Advanced Secure Cloud Encrypted Platform for Internationally Orchestrated Solutions in Healthcare

Project Acronym: **ASCLEPIOS**
Project Contract Number: **826093**

Programme**: Health, demographic change and wellbeing**
Call: **Trusted digital solutions and Cybersecurity in Health and Care to protect privacy/data/infrastructures**
Call Identifier: **H2020-SC1-FA-DTS-2018-2020**

Focus Area: **Boosting the effectiveness of the Security Union**
Topic**: Toolkit for assessing and reducing cyber risks in hospitals and care centres**
Topic Identifier: **H2020-SC1-U-TDS-02-2018**

Funding Scheme: **Research and Innovation Action**

Start date of project: 01/12/2018 Duration: 36 months

Deliverable:

D2.1: Symmetric Searchable Encryption and Integration in Medical Devices

Due date of deliverable: <30/11/2019> Actual submission date: <10/12/2019>
WPL: <WPL Antonios Michalas>
Dissemination Level: <PU>
Version: final

1 Table of Contents

2 List of Figures and Tables

Figures

Tables

3 Status, Change History and Glossary

Status:	Name:	Date:	Signature:
Draft:	Antonios Michalis	March – November 2019	Antonios Michalis
Reviewed:	Gabor Terstyanszky	03-04/12/2019	Gabor Terstyanszky
Approved:	Tamas Kiss	09/12/2019	Tamas Kiss

Table 1: Status Change History

Version	Date	Pages	Author	Modification
V0.1	4/3/2019	11	Alexandros Bakas	Begin working on the introduction
V0.2	15/3/2019	18	Alexandr Zalitko	Begin working on the Notations and Definition for SSE
V0.3	19/3/2019	29	Arash Vahidi	Started Working on the System Model
V0.4	28/3/2019	29	Antonis Michalas	Corrected format and typos
V0.5	1/4/2019	29	Hai-Van Dang	Begin working on State of the Art
V0.6	14/4/2019	30	Arash Vahidi	Finalized the system model
V0.7	27/4/2019	30	Alexandros Bakas	Include more schemes in the state of the art section
V0.8	12/5/2019	31	Antonis Michalas	Corrected format and typos
V0.9	31/5/2019	35	Alexandr Zalitko	Begin working on an SSE scheme for ASCLEPIOS
V1.0	09/6/2019	38	Hai-Van Dang	Formalized the SSE scheme for ASCLEPIOS
V1.1	18/6/2019	41	Arash Vahidi	Proposed a protocol based on the SSE scheme
V1.2	29/6/2019	46	Alexandros Bakas	Designed a detailed protocol based on the SSE scheme
V1.3	15/7/2019	52	Antonis Michalas	Proved the Security of the proposed scheme
V1.4	12/8/2019	52	Alexandr Zalitko	Proved the Security of the protocol
V1.5	20/8/2019	53	Hai-Van Dang	Found datasets to properly evaluate the SSE scheme
V1.7	10/9/2019	56	Antonis Michalas	Started working on the implementation of the SSE scheme
V1.8	01/10/2019	58	Alexandr Zalitko	Finalized the experimental evaluation
V1.9	19/9/2019	58	Hai-Van Dang	Begin working on integrating the SSE scheme to the needs of the demonstrators

V2.0	18/10/2019	58	Alexandros Bakas	Added a Modification Algorithm to the Scheme without affecting neither the experimental results, nor the security of the scheme.
V2.1	27/10/2019	58	Hai-Van Dang	Finalized the Integration for the demonstrators
V2.2	08/11/2019	58	Antonis Michalas	Corrected format and typos
V3.0	04/12/2019	58	Gabor Terstyanszky	Reviewed the report

Table 2: Deliverable Change History

Glossary

SSE	Symmetric Searchable Encryption
PRF	Pseudorandom Function
IPRF	Invertible Pseudorandom Function
negl	Negligible
Pr	Probability
$\mathcal{L}$	Leakage Function
$\mathcal{A}$	Adversary
$\mathcal{S}$	Simulator
CSP	Cloud Service Provider
TA	Trusted Authority
u	User
DO	Data Owner
K	Symmetric Key
$\sigma()$	Signature

Table 3: Glossary

1 Introduction

1.1 Contribution and Organization

D2.1: Symmetric Searchable Encryption and Integration in Medical Devices, as specified in the ASCLEPIOS Grant Agreement, will

- reveal advantages and limitations of Symmetric Searchable Encryption schemes, and
- provide a detailed analysis on how and under which conditions such schemes could be used by Healthcare services.

The contribution of this deliverable is twofold. First, we present a survey on existing SSE schemes, examining in detail the advantages and the disadvantages of each scheme. Then, we proceed by describing an SSE scheme that was designed to meet the needs of ASCLEPIOS according to (Tampere Univeristy (TUNI), Norwegian Centre for eHealth Research (NSE)). The rest of the deliverable is organized as follows:

- In Section 2, we present the notation that will be used throughout the deliverable as well as some important definitions and the threat model.
- Section 3 consists of a survey on Symmetric Searchable Encryption where multiple schemes are described.
- In Section 4, we present the system model needed for our construction. This system model is in total accordance with the one presented in (Tampere Univeristy (TUNI), Norwegian Centre for eHealth Research (NSE)).
- In Section 5, we describe in detail our SSE scheme, designed especially for the ASCLEPIOS project.
- In Section 6, we present a protocol based on the SSE scheme and the reference architecture of ASCLEPIOS (Tampere Univeristy (TUNI), Norwegian Centre for eHealth Research (NSE)).
- In Section 7, we prove the security of the proposed protocol.
- In Section 8, we present several experiments of the SSE scheme.

1.2 Symmetric Searchable Encryption

Symmetric Searchable Encryption (SSE) along with Dynamic Symmetric Searchable Encryption (DSSE) in which file additions and deletions are allowed, are among the most promising encryption techniques that can pave the way to truly secure and privacy-preserving cloud-based services. In general, SSE schemes aim to provide confidentiality and integrity, while retaining main benefits of cloud storage – availability, reliability, and ensuring requirements through cryptographic guarantees rather than administrative controls. SSE allows a client to securely outsource private data to a Cloud Service Provider (CSP) in such a way that the client can later perform keyword searches directly on the stored ciphertexts. To perform such a search, the client sends a query for a specific keyword w to the CSP. By processing his query, the CSP can find all stored ciphertexts containing w without revealing any valuable information about the contents of the files and without even getting to know the actual keyword w that the user searched for. Ideally, an SSE scheme should leak no information at all to the CSP. However, to achieve this, techniques such as oblivious RAM (ORAM) need to be used and according to (M. Naveed), it is even less efficient than downloading and decrypting the entire database locally. Leaked information is a problem of paramount importance in SSE since even a small leakage can lead to attacks that violate users' privacy (Cash, Grubbs and Pery), (Islam, Kuzu and Kantarcioglu). For example, in (Zhang, Katz and Papamanthou), authors assumed that a malicious adversary can add new

files and showed that only after ten file injections, the adversary was able to reveal the contents of a past query thus, violating users' privacy.

Taking this into consideration, we have designed an SSE scheme that squarely fits on ASCLEPIOS's reference architecture, as defined by (Tampere Univeristy (TUNI), Norwegian Centre for eHealth Research (NSE)), and at the same time offers stronger security guarantees. In particular, the scheme presented in this deliverable is both forward and backward private. Informally, forward privacy is achieved if for all file insertions that take place after the initial setup of the SSE scheme, the leakage is limited to the number of distinct keywords of the file, as well as the size of it. On the other hand, an SSE scheme is said to be backward-private if, whenever a keyword/document pair $(w, id(f))$ is added to the database and then deleted, subsequent search queries on w do not reveal $id(f)$.

1.1 Symmetric Searchable Encryption in ASCLEPIOS

One of the core components of ASCLEPIOS, as defined n (Tampere Univeristy (TUNI), Norwegian Centre for eHealth Research (NSE)), is the cryptographic layer. The cryptographic layer is further divided into two subcomponents, namely traditional cryptography and modern cryptographic techniques.

In this deliverable, we present a thorough study of Symmetric Searchable Encryption (SSE), one of the modern cryptographic techniques that constitute the backbone of the ASCLEPIOS project.

By implementing an SSE scheme, we can ensure that the privacy of a patient's data will not be compromised and at the same time, any registered user that satisfies certain access rights, will be able to search on the patient's encrypted data.

2 Notation and Definitions

In this section we present the notations and give definitions that will be used throughout this report. Moreover, we define security properties that are needed to formalize symmetric searchable encryption.

2.1 Primitives

Let $\mathcal{X}$ be a set. We use $x \leftarrow \mathcal{X}$ is x is sampled uniformly from $\mathcal{X}$ and $x \xleftarrow{\$} \mathcal{X}$, if x is chosen uniformly at random. If $\mathcal{X}$ and $\mathcal{Y}$ are two sets, then 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 $\mathcal{Y}$. $R(.)$ is used for a truly random function, while $R^{-1}(.)$ represents the inverse function of $R(.)$. A function $negl(.)$ is called negligible if $\forall n > 0, \exists N_n : \forall x > N_n : |negl(x)| < \frac{1}{poly(x)}$, where $poly(x)$ represents a polynomial of x.

Definition (Pseudorandom Function (PRF)). Let G be a function such that $G: \mathcal{K} \times \mathcal{X} \rightarrow \mathcal{Y}$, where $\mathcal{K}$ denotes the key-space, $\mathcal{X}$ denotes the domain of definition and $\mathcal{Y}$, the range of the function G. Moreover, let $G.Gen(1^\lambda)$ be a probabilistic algorithm that given the security parameter λ, outputs a key $k \in \mathcal{K}$ for G. G is a PRF if for all probabilistic polynomial time adversaries $\mathcal{A}$:

$$\left| \Pr\left[k \leftarrow G.Gen(1^\lambda) : \mathcal{A}^{G(k,.)}(1^\lambda) = 1\right] - \Pr\left[k'[\mathcal{X}, \mathcal{Y}] \xleftarrow{\$} : \mathcal{A}^{R(.)}(1^\lambda) = 1\right] \right| = negl(\lambda)$$

An invertible pseudorandom function (Boneh, Kim and Wu) is defined as follows:

Definition (Invertible Pseudorandom Function (IPRF)). An IPRF G 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 $G.Gen(1^\lambda)$ be a probabilistic algorithm that given the security parameter λ, outputs a key $k \in \mathcal{K}$ for G. The functions G, G^{-1} satisfy the following properties:

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

A function $G: \mathcal{K} \times \mathcal{X} \rightarrow \mathcal{Y}$ is an IPRF if for all probabilistic polynomial time adversaries $\mathcal{A}$:

$$\left| \Pr\left[k \leftarrow G.Gen(1^\lambda) : \mathcal{A}^{G(k,.), G^{-1}(k,.)}(1^\lambda) = 1\right] - \Pr\left[k'[\mathcal{X}, \mathcal{Y}] \xleftarrow{\$} : \mathcal{A}^{R(.), R^{-1}(k,.)}(1^\lambda) = 1\right] \right| = negl(\lambda)$$

Dynamic Symmetric Searchable Encryption (DSSE)

We continue with a formal definition of a Dynamic Symmetric Searchable Encryption Scheme.

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

- K ← $KeyGen(1^{\lambda})$: The data owner generates a **secret key K** that consists of a key K_G for an IPRF G and a key K_{SKE} for an IND-CPA secure symmetric key cryptosystem SKE.
- $(In_{CSP}, C)(In_{TA}) \leftarrow InGen(K, F)$: The data owner runs this algorithm to generate the **CSP index In_{CSP}** and a **collection of ciphertexts C** that will be sent to the CSP. Additionally, the index In_{TA} that is stored both locally and in a remote location, since it is outsourced to a trusted authority TA is generated.
- $(In'_{CSP}, C'), In'_{TA} \leftarrow AddFile(K, f, In_{TA})(In_{CSP}, C)$: The data owner is running this algorithm to **add a file to his/her collection of ciphertexts**. All the indexes and the collection of cipehertexts are updated.
- $(In'_{CSP}, I_w), In'_{TA} \leftarrow Search(K, a, b, In_{TA})(In_{CSP}, C)$: This algorithm is executed by a user in order to **search for all files f containing a specific keyword w**. The indexes are updated and the CSP also returns to the user a sequence of file identifiers I_w.
- $(In'_{CSP}, I_w), In'_{TA} \leftarrow RangeSearch(K, w_i, In_{TA})(In_{CSP}, C)$: This algorithm is executed by a user in order to **search for all files f containing values in the range** $[a, b]$.The indexes are updated and the CSP also returns to the user a sequence of file identifiers I_w.
- $(In'_{CSP}, C'), In'_{TA} \leftarrow Delete(K, c_{id(f)}, In_{TA})(In_{CSP}, C)$: The data owner runs this algorithm to **delete a file from the collection**. All the indexes are updated accordingly.
- $(In'_{CSP}, C'), In'_{TA} \leftarrow Modify(K, f, In_{TA})(In_{CSP}, C)$: The data owner runs this algorithm to **modify a file that already exists in the collection**.

Definition (Search Pattern). The Search Pattern is a mapping between queries and keywords. This mapping is used to tell whether two or more queries were for the same keyword.

Definition (Access Pattern). The Access Pattern is defined to be the outcome of each search query.

Definition (Forward Privacy). An SSE scheme is said to be forward private if for all file insertions after the initial setup the leakage is limited to the size of the inserted file and the number of unique keywords contained in it.

Definition (Backward Privacy). An SSE scheme is said to be backward private if, whenever a keyword/document pair $(w, id(f))$ is added into the database and the deleted, subsequent search queries on w do not reveal $id(f)$.

Moreover, let $L_{InGen}, L_{Add}, L_{Search}, L_{Delete}$ be the leakage functions associated with index creation, file addition and the search and delete operations. We have:

$L_{InGen} = (N,\ n,\ id(f_i),\ |f_i|)$: This function leaks the total size N of all the $(w, id(f))$ mappings, as well as the number of files, their ID's and their sizes.

- L_{Add} = $(id(f),\ |f|,\ \#w_i \in f)$: This function leaks the file id, its size and the number of unique keywords contained in it.
- L_{Search} = $\{Access\ Pattern, Search\ Pattern\}$: This function leaks the Access and Search Patterns.
- L_{Delete} = $(\#w_i \in f)$: This functions leaks the number of unique keywords contained in the deleted file.

Definition (SSE Security). *Let SSE = (KeyGen, InGen, Add, Search, Delete, Modify, RangeSearch, ComplexSearch) be a dynamic symmetric searchable encryption scheme. Moreover, let* $L_{InGen}, L_{Add}, L_{Search}, L_{Delete}$ *be the leakage functions as defined above.* $We\ consider$ *the following experiment between a simulator* $\mathcal{S}$ *and an adversary* $\mathcal{A}$*,*

$\boldsymbol{Real_{\mathcal{A}}(\lambda)}$**:** $\boldsymbol{\mathcal{A}}$ *outputs a set of files* F*.* $\mathcal{C}$ *runs KeyGen to generate a key* K*, and runs InGen.* $\mathcal{A}$ *then makes a polynomial time of adaptive queries* $(q = \{w, f_1, f_2\})$ *such that* f_1 *is contained in a file* $f \in F, f_1 \notin F$ *and* $f_2 \in F$*. For each* q*, she receives bask either a search token* $\tau_s(w)$*, for* w*, an add token* $\tau_\alpha(f_1)$ *and a ciphertext* c_{f_1} *for* f_1*, or a delete token* $\tau_d(f_2)$ *for* f_2*. Finally,* $\mathcal{A}$ *outputs a bit* b*.*

$\boldsymbol{Ideal(\lambda)}$**:** $\boldsymbol{\mathcal{A}}$ *outputs a set of files* F*.* $\mathcal{S}$ *gets* L_{InGen} *as input and simulates InGen.* $\mathcal{A}$ *then makes a polynomial time of adaptive queries* $(q = \{w, f_1, f_2\})$ *such that* f_1 *is contained in a file* $f \in F, f_1 \notin F$ *and* $f_2 \in F$*. For each* q*,* $\mathcal{S}$ *is given either* $L_{Search}(w)$ *,* $L_{Add}(f_1)$ *or* $L_{Delete}(f_2)$*.* $\mathcal{S}$ *then simulates the tokens and, in the case of file addition, a ciphertext. Finally,* $\mathcal{A}$ *outputs a bit b.*

We say that the SSE scheme is secure if for all Probabilistic Polynomial Time adversaries $\mathcal{A}$ *, there exists a simulator* $\mathcal{S}$ *such that:*

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

2.2 Threat Model

The ASCLEPIOS threat model is similar to the one described in (Paladi, Gehermann and Michlas) which is based on the Dolev-Yao adversarial model (Dolev and Yao). More presicely, we assume the following:

- **Hardware Integrity**: Media revelations have raised the issue of hardware tampering en route to deployment sites. We assume that the cloud provider has taken necessary technical and non-technical measures to prevent such hardware tampering.
- **Physical Security**: We assume physical security of the data centres where the IaaS is deployed. This assumption holds both when the IaaS provider owns and manages the data center (as in the case of Amazon WebServices, Google Compute Engine, Microsoft Azure, etc.)and when the provider utilizes third party capacity, since physical security can be observed, enforced and verified through known best practices by audit organizations. This assumption is

important to build higher-level hardware and software security guarantees for the components of the IaaS.

