simulate Setup*. $\mathcal{ADV}$ then makes a polynomial number of adaptive queries $q = \{w, f_1, f_2\}$ such that $f_1 \notin \mathbf{f}$ and $f_2 \in \mathbf{f}$. For each q, $\mathcal{S}$ is given either $\mathcal{L}_{\mathsf{Search}}(w)$ or $\mathcal{L}_{\mathsf{update}}(f_i)$, $i \in \{1, 2\}$. $\mathcal{S}$ then simulates the tokens and, in the case of addition, a ciphertext. Finally, $\mathcal{ADV}$ outputs a bit b.*

We say that the DSSE scheme is secure if $\forall$ PPT adversary $\mathcal{ADV}$, $\exists\ \mathcal{S}$ such that:

$$|Pr[(\mathsf{Real}_{\mathcal{ADV}}) = 1] - Pr[(\mathsf{Ideal}_{\mathcal{ADV},\mathcal{S}}) = 1]| \leq negl(\lambda) \tag{2}$$

Correctness The DSSE scheme is *correct* iff the search protocol returns the correct result for every search query, except with negligible probability.

A DSSE scheme is said to be *forward private* if for all file insertions that take place after the successful execution of the Setup algorithm, the leakage is limited to the size of the file, and the number of unique keywords contained in it. On the other hand, a DSSE scheme is said to be *backward private* if whenever a keyword/document pair $(w, id(f))$ is added into the database and then deleted, subsequent search queries for w do not reveal $id(f)$. More formally:

Definition 4 (Forward Privacy). *An $\mathcal{L}$-adaptively SSE scheme is forward private iff the leakage function $\mathcal{L}_{Update}$ can be written as:*

$$\mathcal{L}_{\mathsf{update}}(op, id(f)) = \mathcal{L}'(op, \#w \in f) \tag{3}$$

Where $\mathcal{L}'$ is a stateless function.

Definition 5 (Backward Privacy). *There are three different flavors of backward privacy (listed in decreasing strength):*

- *Type-I: Backward Privacy with insertion pattern leaks the documents currently matching w and when they were inserted i.e. their timestamps $TimeDB(w)$.*
- *Type-II: Backward Privacy with update pattern leaks the documents currently matching w, $TimeDB(w)$ and a list of timestamps $Updates(w)$ denoting when the updates on w happened.*
- *Type-III: Weak Backward Privacy leaks the documents currently matching they keyword w, $TimeDB(w)$ and $DelHist(w)$, where $DelHist(w)$ reveals the timestamps of the delete updates on w together with the corresponding entries that they remove.*

Our construction satisfies Type-II backward privacy. In particular:

Definition 6. *An $\mathcal{L}$-adaptively SSE scheme is update pattern revealing backward private iff the search and update leakage functions $\mathcal{L}_{\mathsf{search}}, \mathcal{L}_{\mathsf{update}}$ can be written as:*

$$\begin{aligned} \mathcal{L}_{\mathsf{search}}(w) &= \mathcal{L}'(\text{TimeDB(w)}, \text{Updates(w)}) \\ \mathcal{L}_{\mathsf{update}}(op, w, id) &= \mathcal{L}''(op, w_i) \end{aligned} \tag{4}$$

Where the functions $\mathcal{L}'$ and $\mathcal{L}''$ are stateless.

Finally, the leakage function $\mathcal{L}_{stp}$ associated with the setup operation is formalised as follows:

$$\mathcal{L}_{stp} = (N, n, c_{id(f_i)}), \forall f_i \in \mathbf{f} \tag{5}$$

Where N is the total size of all the (keyword/filename) pairs, and n is the total number of files in the collection $\mathbf{f}$

Threat Model Our threat model is similar to the one described in [25] which is based on the Dolev-Yao adversarial model [14]. Furthermore, we assume that the adversary $\mathcal{ADV}$ is allowed to provide SVMs with inputs of her choice and record the output. This assumption significantly strengthens $\mathcal{ADV}$ since we need to ensure that only honest attested programs with correct inputs will run in the SVMs.

4 Architecture

In this section, we introduce the system model by describing the entities participating in our construction.

Users The users in our system model are mainly classified into two categories: data owners and simple registered users that they have not yet upload any data to the CSP. A data owner first needs to locally parse all the data that wishes to upload to the CSP. During this process, she generates three different indexes:

1. $\mathsf{No.Files[w]}$ which contains a hash of each keyword w along with the number of files that w can be found at
2. $\mathsf{No.Search[w]}$, which contains the number of times a keyword w has been searched by a user.
3. Dict a dictionary that maintains a mapping between keywords and encrypted filenames.

Both $\mathsf{No.Files[w]}$ and $\mathsf{No.Search[w]}$ are of size $O(m)$, where m is the total number of keywords while the size of Dict is $O(N) = O(nm)$, where n is the total number of files. To allow the rest of the users to search over her data, the data owner sends both $\mathsf{No.Files}[w]$ and $\mathsf{No.Search}[w]$ while Dict is outsourced to a CSP.

Cloud Service Provider (CSP) The CSP is responsible for storing:

1. The ciphertexts
2. A dictionary Dict generated by a data owner. Each address on the dictionary is computed using a different key $\mathsf{K_w}$. Hence, given this key and the $\mathsf{No.Files}[w]$ for a keyword w, the CSP can locate all the files that contain w.

