A Survey on Design and Implementation of Protected Searchable Data in the Cloud

Rafael Dowsley

Cryptography and Security Research Group, Department of Computer Science, Aarhus University, Aarhus, Denmark

Email: rafael@cs.au.dk

Antonis Michalas*

Cyber Security Group, Department of Computer Science, University of Westminster, London, UK

Email: a.michalas@westminster.ac.uk

Matthias Nagel*

Institute of Theoretical Informatics, Karlsruhe Institute of Technology, Karlsruhe, Germany

Email: matthias.nagel@kit.edu

Nicolae Paladi*

Security Lab, SICS Swedish ICT, Kista, Sweden

Email: nicolae@sics.se

Abstract

While cloud computing has exploded in popularity in recent years thanks to the potential efficiency and cost savings of outsourcing the storage and management of data and applications, a number of vulnerabilities that led to multiple attacks have deterred many potential users.

As a result, experts in the field argued that new mechanisms are needed in order to create trusted and secure cloud services. Such mechanisms would eradicate the suspicion of users towards cloud computing by providing the necessary security guarantees. Searchable Encryption is among the most promising solutions – one that has the potential to help offer truly secure and privacy-preserving cloud services. We start this paper by surveying the most important searchable encryption schemes and their relevance to cloud computing. In light of this analysis we demonstrate the inefficiencies of the existing schemes and expand our analysis by discussing certain confidentiality and privacy issues. Further, we examine how to integrate such a scheme with a popular cloud platform. Finally, we have chosen – based on the findings of our analysis – an

*Corresponding Author.

existing scheme and implemented it to review its practical maturity for deployment in real systems. The survey of the field, together with the analysis and with the extensive experimental results provides a comprehensive review of the theoretical and practical aspects of searchable encryption.

Keywords: Searchable Encryption, Security, Cloud Computing, Cloud Storage

1. Introduction

In recent years we have witnessed an astonishing increase in the offer of cloud computing solutions. Leveraging savings through large scale optimizations and reduction of wasted resources (inactive computer time, unused hardware space, etc), this business model offers clear economic advantages. Along with the continuous increase in the amount of data, this provides a strong incentive for both organizations and private users to opt for storing their data with cloud service providers (CSPs). However, this trend raises a security issue, since many clients want to keep their files confidential. The solution may be to encrypt the files before sending them to the CSP, but there are two seemingly contradictory goals that an encryption scheme should achieve in order to be useful in this scenario. On one hand, the encryption must satisfy a strong notion of security in order to keep the data hidden from the CSP. On the other hand, the scheme should allow the clients to continue performing their operations efficiently, i.e., with time and computational costs comparable to the ones for locally stored files. Searching often a quintessential requirement for many clients. It is therefore essential to develop and employ encryption schemes that allow for efficient searching of the data stored in the cloud; if the clients have to download the entire data set and perform the search locally, then the scheme is completely impractical.

Searchable Encryption (SE) is an enhanced encryption technique that allows encryption while enabling search for keywords in the encrypted data (as it would be possible in the plaintext). Its quintessential application is cloud storage. Using searchable encryption should enable a CSP – with the help of a search token sent by the client – to locally perform some operations and then send the relevant data to the client. The relevant data should on one hand contain the matching documents (i.e., the documents that contain the searched keyword), while on the other hand its size should be not much larger than that of the matching documents (i.e., the server cannot simply transfer a large part of the database to the client on every query). Of course the CSP should not learn the keyword that is being searched or the search query, otherwise he is learning partial information about the documents.

Searchable encryption clearly displays trade-offs between efficiency, functionality and security. From an efficiency point of view, it is desirable to reduce as much as possible the number of operations performed by the server during a search. It is also highly important to make these operations parallelizable and increase their locality (in order to improve I/O performance), in order to

improve the search time. From the functionality point of view, one important parameter is the query expressiveness. An SE scheme should support as powerful queries as possible, thus increasing the usefulness of the scheme to the clients. Other important parameters are whether a single or multiple clients should be able to write data to the cloud and whether a single or multiple clients should be able to read the data. Additionally, schemes for practical applications should be dynamic, i.e., they should allow database updates without additional leakage. From a security point of view, it is essential to reduce the leakage caused by all operations as much as possible.