- **Low-Level Software Stack**: We assume that at installation time, the IaaS provider reliably records integrity measurements of the low-level software stack: the Core Root of Trust for measurement; BIOS and host extensions; host platform configuration; Option ROM code, configuration and data; Initial Platform Loader code and configuration; state transitions and wake events, and a minimal hypervisor. We assume the record is kept on protected storage withread-only access and the adversary cannot tamper with it.
- **Network Infrastructure**: The IaaS provider has physical and administrative control of the network. ADV is in full control of the network configuration, can overhear, create, replay and destroy all messages communicated between DM and their resources (VMs, virtual routers, storage ab-straction components) and may attempt to gain access toother domains or learn confidential information.
- **Cryptographic Security**: We assume encryption schemes are semantically secure and the ADV cannot obtain the plaintext of encrypted messages. We also assume the signature scheme is unforgeable, i.e. the ADV cannot forge the signatures and that the MAC algorithm correctly verifies message integrity and authenticity. We assume that the ADV, with a high probability, cannot predict the output of a pseudorandom function. We explicitly exclude denial-of-service attacks and focus on ADV that aim to compromise the confidentiality of data in IaaS.
- Furthermore, we assume that the adversary $\mathcal{A}$ is allowed to provide the enclaves with inputs of his/her choice and record the output. This assumption significantly strengthens the adversary since we need to ensure that only honest attested programs with correct inputs will run in the enclaves. At this point it is worth mentioning that, like similar works in the area (Dowsley, Michalas and Nagel), the threat model of the actual construction of ASCLEPIOS DSSE scheme is the, not so realistic, semi-honest model. To make our work secure under the threat model we described, earlier, we designed a protocol that shows how our scheme can be used in a cloud-based service.

3 Existing Encryption Approaches – State of the Art

3.1 Two-Layered Encryption Scheme

SSE was first introduced in (Song, Wagner and Perrig) where the authors presented a **forward index scheme** based on two different layers of encryption. The main idea of the scheme is to compute the deterministic encryption of each keyword and then use a stream cipher as the second layer of the encryption. In particular, for each keyword we compute $x = Enc(w)$ and then we parse as $x = x_l||x_r$. The x_l component is used for the creation of a key k of a hash function h, while at the same time, the exclusive bitwise or (XOR) of x_l with a random seed s is computed. This will finally allow us to compute $h(k,s) \oplus x_r$. The search token for a keyword w is then $E(w)$ and the key k is the one derived by x_l. The server can then perform the search operation by checking for each ciphertext c whether it is of the form $s||h(k,s)$.

3.1.1 The Basic SSE Scheme

Assume that a user Alice wants to encrypt a sequence of keywords $W_1, W_2, \dots, W_\ell$. Intuitively, the scheme works by computing the XOR of the plaintexts with a sequence of pseudorandom bits which have a special structure. This structure will allow to search on the data without revealing anything else about the clear text. More specifically, the basic scheme is as follows: Alice generates a sequence of pseudorandom values $S_1, S_2, \dots, S_\ell$ using some stream cipher (namely, a pseudorandom generator G), where each S_i is $n-m$ bits long. To encrypt an n-bit word W_i that appears in position i, Alice takes the pseudorandom bits S_i, sets T_i=< $S_i||F_{k_i}(S_i)$ >, where F is a pseudorandom function such that: $F: K_F \times \{0,1\}^{n-m} \rightarrow \{0,1\}^m$, and ouputs $C_i = W_i \oplus T_i$.

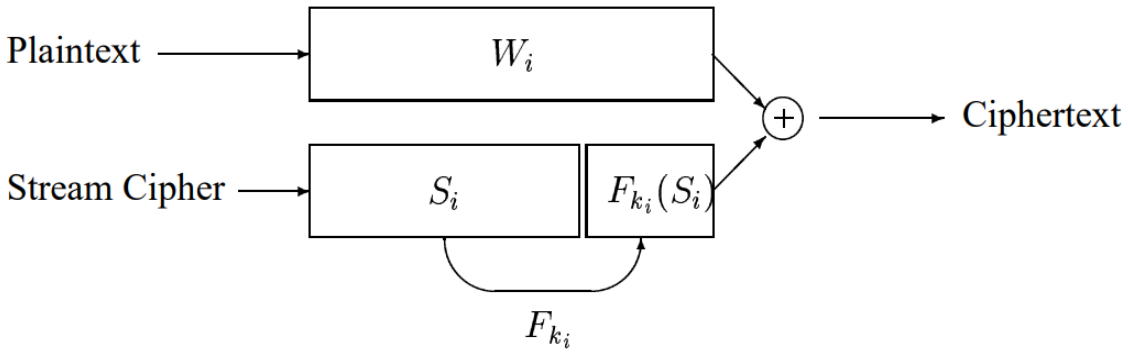

Figure 1: Basic SSE Scheme

Note that only Alice can generate the pseudorandom stream $T_1, T_2, \dots T_\ell$ so no one else can decrypt. This scheme support searches over the ciphertexts in the following way: if Alice wants to search the word W, she can tell Bob W and the k_i corresponding to each location i where a word W may occur. Bob can then search for W in the ciphertext by checking whether $C_i \oplus W_i$ is of the form $s||F_{k_i}(s)$ for some s. At the positions where Bob does not know k_i, he learns nothing about the plaintext. Thus, the scheme allows a limited form of control: if Alice only wants Bob to be able to search over the first half of the ciphertext, Alice should reveal only the K_i, corresponding to those locations and none of the k_i used in the second half of the ciphertext. However, for Alice to help Bob search for a word W, either Alice must reveal all the k_i (thus revealing the entire document), or Alice must know in

advance which location W may appear at- which seems to defeat the purpose of remote searching.

3.1.2 The Final SSE Scheme

In this final scheme authors tried to deal with the aforementioned problems. To this end, Alice computes a deterministic encryption of the words W as $X = E(W)$. She then parses X as $X_i = L_i||R_i$, where L_i consists of the first $n - m$ bits of X_i and R_i of the last m bits. Alice now computes the keys as $k_i = F_k(L_i)$.

To decrypt, Alice computes S_i, using a pseudorandom generator, and can retrieve S_i by XORing $S_i$with the first $(n - m)$ bits of C_i. Finally, by knowing L_i, Alice can compute k_i and hence, finish the encryption.

This correction would not be correct if the words were not encrypted first since it is very common for different words to have the same $n - m$ first bits. By encrypting the keywords in the first step, diminishes this problem since $|\Pr[L_i = L_j] = 1| = negl(.)$. In particular, the authors assumed that the initial encryption is a pseudorandom permutation, and by the birthday paradox, the probability of collision is extremely low. After Alice is done with encrypting the words, she can re-order the ciphertexts using a pseudorandom permutation. This way, when Bob searches for a word, he will now knot the position of the word in the plaintext. This scheme deals with a number of problems: First of all, it uses fixed size words and then, even if Alice re-orders the ciphertexts, the position of the word can still be leaked. Finally, the complexity of a search operation is $O(m)$, where m is the total number of words in a file. In other words, the complexity is linear to the total number of keywords in a file, which is the worst case.

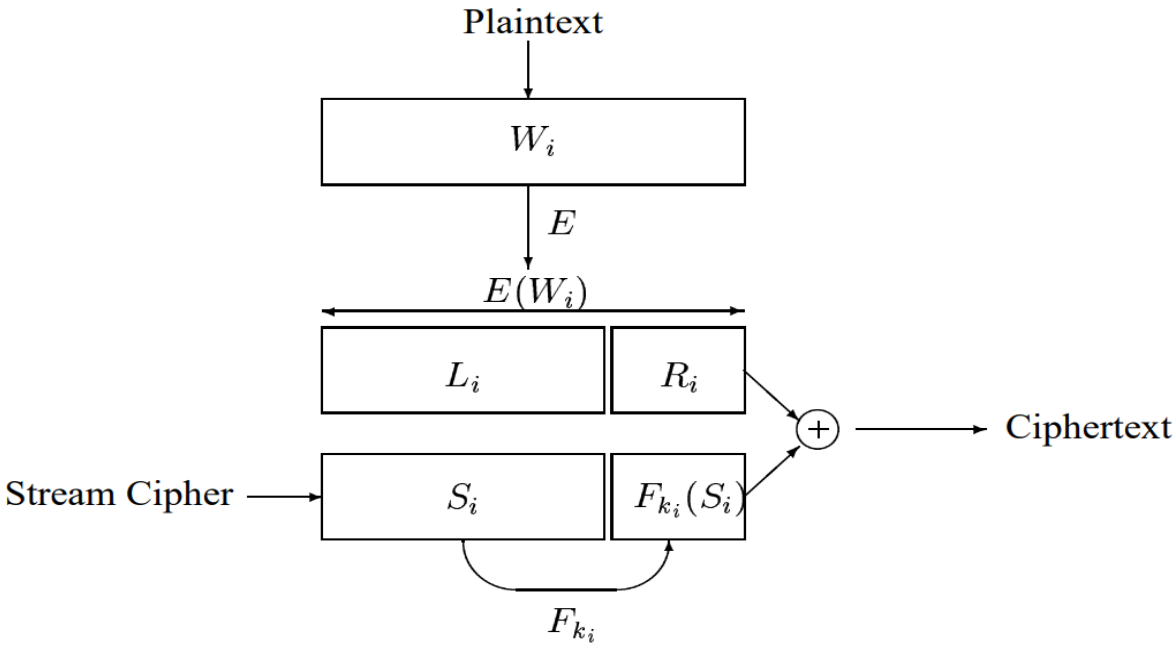

Figure 2: Final SSE Scheme

3.2 Forward Index

In (Goh), authors suggested to use an encrypted data structure for searching on keywords and in particular, a bloom filter. Bloom filters are data structures that can efficiently check whether an element is part of a set.

More specifically, they use an array of length n whose bits are initially 0. For each keyword w to be inserted to the set, t different hashes $(h_1, \dots, h_t)$ of the word are computed, where each h_i hashes into the set $\{1, \dots, n\}$ and then the bits $h_i(w)$ are set to 1. Using his data structure, it is possible to check whether the keyword is present in the document or not by checking if all the bits outputted by $h(w_i)$ are set to 1 or not.

To limit the information leakage about the keywords, the authors make use of two pseudorandom functions before inserting they keywords in the Bloom filter where the first function takes as input they keyword, and the second also takes as input the unique identifier of the file in order to avoid leaking similarities between the documents.

There are two main problems with his approach:

- Bloom Filters inherently return false positives;
- The number of 1s in the bloofm filter, depends on the number of unique keywords in a file. As a result, the proposed scheme leaks the total number of keywords in each file.

3.3 Hierarchical Structure of Logarithmic Levels

Stefanov et al. (Stefanov, Papamantou and Shi) proposed a dynamic SSE scheme that uses a hierarchical structure of logarithmic levels (which is reminiscent from techniques for oblivious RAMs). For *P* pairs of file/keywords, the server stores a hierarchical data structure containing log *P* + 1 levels. Each level ℓ can store up to 2^ℓ entries, where each entry encrypts the information about one keyword *k*, one identifier of a file *f* that contains *w*, the type of operation performed (either add or delete) and a counter for the number of occurrences of keyword *w* in the level ℓ. The scheme ensures that within the same level only one operation is stored for each pair of file/keyword. One search token per level of the structure is used to perform the search operation. In this scheme, every update induces a rebuild of levels in the data structure. The basic idea is to take the new entry together with the entries in consecutive full levels $1,\dots,\ell-1$ and merge them at level ℓ.

This scheme has small leakage, a data structure of linear size (in the number of file/keyword pairs), and both updates and searches are in sub-linear time. In contrast to the other schemes, it achieves the notion of forward security: the search tokens used in the past cannot be used to search for the keyword in the documents that are added afterwards. It is achieved due to the fact that every time a level is rebuilt a new key is used to encrypt the entries within that level. However, this smaller leakage comes at the expense of poly-logarithmic overhead (in the number of file/keyword pairs) on top of Dynamic SSE overhead of other schemes.

3.4 Inverted Index

The main idea proposed in (Kamara, Papamanthou and Roeder, Dynamic Searchbale Encryption) is to use an inverted index for each keyword instead for each file. The result is to reduce the search time from linear to the number of files to linear to the number of files containing the keyword w we are searching for.

For each keyword w, there exists a linked list L_w containing identifiers for the files containing w. The linked lists are encrypted to avoid information leakage. Hence, all nodes of L_w are stored in an array A, unsorted and encrypted. The plaintext of each node contains three things:

The unique file identifier of a file.
The encryption key of the next node.
A pointed to the next node.

Thus, to search for a keyword w, we need the encryption key of the first node of L_w and a pointer to its position in A. This information is stored encrypted in a pseudorandom position of a look-up table T. In other words, the user generates A and T along with her ciphertexts, and stores all of this information in the server.

With this setup, the search token consists of the position of w in T, along with the key used to insert the entry in T.

3.4.1 Achieving Dynamicity using a Deletion Array

The difficulty is that the addition, deletion or modification of a file requires the server to add, delete or modify nodes in the encrypted lists stored in As. This is difficult for the server to do since: (1) upon deletion of a file f , it does not know where (in A) the nodes corresponding to f are stored; (2) upon insertion or deletion of a node from a list, it cannot modify the pointer of the previous node since it is encrypted; and (3) upon addition of a node, it does not know which locations in A_s are free.

At a high-level, these limitations are addressed as follows:

1. **file deletion.** We add an extra (encrypted) data structure A_d called the *deletion array* that the server can query (with a token provided by the client) to recover pointers to the nodes that correspond to the file being deleted. More precisely, the deletion array stores for each file f a list L_f of nodes that point to the nodes in A_s that should be deleted if file f is ever removed. So, every node in the search array has a corresponding node in the deletion array and every node in the deletion array points to a node in the search array.
2. **pointer modification**. The pointers are stored encrytped in a node with a homomorphic encryption scheme. By providing the server with an encryption of an appropriate value, it can then modify the pointer without ever having to decrypt the node. We use the "standard" private-key encryption scheme which consists of XORing the message with the output of a PRF. This simple construction also has the advantage of being non-committing (in the private-key setting) which we make use of to achieve CKA2-security.
3. **memory management.** to keep track of which location in A_s are free, authors add and manage extra space, comprising a *free list* that the server uses to add new nodes.

Example: The index is built on three documents, namely f_1, f_2, f_3 over three keywords, namely w_1, w_2, w_3. All the documents contain keyword w_1, keyword w_2 is only contained in document f_2 and w_3 is contained in documents f_2 and f_3. The respective search table T_s, the deletion table T_d, the search array A_s and the deletion array A_d are also shown in the figure below. Note that in a real DSSE index, there would be padding to hide the number of file-word pairs; we omit padding for simplicity in this example.