Trusted Authority (TA) TA is mainly responsible for storing the $\mathsf{No.Files[w]}$ and $\mathsf{No.Search[w]}$ indexes generated by the data owner. For a registered user to create a consistent search token for a keyword w, she must first contact TA in order to get access to the corresponding $\mathsf{No.Files[w]}$ and $\mathsf{No.Search[w]}$ values. TA is alsi SEV-enabled.

Deletion Authority (DelAuth) DelAuth is responsible for the deletion of files. Every time a user performs a search operation, the CSP forwards the result R to DelAuth. DelAuth decrypts the result, removes the Dict entries to be deleted and then re-encrypts the remaining filenames and sends them back to the CSP. Like the CSP and TA, DelAuth is also SEV enabled.

TA and DelAuth can be individual entities. For simplicity, we will assume that they are part of the same host.

SEV . We provide a brief description of the main SEV functionalities. More details can be found in [2].
Isolation: SEV is a security feature that mainly addresses software attacks by isolating VMs from the hypervisor or other VMs on the same platform. It uses AES128 to encrypts the VM's memory space with an encryption key that is unique for each VM. The encryption keys are handled by the AMD secure processor which lies on the System on a Chip (SoC). SEV is particularly applicable to cloud services where VMs should not trust the hypervisor and administrator of their host system.
Attestation: Once the VM save area is encrypted, the hypervisor issues a LAUNCH_MEASURE command to produce a measurement of the data encrypted by the launch flow. This measurement is sent back to the guest owner (a user or another VM) as a receipt. Upon reception of the receipt, the guest owner can start providing the VM with confidential information. Since the guest owner knows the initial contents of the VM at boot, the attestation measurement can be verified by comparing it to what the guest owner expects.
Why SEV? The main advantages of SEV in comparison to its main competitor -Intel SGX- are *(1)* memory size, *(2)* efficiency and *(3)* No SDK or code refactoring are required. In particular, SGX allocates only 128MB of memory for software and applications and thus, making it a good candidate for microtranscations and login services. However, SEV's memory is up to the available RAM and hence, making it a perfect fit for securing complex applications. Moreover, in situations where many calls are required, like in the case of a multi-client cloud service, SEV is known to be much faster and efficient than SGX. The above are summarized in Table 2. More information can be found in [23].

TEE	**Memory Size**	**SDK**	**Attestation**	**Code Refactoring**
SEV	Up to Available System Ram	Not Required	Through AMD Secure Processor	Not Required
SGX	Up to 128MB	Required	Through Intel Remote Attestation Protocol	Major Refactoring Required

Table 2: SEV-SGX Comparison

The main drawback of SEV is that it is known for not offering memory integrity, which can lead to replay attacks by a malicious hypervisor. However, as of January 2020 AMD has launched an updated version of SEV, called SEV-SNP, which further strengthens VM's isolation with memory integrity [3].

5 Nowhere to Leak

Before we proceed to the formal description of our scheme, we present a high-level overview.

Workflow: We assume that a data owner u_i has encrypted a collection of files $\mathbf{f}$ under an SSE key $\mathsf{K}_{\mathsf{SKE}}$ and stored them on the CSP. Additionally, u_i allows another user u_k to access her files in $\mathbf{f}$ by sharing $\mathsf{K}_{\mathsf{SKE}}$[1]. Now, u_k can search directly on u_i's encrypted data. To do so, u_k first hashes the keyword w that wishes to search for and sends $h(w)$ to TA. Upon reception, TA retrieves $\mathsf{No.Files}[w]$ and $\mathsf{No.Search}[w]$ based on which, it can create the search token $\tau_s(w)$. The search token is then forwarded to the CSP, who uses it to find all the Dict entries associated with w. The output is stored in a list R_w. This list is then sent to u_k through DelAuth. The reason for sending this through DelAuth is to provide our scheme with backward privacy. More precisely, if an authorized user wishes to delete a document f_n, she sends $id(f_n)$ to DelAuth where it will be stored in a *deletion list*. On a *later* round, if $id(f_n)$ appears in a search result, DelAuth will delete the corresponding entry, re-encrypt the rest of the elements in the result list, and re-insert them in Dict.

Formal Construction: Our construction constitutes of three different algorithms, namely $\mathsf{Setup}, \mathsf{Search}$ and Update. Let $G : \{0,1\}^\lambda \times \{0,1\}^* \rightarrow \{0,1\}^*$ be an IPRF. Moreover, let $\mathsf{SKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be an IND-CPA secure symmetric key cryptosystem and $h = \{0,1\}^* \rightarrow \{0,1\}^\lambda$ be a cryptographic hash function.

Setup A data owner u_i generates the secret key K required for our construction. Then, to execute the Setup algorithm, she internally runs the Update algorithm (Algorithm 3) with op = add for every file in her collection $\mathbf{f}$. By doing this she processes all of her files $f_i \in \mathbf{f}$, extracts each ***unique*** keyword $w_{i_j} \in f_i$ and computes $\mathsf{K}_{\mathsf{w_{i_j}}} = G(\mathsf{K}_\mathsf{G}, h(w_{i_j})||\mathsf{No.Search}[\mathsf{w_{i_j}}])$. Using this key, she computes the addresses for Dict as $addr_{w_{i_j}} = h(\mathsf{K}_{\mathsf{w_{i_j}}}, \mathsf{No.Files}[\mathsf{w_{i_j}}])$ and using $\mathsf{K}_{\mathsf{SKE}}$ she computes the values $val_{w_{i_j}} = \mathsf{Enc}(\mathsf{K}_{\mathsf{SKE}}, id(f_i)||\mathsf{No.Files}[\mathsf{w_{i_j}}])$. Each $\{addr_{w_{i_j}}, val_{w_{i_j}}\}$ pair is stored in a list $\mathsf{Map_i}$. Finally, u_i encrypts f_i with $\mathsf{K}_{\mathsf{SKE}}$ and stores c_{f_i} and $\mathsf{Map_i}$ into a list AllMap that will be sent to the CSP who stores it in Dict.