Depending on the requirements of the desired scheme, it possible to use either public-key cryptography or symmetric-key cryptography. However, often searchable public-key encryption schemes with good security guarantees do not scale well because they have search time which is linear in the number of documents.

Symmetric searchable encryption was introduced by Song et al. [1], who presented a scheme that allowed linear search time (in the number of documents) by the server. Unfortunately their scheme does not achieve a strong notion of security: it has no security guarantees related to the leakage that can be caused by the use of the search tokens that are given to the server in order to allow the search to be performed on the server side. Goh [2] introduced the approach of using secure indexes in order to achieve linear search time with stronger security guarantees. Unfortunately the search time of this approach is inherently linear in the number of files. Curtmola et al. [3] presented the first secure scheme with sub-linear search time using an inverted index approach (uses the keywords as index) and also introduced a strong security model for searchable encryption which became the standard security notion for searchable encryption in the last several years. The inverted index approach is quite efficient and is in fact optimal for the number of operations that the server has to perform during a search, which led to it being used in many subsequent works (e.g., [4, 5, 6]). One limitation of this method is that it is inherently sequential, preventing it from taking advantage of parallelism to improve performance. Another issue is that it is not well-suited for dynamic databases, which is the case of most applications. Recent works made progress in the direction of dynamic [5, 7, 8, 9, 10] and parallel [7, 11, 8] schemes.

Symmetric searchable encryption perfectly fits the scenario of a single user writing to/reading from the database. However, there is a generic construction that combines a single writer/reader scheme with broadcast encryption in order to obtain a scheme that supports multiple readers [3]. One additional issue in this case is revocation: a revoked user should not be able to perform searches after the revocation has occurred.

In terms of query expressiveness, most symmetric searchable encryption schemes focus on single equality queries. Some recent works [11, 12] demonstrated that it is possible to extend data structures for single keyword symmetric searchable encryption in order to deal with more complex queries, such as conjunctive queries for keyword combinations and general Boolean queries.

Public-key searchable encryption was introduced by Boneh et al. [13]. It

allows multiple clients to encrypt data into the database, which can be decrypted by the data owner that has the secret-key. Other solutions allow conjunctive, subset and range queries [14]. The efficiency of these schemes is limited by the cost of public-key operations. Another problem of the proposed schemes with strong security assurances is their linear search time, which limits scalability.

So far, to the best of our knowledge, no public cloud offering is known to support storage protection with searchable encryption support. To explore the feasibility of searchable encryption for cloud storage, we have chosen to implement it using a popular open-source cloud platform. This project is supported by more than 200 companies around the world, including key industry players.

1.1. Our Contribution

The contribution of this paper is twofold. First, we present a theoretical analysis of the existing *Symmetric Searchable Encryption* schemes. In light of this analysis, we demonstrate the inefficiencies of the existing schemes while we expand our analysis by discussing certain privacy issues. Apart from that, we focus on integrating such a scheme with Openstack – an open-source popular cloud platform. Finally, based on the findings of our analysis, we have chosen to implement and test one of the existing schemes, in order to understand whether it is practical enough for deployment in real systems. As a result, our theoretical analysis is coupled with extensive experimental results. We hope that the findings of this work will give valuable insights to protocol designers and will spawn further research in the area.

1.2. Organization

In Section 2 we discuss in more detail why searchable encryption fits perfectly the cloud. In Section 3 we present in more detail the concept of searchable encryption and its security model. In Section 4 we survey the current known methods for building symmetric searchable encryption schemes. Then in Section 5 we highlight some considerations regarding the privacy of such schemes while in Section 6 we elaborate on their efficiency. Section 7 presents the architecture of OpenStack. Then in Section 8 we give our recommendation of the scheme that seems more appropriate for the integration with OpenStack-based solutions. Section 9 reports on the performance of the implemented scheme. Finally, in Section 10 we conclude the paper.