Searching: Searching is the simplest operation in the scheme. Suppose the client wishes to search for all the documents that contain keyword w_1. He prepares the

search token, which among others contains $F_{k_1}(w_1)$ and $G_{k_2}(w_1)$. The first value $F_{k_1}(w_1)$ will enable the server to locate the entry corresponding to keyword w_1 in the search table T_s. In our example, this value is $x = (4||1) \oplus G_{k_2}(w_1)$. The server now uses the second value $G_{k_2}(w_1)$ to compute $x \oplus G_{k_2}(w_1)$.. This will allow the server to locate the right entry (4 in our example) in the search array and begin "unmasking" the locations storing pointers to the documents containing w_1. This unmasking is performed by means of the third value contained in the search token

File Addition: Assume now the client wishes to add a document f_4 containing keywords w_1 and w_2. Note that the search table does not change at all since f_4 is going to be the last entry in the list of keywords w_1 and w_2 and the search table only stores the first entries. However, all the other data structures must be updated in the following way. First the server uses free to quickly retrieve the indices of the "free" positions in the search array A_s, where the new entries are going to be stored. In our example these positions are 2 and 6. The server stores in these entries the new information (w_1, f_4) and (w_2, f_4). Now the server needs to connect these new entries to the respective keywords lists: using the add token, it retrieves the indices i = 0 and j = 3 in the search array A_s of the elements x and y such that x and y correspond to the last entries of the keyword lists w_1 and w_2. In this way the server homomorphically sets A_s[0]'s and A_s[3]'s "next" pointers to point to the newly added nodes, already stored in the search array at positions 2 and 6.

Note that getting access to the free entries in the search array also provides access to the respective free positions of the deletion array A_d. In our example, the indices of the free positions in the deletion array are 3 and 7. The server will store the new entries (f_4, w_1) and (f_4, w_2) at these positions in the deletion array and will also connect them with pointers. Finally, the server will update the deletion table by setting the entry $F_{k_1}(f_4)$ to point to position 3 in the deletion array, so that file f_4 could be easily retrieved for deletion later.

File Deletion: Suppose now the client wants to delete a document already stored in our index, say document f_3, containing keywords w_1 and w_3. The deletion is a "dual operation" to addition. First the server uses the value F_K1 (f_3) of the deletion token to locate the right value $4 \oplus tt_K2$ (f_3) in the deletion table. This will allow the server to get access to the portion of the remaining data structures that need to be updated in a similar fashion with the addition algorithm. Namely it will "free" the positions 4 and 6 in the deletion array and positions 1 and 3 in the search array. While "freeing" the positions in the search array, it will also homomorphically update the pointers of previous entries in the keyword list w_1 and w_3 to point to the new entries (in our example, to the end of the lists–generally in the next pointers of the deleted items). Note that no such an update of pointers is required for the deletion array.

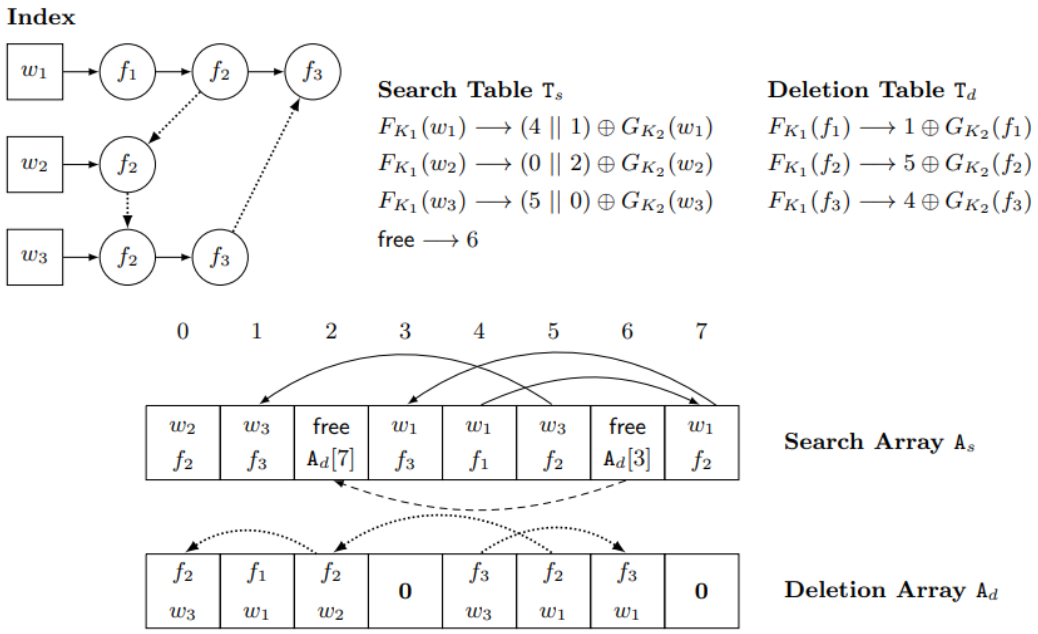

Figure 3: Example of a dynamic encrypted index

3.5 Keyword Red-Black Tree (KRB tree)

In (Kamara and Papamanthou, Parallel and Dynamic Searchable Symmetric Encryption) authors, proposed a construction based on Red-Black trees. Assume that $f = (f_{i_1}, \ldots, f_{i_n})$ is a sequence of documents with corresponding identifiers $i = (i_1, \ldots, i_n)$ over a set of keywords $w = (w_1, \ldots, w_n)$. Each individual f_i is treated as a bit string of polynomial length, (i.e. $f_i = \{0,1\}^{poly(k)}$). Authors use an inverted index, in the sense that each $w_i \in w$ is mapped to a set of document identifiers.

Moreover, authors make the assumption that the universe of keywords is fixed. This is a significant drawback, since it does not allow for a universal solution. Apart from that, their construction is further based on the assumption that the total number of keywords is much smaller than the total number of files, which in our opinion is not realistic.

Their construction makes use of a KRB tree. The KRB tree is a dynamic data structure that can be used to efficiently answer multi-map queries. A KRB tree δ is constructed from a set of documents$f = (f_{i_1}, \ldots, f_{i_n})$ and a universe of keywords $w = (w_1, \ldots, w_n)$. The data structure is constructed as follows:

1. Assume a total order on the documents $\mathbf{f} = (f_{i1}, \ldots, f_{in})$, imposed by the ordering of the identifiers $\mathbf{i} = (i_1, \ldots, i_n)$. Build a red-black tree T on top of $i_1, \ldots, i_n$. At the leaves, store pointers to the appropriate documents. We assume the documents are stored separately, e.g., on disk. Note that this is a slight modification of a red black tree since the tree is constructed on top of the identifiers but the leaves store pointers to the files.
2. At each internal node u of the tree, store an m-bit vector $data_u$. The i-th bit of $data_u$ accounts for keyword w_i, for $i = 1, \ldots, m$. Specifically, if $data_u[i] = 1$, then there is at least one path from u to some leaf that stores some identifier j, such that f_j contains w_i;
3. We guarantee the above property of vectors $data_u$, by computing $data_u$ as follows:

for every leaf l storing identifier j, set $data_l[i] = 1$ if and only if document f_j contains keyword w_i. Now let u be an internal node of the tree T with left child v and right child z. The vector $data_u$ of the internal node u is computed recursively as follows:

$$data_u = data_v \oplus data_z$$

To search for a keyword w in a KRB tree T one proceeds as follows. Assuming that w has position i in the m-bit vectors stored at the internal nodes, check the bit at position i of node v and examine v's children if the bit is 1. When this traversal is over, return all the leaves that were reached. The intuitive reason the KRB tree is useful is that it allows *both* keyword-based operations (by following paths from the root to the leaves) and file-based operations (by following paths from the leaves to the root).

Example: The construction of a dynamic symmetric searchable encryption (DSSE) scheme using the KRB tree data structure, for a collection of $n = 8$ documents indexed over $m = 5$ keywords. Note that for each node v we store two vectors. The encryption of the actual bit of position i at node v is stored to either hash table $\lambda 0_v$ or hash table $\lambda 1_v$, depending on the output of the random oracle. The red arrows indicate the search for keyword 5, returning documents $f3, f6, f7$. Note that the two searches displayed can be parallelized

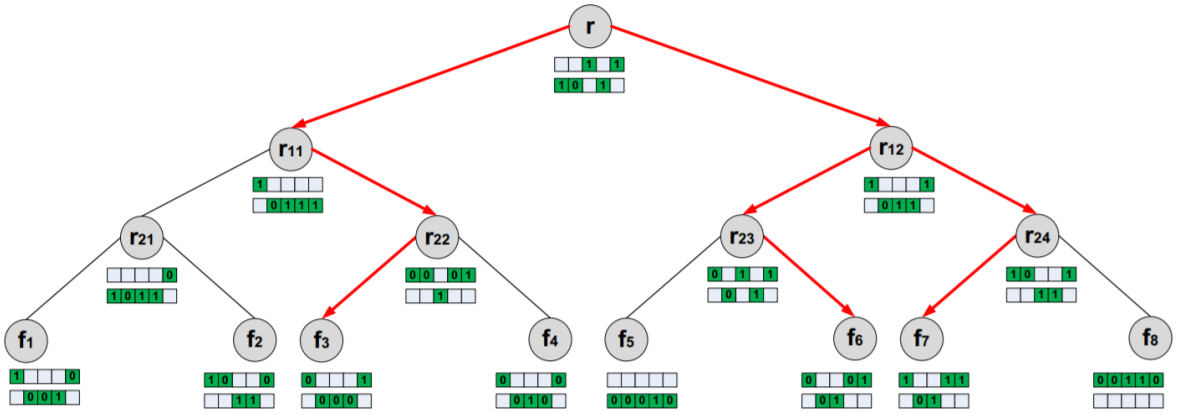

Figure 4: Example of Searching on a KRB Tree

3.6 Blind Storage

Blind storage is a technique introduced in (Naveed, Prabhakaran and Gunter). The scheme is considered to be simpler than the ones described previously. In particular, it does not require the server to support any operation other than upload and download the data. Thus, the server in that scheme can be based solely on a cloud storage service, rather than a cloud computation service as well. The important feature is that the server does not need to carry any decryptions. In linked-lists based constructions as the one described in Section 3.4, each node in the list is progressively revealed. In contrast, Blind Storage allows the server to be free from cryptographic operations and still have a constant number of rounds of interaction. Indeed, the only operations that need to be supported by the server are uploading and downloading blocks of data, and if possible, parallel. Most of the previous SSE schemes that have been presented, use a dedicated server, that performs both storage and computation like in the figure below:

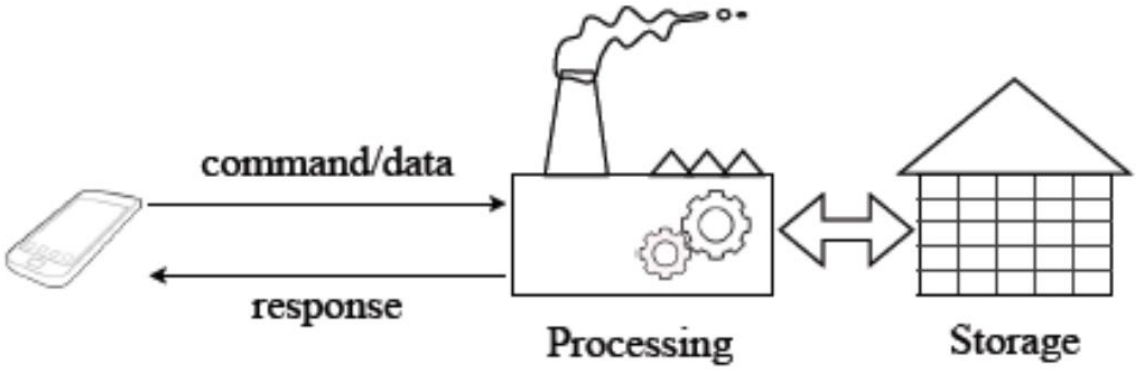

Figure 5: Topology of a Generic SSE scheme

Naturally, to deploy such a scheme, one would need to rely not only on cloud storage services but also on cloud computation services.

The Blind Storage scheme allows a client to store a set of files on a remote server in such a way that the server does not learn how many files are stored, or the lengths of the individual files. Only when a file is retrieved, the server will learn about its existence. A drawback is that the storage server can notice if the same file being downloaded subsequently, but again the file's name and its content are not revealed.

In building the SSE scheme, the search index entries for all the keywords are stored as individual files in the Blind Storage scheme.

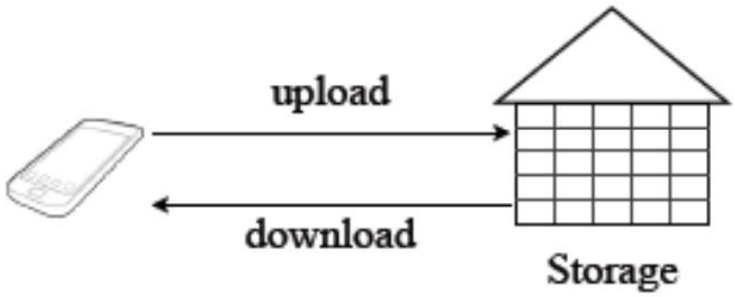

Figure 6: Topology of Blind Storage

Each file is stored as a collection of blocks that are kept in pseudorandom locations, as can be seen in the figure below.

The server sees only a super-set of the locations where the file's blocks are kept, and not the exact set of locations. The key security property, from the point of view of the server, is that each file is associated with a set of locations independently of the other files in the system. On the other hand, the sets of locations for two files can overlap. The only cryptographic tools used in this scheme are block ciphers for standard symmetric key cryptography, as well as for generating pseudorandom locations where the data blocks are kept, and collision resistant hash functions.

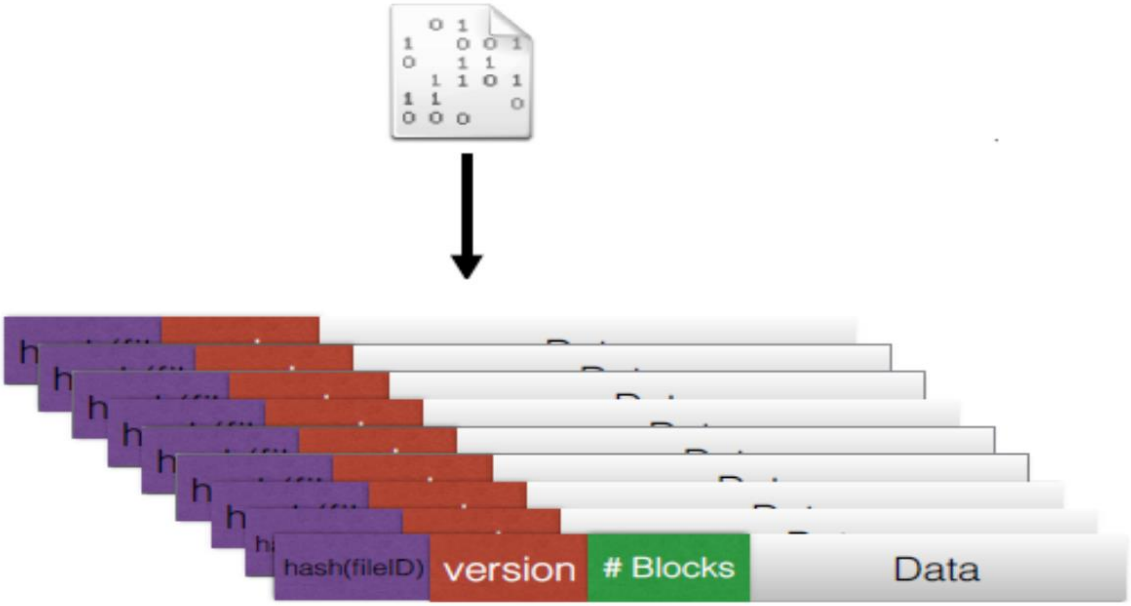

Figure 7: File transforming to a collection of blocks

There is however, an upper bound N on the number of data blocks that can be stored. Given a file f with n blocks, αn locations on the set $\{1, \dots, N\}$ are chosen using a pseudorandom number generator and the n blocks of f are stored in n of these positions. The reason to choose α as many blocks as necessary to store f is that there may be collisions with the storage positions of other files, as already stated. Hence, the αn positions that are retrieved from the server to access f are chosen completely independently from the other files (and so this does not leak any information to the server) and then f is stored encrypted in n of these positions. One issue is that the client needs to know the number of blocks in f to retrieve it, This can be achieved by either storing these information on the client (which is practical if the data collections consists of a small number of relatively large files), or by storing this information in the first block and adding one additional round of interaction, in which the client retrieves the κ first blocks of f. This construction also supports dynamic blind storage, but the updates leak the size of the files. For a typical scenario, one can have a blowup factor $\alpha = 4$.