[1] The key sharing protocol is out of the scope of this paper. Some constructions that squarely fit our scheme are presented in [5, 6, 20, 22]

Algorithm 1 Setup

```
1: $\mathsf{K}_G \leftarrow \mathsf{GenIPRF}(1^\lambda)$        ▷ For the IPRF G
2: $\mathsf{K}_{\mathsf{SKE}} \leftarrow \mathsf{SKE.Gen}(1^\lambda)$
3: return $\mathsf{K} = (\mathsf{K}_G, \mathsf{K}_{\mathsf{SKE}})$
4: Send $\mathsf{K}_G$ to the TA and $\mathsf{K}$ to DelAuth
5: $\mathbf{c} = \{\}$
6: AllMap $= \{\}$
7: for all $f_i$ do
8:     Run Update (Algorithm 3) with $op =$ add to generate $c_{f_i}$ and Map$_i$  ▷ Results are NOT sent to the CSP
9:     $\mathbf{c} = \mathbf{c} \cup c_{f_i}$
10:    AllMap $= [\{\text{AllMap} \cup \text{Map}_i\},\ c_{id(f_i)}]$
11: Send (AllMap, $\mathbf{c}$) to the CSP
12: Send $\mathsf{In}_{\mathsf{TA}} = \{\mathsf{No.Files}, \mathsf{No.Search}\}$ to the TA
13: CSP stores AllMap in $\mathsf{In}_{\mathsf{CSP}} = \mathsf{Dict}$
```

Search After the successful execution of the Setup algorithm and the generation of all the required indexes, u_i shares her secret key $\mathsf{K}_{\mathsf{SKE}}$ with all users she wishes to share her files with. Assume that a user u_k wishes to search over u_i's stored ciphertexts. To do so, she hashes the keyword w_j that she is looking for and sends it to the TA. Upon reception, TA retrieves the $\mathsf{No.Search[w_j]}$ and $\mathsf{No.Files[w_j]}$ values and calculates both the unique key $\mathsf{K}_{\mathsf{w_j}}$ for w_j as well as the addresses on Dict. As a next step, TA increments $\mathsf{No.Search[w_j]}$ by one and calculates the new addressees. The addresses are stored in a list L_{up}. Finally, TA outsources the search token $\tau_s(w_j) = (\mathsf{K}_{\mathsf{w_j}}, \mathsf{No.Files}[w_j], L_{up})$ to the CSP (lines 2-12 of Algorithm 2). Upon reception, the CSP locates all the encrypted filenames of the files that contain w_j and stores them in a result list R_{w_j} which is sent to u_k through DelAuth. Apart from that, the CSP sends $\tau_s(w_j)$ to the DelAuth as well[2]. Finally, DelAuth authorizes the CSP to delete all the Dict addresses associated with w_j and inserts the ones contained in L_{up}.

[2] The reason for forwarding $\tau_s(w_j)$ to DelAuth, as we will see in Algorithm 3, is to provide our scheme with backward privacy.

Algorithm 2 Search

```
1: User sends $h(w_j)$ to the TA
   TA:
2: $\mathsf{K}_{\mathsf{w_j}} = G(\mathsf{K_G}, h(w_j)||\mathsf{No.Search}[\mathsf{w_j}])$
3: $L_{up} = \{\}$
4: $\mathsf{No.Search}[\mathsf{w_j}] ++$
5: $\mathsf{K}'_{\mathsf{w_j}} = G(\mathsf{K_G}, h(w_j)||\mathsf{No.Search}[\mathsf{w_j}])$
6: for $i = 1$ to $i = \mathsf{No.Files}[\mathsf{w_j}]$ do
7:     $\mathrm{addr}'_{\mathsf{w_j}} = h(\mathsf{K_w}', i)$
8:     $L_{up} = L_{up} \cup \{\mathrm{addr}_{\mathsf{w_j}}\}$
9: Send $\tau_s(w_j) = (\mathsf{K}_{\mathsf{w_j}}, \mathsf{No.Files}[\mathsf{w_j}], L_{up})$ to the CSP
   CSP:
10: $R_{w_j} = \{\}$
11: for $i = 1$ to $i = \mathsf{No.Files}[\mathsf{w_j}]$ do
12:     $\mathrm{val}_{\mathsf{w_j}} = \mathsf{Dict}[h(\mathsf{K}_{\mathsf{w_j}}), i]$
13:     $R_{w_j} = R_{w_j} \cup \{\mathrm{val}_{\mathsf{w_j}}\}$
14:     Forward $R_{w_j}$ and $\mathsf{K}_{\mathsf{w_j}}'$ to the DelAuth
15: DelAuth:
16: Send $R_{w_j}$ to the user and an acknowledgement to the CSP.
    CSP:
17: Delete all Dict entries associated with $w_j$ and insert the addresses contained in $L_{up}$
```

Update Users that hold $\mathsf{K_{SKE}}$ can also update the data owner's encrypted database by either adding new files or deleting existing ones. To add a new file (f_i) to the existing collection of ciphertexts, u_k executes the same steps as in Algorithm 1 but only for the file she wishes to add. Moreover, for each file added, an acknowledgment needs to be sent to the TA to update the $\mathsf{No.Files}[w]$ index accordingly.