2. Why Searchable Encryption Squarely Fits the Cloud

While cloud computing has exploded in popularity in recent years thanks to the potential efficiency and cost savings of outsourcing the management of data and applications, a number of vulnerabilities that led to various attacks have left many potential users worried [15]. As a result, experts in the field argued that new technologies are needed in order to create trusted cloud services [16, 17] – services that will eventually eradicate the suspicion of users for cloud computing

by providing the necessary security guarantees. More precisely, despite significant improvements regarding availability and scalability of cloud services, it has been observed that the greatest concern of users that hinders the adoption of cloud computing is the fear of storing sensitive data online. Without proper security mechanisms to protect users' data from unauthorized access, sensitive information is at risk of being leaked to interested third parties.

The most common solution to this problem is to make sure that users' data is always encrypted when it is placed on the provider's storage hosts and while it is in use by the cloud service. However, such an approach does not always provide full security since all of the trust is placed on the party that is encrypting the data and storing the encryption key. More precisely, once the cloud provider is responsible for encrypting the data it becomes aware of the encryption/decryption key, casting doubts on the security of users' data in case of a malicious provider or a malicious administrator.

One of the most promising concepts first introduced by Song et al. [1] is the so called searchable encryption where users can search directly on encrypted data without having to decrypt them first. In general, searchable encryption schemes aim to provide confidentiality and integrity, while retaining main benefits of cloud storage – availability, reliability, data sharing, and ensuring requirements through cryptographic guarantees rather than administrative controls. However, until to this day there is a lack of practical applications that rely on searchable encryption schemes. To the best of our knowledge, there is no public cloud provider that supports such functionality and the main reason for that is the fact that in order to provide a reliable and efficient implementation requires additional research.

Furthermore, the latest advancements in the field of searchable encryption have the potential to allow cloud providers to build different kinds of security levels, which will eventually lead to various business models. Therefore, building a concrete searchable encryption scheme for the cloud will give the opportunity to cloud providers to offer a range of security options for the users. More precisely, in an ideal scenario users will be able to configure the level of security based on what kind of searchable encryption they want to use. For example, options such as the blind storage that were proposed in [9], where users can encrypt their data locally before sending them to the cloud and then can search directly over the encrypted data stored in the cloud provider, will provide a set of strong security guarantees to the users since they will be sure that even in the case of a malicious cloud provider or a corrupted administrator the stored data will be secured since the users will be the only ones who have access to the encryption key. In other words, even if the cloud provider tries to expose the privacy of users by looking at the stored data it will not be able to find any valuable information as long as the underlying cryptosystem is secure. As a second example, we can consider a protocol that will be based on proxy re-encryption that was first introduced in [18] and allows a semi-trusted party to search through the data stored in the cloud by using a searchable encryption key. In contrast to the previous example, such a scenario will weaken the adversarial model since the users will have to trust a third party – the proxy server – but

at the same time will offer better efficiency since all the computations will not take place on user's machine but on the proxy. Furthermore, by using searchable encryption cloud providers will be able to offer a plethora of options to the users and will eventually be able to address even the more demanding needs in the sense of data protection.

In addition to that, cloud services that are solely based on searchable encryption schemes are the perfect candidates for providing a realistic and reliable solution for the increasingly urgent problem of physical location of data in cloud storage. In a short time, the aforementioned problem has evolved from the concern of a few regulated businesses to an important consideration for many cloud storage users. One of the characteristics of cloud storage is fluid transfer of data both within and among the data centres of a cloud provider. However, this has weakened the guarantees with respect to control over data replicas, protection of data in transit and physical location of data. Moreover, after the revelations of E. Snowden some months ago and the NSA scandal the significance for finding a reliable solution that will tackle this problem is of paramount importance. Even though, searchable encryption will not provide a direct solution for a trusted geolocation-based mechanism [19] for data placement control, it has the potential to protect users' private data from unauthorized access by providing the indispensable proofs ensuring that unencrypted data will only be available in jurisdictions allowed by policies and defined by the data owner.