The idea to obtain an SSE scheme from this blind storage scheme is to store, for all keywords, the search index entries (which lists all the files containing the keyword) as individual files in the blind storage scheme. For dynamic SSE, the original files and the added files are treated by their scheme, which uses two different indexes. The index corresponding to the original files is done using the blind storage scheme and lazy deletion (i.e. after the deletion of one of the original files, the index file of a keyword is not updated before the first search is done for that keyword).

One advantage of this scheme is that the server does not need to perform any computation, but only provide interfaces for uploading and downloading files, which makes the scheme much more transparent for using in cloud environments. However, a significant disadvantage is that it does not provide the same level of security for original and added files. The updates leak a deterministic function of the keywords and so, the security

guarantees for the added files are much weaker than for the original files. This is particularly worrisome for databases that start almost empty and grows over the times-which is often the case in practice.

3.7 Limitations of the SSE schemes and Future Direction

Symmetric Searchable Encryption faces three problems that renders it impractical for wide commercial use. In this sub-section we address both of them and describe the steps we plan to take in order to face them.

- ***Revocation:*** Taking into consideration the aforementioned works, is seems as if there is no efficient way to embed a revocation mechanism in an SSE scheme. Thus, the only way to revoke a user and deny his/her access to the encrypted database, is to download the entire database, decrypt it, re-encrypt it with a fresh key and upload everything again on the storage server. Naturally, this solution does not scale well since for big data volumes, this procedure is very computationally expensive. To address the problem of revocation, we plan to develop hybrid encryption techniques that will combine Symmetric Searchable Encryption with Attribute-Based Encryption (ABE) (Bethencourt, Sahai and Waters) and will make further use of a trusted execution environment. ABE allows users to encrypt their data, and bind them with a policy. At the same time, each user receive a unique secret key with attached attributes. This way, a user can decrypt the ciphertext if and only if his/her attributes satisfy the policy bound on it. Such a technique can be used to encrypt the SSE secret key along with a policy. As a result, only users that satisfy the policy bound to the key will be able to access the database. Moreover, by changing the policy bound to the key, the data owner will be able to efficiently revoke users from her database.
- ***Multi Writers/Multi Readers:*** The most efficient SSE schemes so far, address the single client model. In other words, they propose solutions where the data owner encrypts his/her data, uploads them to the storage server and then searches through his/her encrypted database. Obviously, even if such a scheme seems ideal for a casual user who simply wants to store data online, this is a limitation that prevents companies and big industrial players to adopt SSE in their work environments. To this end, we have designed a scheme that will allow data owners to securely share their SSE keys with multiple other users so that they will all be able to access and edit the same database. Naturally, giving the ability multiple users to add files , renders the scheme vulnerable to file injection attacks. Thus, the aforementioned scheme will need to be forward private, to offer the maximum level of security.
- ***Threat Model:*** Most constructions so far are purely theoretical and their security is proven in the random oracle model by assuming a semi-honest adversary. This prevents key industrial players from adopting such a techniquq since they are looking for realistic and practical scenatios. To deal with this problem, we present a protocol in Section 6 those security is proven against realistic malicious adversaries.

Table 4: Comparison

SCHEME	**Dynamic**	**Multi-Client**	**Forward Privacy**	**Range Queries**
Two-Layered Encryption Scheme	✗	✗	✗	✗
Forward Index	✓	✗	✗	✗
Inverted Index	✓	✗	✗	✗
KRB	✓	✗	✗	✗
Blind Storage	✓	✗	✗	✗
Ours	✓	✓	✓	✓

4 System Model

The system model is derived from (Tampere Univeristy (TUNI), Norwegian Centre for eHealth Research (NSE)) where the reference architecture was described in detail. In this section, we describe the entities needed for the proper run of the DSSE scheme that we will describe later.

Users: We denote by $\mathcal{U} = \{u_1, u_2, \dots, u_n\}$ the set of all users that have been already registered in a cloud service that allows them to store, retrieve, update, delete and share encrypted files while at the same time being able to search over encrypted data by using the ASCLEPIOS DSSE scheme. The users in the ASCLEPIOS system model are mainly classified into two categories: data owners and registered users that they have not yet upload any data to the CSP. The role of data owner is the most important for this study since he/she is the one who actually uses the main functions of this scheme to encrypt data locally and create a dictionary that will be sent to the CSP. More precisely, a data owner first needs to locally parse all the data that wishes to upload to the CSP. During this process, the data owner generates three different indexes:

- No.Files[w] contains a hash of each keyword *w* along with the number of files that *w* can be found at.
- No.Search[w] contains the number of times a keyword *w* has been searched by a user.
- Order[w] contains hashes of the keywords, sorted by the plaintext.
- Dict is a dictionary that maintains a mapping between keywords and filenames.

No.Files[w], No.Search[w], Order[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 achieve the multi-client model, the data owner outsources *No.Files[w], No.Search[w]* and *Order[w]* to a trusted authority TA on the cloud, but also keeps a copy locally. These indexes will allow registered users to create consistent search tokens. *Dict* is finally sent to the CSP.

Cloud Service Provider(CSP): We consider a cloud computing environment similar to the one described in (Paladi, Gehermann and Michlas). The CSP must support TEE since core entities will be running in the trusted execution environment offered by TEE. The CSP storage will consist of the ciphertexts as well as of the dictionary *Dict*. Each entry of *Dict* is encrypted under a different symmetric key K_W. Thus, given K_W and the number of files containing a keyword *w*, the CSP can recover the files containing *w*.

Trusted Authority (TA): TA is an index storage that stores the *No.Files* and *No.Search* indexes that have been generated by the data owner. All registered users can contact the TA to access the *No.Files w* and *No.Search w* values for a keyword *w*. These values are needed to create the search tokens that will allow users to search directly on the encrypted database. Similarly to the CSP, the TA is also TEE enabled.

Trusted Execution Environment (TEE): Our construction requires a TEE like the one described in (Tampere Univeristy (TUNI), Norwegian Centre for eHealth Research (NSE)). In particular, we want that the TEE should satisfy the following properties:

- **Isolation:** We require the TEE to offer an isolated environment located in a hardware guarded area of memory. The TEE needs to be based on memory isolation built in the processor along with strong cryptogaphy. The processor will track which parts of memory belong to which enclave and ensures that only enclaves can attest their own memory.
- **Sealing:** The TEE must come with a special Key with which, data is encrypted when stored in untrusted memory. Sealed data should be able to be recovered even after the isolated environment is destroyed and rebooted on the same platform
- **Attestation:** The TEE should finally support local and remote attestation. In the case of local attestation, the TEE will be able to verify another TEE, as well as the program/software running in the later, on the same platform. This will be achieved through a cyrptographic message authentication code (MAC) in which all the isolated environments on the same platform, share a key. In the case of remote attestation, the verfication will be achieved by a public key signature scheme. Thus, an Attestation

server is required (like the one described in (Tampere Univeristy (TUNI), Norwegian Centre for eHealth Research (NSE))).

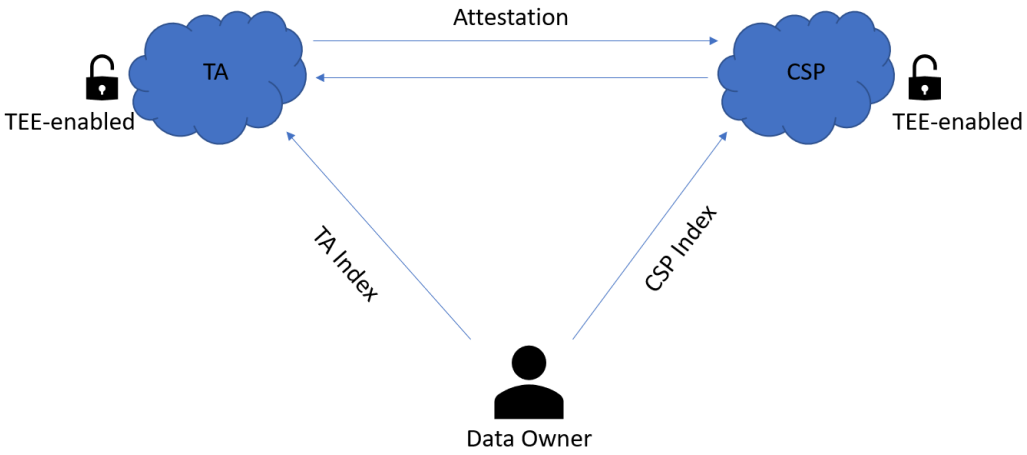

Figure 8: System Model

5 ASCLEPIOS SSE

5.1 High-Level Overview

In this section we provide a high-level overview of the ASCLEPIOS encryption scheme.

The server stores an encrypted index that associates each keyword with a set of file identifiers representing the files that contain that keyword. The entries associated with a keyword *w* are encrypted with a key K_w that depends on both *w* and the number of times *w* has been searched for. K_w is derived using a pseudorandom function from a master key that is stored by the client and by the TA. So, the TA must store the number of times each keyword has been searched for so far. We do this in a dictionary called *No.Search[w]*. To search for a keyword, the client generates and reveals the key K_w to enable the server to operate on the appropriate entries in the encrypted index, find the identifiers of files sharing the keyword, and return the corresponding (encrypted) files.

Incrementing the number of times *w* has been searched for on each search leads to a new key K_w be generated for *w* and invalidates the previous key revealed to the server. This ensures the freshness of K_w on each search. Therefore, if *w* appears in a new file being added, the corresponding entry will be encrypted under a fresh key and the server cannot link it to the previous searches and realize that the new file contains *w*. This provides the essential *forward privacy* property. On the downside, this requires another round of interaction (at the end of search) to encrypt the accessed index entries with the new key and upload them back to the server. (We ask the server to remove the accessed entries from the index during a search. The server can keep the deleted entries, but they have already been leaked and contain no new information.) Note that this does not increase the asymptotic search cost. Besides, we can eliminate the extra round using piggybacking and upload the current updated entries together with the next search token. The whole process ensures that no entry of the outsourced index is encrypted under a revealed key. Furthermore, the revealed keys will never be used again.

Another requirement for forward privacy is that the identifiers of all files containing a keyword cannot be stored in an easily linkable fashion (e.g., in a set or a file). Otherwise, adding a new file would trivially reveal which of the previously searched for keywords are contained in this new file. It may further leak information about other files the new file shares keywords with. This requires the identifiers of all files containing each keyword *w* to be stored at random locations in the index that are also determined by K_w.

This solution immediately enables parallelism for efficiency. A sequence number is assigned to each file ID among the set of files containing a given keyword. These sequence numbers are used to generate the addresses at which the respective encrypted file IDs will be stored. Therefore, given the total number of file IDs, the provider can divide it by the number of available servers and ask each one to extract a subset of file IDs to be returned. The client again needs to store the total number of file IDs sharing a keyword in a dictionary named *No.Files[w]*.

Example: We give an example to better illustrate the client (data owner and TA) and server indexes and how the protocols operate on them. Assume that there are three files and four keywords: f_1 contains w_1, w_3, w_4, f_2 contains w_1, w_2, f_3 contains w_1, w_4. Let $g(w, f, i)$ denote the masked version of the id of the file after the i^{th} search for *w* to be stored in *Dict*. Hence, $g(w_1, f_1, 0), g(w_3, f_1, 0), g(w_4, f_1, 0)$ are added into *Dict* for f_1, $g(w_1, f_2, 0), g(w_2, f_2, 0)$ for f_2 and $g(w_1, f_3, 0), g(w_4, f_3, 0)$ for f_3.

No.Files	
w_1	3
w_3	1
w_4	2
w_2	1

No.Search	
w_1	0
w_3	0
w_4	0
w_2	0

Dict
$g(w_4, f_3, 0)$
$g(w_1, f_1, 0)$
$g(w_2, f_2, 0)$
$g(w_1, f_2, 0)$
$g(w_3, f_1, 0)$
$g(w_4, f_1, 0)$
$g(w_1, f_3, 0)$

Figure 9: TA and Server indexes after adding files

Now, the user wishes to search for w_1. She sends the server the key K_1 and the number of files w_1 appears in (i.e. 3). The server locates the given number of *Dict* entries, decrypts the contents and sends the files to the user. The user increments the number of times w_1 is searched for as: $No.Search[w] + +$. Then it re-encrypts the received pairs with a fresh key generated using the updated $No.Search[w]$, and sends them back to the server. The server stores them in their new locations in *Dict*. Finally, an acknowledgement is sent to the TA, so that it will also update its local indexes.

No.Files	
w_1	3
w_3	1
w_4	2
w_2	1

No.Search	
w_1	1
w_3	0
w_4	0
w_2	0

Dict
$g(w_1, f_1, 1)$
$g(w_4, f_3, 0)$
$g(w_2, f_2, 0)$
$g(w_1, f_3, 1)$
$g(w_3, f_1, 0)$
$g(w_1, f_2, 1)$
$g(w_4, f_1, 0)$

Figure 10: TA and server indexes after searching for w1.

5.2 The Encryption Scheme

Our scheme consists of seven algorithms:

- KeyGen: Generates all the necessary keys.
- InGen: Generates all the indexes.
- Search: Allows users to search for a specific keyword over the encrypted data.
- Add: Allows users to add new files to the encrytped database.
- Delete: Allows users to delete files from the encrypted database.
- Range Search: Allows users to issue range queries over the encrypted data
- Modify: Allows a user to modify an encrypted file.

Below, we provide a detailed description of all seven algorithms.

Key Generation: This algorithm generates the secret key $\boldsymbol{K} = (K_G, K_{SKE})$, where K_G is a key for the *IPRF G* and K_{SKE} is the key for the CPA-Secure symmetric key encryption scheme. K_G is also sent to the TA. This is a probabilistic algorithm run by the data owner.

Algorithm 1: ASCLEPIOS.KeyGen

Input: Security parameter λ
Output: K ← Gen$(1^\lambda)(1^\lambda)$
1. $K_G \leftarrow$ GenIPRF(1^λ)
2. $K_{SKE} \leftarrow$ SKE.Gen(1^λ)
3. return **K** = (K_G, K_{SKE})

Indexing: After the data owner generates the secret key ***K,*** he/she will generate the indexes needed for the scheme. In particular, she will generate three indexes in total: *No.Files[w],* where she stores the total number of files sharing a keyword, *No.Search[w],* where he/she stores the number of times each keyword has been searched for, *Order[w]* where he/she stores hashes of the keywords sorted by the plaintext and finally, *Dict*, in which every keyword is mapped to the set of file identifiers in which the keyword appears. *No.Files[w], No.Search[w]* and *Order[w]* are outsourced to the TA but the data owner also keeps a copy locally. On the other hand, *Dict* is sent directly to the CSP. This protocol is treated like a set of AddFile protocols; thus the data owner is required to internally run AddFile. Note that upon its generation, *Dict* is already encrypted and thus, is directly sent to the CSP. However, this is not the case for *No.Files[w], No.Search[w]* and *Order[w]*. As a result, before outsourcing these indexers to the TA, they need to be encrypted under TA's public key. Although this process is not characterized by its efficiency, it will only occur once and it is a necessary trade-off to achieve a multi-client scheme. Upon reception, TA decrypts the indexes using its private key and stores them in plaintext.