While adding a file is a relatively easy process, deleting a file is considered as a more demanding procedure. This is because since we had to follow certain steps to ensure that our scheme preserves the crucial notion of backward privacy. Assume u_k wishes to delete a file f_ℓ. To do so, she sends $id(f_\ell)$ to DelAuth where it will be stored in a list L_{TBD}. Later, when a user $u_n \in \mathcal{U}$ searches for a keyword w_{ℓ_m}, $c_{id(f_\ell)}$ will be naturally contained in the result list R. After the CSP forwards R and $\mathsf{K}_{\mathsf{w}_{\ell_m}}$ to DelAuth, DelAuth decrypts every element in R and checks whether $id(f_\ell) \in R$. If $id(f_\ell) \notin R$, DelAuth simply sends an acknowledgement to the CSP. However, if $id(f_\ell) \in R$, DelAuth inverts the IPRF G (Recall that $\mathsf{K_w} = G(\mathsf{K_G}, h(w)||\mathsf{No.Search}[\mathsf{w}]))$, retrieves $h(w_{\ell_m})$ and $\mathsf{No.Searh}[\mathsf{w}_{\ell_m}]$ and requests the $\mathsf{No.Files}[w_{\ell_m}]$ value from the TA. Upon reception of $\mathsf{No.Files}[w_{\ell_m}]$, DelAuth decreases it by one and re-encrypts every $id(f_i) \in R$ except for $id(f_\ell)$. Apart from that, it also computes the new addresses and sends them to the CSP in a list L_{del} (lines 11-25 of Algorithm 3). The CSP deletes the Dict row associated with $\{w_{\ell_m}, id(f_\ell)\}$ and inserts the new ones contained in L_{del}.

Algorithm 3 Update

```
1: if $op$ = add then
2:     Map = {}
3:     for all $w_{i_j} \in f_i$ do
4:         $\mathsf{No.Files}[w_{i_j}]++$
5:         $\mathsf{K}_{\mathsf{w}_{\mathsf{i_j}}} = G(\mathsf{K}_\mathsf{G}, h(w_{i_j})||\mathsf{No.Search}[\mathsf{w}_{\mathsf{i_j}}])$
6:         $\mathrm{addr}_{\mathrm{w}_{\mathrm{i_j}}} = h(\mathsf{K}_{\mathsf{w}_{\mathsf{i_j}}}, \mathsf{No.Files}[\mathsf{w}_{\mathsf{i_j}}])$
7:         $\mathrm{val}_{\mathrm{w}_{\mathrm{i_j}}} = \mathsf{Enc}(\mathsf{K}_{\mathsf{SKE}}, id(f_i)||\mathsf{No.Files}[\mathsf{w}_{\mathsf{i_j}}])$
8:         $\mathrm{Map} = \{\mathrm{addr}_{\mathrm{w}_{\mathrm{i_j}}}, \mathrm{val}_{\mathrm{w}_{\mathrm{i_j}}}\}$
9:         $c_{f_i} \leftarrow \mathsf{Enc}(\mathsf{K}_{\mathsf{SKE}}, f_i)$
10:        Send $\tau_\alpha(f_i) = (c_{f_i}, \mathrm{Map})$ to the CSP
11: else                                                    ▷ $op$ = delete
12:     Initiate the Search protocol for a keyword $w_j$
    After DelAuth forwards $R$ to the user:                 ▷ Line 9 of Algorithm 2
13:     for all $c_{id(f_j)} \in R$ do
14:         $\mathsf{Dec}(\mathsf{K}_{\mathsf{SKE}}, c_{id(f_j)}) \rightarrow id(f_j)$
15:     if $\exists \ell : id(f_\ell) \in L_{TBD}$ then
16:         $L_{del} = \{\}$
17:         Compute $\mathsf{No.Search}[\mathsf{w_j}]$ from $\mathsf{K}_{\mathsf{w_j}}$
18:         $\mathsf{No.Files}[\mathsf{w_j}]--$
19:         $\mathsf{No.Search}[\mathsf{w_j}]++$
20:         $\mathsf{K}_{\mathsf{w_j}} = G(\mathsf{K}_\mathsf{G}, h(w_j)||\mathsf{No.Search}[w_j])$
21:         for $i = 1$ to $i = \mathsf{No.Files}[\mathsf{w_j}] - 1$ do
22:             $\mathrm{new\_addr} = h(K_{w_j}||i)$
23:             $\mathrm{new\_value} = \mathsf{Enc}(\mathsf{K}_{\mathsf{SKE}}, id(f_i)||\mathsf{No.Files}[w_j])$
24:             $L_{del} = \{(\mathrm{new\_addr}, \mathrm{new\_value})\}$
25:     else
26:         Send $\tau_d(f_\ell) = L_{up}$ to the CSP               ▷ The list from TA
    CSP:
27:     Delete all Dict entries associated with $w_j$ and insert the addresses contained in
    $L_{del}$
    User:
28:     Update the local indexes in send an acknowledgement to the TA to update its
    indexes as well
```

6 Security Analysis

In this Section, we prove that our construction is secure against the threat model defined in Section 3. More precisely, we prove the existence of a simulator $\mathcal{S}$ that can simulate the SSE functionality, in such a way that any PPT adversary $\mathcal{ADV}$ will be able to distinguish between the real and ideal experiments with only negligible probability.