Searchable Encryption is not the only technique that allows users to perform operations directly on encrypted data. One of the most well-known schemes is the so called fully homomorphic encryption (FHE) [20] that allows a user to perform a set of operations directly on ciphertexts. In addition to that, if multiparty computation (MPC) [21, 22] schemes are combined with FHE schemes have the potential to provide really secure and privacy-preserving solutions for the cloud. However, while both FHE and MPC can perform some computations on encrypted data they cannot search ciphertexts for specific keywords. Likewise, FHE is currently considered as a computationally heavy technique that prevents any service from using such a scheme. To this end, FHE is currently on an experimentation stage and cannot be used to run real services. Hence, comparing the two techniques would be a rather unfair battle since there is a huge efficiency gap but most importantly they serve different purposes.

3. General Model of Searchable Encryption

Searchable encryption allows a client to encrypt its data in such a way that he can generate search tokens that allows the storage server to search over the encrypted data. The data can be viewed as a collection $\mathbf{f} = (f_1, \ldots, f_n)$ of n files where file f_i is a sequence of words $(w_1, \ldots, w_m)$ from some keyword space $\mathcal{W}$. Additionally, each file f_i has an unique identifier $\mathsf{id}(f_i)$. The data is dynamic, thus file additions or removals are allowed. In addition to the search tokens, the client also generates and sends to the server add/delete tokens when he wants to add/delete files from the encrypted database. We formalize the notion of dynamic symmetric searchable encryption (SSE) scheme using the extensions

to the dynamic setting by Kamara et al. [5] of the definition of Curtmola et al. [3].

Definition 1 (Dynamic Index-based SSE). *A dynamic index-based symmetric searchable encryption scheme is a tuple of nine polynomial algorithms* $\mathsf{SSE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{SearchToken}, \mathsf{AddToken}, \mathsf{DeleteToken}, \mathsf{Search}, \mathsf{Add}, \mathsf{Delete}, \mathsf{Dec})$ *such that:*

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

is used by the storage server upon receiving a delete token in order to update the encrypted index and the ciphertext vector to delete the data corresponding to the deleted file.

- Dec *is a deterministic algorithm that takes as input a secret key K and a ciphertext c and outputs a file f. It is used by the client to decrypt the ciphertexts that obtained from the storage server.*

A dynamic SSE scheme is correct if for all possible security parameters and file collections, and for secret keys, encrypted indexes and ciphertexts created using the respective algorithms and for any sequences of add, delete and search operations handled using the respective algorithms. It holds that the search operation always returns the correct set of indices corresponding to the searched keyword and the returned ciphertexts can be correctly decrypted. A static SSE scheme can be defined by omitting the algorithms AddToken, DeleteToken, Add and Delete from the definition.

On an intuitive level, a good security notion for searchable encryption would be to require that nothing is leaked to the storage server beyond the outcome of the search (also known as *access pattern*), i.e., the identifiers of the documents that contain the queried keyword. Note that the access pattern can only be hidden using expensive techniques as oblivious RAMs [23, 24]. But the practical searchable encryption schemes normally leak more than that: they also leak whether two queries were for the same keyword or not, which is called the *search pattern*. The search pattern is leaked for instance if deterministic search tokens are used, which is the case in the most efficient solutions. Given this, a reasonable definition of security for searchable encryption is requiring that nothing is leaked beyond the access and search patterns. We should mention that some dynamic SSE schemes also leak information during the add/delete operations.