Algorithm 2: ASCLEPIOS.InGen

Input: K, F
Output: $(In_{CSP}, C) \leftarrow$ InGen$(K)(F)$
1. $Cipher = \{\}$
2. $AllMap = \{\}$
3. **for all f_i do**
4. Run AddFile to generate c_i and Map_i (Results are NOT sent to the CSP)
5. $Cipher = Cipher \cup c_i$
6. $AllMap = AllMap \cup Map_i$
7. $In_{TA} = (\text{No. Files}[w], \text{No. Search}[w], \text{Order}[w])$
8. Send $(AllMap, C)$
9. Send In_{TA}
10. CSP stores $AllMap$ in a dictionary Dict
11. $In_{CSP} = \{\text{Dict}\}, In_{TA} = \{\text{No. Files}, \text{No. Search}, \text{Order}\}$

File Insertion: The data owner u_i can add files to his/her collection, even after the execution of Algorithm 1. To do so, he/she retrieves the *No.Files[w], No.Search[w]* and *Order[w]* indexes that are stored locally on his/her device. These indexes will allow her to create an add token $\tau_\alpha(f)$, for the file f that he/she wishes to add to his/her collection. For each distinct keyword $w_i \in f$, he/she increments *No.Files[w_i]* by one and then computed the corresponding address on *Dict.* Moreover, he/she encrypts the files wishes to add as $c \leftarrow Enc(K_{SKE}, f)$ and sends the results to the CSP. As a last step, u_i sorts the new keywords w_i, hashes them, inserts them to *Order[w]* and sends an acknowledgement to the TA to update its indexes as well.

Algorithm 3: ASCLEPIOS.AddFile

Input: K, f, In_{TA}
Output: $(In'_{CSP}, C'), In'_{TA} \leftarrow$ Add$(K, f, In_{TA})(In_{CSP}, C)$
1. $Map = \{\}$
2. **for all** $w_i \in f$ **do**
3. $\text{No. Files}[\text{w}_\text{i}] + +$

4. $K_{w_i} = G(K_G, h(w_i)||\text{No. Search}[\text{w}_\text{i}])$
5. $addr_{w_i} = h(K_{w_i}||\text{No. Files}[\text{w}_\text{i}])$
6. $val_{w_i} = Enc(K_{SKE}, id(f_i)||\text{No. Files}[\text{w}])$
7. $Map = Map \cup \{addr_{w_i}, val_{w_i}\}$
8. $c \leftarrow SKE.Enc(K_{SKE}, f)$
9. $\tau_\alpha(f) = (c, Map)$
10. Send $\tau_\alpha(f)$ to the CSP
11. CSP adds c_i to C and Map to Dict
12. Send the updated value of No.Files to the TA
13. TA updates No.Files

Search: We now assume that the data owner has successfully shared his/her secret key **K** with multiple users to give them access to his/her encrypted data. For a user u_j to create the search token $\tau_s(w)$ for a specific keyword w, the user first needs to request the corresponding *No.Files[w]* and *No.Search[w]* values from the TA. After u_j receives these values, the user can compute the key K_w as $K_w = G(K_G, h(w)||\text{No. Search}[\text{w}])$. Apart from that, he/she increments the *No.Search[w]* value by one and computes the updated key for w, K_w', and the new addresses $addr_w$ for *Dict*. The user will finally store the new addresses in a list L that will be sent to the CSP. Upon reception, *Dict* will forward $(K_w, \text{No. Files}[\text{w}])$ to the TA to ensure that u_j sent the correct values. At this point, TA will retrieve w and *No.Search[w]* by inverting the *IPRF G*. In particular, since the TA holds the key for the IPRF G (K_G), it can compute the following:

$$G^{-1}(K_G, K_w) = G^{-1}(K_G, G(h(w)||\text{No. Search}[\text{w}]) = h(w)||\text{No. Search}[w]$$

As soon as TA retrieves these values, is can compute K_w' by incrementing the *No.Search[w]* value by one. Furthermore, it will also compute the updated addresses for *Dict*. These addresses will be stored in a list L_{TA} that will be sent to the CSP. Upon reception, the CSP will check whether $L = L_{TA}$ or not. If $L \neq L_{TA}$, the CSP will output an error message $\perp$ and abort the protocol. Otherwise, the CSP locates the file identifiers from the list of addresses it received, sends the files identifiers back to u_j and updates the addresses with the new ones. Finally, the CSP sends an acknowledgement to the TA that the search is completed so that the TA will increment *No.Search[w]* by one. This acknowledgement is also sent to the data owner, to update his/her local indexes.

Algorithm 4: ASCLEPIOS.Search

Input: K, w, $\boldsymbol{In_{TA}}$
Output: $(\boldsymbol{In'_{CSP}}, \boldsymbol{I_w}), \boldsymbol{In'_{TA}} \leftarrow \boldsymbol{Search}(\boldsymbol{K}, \boldsymbol{w}, \boldsymbol{In_{TA}})(\boldsymbol{In_{CSP}}, \boldsymbol{C})$
User:

1. Request the values No.Files[w] and No.Search[w] from the TA

TA:

2. Verifies the user and send back the values

User:

3. $K_w = G(K_G, h(w)||\text{No. Search}[\text{w}])$
4. $No.Search[w] + +$
5. $K'_w = G(K_G, h(w)||\text{No. Search}[\text{w}])$
6. $L_u = \{\}$
7. **for** $i = 1$ **to** $i = \text{No. Files}[\text{w}]$ **do**
8. $\quad addr_w = h(K'_w, i||0)$
9. $\quad L_u = L_u \cup \{addr_w\}$
10. Send $\tau_s(w) = (K_w, \text{No. Files}[\text{w}], L_u)$ to the CSP

CSP:

11. Forward $K_w, \text{No. Files}[\text{w}]$ to the TA

TA:

12. $w||\text{No. Search}[\text{w}] = G^{-1}(K_G, K_w)$
13. $\text{No. Search}[\text{w}] + +$
14. $K'_w = G(K_G, h(w)||\text{No. Search}[\text{w}])$
15. $L_{TA} = \{\}$
16. **for** $i = 1$ **to** $i = \text{No. Files}[\text{w}]$ **do**
17. $addr_w = h(K'_w, i||0)$
18. $L_{TA} = L_{TA} \cup \{addr_w\}$
19. Send L_{TA} to the CSP

CSP:

20. **if** $L_u = L_{TA}$ **then**
21. $I_w = \{\}$
22. **for** $i = 1$ **to** $i = \text{No. Files}[\text{w}]$ **do**
23. $c_{id(f_i)} = \text{Dict}[h(K_w, i||0)]$
24. $I_w = I_w \cup \{c_{id(f_i)}\}$
25. Delete $\text{Dict}[h(K_w, i||0)]$
26. Add the new addresses as specified by L_u
27. **else**
28. Output $\perp$
29. Send an acknowledgement to the user and acknowledgement to the TA and the data owner

TA:

30. $\text{No. Search}[\text{w}] + +$

User:

31. $\text{No. Search}[\text{w}] + +$

Range Queries: For a user u_j to issue a range query to the CSP, he/she first needs to contact TA. In particular, if u_j wishes to search for all files containing values in the range $[a, b]$, the user first hashes the extreme values of the range (i.e. $h(a)$ and $h(b)$) and sends them to the TA. Upon reception, TA retrieves the sorted index *Order[w],* locates the values $h(a)$ and $h(b)$ and sends back to u_j every entry that lies in between of $h(a)$ and $h(b)$. Finally, for each $h(w_i)$ returned, u_j performs a search operation just as described in Algorithm 3.

Algorithm 5: ASCLEPIOS.RangeSearch

Input: K, $\boldsymbol{h(\alpha)}$**,** $\boldsymbol{h(\beta)}$ $\boldsymbol{In_{TA}}$
Output: $(\boldsymbol{In'_{CSP}}, \boldsymbol{I_w}), \boldsymbol{In'_{TA}} \leftarrow \boldsymbol{Search}(\boldsymbol{K}, \boldsymbol{w_i}, \boldsymbol{In_{TA}})(\boldsymbol{In_{CSP}}, \boldsymbol{C})$

User:

1. Send $h(\alpha), h(\beta)$ to the TA

TA:

2. Locate $h(\alpha), h(\beta)$ on Order [w]. *Let* Order$[\alpha] = h(a)$ *and* Order $[\beta] = h(\beta)$
3. $L_{ord} = \{\}$
4. **for** $i = \alpha$ **to** $i = \beta$ **do**
5. $L_{ord} = L_{ord} \cup Order[i]$
6. Return L_{ord} to u_j

User:

7. **for all** $\boldsymbol{h(w_i)} \in \boldsymbol{L_{ord}}$ **do**
8. Run ASCLEPIOS.Search

File Deletion: A data owner u_i can also delete files. To do so, he/she sends a request to the CSP requesting the file f to be deleted. Note here that since the file names are encrypted, the CSP can not learn which file will be deleted. After u_i receives f, he/she decrypts it locally, extracts every keyword and updates his/her local indexers accordingly. The data owner will also send an acknowledgement to the TA to also update its indexers. Finally, u_i needs to

compute the new addresses and values for each keyword $w_i \in f$. These addresses will be sent to the CSP who will proceed with the deletion of every *Dict* entry associated to f.

Algorithm 6: ASCLEPIOS.Delete

Input: $\boldsymbol{c_{id(f)}}$
Output: $(\boldsymbol{In'_{CSP}}, \boldsymbol{C'}), \boldsymbol{In'_{TA}} \leftarrow \boldsymbol{Delete}(\boldsymbol{K}, \boldsymbol{c_{id(f)}}, \boldsymbol{In_{TA}})(\boldsymbol{In_{CSP}}, \boldsymbol{C})$
Data Owner:

1. $FileNumber = \{\}$
2. **for all** $w_i \in f$ **do**
3. **if** $\text{No. Files}[\text{w}_\text{i}] > 1$ **then**
4. $addr_{w_i} = h(K_{w_i}, \text{ No. Files}[\text{w}_\text{i}]||0)$
5. $val_{w_i} = Enc(K_{SKE}, id(f), \text{No. Files}[\text{w}_\text{i}])$
6. $No.Files[w_i]$
7. $newaddr_{w_i} = h(K_w, \text{No. Files}[\text{w}]||0)$
8. $newval_{w_i} = Enc(K_{SKE}, id(f)||\text{No. Files}[\text{w}])$
9. **else**
10. $newaddr_{w_i} = 0$
11. $newval_{w_i} = 0$
12. Delete No.Files[w], No.Search[w] and Order[w]
13. $FileNumber = FileNumber \cup \{h(w), \text{No. Files}[\text{w}]\}$
14. Send $FileNumber$ to the TA
15. $\tau_d(f) = \{K_w, (addr_{w_i}, newaddr_{w_i}), (val_{w_i}, newval_{w_i})\}_{i=1}^{\#w \in f}$.
16. Send $\tau_d(f)$ to the CSP

TA:

17. **for all** $w_i \in f$ **do**
18. **if** $No.Files[w] > 1$ **then**
19. $No.Files[w] - -$
20. **else**
21. Delete No.Files[w_i], No.Search[w_i] and Order[w_i]

CSP:

22. **For** $i = 1$ **to** $i = \#w \in f$ **do**
23. **if** $newaddr_{w_i} = 0$ **then**
24. Delete $addr_{w_i}$ and val_{w_i}
25. **else**
26. $addr_{w_i} = newaddr_{w_i}$
27. $val_{w_i} = newval_{w_i}$

File Modification: Finally, the dataowner u_i can modify/update a file. If u_i wishes to modify a file f that is stored online, he/she first needs to run the Delete algorithm to make sure that each entry associated with f will be deleted and that all indexes will be updated accordingly. Recall that during the delete process, u_i has received a local copy of f that has also decrypt it. As a result, u_i modifies the decrypted f locally and then runs the AddFile algorithm for the updated file.

Algorithm 1: ACLEPIOS.Modify

Input: $\boldsymbol{c_{id(f)}}$
Output: $(\boldsymbol{In'_{CSP}}, \boldsymbol{C'}), \boldsymbol{In'_{TA}} \leftarrow \textbf{Add}(\boldsymbol{K}, \boldsymbol{f}, \boldsymbol{In_{TA}})$

1. Run ASCLEPIOS.Delete with f as input
2. Modify f
3. Run ASCLEPIOS.AddFile with f as input

6 A Security Protocol Based on ASCLEPIOS SSE

In this section we present a detailed security protocol that shows how the scheme described earlier can be incorporated into a cloud-based service. The description of such a protocol is important not only because it allows us to assume a stricter adversarial model but also because it can give valuable insights to developers on how to incorporate such schemes into their existing cloud-based services.

For the description of the protocol we assume the existence of an IND-CCA2 secure public key cryptosystem PKE= (Gen, Enc, Dec) and an EUF-CMA secure signature scheme $S = (\text{sign}, \text{verify})$. Every entity participating in the protocol has a public/private key pair, as well as a signature/verification key pair. Our protocol protocol is divided into two main phases, the ***Setup phase***, and the ***Running Phase***.

6.1 Setup Phase

6.1.1 Key Generation

In the Setup Phase, a data owner u_i runs the KeyGen algorithm to generate the secret key **K**= (K_G, K_{SKE}). K_G will then be sent to the TA. To this end, u_i generates and sends $m_1 = < r_1, Enc_{pk_{TA}}(K_G), \sigma_i(H(r_1||K_G)) >$ where r_1 is a random number used to ensure the freshness of the message, σ_i is u_i's signature and H is a cryptographic hash function.

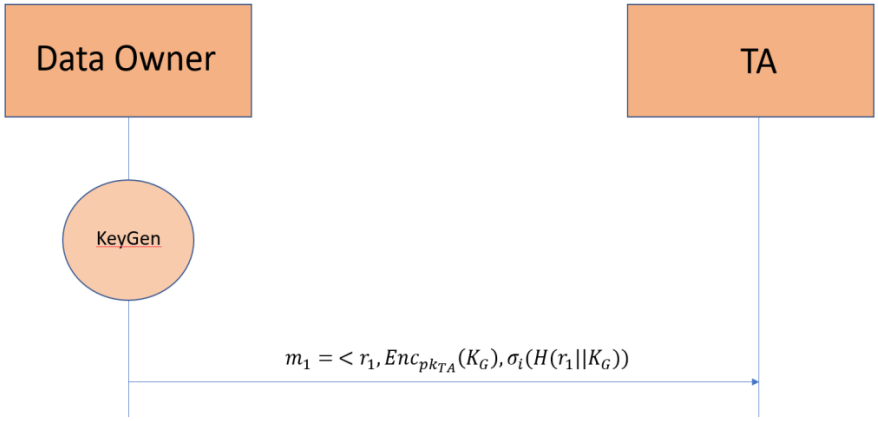

Figure 11: Key Exchange

6.1.2 Index Generation

As a next step, u_i runs the InGen algorithm to generate the four indexes needed and the encrypted files. After the successful execution on InGen the data owner sends $m_2 = < r_2, Enc_{pk_{TA}}(In_{TA}), \sigma_i(H(r_2||In_{TA})) >$ to the TA. He/she also sends $m_3 = < r_3, C, Dict, s_i(H(r_3||C||Dict)) >$ to the CSP. Upon reception, both entities will verify the integrity and the freshness of the messages and will store the indexes locally. Note here, that *Dict* is already encrypted upon its generation, so there is no need to encrypt it with the CSP's public key. Moreover, the size of the indexes stored in the TA are all $O(m)$, where m is the total number of keywords. Hence, if we consider a case with 1 million keywords with the size of an integer being at 4 bytes and the average size of a keyword at 10 bytes, then the total size required for the three indexes will be:

$$1 \times 10^6 \times (10 + 4 + 10 + 4 + 10 + 4) = 10^6 \times 42 = 42MB$$

As a result, storing a very large number of keywords does not require computationally powerful machines.