Theorem 1. *Let* $\mathsf{SKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *be a CPA-secure symmetric key encryption scheme. Moreover, let G be an IPRF and h a cryptographic hash function. Then our construction is $\mathcal{L}$-adaptively secure according to definition 3.*

Proof. We design a simulator $\mathcal{S}$ that simulates the real algorithms in a way that no PPT adversary $\mathcal{ADV}$ can distinguish between the real and ideal experiments. To prove the security of our construction, we rely on a hybrid argument in which $\mathcal{S}$ gets as input the leakage function $\mathcal{L} = (\mathcal{L}_{stp}, \mathcal{L}_{search}, \mathcal{L}_{update})$ and simulates the SSE functionalities.
Hybrid 0: This is the real experiment.
Hybrid 1: Like Hybrid 0, but instead of the Setup Algorithm, $\mathcal{S}$ is given $\mathcal{L}_{stp} = (N, n, |f_i|), \forall f_i \in \mathbf{f}$ and proceeds as follows:

Algorithm 4 Setup Simulation

```
1: $\mathsf{K_{EXP}} \leftarrow \mathsf{SKE.Gen}(1^\lambda)$
2: for $i = 1$ to $i = N$ do
3:     Simulate $(a_i, v_i)$ pairs
4:     Store all $(a_i, v_i)$ pairs in a dictionary Dict
5: for all $f_i \in \mathbf{f}$ do
6:     $c_i \leftarrow \mathsf{SKE.Enc}(\mathsf{K_{EXP}}, 0^{|\mathsf{f_i}|})$
7: Create a dictionary KeyStore to store the last $\mathsf{K}_w$ of each keyword.
8: Create a dictionary Oracle to reply to the random oracle queries.
```

The (a_i, v_i) pairs simulate the real addresses and encrypted filenames respectively. The two dictionaries have exactly the same format and size and hence, $\mathcal{ADV}$ cannot distinguish between the two. Apart from that, since we have assumed that SKE is CPA-secure, $\mathcal{ADV}$ can only distinguish between encryptions of filenames and encryptions of strings of zeroes, with negligible probability. Thus, we conclude that:

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

Hybrid 2: Like Hybrid 1, but $\mathcal{S}$ is now given $\mathcal{L}_{search} = \mathcal{L}'(\text{TimeDB(w)}, \text{Updates(w)})$ and proceeds as shown in Algorithm 5.

In particular, the $\mathsf{KeyStore}[w]$ dictionary stores the last key K_{w_j} used for each keyword w_j. Moreover, $\mathsf{Oracle}[\mathsf{K}_w][j][i]$ is used to reply consistently to queries issued by $\mathcal{ADV}$. For example, $\mathsf{Oracle}[\mathsf{K}_w][0][i]$ refers the address of a Dict entry that corresponds the $i-th$ file in the file collection. Similarly, $\mathsf{Oracle}[\mathsf{K}_w][1][i]$ refers to the masked value required to recover the filename. From the design of the simulator, we observe that the simulated token has the same size and format as the real one. Hence, no PPT adversary can distinguish between them with probability greater than negligible. Apart from that, $\mathcal{ADV}$ cannot tamper with the measurement of the SVM's during the execution of the attestation protocols. This is because tampering with this measurement would produce a different receipt than the expected one (see Section 4) and thus the protocol would abort. Hence, we conclude that:

$$|Pr[(Hybrid\ 1) = 1] - Pr[(Hybrid\ 2) = 1]| \leq negl(\lambda) \tag{7}$$

Algorithm 5 Search Token Simulation

```
1: $d = |\mathbf{f}_{w_j}|$                                   ▷ Number of files containing $w_j$
2: if $\mathsf{KeyStore}[w_j] = Null$ then                   ▷ First search for $w_j$
3:     $\mathsf{KeyStore}[w_j] \leftarrow \{0,1\}^\lambda$
4:     $\mathsf{K}_w = \mathsf{KeyStore}[w]$
5: for $i = 1$ to $i = d$ do
6:     if $\mathsf{Oracle}[\mathsf{K}_{w_j}][0][i]$ is Null then
7:         if $f_i$ is added after the Setup Algorithm then
8:             Pick a $(c_{id(f_i)}, \{a_i, v_i\})$ pair
9:         else
10:            Pick an unused $\{a_i, v_i\}$ at random
11:        $\mathsf{Oracle}[\mathsf{K}_{w_j}][0][i] = a_i$
12:        $\mathsf{Oracle}[\mathsf{K}_{w_j}][1][i] = v_i||c_{id(f_i)}$          ▷ $v_i$ is resized so that
    $|v_i||c_{id(f_i)}| = |c_{id(f_i)}||\mathsf{No.Files}[w_j]|$
13:    else
14:        $a_i = \underline{\mathsf{Oracle}[\mathsf{K}_w][0][i]}$
15:        $v_i = \overline{\mathsf{Oracle}[\mathsf{K_w}][1][i]}(|\mathsf{Oracle}[\mathsf{K_w}][1][i]| - |c_{id(f_i)}|)$     ▷ Prefix of the string
16:    Remove $a_i$ from the dictionary but keep $v_i$
17: $UpdatedVal = \{\}$                                        ▷ Generate new pairs
18: $\mathsf{K}'_{w_j} \leftarrow \{0,1\}^\lambda$
19: $\mathsf{KeyStore}[w_j] = \mathsf{K}'_{w_j}$
20: for $i = 1$ to $i = d$ do
21:     Generate new $a_i$ and match it with $v_i$ from step 16
22:     Add $(c_{id(f_i)}, \{a_i, v_i\})$ to the dictionary
23:     $UpdatedVal = UpdatedVal \cup \{c_{id(f_i)}, a_i\}$
24:     $\mathsf{Oracle}[\mathsf{K}'_{w_j}][0][i] = a_i$
25:     $\mathsf{Oracle}[\mathsf{K}'_{w_j}][1][i] = v_i||c_{id(f_i)}$
26: $\tau_s(w) = (\mathsf{K}_w, d, UpdatedVal)$
```