This intuitive idea is captured using the extension to the setting of dynamic SSE schemes (as in [5]) of the security definition of Curtmola et al. [3], the so-called *security against adaptive chosen-keyword attacks* (CKA2). The leakage functions associated to index creation, search, addition and delete operations are denoted as $\mathcal{L}_I$, $\mathcal{L}_S$, $\mathcal{L}_A$, $\mathcal{L}_D$ respectively. Then the security is defined using the simulation paradigm, which is the standard way of defining strong security guarantees in cryptography.

Definition 2 (Dynamic CKA2-Security). *Let* SSE = (Gen, Enc, SearchToken, AddToken, DeleteToken, Search, Add, Delete, Dec) *be a dynamic index-based symmetric searchable encryption scheme and $\mathcal{L}_I$, $\mathcal{L}_S$, $\mathcal{L}_A$, $\mathcal{L}_D$ be leakage functions. Then the following experiments are considered:*

- $\mathbf{Real}_{\mathcal{A}}()$*: The secret key K is generated by running* $\mathsf{Gen}(1^{)}$*. The adversary $\mathcal{A}$ chooses a file collection $\mathbf{f}$ and then receives an encrypted index γ and the ciphertexts $\mathbf{c}$ such that $(\gamma, \mathbf{c})\mathsf{Enc}(K, \mathbf{f})$. The adversary $\mathcal{A}$ can make a polynomial number of adaptive queries to get search, add and delete tokens. The tokens are generated using the respective algorithms of* SSE

(the ciphertext is also generated in the case of an addition) and given to the adversary. Finally $\mathcal{A}$ outputs a bit b indicating whether he thinks he is the real or ideal experiment.

- $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}()$*: The adversary $\mathcal{A}$ chooses a file collection $\mathbf{f}$. The simulator $\mathcal{S}$ only gets $\mathcal{L}_I(\mathbf{f})$ and has to simulate an encrypted index γ and ciphertexts $\mathbf{c}$ to send to the adversary. The adversary $\mathcal{A}$ is again allowed to make adaptive queries to get search, add and delete tokens; but the simulator has to generate the tokens (and also the ciphertext in the case of additions) to sent to the adversary given only the leakage from either $\mathcal{L}_S$, $\mathcal{L}_A$ or $\mathcal{L}_D$. Finally $\mathcal{A}$ outputs a bit b indicating whether he thinks he is the real or ideal experiment.*

SSE *is ($\mathcal{L}_I$, $\mathcal{L}_S$, $\mathcal{L}_A$, $\mathcal{L}_D$)-secure against adaptive dynamic chosen-keyword attacks if for all probabilistic polynomial time adversaries $\mathcal{A}$, there exists a probabilistic polynomial time simulator $\mathcal{S}$ such that* $|\mathbf{Real}_{\mathcal{A}}() = 1 - \mathbf{Ideal}_{\mathcal{A},\mathcal{S}}() = 1| \leq$ negl().

The intuition behind this definition is that if every adversary cannot distinguish whether the encrypted index, ciphertexts and tokens given to him were generated using the real data and the scheme SSE or by a simulator which only gets as input the information specified by the leakage functions, then SSE only leaks the information specified by the leakage functions.

Using this security definition the leakage of the scheme SSE can be formally defined. As dynamic index-based symmetric searchable encryption schemes should leak as little information as possible, a good example would be: $\mathcal{L}_I$ leaking only the number of files and unique keywords, the identifiers of the files and the size of the files, $\mathcal{L}_S$ leaking only the search and access patterns, $\mathcal{L}_A$ leaking only the size and identifier of the added file as well as the updated number of unique keywords and $\mathcal{L}_D$ leaking only the updated number of unique keywords.