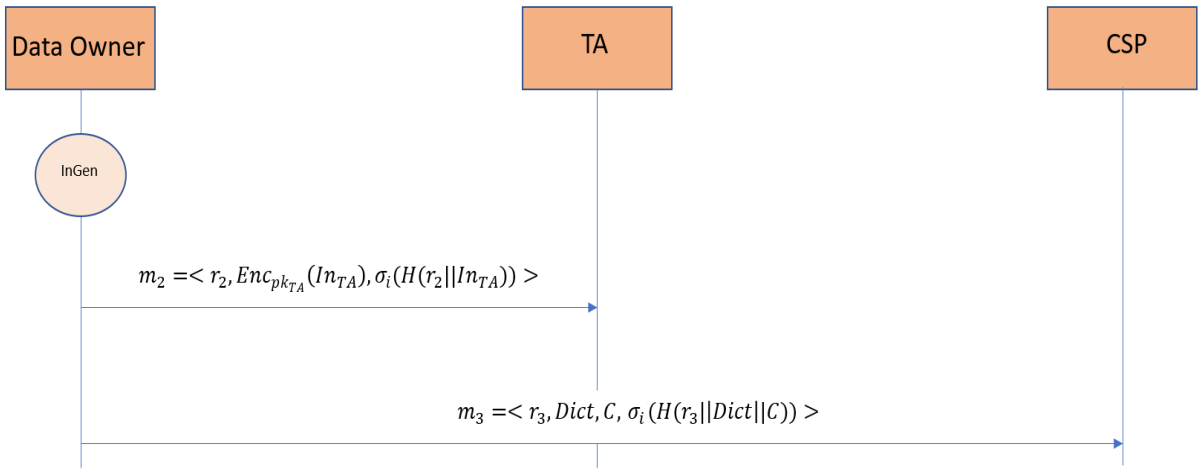

Figure 12: Outsourcing the Indexes

6.2 Running Phase

In the Running Phase, the data owner u_i can update his/her collection of encrypted files by adding or deleting files but most importantly, share these files with other users.

6.2.1 File Addition

After the completion of the Setup Phase, the data owner u_i can still add files to his/her encrypted database by running the AddFile algorithm. To do so, the data owner first genrates the add token $\tau_\alpha(f)$ for the file f, and then sends $m_4 =< r_4, \tau_\alpha(f), \sigma_i(H(r_4||\tau_\alpha(f))) >$ to the CSP and $m_5 =< r_5, \{h(w_i), Enc_{pk_{TA}}(n_i)\}_1^{\#w\in f}, \sigma_i(H(r_5||\{h(w_i), Enc_{pk_{TA}}(n_i)\}_1^{\#w\in f})) >$, where $n_i \in \mathbb{N}$ represents the No.Files[w_i], to the TA.

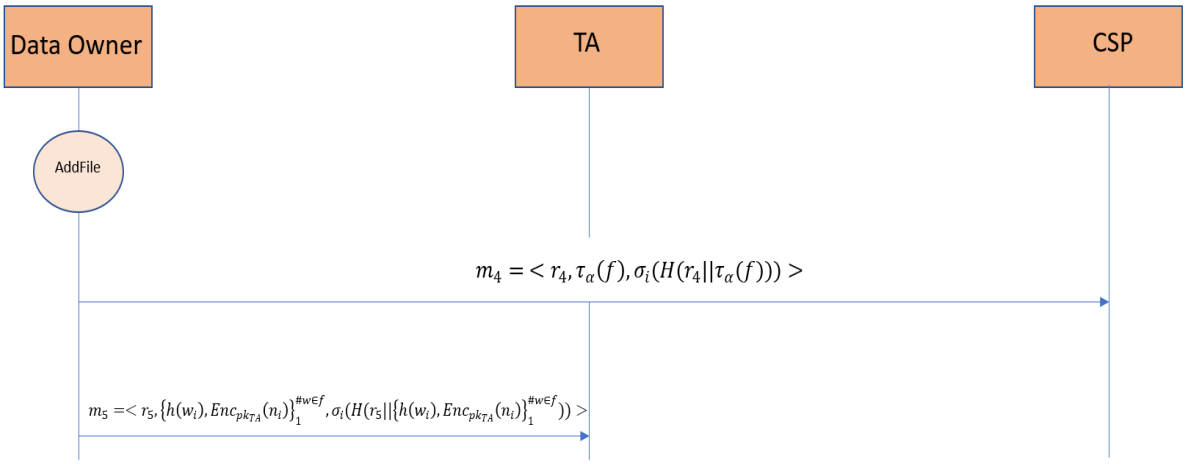

Figure 13: File Addition

6.2.2 Searching

We now assume that the data owner u_i has successfully shared his/her secret key **K** with other users. For a user u_j to search for the files containing a keyword w, the user first contacts TA to get $n_1 = No.Files[w]$ and $n_2 = No.Search[w], (n_1, n_2 \in \mathbb{N})$, to be able to compute K_w. To this end, he/she sends $m_6 =< r_6, h(w), \sigma_j(H(r_6||h(w)))$ to the TA and receives back $m_7 = < r_7, Enc_{pk_{u_j}}(n_1, n_2), \sigma_{TA}(r_7||n_1||n_2) >$. Now, u_j can finally construct the search token $\tau_s(w)$. And so the user sends $m_8 =< r_8, \tau_s(w), \sigma_{u_i}(H(r_8||\tau_s(w))) >$ to the CSP. Upon reception, the CSP forwads this message to the TA that will compute L_{TA} based on K_w and $No.Files[w]$. Then, TA sends $m_9 =< r_9, , \sigma_{TA}(H(r_9||L_{TA})) >$ back to the CSP. The CSP will then check whether $L_{TA} = L_{u_j}$, where $L_{u_j} \in \tau_s(w)$. If not, the CSP will output $\perp$ and abort the protocol,

Otherwise, it will find the files containing w and will send them back to u_j. Furthermore, the CSP sends an akcnowledgement to both the TA and the data owner u_i, so that they will increment the $No.Search[w]$ value by one.

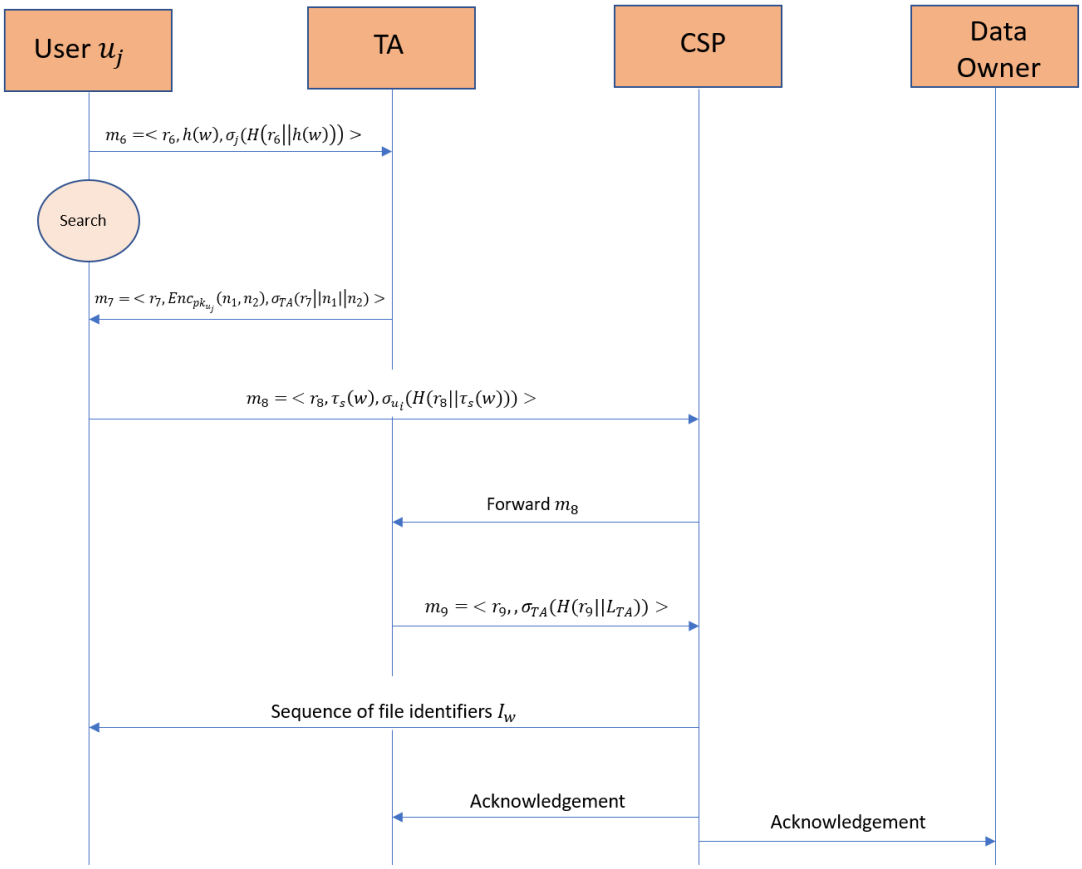

Figure 14: Search

Range Queries: A user can also perform range queries. A range query could be the following actions. "Search for all patients with disease D_1 that are older than 25 years old." Such queries can be very useful in the healthcare section, since they allow the interested party to search for groups of patients that their data will be later used for privacy preserving analytics. For a user to perform a range query, one extra round of interaction between the user and the TA is needed. In particular, the user u_j initiates the protocol by sending $m_{range} =< r_{range}, h(\alpha), h(\beta), \sigma_{u_j}(r_{range}||h(\alpha), h(\beta)) >$ to the TA. The TA then retrieves *Order[w]*, where all the keywords are sorted by the plaintext, locates $h(\alpha)$ and $h(\beta)$ and sends back to u_j every *Order[w]* entry that lies between $h(\alpha)$ and $h(\beta)$ via $m_{values} =< r_{values}, \{h(w_i)\}_{\alpha \le w_i \le \beta}, \sigma_{TA}(H(r_{values}||\{h(w_i)\}_{\alpha \le w_i \le \beta})) >$. Upon reception, u_j creates a search token for each $h(w_i)$ that she received and initiates the search protocol.

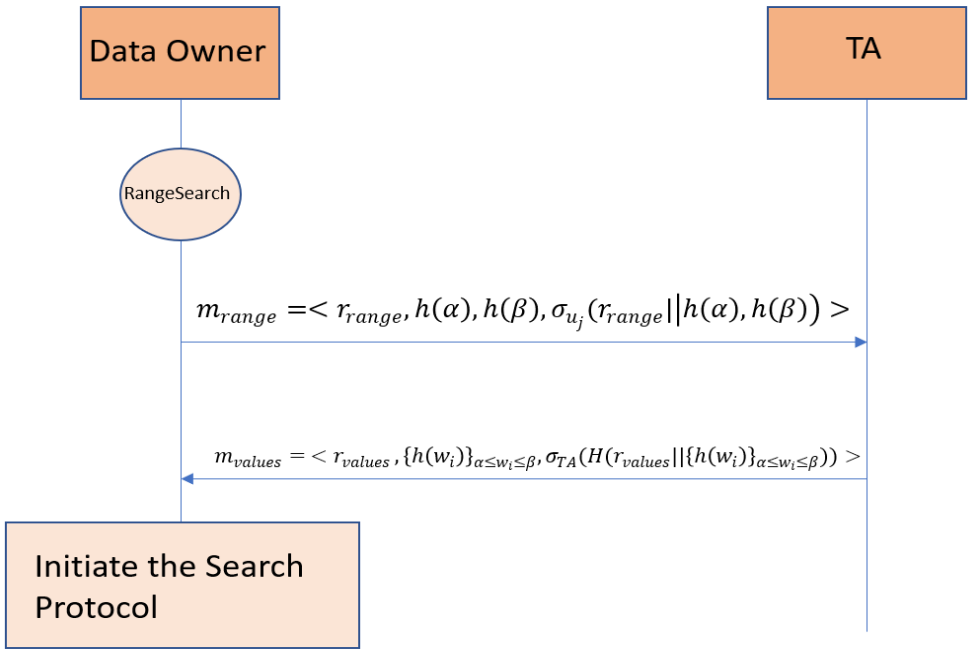

Figure 15: Range Queries

6.2.3 File Deletion

The data owner u_i can also delete files from his/her database. To do so, the owner first requests from the CSP the the file he/she wants to delete. Note, that since the filenames are encrypted, the CSP does not learn which file will be deleted. After, u_i receives the file, he/she can create the delete token. Next, the data owner u_i sends the delete token to the CSP via $m_{10} =< r_{10}, \tau_d(f), \sigma_{u_i}\left(H\left(r_{10}||\tau_d(f)\right)\right) >$. Moreover, he/she sends the list FileNumber to the TA via $m_{11} =< r_{11}, FileNumber, \sigma_{u_i}(H(r_{11}||FileNumber)) >$. Finally, the CSP deletes each *Dict* entry associated to the file f.

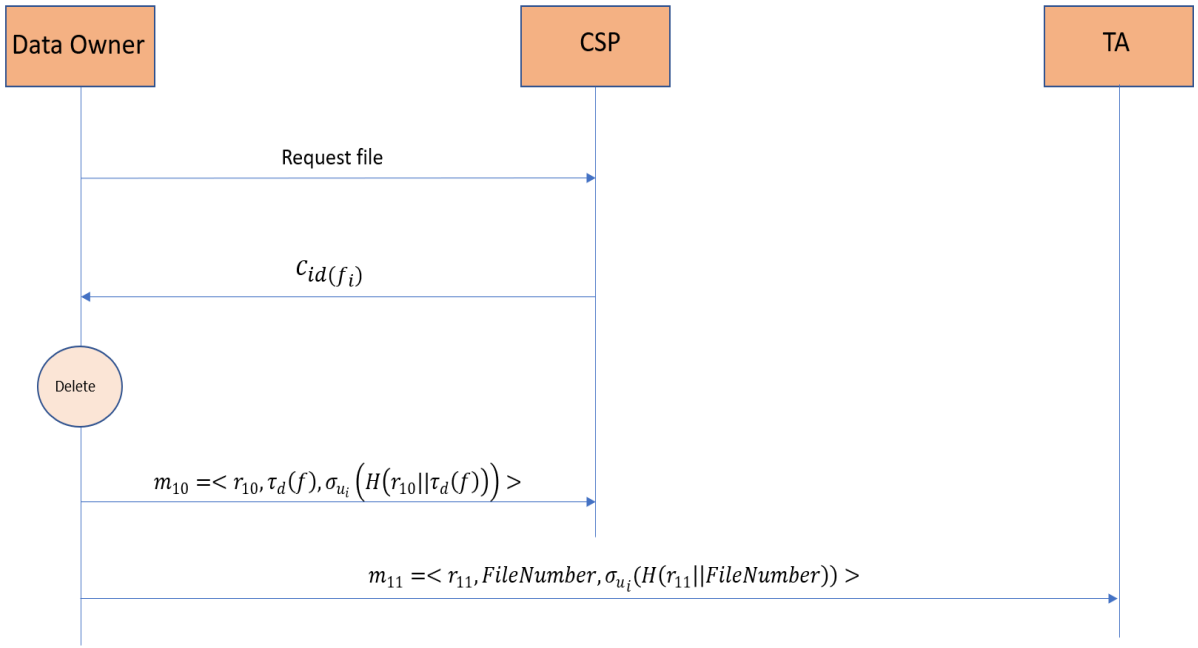

Figure 16: File Deletion

6.2.4 File Modification

Finally, as already stated the data owner can also modify one or more of his/her files. As already described, to do so the data owner first needs to run the delete protocol for the file he/she wants to modify. This will result to the deletion of both the file and every *Dict* entry associated with the file. As a next step, the data owner will modify the file as he/she wishes, and will initiate the AddFile protocol to add the modified file in his/her collection as a new file. This way the CSP does not learn that this a new file that is a modified version of an old file. As a result, the CSP cannot correlate between these files.