Hybrid 3: Like Hybrid 2. but $\mathcal{S}$ gets as input $\mathcal{L}_{\mathsf{Update}}(op, w, id) = \mathcal{L}''(op, w_i)$. Moreover, $\mathcal{S}$ maintains a list L where it stores deletion requests (encrypted filenames). $\mathcal{S}$ proceeds as shown in Algorithm 6.

Algorithm 6 Update Token Simulation

```
1: if $op = $ add then
2:     $\mathcal{L}_{add} = \{\}$
3:     for $i = 1$ to $i = \#w_i$ do
4:         Simulate $\{a_i, v_i\}$ pairs
5:         Add $(c_{id(f)}, \{a_i, v_i\})$ in Dict
6:         $\mathcal{L}_{add} = \mathcal{L}_{add} \cup \{a_i, v_i\}$
7:     $c \leftarrow \mathsf{SKE.Enc}(\mathsf{K_{EXP}}, 0^{|\mathsf{f}|})$
8:     $\tau_\alpha(f) = (c_{id(f)}, c, \mathcal{L}_{add})$
9: else                                              ▷ $op = delete$ for a file $f_\ell$
10:    if $c_{id(f_\ell)} \in L$ then
11:        $L_{del} = \{\}$
12:        for all $c_{id(f_i)} \notin L$ do
13:            Simulate a fake address $a_i$
14:            $v_i \leftarrow \mathsf{SKE.Enc}(\mathsf{K_{EXP}}, 0^{|\mathsf{f}|})$
15:            $L_{del} = L_{del} \cup \{(a_i, v_i)\}$
16:        $\tau_d = (L_{del})$
```

The simulated addition token, allows the simulator to keep its dictionary consistent as $\mathcal{ADV}$ inserts new files. It can be seen, that both the simulated and real tokens have the same format and size and hence we can conclude that they are indistinguishable from $\mathcal{ADV}$'s point of view. Furthermore, since SKE satisfies the CPA-security property, $\mathcal{ADV}$ can only distinguish between encryptions of files and zeroes with negligible probability. Apart from that, just like the addition token, the simulated deletion token also has the same format and size with the real deletion token and hence, they are also indistinguishable. Similarly to the simulation of the search token, $\mathcal{ADV}$ cannot tamper with the receipts generated by the SEV-enabled VMs, as this would imply that $\mathcal{ADV}$ can forge a valid SVM's measurement, which can only happen with negligible probability. As a result:

$$|Pr[(Hybrid\ 2) = 1] - Pr[(Hybrid\ 3) = 1]| \leq negl(\lambda) \tag{8}$$

By combining inequalities 6,7 and 8 we get:

$$|Pr[(Hybrid\ 0) = 1] - Pr[(Hybrid\ 3) = 1]| \leq negl(\lambda) \tag{9}$$

Which is equivalent to:

$$|Pr[(\mathsf{Real}_{\mathcal{ADV}}) = 1] - Pr[(\mathsf{Ideal}_{\mathcal{ADV},\mathcal{S}}) = 1]| \leq negl(\lambda) \tag{10}$$

And thus, our proof is complete.

Finally, it is important to note that since we managed to simulate consistent update tokens given only $\mathcal{L}_{update}$, we proved that our scheme satisfies the notions of forward and backward privacy.

□

Side-Channel Attacks Encryption and Decryption using the AES-NI hardware instruction, ensures there is no leakage of the key during search and update. This is because AES encryption/decryption with these instructions have data-independent timing and involve only data-independent memory access.

6.1 Does the Removal of TEE Affects the Security of the Scheme?

While the use of a TEE can be seen as a subterfuge to improve the security of a scheme this is not true in our case. More precisely, the security of our construction is not affected by removing the TEE and the crucial notions of froward and backward privacy are still preserved. However, the efficiency of the scheme it will be influenced since an extra round of communication will be required. More precisely, by removing the TEE, the trust that is now given to the TEE, will have to be distributed to the users. In particular, the role of the Deletion Authority can be played by any arbitrary user that performs a search operation. When a user performs a search operation, after she recieves the results from the CSP, she can filter out the entries to be deleted, re-encrypt the rest and send them back to the CSP. Hence, assuming that users are trusted, we achieve the same level of security by adding one more round of communication between the users and the CSP. In our work, the main reason for the use of the TEE, is to support the multi-client model. As a result, while removing the TEE does not affect the overall security of the scheme, it will result to a single-client model.