4. Existing Approaches

4.1. Two-Layered Encryption Scheme

The first construction of SSE was presented by Song et al. [1], who developed a solution based on a special two-layered encryption scheme. The idea is to encrypt each keyword separately using a deterministic encryption scheme in the first layer and then use a stream cipher with a special structure for the second layer of the encryption. The keystream for the second level is generate in a special way which allows the detection of the keywords during an execution of the search algorithm. More specifically, for a keyword w, in the first layer a deterministic encryption $x = E(w)$ of w is computed and then parsed in two parts $x = x_\ell \| x_r$. The first part x_ℓ is then used to generate a key k for a hash function h. Finally the keystream is chosen by picking a random seed s, which is xored with x_ℓ, and then computing $h(k, s)$, that is xored with x_r. In order to perform a search for the keyword w the search token $\tau_s(w)$ is $x = E(w)$

and the key k generated from x_ℓ. With this token the server can perform the search by testing for each ciphertext c whether cx has the format $s\|h(k,s)$ for some s or not. Unfortunately this schemes has some problems: first, the scheme uses fix-sized keywords and is not compatible with existing encryption standards; second, it does not achieve a strong notion of security – it has no security guarantees related to search capabilities of the scheme, the only security guarantees is about the ciphertext themselves (which are IND-CPA secure). Indeed the scheme leaks the position of the keyword within the document, which can lead to attacks based on statistical analysis; finally, the search time is linear in the total number of words contained in the documents.

4.2. (Forward) Index Approach

The first approach for designing SSE schemes with stronger security guarantees and linear search time in the number of documents was the (forward) index approach introduced by Goh [2]. In such approach, for each document, there is an associated encrypted data structure that is used for searching the keywords. The index is independent of the underlying encryption algorithm. A user that possess the secret key can generate a search token for a specific keyword, which allows the server to search for the files containing that keyword using the index. Goh's scheme [2] uses Bloom filters [25] to build the index. Bloom filters are a data structure that can be used to answer set membership queries. It uses an array of ℓ bits which are initially 0. For each element w to be added into the set, t independent hashes of w are computed, where each hash function h_i hashes into the set $\{1,\ldots,\ell\}$, and then the bits $h_i(w)$ are set to 1. Using this data structure, it is possible to check whether the keyword is present in a document or not by checking whether all the bits outputted by $h_i(w)$ are set to 1 or not. But this method inherently produces false positives. To avoid leaking information about the keywords, Goh's scheme first process the keyword using two pseudorandom functions before inserting them in the Bloom filters (the second function also takes as input an unique document identifier in order to avoid leaking similarities between the documents). One problem with this approach is that the number of *1*s in the Bloom filter leaks information about the number of keywords associated with that document.

Chang and Mitzenmacher [26] developed a solution without false positives. The idea is to use a prebuilt dictionary of keywords to build an index per document. It is represented as an array with ℓ bits, where ℓ is the number of distinct keywords and each bit represents a keyword. A pseudorandom permutation is used to hide which keyword corresponds to each bit.

The main drawback of the forward index approach is that its search time is inherently linear in the number of files since the search is performed by using the encrypted data structure that is associated with each specific file. Additionally, the security notions used on the works mentioned above do not guarantee the security of the search tokens.

4.3. Inverted Index Approach

The central idea of the inverted index approach is to use an index per distinct keyword instead of per distinct document. This change reduces the search time from linear in the number of documents to linear in the number of documents that contain the searched keyword, which is optimal. The first schemes using this approach were presented by Curtmola et al. [3].

The idea of the scheme is that for each keyword w there is a linked list L_w which contains the identifiers of the documents that contain the keyword w. But these linked lists cannot be store in a straight-forward and unencrypted way, since this would leak information. The idea is that the nodes of all linked lists are stored together in an array A, in a scrambled order and in an encrypted format. The plaintext of each node consists of three parts: the identifier of one document, the key used to encrypt the next node of the linked list and the pointer to the next node of the linked list. What is then needed in order to perform the search for keyword w is the key used to encrypt the first node of L_w and a pointer to its location within A. This information is stored encrypted in a pseudorandom position of a look-up table T. The search token $\tau_s(w)$ then consists of the position in T used for keyword w together with the key that was used to encrypt this entry of T. This scheme achieves security according to the strong security notion of Curtmola et al. [3] against non-adaptive adversaries, i.e., the adversary has to choose the values it will query at onset before seeing any other information.