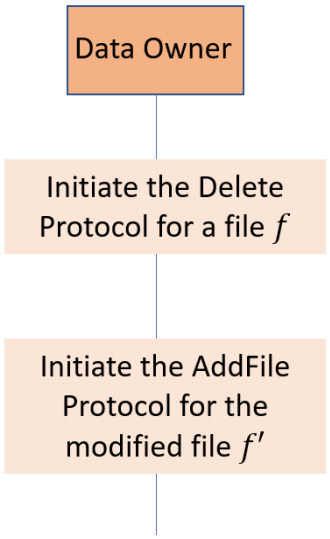

Figure 17: File Modification

7 Security Analysis

In this section we prove the security of our construction against the threat model defined in Section 2.2. We will construct a simulator that will simulate the ASCLEPIOS SSE scheme as well as the communications between the TEE's.

Theorem: *Let SKE be a CPA-secure symmetric key encryption scheme, G an invertible pseudorandom function and h a hash function. Then our construction is secure according to the definition of the security game in Section 2.*

We will construct a simulator $\mathcal{S}$ that will simulate all the ASCLEPIOS.SSE algorithms but range search in a way that no probabilistic, polynomial time (PPT) adversary will be able to distinguish between the real algorithms and the simulator. To this end, we will make use of a Hybrid Argument.

7.1 Hybrid Argument

Before we continue with the proof of this theorem, we briefly present a general model of the Hybrid Argument. This section is purely theoretical and aims at helping the reader understand the proof of the theorem.

Suppose we have two distributions $\mathcal{O}_0, \mathcal{O}_1$ and we want to prove that they are indistinguishable, i.e. for every PPT distinguisher $\mathcal{D}$, the following must hold:

$$\left|\Pr\left[D^{\mathcal{O}_1} = 1\right] - \Pr\left[D^{\mathcal{O}_0} = 1\right]\right| = negl(.)$$

The Hybrid Argument allows us to take multiple steps, using the triangle inequality:

- First, it allows definition of a polynomial set of hybrids. In other words, let $q(n)$ be a polynomial function of the security parameter, and $\mathcal{H}_i$ are hybrid oracles (or input distributions) for all $i \in \{0, 1, 2, \dots, q(n)\}$, where $\mathcal{H}_0 = \mathcal{O}_0$ and $\mathcal{H}_{q(n)} = \mathcal{O}_1$. We choose hybrids $\mathcal{H}_i$, for $i \in \{1, 2, \dots, q(n) - 1\}$ to be indistinguishable, intermediate steps between $\mathcal{O}_0$ and $\mathcal{O}_1$.
- Second, it states that, according to the triangle inequality, as illustrated in Fig. 18, the following is true:

$$\left|\Pr[D^{\mathcal{O}_1} = 1] - \Pr[D^{\mathcal{O}_0} = 1]\right| \leq \sum_{i=1}^{q(n)} |\Pr[D^{\mathcal{H}_i}] - \Pr[D^{\mathcal{H}_{i-1}}]|$$

  Therefore, it suffices to show that every $\mathcal{H}_{i-1}$ and $\mathcal{H}_i$ are indistinguishbale.

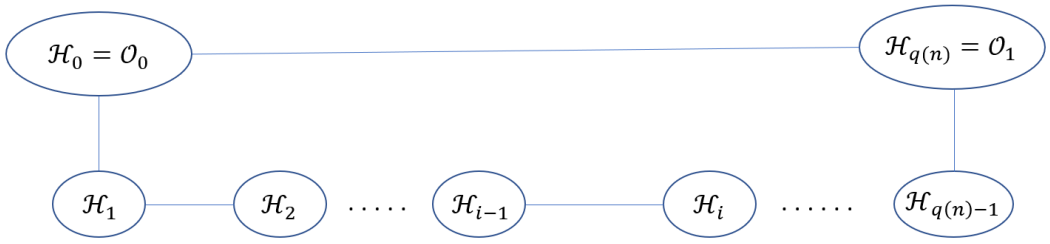

Figure 18: The triangle inequality applied to the general hybrid argument

  For each $i \in \{1, 2, \dots, q(n)\}$, we prove, using reduction or a probabilistic argument, that $\mathcal{H}_{i-1}$ and $\mathcal{H}_i$ are indistignuishable, i.e. , that for every PPT distinguisher $\mathcal{D}$:

$$\left|\Pr\left[D^{\mathcal{H}_i}\right] - \Pr\left[D^{\mathcal{H}_{i-1}}\right]\right| = negl(.)$$

  The same argument can be used for several steps.

- Finally, because of the previous two steps, we know that:

$$\begin{aligned}\left|\Pr[D^{\mathcal{O}_1}=1]-\Pr[D^{\mathcal{O}_0}=1]\right| &\leq \sum_{i=1}^{q(n)}\left|\Pr[D^{\mathcal{H}_i}]-\Pr[D^{\mathcal{H}_{i-1}}]\right| \\ &= \sum_{1}^{q(n)} negl(.) \\ &= q(n)\times negl(.) \\ &= negl(.) \\ &\Longrightarrow \mathcal{O}_0 \approx \mathcal{O}_1\end{aligned}$$

And this completes the proof that $\mathcal{O}_0$ and $\mathcal{O}_1$ are indistinguishable.

7.2 Proof of the Theorem

We are now ready to prove the indistinguishability of the Real and Ideal games as defined in Section 2.2. Our goal is to prove the existence of a simulator $\mathcal{S}$ such that, for all PPT adversaries $\mathcal{A}$:

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

In this proof, the adversary plays the role of the distinguisher.

$\mathcal{H}_0$: The real experiment runs.

$\mathcal{H}_1$: Like $\mathcal{H}_0$ but instead of ASCLEPIOS.InGen $\mathcal{S}$ is given $\mathcal{L}_{InGen}$ and proceeds as follows:

1. $k \leftarrow SKE.Gen(1^{\lambda})$
2. **for** $i=1$ **to** $i=N$ **do**
   a. Simulate (a_i, v_i) pairs
   b. Store all(a_i, v_i) pairs in a dictionary Dict
3. **for all** $f_i \in F$ **do**
   a. $c_i \leftarrow SKE.Enc(k, 0^{|f(i)|})$
4. Create a dictionary KeyStore to store the last K_w for each keyword
5. Create a dictionary Oracle to reply to the random oracle queries

Lemma 1: $\mathcal{H}_0 \approx \mathcal{H}_1$

Proof: *$\mathcal{A}$ cannot distinguish between the two Hybrids since the simulated dictionary has exactly the same size as the real one. Moreover, the CSP security of the symmetric encryption scheme, ensures us that $\mathcal{A}$ cannot distinguish between the encryption of the actual files and that of a string of zeros. Hence, $\mathcal{H}_0 \approx \mathcal{H}_1$.*

$\mathcal{H}_2$: Like $\mathcal{H}_1$ but instead of ASCLEPIOS.AddFile, $\mathcal{S}$ is given $\mathcal{L}_{Add}$ and proceeds as follows:

1. $L_a = \{\}$
2. **for** $i=1$ **to** $i=\#w_i$ **do**
   a. Simulate (a_i, v_i) pairs
   b. $c_{id(f_i)} \leftarrow SKE.Enc(k, 0^{|id(f)|})$
   c. Add $(c_{id(f)}, \{a_i, v_i\})$ in Dict
   d. $L_a = L_a \cup \{a_i, v_i\}$
3. $c \leftarrow SKE.Enc(k, 0^{|f|})$
4. $\tau_\alpha(f) = (c_{id(f)}, c, L_a)$

Lemma: $\mathcal{H}_1 \approx \mathcal{H}_2$

Proof: *The simulated add token, allows $\mathcal{S}$ to keep its dictionary up to date with files provided by $\mathcal{A}$, after the execution of InGen. The token provided by the simulator has exactly the same format and size as the real add token. Moreover, we show that the add token can be simulated*

by only knowing $\mathcal{L}_{Add}$, and as a result, we prove that our scheme preserves the property of forward privacy. Finally, once again the CPA security of the symmetric encryption scheme, ensures us that $\mathcal{A}$ cannot distinguish between the encryption of actual files and that of a string of zeros. Hence, $\mathcal{H}_1 \approx \mathcal{H}_2$.

$\mathcal{H}_3$: Like $\mathcal{H}_2$ but instead of ASCLEPIOS.Search, $\mathcal{S}$ is given the access and search patterns and proceeds as follows:

1) $d = |F_w|$ (Number of files containing w)
2) **if** $KeyStore[w] = null$ **then**
   a) $KeyStore[w] \leftarrow \{0,1\}^\lambda$
   b) $K_w = KeyStore[w]$
3) **for** $i = 1$ **to** $i = d$ **do**
   a) **if** $Oracle[K_w][0][i]$ is null **then**
      i) **if** f_i is added after InGen **then**
         (1) Pick a $(c_{id(f_i)}, \{a_i, v_i\})$ pair
      ii) **Else**
         (1) Pick an unused $\{a_i, v_i\}$ pair at random
      iii) $Oracle[K_w][0][i] = a_i$
      iv) $Oracle[K_w][1][i] = v_i || c_{id(f_i)}$
   b) **else**
      i) $a_i = \overline{Oracle[K_w][0][i]}$
      ii) $v_i = \overline{Oracle[K_w][1][i]}(Oracle[K_w][1][i] - |c_{id(f_i)}|)$
   c) Remove a_i from the dictionary
4) $UpdatedVal = \{\}$
5) $K'_w \leftarrow \{0,1\}^\lambda$
6) $KeyStore[w] = K'_w$
7) **for** $i = 1$ **to** $i = d$ **do**
   a) Generate a new a_i and match it with the v_i from step (c)
   b) Add $(c_{id(f_i)}, \{a_i, v_i\}$ to the dictionary
   c) $UpdatedVal = UpdatedVal \cup \{c_{id(f_i)}, a_i\}$
   d) $Oracle[K'_w][0][i] = a_i$
   e) $Oravle[K'_w][1][i] = v_i || c_{id(f_i)}$
8) $\tau_s(w) = (K_w, d, UpdatedVal)$

Lemma 2: $\mathcal{H}_3 \approx \mathcal{H}_2$

Proof: *The KeyStore[w] is used to keep track of the last key K_w used for each keyword w. The $Oracle[K_w][j][i]$ dictionary is used to reply to $\mathcal{A}$'s queries. Foe example, $Oracle[K_w][0][i]$ represents the address of a Dict entry assigned to the i-th file in the collection F. Similarly, $Oracle[k_w][w][i]$ represents the masked value needed to recover $c_{id(f)}$. It is clear, that the simulated search token has exactly the same size and format as the real one and as a result, no PPT adversary $\mathcal{A}$ can distinguish between them. Moreover, $\mathcal{A}$ cannot tamper with the messages that are constructed by the TEEs. The reason for this, is that these messages are signed with a secret key that is only known to the enclaves. As a result, tampering with the reports, implies producing a valid TEE's signature, which can only happen with negligible probability. Hence,* Lemma: $\mathcal{H}_3 \approx \mathcal{H}_2$.

$\mathcal{H}_4$: Like $\mathcal{H}_3$ but instead of ASCLEPIOS.Delete, $\mathcal{S}$ is given $\mathcal{L}_{Delete}$ and proceeds as follows:

1) $L_d = \{\}$
2) $L_s = \{\}$

3) $c_{id(f)} \leftarrow SKE.Enc(k, 0^{|id(f)|})$
4) $L_{\mathcal{S}} = L_{\mathcal{S}} \cup c_{id(f)}$
5) **for** $i = 1$ **to** $i = \ell$ **do** (Number of keywords that exist in more files)
   a) Generate a new $\{a_i', v_i'\}$ pair
   b) Select an unused $\{a_i, v_i\}$ pair
   c) $L_d = L_d \cup \{\{a_i', v_i'\}, \{a_i, v_i\}\}$
   d) Replace $\{a_i, v_i\}$ with $\{a_i', v_i'\}$
6) **for** $i = \ell + 1$ **to** $i = \#w$ (Number of keywords that only exist in the deleted file)
   a) Generate a new v_i'
   b) Pick an unused $\{a_i, v_i\}$ pair and delete it
   c) $L_d = L_d \cup \{v_i', 0\}$
7) Output $\tau_d(f) = (c, L_d)$

Lemma 3: $\mathcal{H}_4 \approx \mathcal{H}_3$

Proof: *The list $L_{\mathcal{S}}$ is used by the simulator to reply correctly to future queries. The simulated delete token is indistinguishable from the real one as they share the same format and size. Moreover, $\mathcal{A}$ could once again try to tamper the TEE's messages. However, as already stated before, tampering the messages implies that $\mathcal{A}$ could produce a valid TEE's signature without owning the corresponding key. However, this can only happen with negligible probability. Hence, $\mathcal{H}_4 \approx \mathcal{H}_3$*

With this last hybrid our proof is complete. We constructed a simulator $\mathcal{S}$ that simulates all the real protocols in a way that no PPT adversary $\mathcal{A}$ can distinguish between the real and the ideal experiments. In other words we showed that $\mathcal{H}_0 \approx \mathcal{H}_4$.

Range Search: The reason that we did not include this algorithm in the security analysis, is that after the initial communication with the TA, this algorithm is reduced to the original Search algorithm for which we already proved its security. The initial exchange of messages between the user and the TA is not vulnerable to any attacks either. The messages are constructed in such a way that their freshness, integrity and authenticity are maintained.

8 Experimental Results

Our experiments mainly focused at analysing the performance of the our scheme. To do so, we implemented it in Python 2.7 using the PyCrypto library ((Ed.)). To test the overall performance of the underlying dynamic SSE scheme, we used files of different size and structure. More precisely, we selected random data from the Gutenberg dataset (Gutenberg Project). Our experiments focused on three main aspects:

(1) Indexing,
(2) Searching for a specific keyword, and
(3) Deleting a specific 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 over- all performance of the scheme it is considered as more durable and closer 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 of our construction under realistic conditions, we decided to use different machines. To this end, we ran our experiments in the following three different machines:

- Intel Core i7-8700 at 3.20GHz (6 cores), 32GB of RAM running Ubuntu 18.04 Desktop operating system.
- Microsoft Surface Book laptop with a 4.2GHz Intel Core i7 processor (4 cores) and 16GB RAM running Windows 10 64-bit.
- Microsoft Surface Book *tablet mode* with a 1.9GHz Intel Core i7 processor 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 lot of computational power and resources, the other two machines are considered as commodity machines that a typical user can own (especially the tablet). The reason for measuring the performance of the scheme 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.

8.1 Dataset

For the needs of our experiments, we created a dataset containing five different sub-datasets with random text files (i.e. e-books in .txt format) from the Gutenberg dataset. The selected datasets ranged from text files with a total size of 184MB to a set of text files with a total size of 1.7GB. It is important to mention that using text files (i.e. pure text in comparison to other formats such as PDF, word, etc.) resulted in a very large number of extracted keywords – thus creating a dictionary containing more than 12 million distinct keywords (without counting stop words). Furthermore, in our implementation we also incorporated a stop words (such as "the", "a", "an", "in") removal process. This is a common technique used by search engines where they are programmed to ignore commonly used words both when indexing entries for searching and when retrieving them as the result of a search query. This makes both the searching and indexing more efficient while also reducing the total size of the dictionary. Table 4 shows the five different sub-datasets that we used for our experiments as well as the total number of unique keywords that were extracted from each of the incorporated collections of files.