7 Experimental Results

We implemented our scheme in Python 2.7 using the PyCrypto library [1]. To test the overall performance, we used files of different sizes and structures. More precisely, we used a collection of five datasets provided in [21]. Table 4 shows the datasets used in our experiments as well as the total number of unique keywords extracted from each set. Our experiments focused on two main aspects: *(1)* Indexing and *(2)* Searching for a specific keyword. Deletion cannot be realistically measured since to completely delete all entries corresponding to a file, we first need to search for all the keywords contained in the file. Additionally, our dictionaries were implemented as tables in a MySQL database. In contrast to other similar works, we did not rely on the use of data structures such as arrays, maps, sets, lists, trees, graphs, etc. and we decided to build a more durable implementation with an actual database that properly represents a persistent storage. While the use of a database system decreases the overall performance of the scheme it is considered as more durable and close to a production level. Conducting our experiments by solely relying on data structures would give us better results. However, this performance would not give us any valuable insights about how the scheme would perform outside of a lab. Hence, we would not be able to argue about the actual practicality of our scheme in a proper cloud-based service. Additionally, storing the database in RAM raises several concerns. For example a power loss or system failure could lead to data loss (because RAM is volatile memory). Further to the above mentioned, since we wanted to evaluate the performance under realistic conditions, we decided to use different machines. To this end, we ran our experiments in the following two different machines:

- AMD Ryzen™ 7 PRO 1700 Processor at 3.0GHz (8 cores), 32GB of RAM running Windows 10 64-bit with AMD SEV Support;
- Microsoft Surface Book laptop with a 4.2GHz Intel Core i7 processor (4 cores) and 16GB RAM running Windows 10 64-bit;

As can be seen, apart from the first test-bed where we used a powerful machine with a lot of computational power and resources, the other machine is a commodity laptop that a typical user can own. The reason for measuring the performance on such machines and not only in a powerful desktop – like other similar works – is that in a practical scenario, the most demanding processes of any SSE scheme (e.g. the creation of the dictionary) would take place on a user's machine. Hence, conducting the experiments only on a powerful machine would result in a set of non-realistic measurements.

Indexing & Encryption: In our experiments, we measured the total setup time for each one of the datasets shown in table 4. Each process was run ten times on the commodity laptop and the average time for the completion of the entire process was measured. As can be seen from table 5, the setup time can be considered as practical and can even run in typical users' devices. In figure 1, we compare the setup times for the commodity laptop and the powerful desktop. Based on the fact that this phase is the most demanding one in an SSE scheme the time needed to index and encrypt such a large number of files is considered as acceptable not only based on the size of the selected dataset but also based on the results of other schemes that do not even offer forward privacy as well as on the fact that we ran our experiments on commodity machines and not on a powerful server. This is an encouraging result and we hope that will motivate researchers to design and implement even better and more efficient SSE schemes but most importantly we hope that will inspire key industrial players in the field of cloud computing to create and launch modern cloud-services based on the promising concept of Symmetric Searchable Encryption.

Table 3: Dataset Sizes and Setup Times

No of TXT Files	**Dataset Size**	**Unique Keywords**	**$(\mathbf{w}, \mathbf{id})$ pairs**
425	184MB	1,370,023	5,387,216
815	357MB	1,999,520	10,036,252
1,694	670MB	2,688,552	19,258,625
1,883	1GB	7,453,612	28,781,567
2,808	1.7GB	12,124,904	39,747,904

Table 4: Size of Datasets and Unique Keywords

Dataset \ Testbed	Laptop	Desktop
184MB	22.48m	8.49m
357MB	40.00m	13.51m
670	86.43m	29.51m
1GB	141.60m	48.99m
1.7GB	203.28m	68.44m

Table 5: Setup time (in minutes)

Search: In this part of the experiments we measured the time needed to complete a search over encrypted data. In our implementation, the search time is calculated as the sum of the time needed to generate a search token and the time required to find the corresponding matches at the database. It is worth mentioning that the main part of this process will be running on the CSP (i.e. a machine with a large pool of resources and computational power). To this end, the time to generate the search token was measured on the laptop while the actual search time was measured using the desktop machine described earlier. On average, the time needed to generate the search token on the Surface Book laptop was $9\mu s$. Regarding the actual search time, searching for a specific keyword over a set of 12,124,904 distinct keywords and 39,747,904 addresses required 1.328sec on average while searching for a specific keyword over a set of 1,999,520 distinct keywords and 10,036,252 addresses took 0.131sec.

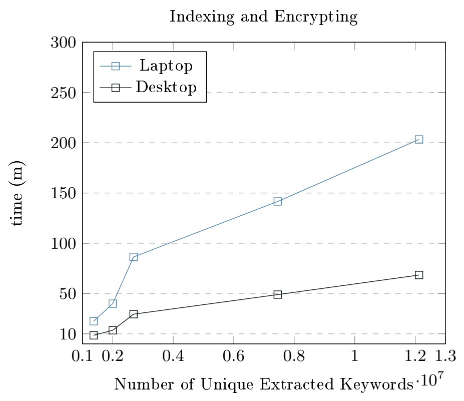

Fig. 1: Indexing and Encrypting Files

8 Conclusion

In this paper, we presented a forward/backward private SSE scheme in the multi-client setting based on the work proposed in [15]. The forward and backward privacy properties of our scheme have been demonstrated by showing that consistent simulated tokens can be constructed by a simulator that only has access to the corresponding leakage functions as they were formalized in [10]. Additionally, to the best of our knowledge, our construction, along with the one presented in [4] are the only ones that satisfy Type-II backward privacy and require only one round of interaction per search query. Finally, our scheme is currently the only approach that achieves this level of security in the multi-client setting – a clear vantage point over the rest of the existing approaches.