In order to obtain security against adaptive adversaries, Curtmola et al. [3] also proposed a second scheme, with increased communication and storage complexities. The idea is to use a look-up table T directly, but with extended labels. For a keyword w appearing in n documents, the extended labels are $w\|1, \ldots w\|n$ and for each of them there is an associated pseudorandom entry of T containing the identifier of one of the documents in which w appears. The keyword w_{MAX} that appears more often on distinct documents has to be determined and in also how many documents MAX it appears. The search token for w consists of the outputs of permutation that scrambles T applied on the inputs $w\|1, \ldots w\|\mathsf{MAX}$. The scheme pads the table with dummy entries so that the identifier of each document appears in the same number of entries. The search in this scheme is linear in the maximum number of documents that contain a single keyword, i.e. MAX.

Chase and Kamara [4] proposed structured encryption, which is a generalization of index-based SSE schemes. They also noticed that the simpler scheme of Curtmola et al. [3] (i.e., the one that is only secure against non-adaptive adversaries) can be also be made secure against adaptive adversaries by requiring the symmetric encryption scheme that is used to encrypt the nodes to be non-committing.

Kurosawa and Ohtaki [6] showed that it is possible to extend the second SSE scheme of Curtmola et al. [3] (i.e., the one that is secure against adaptive adversaries and has linear search time) in order to achieve a stronger notion of security (UC security [27]) that guarantees security against active adversaries

(instead of only against passive ones, as considered in the other works). The idea is to extend the scheme by using message authentication codes in order to make it a verifiable SSE scheme. The biggest limitation of the resulting scheme is its linear search time.

One big limitation of the above schemes is that they are not explicitly dynamic. The arrays would need to be updated when a file addition/deletion is performed, and using general techniques for making it dynamic would result in an inefficient final scheme. Another significant limitation is that they are not parallelizable since the encrypted indexes used in these schemes store data at random positions and the location of the next position to be accessed is only learned when the data in the current one is retrieved.

4.3.1. Achieving Dynamicity Using a Deletion Array

One idea to obtain a dynamic SSE is to use a deletion array [5]. Using the simpler scheme of Curtmola et al. [3] (which is secure against non-adaptive) as a starting point, Kamara et al. [5] were able to perform modifications in order to obtain the first secure dynamic SSE scheme[1], which is proven secure in the random oracle model. The two limitations of the original scheme are that it is only secure against non-adaptive adversaries and that it is not explicitly dynamic. The first limitation can be overcome by using a non-committing symmetric encryption scheme as mentioned above, but the second one is more difficult to overcome.

The problem is that when a file is added/deleted, the nodes in the search array A have to be updated. More specifically, when a file f is deleted, the nodes in A corresponding to f should be cleared. When a file f is added, it is necessary to locate free locations in A to add the nodes corresponding to f. Additionally, when a file is added or deleted, some pointers in the linked list have to be updated (but they are encrypted). To deal with this, Kamara et al. [5] use the following techniques: (1) a deletion array keeps track of the search array positions that need to be modified if a file deletion occurs. This deletion array can be queried given a token that is generated by the client. (2) There is a list of free nodes which keeps tracks of the free positions in the search array A and can be used by the server when a file is added. (3) The pointers are encrypted using a homomorphic encryption scheme in order to allow modifications without decrypting. Specifically, the encryption is done by XORing the message with the output of a PRF (note that this construction is also non-committing).