Is our Dataset Realistic (i.e. big enough)? As can be seen from Table 4, our dataset is divided into five different sub-datasets ranging from 1,370,023 to 12,124,904 distinct keywords mainly collected from English books. During a project (Jean-Baptiste, Yuan Kui and Aviva P) researchers from Harvard University and Google in 2010 were looking at words in digitised books. They estimated a total of 1,022,000 words and estimated that this number

would grow by several thousand each year. Furthermore, it is important to mention that this number also includes different forms of the same word. It also includes many archaic words not used in modern English. In addition to that, in the second edition of the Oxford English dictionary, there are approximately 600,000 word forms defined. Again, this includes many words not in common use any more. As a result, even our smallest sub-dataset is almost double in the size of the Oxford English dictionary as well as slightly larger than the total number of words found in digitized books in 2010. As a result our dataset that contains more 12,000,000 distinct keywords can be considered as realistic and can give us valuable insights about how our scheme would behave in real-life scenarios where users would use a cloud service based on our scheme to store their personal files online in encrypted forms.

Number of TXT Files	Dataset Size	Unique Keywords
425	184MB	1,370,023
815	357MB	1,999,520
1,694	670MB	2,668,552
1,883	1GB	7,453,612
2,808	1.7GB	12,124,904

Table 5: Size of Datasets and Unique Keywords

8.2 Indexing and Encryption

The indexing phase is considered as the setup phase of the SSE scheme. During this phase the following three steps take place:

(1) reading plaintext files and generating the dictionary,
(2) encrypting the files, and
(3) building the encrypted indexes.

In our experiments, we measured the total setup time for each one of the sub-datasets shown in Table 4. Each process was run ten times on each machine and the average time for the completion of the entire process on each machine was measured. Figure 19 illustrates the time needed for indexing and encrypting text files ranging from 184MB to 1.7GB that resulted to a set of more than 12 million unique keywords. As can be seen from Figure 19 the desktop machine needed the less time to complete the setup phase while the tablet took significantly more time not only compared to the desktop but also in comparison to the time needed by the laptop. However, in all three cases it is evident that the scheme can be considered as practical and can even run in typical users' devices. 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 6 summarizes our measurements from this phase of the experiments. As can be since, to index and encrypt text files that contained 1,370,023 distinct keywords the average processing time was 8.49m, 22.48min and 68.75m for the desktop, laptop and tablet accordingly while for a set of files that resulted in 12,124,904 distinct keywords the average processing time was 68.44m, 203.28m and 545.28. 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 offer forward privacy (Dowsley, Michalas and Nagel) as well as on the fact that we ran our experiments on commodity machines and not only on a powerful server.

Unique Keywords	(w, id) Pairs
1,370,023	5,387,216
1,999,520	10,036,252
2,668,552	19,258,625
7,453,612	28,781,567
12,124,904	39,747,904

Table 6: Keywords and Filenames pairs

Testbed Dataset	Tablet	Laptop	Desktop
184MB	68.75m	22.48m	8.49m
357MB	109.36m	40.00m	13.51m
670GB	195.09m	86.43m	29.51m
1GB	367.75m	141.60m	48.99m
1.7GB	545.28m	203.28m	68.44m

Table 6: Setup time (in minutes) of ASCLEPIOS.SSE

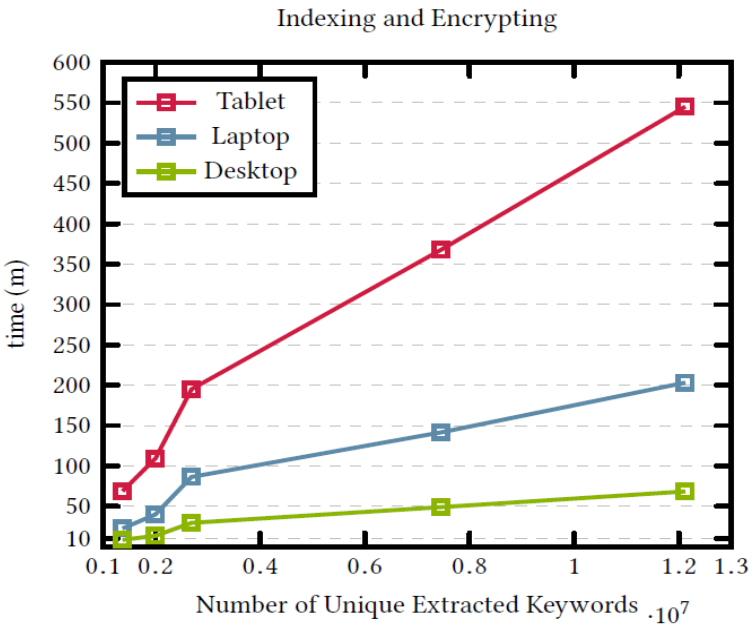

Figure 19: Indexing and Encrypting Files

8.3 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, in our experiments we measured the time to generate the search token on the laptop and the tablet (i.e. typical users' machines) 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 9μs while on the Surface tablet the time for token generation slightly increased to 13μs. Regarding the actual search that is taking place on the CSP it needs to be noted that the actual process is just a series of SELECT and UPDATE queries to the

database. More precisely, 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.

8.4 Delete

To measure the performance of the delete process, we randomly selected 100 different files, performed the delete operation and measured the average processing time for the delete process to be completed. We performed the delete queries in our largest dataset containing 12,124,904 distinct keywords and 39,747,904 addresses. The average time to delete a single file and update all the relevant addresses in the database was 1.19min. Even though this time might be considered as high for just deleting a single file it is important to mention that this process will be running on a CSP with a large pool of resources and computational power (e.g. Amazon EC2). Hence, this time is expected to drop significantly on such a computer where more cores will be also utilized. Furthermore, it is important to mention that to properly test the performance of our delete function we need to conduct further experiments where we will also consider the number of keywords contained in the file to be deleted as well as the number of other files that each keyword can be also found at. This is important because it is expected that these factors will heavily affect the performance of the delete function. We plan to conduct further and more detailed experiments on the delete function in our future works.

9 Symmetric Searchable Encryption and Demonstrators Integration

As described in Deliverable 1.2 (section 6.7, 7.14, 8.21), the Symmetric Searchable Encryption (SSE) schemes can be utilized by the ASCLEPIOS demonstrators to achieve data sharing feature. However, the integration between these demonstrators and the SSE scheme is not straightforward due to differences in their supported data. The SSE supports data as text files, while the demonstrators support structured data (i.e. relational database) and/ or blob data (i.e. images). More specifically, the input/ output of the SSE scheme are text files, and the operations (add, search, delete, modify) are file-based. In the meanwhile, the demonstrators do not produce any text files. Instead, they work with relational databases and/ or image files. This difference raises a need to coordinate the data standard for communication and integration between the demonstrators and the SSE scheme.

The solution is to use JavaScript Object Notation (JSON). Considering the demonstrators as web applications, JSON can be used by the applications to fetch inputs to the SSE scheme. Additionally, the SSE scheme will be changed to adapt this new kind of data input. Based on such changes, the following scenarios describe how the demonstrators will supply the SSE scheme with data, and how they will search and retrieve data using the SSE scheme. In both scenarios, we have three main components:

- **Demonstrator's Web Application:** This application is developed by the demonstrator, and integrated with ASCLEPIOS. Users will interact with this application;
- **SSE Client:** A JavaScript program running on the client side, and implementing the required SSE functionality based on a JSON object that is given as input;
- **SSE Server:** Server side part of the SSE scheme which is responsible for storing the received encrypted data in a MySQL database on ASCLEPIOS cloud storage (server side).

a. Scenario 1: The demonstrator supplies data to the SSE scheme

This scenario happens when a user of the demonstrator wants to add data, for i.e. medical data, to the server.

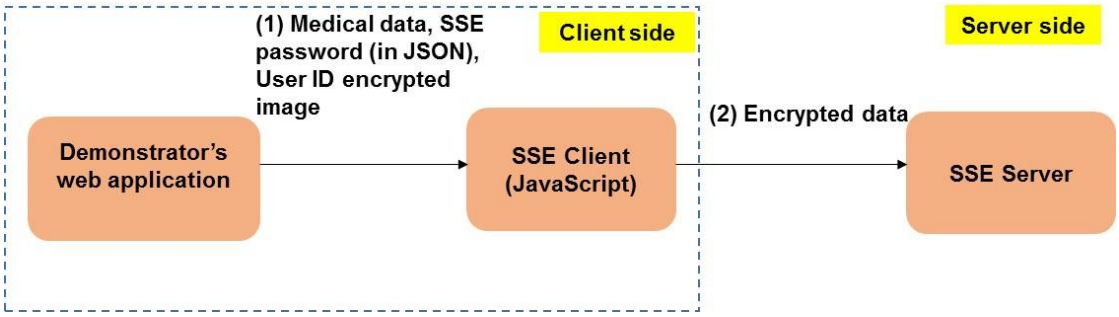

Figure 20: Demonstrator forwards data to the SSE scheme

(1) The demonstrator's web application supplies data to the SSE client

The demonstrator collects data from a user, which involves medical data and/ or encrypted medical images. In addition to such data, the user needs to provide a password, which will be used by the SSE scheme to generate an AES symmetric key for encrypting data prior to sending to the server. Apart from that, a User ID is needed for the SSE Server to categorize the datasets among different users. All information will be converted into a JSON object that will be given as input to the SSE client as described below.

- **Medical data (in text)**: The demonstrator's web application receives medical data from a user, and converts it into JSON that will be given as input to the SSE client.
- **Password:** The password is integrated as part of the JSON object. Upon reception, the SSE client will use this password to generate an AES symmetric secret key with which the data in the JSON data will be encrypted.

- **Medical Images:** In the case where a user wishes to upload medical images in addition to medical text data:
  - The demonstrator's application will extract all metadata from the underlying image file, and will add them at the corresponding JSON that already contained the medical text data.
  - The underlying image, will be encrypted locally prior to sending to the SSE server along with the encrypted JSON data.

Example of JSON data:

```
{ ``patient'':
   { ``id'': 155415463 ``name'': John Snow,},
   ``image_metadata'':
   { ``name'': 'x-ray'}
 ``user.password: 'pass'
 ``user.id'': 12345
}
```

(2) The SSE client sends the encrypted data

The SSE Client uses the user's password to generate an AES symmetric secret key, which is then used to encrypt each pair of name/value in the underlying JSON object. Then, the SSE Client sends the encrypted JSON data (along with any received encrypted medical image) to the SSE Server. The encrypted data will be maintained on ASCLEPIOS cloud storage in an encrypted form (server side).

b. Scenario 2: The demonstrator uses the search function of the SSE scheme

This scenario happens when a user of the demonstrator wishes to search for data (i.e. medical data) by some criterion (i.e. by name).

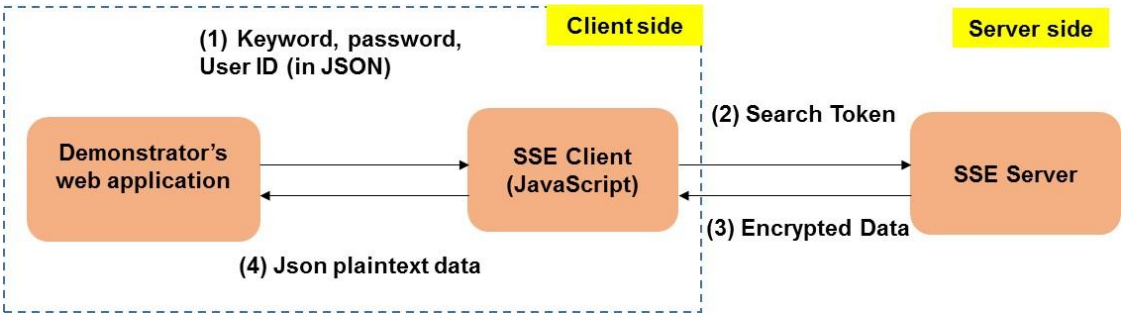

Figure 21: Demonstrator runs search operation using the SSE scheme

(1) The demonstrator's web application supplies the SSE client with keyword, password, User ID (in JSON)

When a user wishes to search for medical data by keyword (for e.g. the user searches by patient's name with the value ``John Snow''), she provides the demonstrator's web application with the keyword (which is used for searching), a password (which is the same password provided when data is created in Scenario 1), and User Id. All information will be converted into JSON data object to supply to the SSE client. More details are described as follows.

- **Keyword and User ID**: The demonstrator's web application receives a keyword from a user. The demonstrator will convert it into JSON which is then given as input, along with the user's specific info (e.g. user id, which can be used for authentication purpose from other components), to SSE client.
- **Password:** The password is integrated as part of the JSON object. Upon reception, the SSE client will use this password to generate an AES symmetric secret key with which the search token is created, and the returned data will be decrypted.

Example:

```
{
    ``user.password'': pass,
    ``user.id'': 12345,
     ``keyword'': { patient.name: John Snow }
}
```

(2) The SSE client sends search token to the SSE server

Upon reception of the JSON object, the SSE client generates a search token that is sent to the CSP. The search token is generated based on the provided keyword and password.

(3) The SSE server returns the encrypted data

The SSE server executes the user's search query by using the received search token. The result, which is encrypted data, will be sent back to the SSE client.

(4) The SSE client sends the results to the demonstrator's web application

Upon reception, the SSE client will decrypt the result and will return the data to the demonstrator's web application in a JSON format. After that, the demonstrator will show the received data to the user.

10 Conclusion

In this deliverable we described the core functionality of the ASCLEPIOS cryptographic layer. The contribution of this document is fourfold.

- First, we presented various SSE schemes pointing out their advantages and disadvantages.
- Second, taking into consideration the requirements of ASCLEPIOS, we designed an SSE scheme that squarely fits the needs of the project. To prove the security of our construction against malicious adversaries, we designed a detailed protocol that defines how different entities communicate with each other. The system model of the ASCLEPIOS SSE scheme is in accordance with the reference architecture as specified in D1.2: ASCLEPIOS Reference Architecture, Security and eHealth Use Cases, and Acceptance Criteria.
- Third, we conducted extensive experiments to test the performance of our construction. In the conducted experiments we focused on measuring the performance of our scheme under a wide range of devices with different characteristics. The results showed that the designed and developed SSE scheme can run smoothly even in devices with low computational resources.
- Fourth, we explained how the ASCLEPIOS SSE scheme can be modified to fit the needs of the ASCLEPIOS demonstrators.

11 References

A Survey on Design and Implementation of Protected Searchable Data in the Cloud*Computer Science Review*201717-30

Acess pattern disclosureon searchable encryption: Ramification, attack and mitigation.*Netwtork and Distributed System Security Symposium*2012

All your queries bleong to us: The power of file-injection attacks on searchable encryption.*25th USENIX Conference on Security Symposium*Berkeley2016

Ciphertext-policy attribute-based encryption. *IEEE Computer Society*Washington, DC2007321-334

Constrained Keys for Invertible Pseudorandom Functions*Theory of Cryptography (TCC)*Baltimore, USASpringer2017237-263

Dynamic searchable encryption*IEEE Symposium on Security and Privacy*IEEE2014639-654

Dynamic Searchbale Encryption2012

Gutenberg Project*www.gutenberg.org*

*https://pypi.org/project/pycrypto/*2013

Leakage-abuse attacks against searchbale encryption *ACM CCS'15*2015

On the security of public key protocols*Information Theory, IEEE Transactions* 1983

SpringerParallel and Dynamic Searchable Symmetric Encryption*Financial Cryptography and Data Security.*OkinawaSpringer2013258-274

Practical Dynamic Searchable Encryption with Small Leakage*NDSS Symposium*San Diego, California2014

Practical Techniques for searches on encrypted data*IEEE Symposium on Security and Privacy*Washington DCIEEE200044-

Providing User Security Guarantees in Public Infrastructure Clouds*IEEE Transactions on Cloud Computing*2017405-419

Quantitative analysis of culture using millions of digitized books*Science*2011176-182

Secure Indexes2003

Tampere Univeristy (TUNI), Norwegian Centre for eHealth Research (NSE)ASCLEPIOS Reference Architecture, Security and E-health Use Cases, and Acceptance CriteriaProject Deliverable2019

The Fallacy of Composition of Oblivious RAM and Searchbale EncryptionIACR Cryptology2015

Deliverable 1.2 - ASCLEPIOS Reference Architecture, Security and E-health Use Cases, and Acceptance Criteria