References

1. PyCrypto – the Python cryptography toolkit (2013), https://pypi.org/project/pycrypto/
2. AMD: Secure Encrypted Virtualization API Version 0.22. Tech. rep. (07 2019)
3. AMD: AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and more. Tech. rep. (01 2020)
4. Amjad, G., Kamara, S., Moataz, T.: Forward and backward private searchable encryption with sgx. In: Proceedings of the 12th European Workshop on Systems Security. EuroSec '19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3301417.3312496, https://doi.org/10.1145/3301417.3312496
5. Bakas, A., Dang, H.V., Michalas, A., Zalitko, A.: The cloud we share: Access control on symmetrically encrypted data in untrusted clouds. IEEE Access **8**, 210462–210477 (2020). https://doi.org/10.1109/ACCESS.2020.3038838
6. Bakas, A., Michalas, A.: Modern family: A revocable hybrid encryption scheme based on attribute-based encryption, symmetric searchable encryption and sgx. In: Chen, S., Choo, K.K.R., Fu, X., Lou, W., Mohaisen, A. (eds.) Security and Privacy in Communication Networks. pp. 472–486. Springer International Publishing, Cham (2019)
7. Bakas, A., Michalas, A.: Power range: Forward private multi-client symmetric searchable encryption with range queries support. In: 2020 IEEE Symposium on Computers and Communications (ISCC). pp. 1–7 (2020). https://doi.org/10.1109/ISCC50000.2020.9219739
8. Boneh, D., Kim, S., Wu, D.J.: Constrained keys for invertible pseudorandom functions. In: Theory of Cryptography Conference. pp. 237–263. Springer (2017)
9. Bost, R.: $\sum o\varphi o\varsigma$: Forward secure searchable encryption. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016 (2016). https://doi.org/10.1145/2976749.2978303, https://doi.org/10.1145/2976749.2978303
10. Bost, R., Minaud, B., Ohrimenko, O.: Forward and backward private searchable encryption from constrained cryptographic primitives. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 1465–1482. CCS '17, ACM, New York, NY, USA (2017). https://doi.org/10.1145/3133956.3133980, http://doi.acm.org/10.1145/3133956.3133980
11. Brasser, F., Hahn, F., Kerschbaum, F., Sadeghi, A.R., Fuhry, B., Bahmani, R.: Hardidx: Practical and secure index with sgx (2017)

12. Cash, D., Grubbs, P., Perry, J., Ristenpart, T.: Leakage-abuse attacks against searchable encryption. In: Proceedings of the 22nd ACM SIGSAC conference on computer and communications security. ACM (2015)
13. Costan, V., Devadas, S.: Intel sgx explained. IACR Cryptology ePrint Archive **2016**(086), 1–118 (2016)
14. Dolev, D., Yao, A.C.: On the security of public key protocols. Information Theory, IEEE Transactions on **29**(2) (1983)
15. Etemad, M., Küpçü, A., Papamanthou, C., Evans, D.: Efficient dynamic searchable encryption with forward privacy. Proceedings on Privacy Enhancing Technologies **2018**(1), 5–20 (2018)
16. Garg, S., Mohassel, P., Papamanthou, C.: Tworam: efficient oblivious ram in two rounds with applications to searchable encryption. In: Annual International Cryptology Conference. pp. 563–592. Springer (2016)
17. Ghareh Chamani, J., Papadopoulos, D., Papamanthou, C., Jalili, R.: New constructions for forward and backward private symmetric searchable encryption. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. pp. 1038–1055 (2018)
18. Islam, M.S., Kuzu, M., Kantarcioglu, M.: Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In: Ndss. vol. 20, p. 12. Citeseer (2012)
19. Kim, K.S., Kim, M., Lee, D., Park, J.H., Kim, W.H.: Forward secure dynamic searchable symmetric encryption with efficient updates. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 1449–1463 (2017)
20. 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. p. 146–155. SAC '19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3297280.3297297, https://doi.org/10.1145/3297280.3297297
21. Michalas, A.: Text files from gutenberg database (Aug 2019). https://doi.org/10.5281/zenodo.3360392, https://doi.org/10.5281/zenodo.3360392
22. Michalas, A., Bakas, A., Dang, H.V., Zalitko, A.: Abstract: Access control in searchable encryption with the use of attribute-based encryption and sgx. In: Proceedings of the 2019 ACM SIGSAC Conference on Cloud Computing Security Workshop. p. 183. CCSW'19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3338466.3358929, https://doi.org/10.1145/3338466.3358929
23. Mofrad, S., Zhang, F., Lu, S., Shi, W.: A comparison study of intel sgx and amd memory encryption technology. In: Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy. pp. 1–8 (2018)
24. Naveed, M.: The fallacy of composition of oblivious ram and searchable encryption. IACR Cryptology ePrint Archive **2015**, 668 (2015)
25. 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
26. Stefanov, E., Papamanthou, C., Shi, E.: Practical dynamic searchable encryption with small leakage. In: NDSS. vol. 71, pp. 72–75 (2014)

27. Sun, S.F., Yuan, X., Liu, J.K., Steinfeld, R., Sakzad, A., Vo, V., Nepal, S.: Practical backward-secure searchable encryption from symmetric puncturable encryption. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. pp. 763–780 (2018)
28. Zhang, Y., Katz, J., Papamanthou, C.: All your queries are belong to us: The power of file-injection attacks on searchable encryption. In: 25th USENIX Security Symposium). pp. 707–720 (2016)