In the proof of security against adaptive adversaries of static SSE schemes, the queried keywords can be chosen based on the encrypted index and the results of the previous queries, and this requires the simulator to create an encrypted index which is equivocable, i.e., the simulator creates a "fake" encrypted index. Later, when a keyword is queried for the first time, the simulator can generate

[1] van Liesdonk et al. [28] designed an explicitly dynamic SSE scheme. However, they only presented a formal security proof for the case of static file collections. Additionally, the encrypted index in their scheme is relatively large.

an appropriate search token $\tau_s(w)$. This level of equivocation was achieved by simply using non-committing encryption schemes [3, 4]. However, in the case of dynamic SSE schemes, a higher level of equivocation is required. The adversary can initially query a keyword w in order to commit the simulator to a search token $\tau_s(w)$, then add a file f that contains w (the simulator does not know about this fact, and thus cannot modify the encrypted index in a meaningful way) and finally query w again, at which point the simulator is already committed to the search token $\tau_s(w)$ but was unable to update the encrypted index to reflect the changes. To address this, Kamara et al. [5] designed the scheme so that the adversary needs to query a random oracle during the search algorithm execution. The random oracle then provides the required level of equivocation for the simulator.

The main problem with this scheme is that the leakage function associated with the addition/deletion of files leaks too much information, namely the search tokens corresponding to the keywords contained in the added/deleted file. In the important case in which the database is initially empty and the files are incrementally added by the client, this scheme is no more secure than using a deterministic encryption scheme.

4.3.2. Achieving Dynamicity by Learning the Inverted Index On-the-Fly

Another idea to obtain dynamic SSE schemes is to build the inverted index on-the-fly, as proposed by Hahn and Kerschbaum [10]. It is based on the idea of learning the inverted index for efficient access from the access pattern itself. With this approach, one starts with a forward index based searchable encryption scheme (using the files as index) that requires linear scans and an empty inverted index. When a keyword is searched for the first time, its access pattern and deterministic search token (for the inverted index) are learned. Next, the keyword is incorporated into the inverted index. When new searches are done for the same keyword, the inverted index is used to search in sub-linear time. Additionally, if an added/deleted file contains a keyword which is already in the inverted index, then the entry corresponding to that keyword in the inverted index is updated.

The central observation used in this approach is that the search tokens of the known SSE constructions remain valid for future usage (until the entire system is rekeyed). Hence, if a keyword was already searched and its search token learned by the server, then updating the inverted index entry corresponding to that keyword can be done without leaking additional information to the server (the server could already use the old search token to test if the added/deleted files contained that keyword anyway).

Using this approach it is possible to obtain a scheme which has asymptotically optimal amortized search time (if the number of search queries is large enough) and small index size, and for which it is proved in the random oracle model that the updates leak no more information than the access pattern (i.e., no more than what can be inferred from the search tokens). The obtained scheme can either have no storage on the client side other than the keys, or store the search history in the client in order to improve the performance of the

update procedure. The main drawback of this approach is that the time for the first search of a keyword is linear.

4.4. Keyword Red-Black Tree

Given the inherently sequential nature of the inverted index approach and the fact that the dynamic SSE schemes based on that approach are very complex and difficult to implement, Kamara and Papamanthou [7] developed an alternative method for obtaining SSE schemes, which also enjoys sub-linear search time but is highly parallelizable and easily handles dynamic file collections. It uses a structure similar to red-black trees and so was named as *keyword red-black tree*. The keyword red-black tree is then encrypted using pseudorandom functions and permutations and a random oracle. The final scheme has the same asymptotic efficiency as an unencrypted keyword red-black tree.

The keyword red-black tree is binary tree-based multi-map data structure. It is assumed that the universe of keywords is fixed (m in total) and much smaller than the number of files, which can grow dynamically. Additionally, a total order on the documents $\mathbf{f} = (f_1, \ldots, f_n)$ is imposed by the ordering of the identifiers. At the leaves of tree, pointers to the appropriate documents are stored. At each internal node u of the tree, a m-bit vector $d_u = d_{u,1} \ldots d_{u,m}$ is stored, in which $d_{u,i}$ corresponds to the i-th keyword w_i of the universe. The bit $d_{u,i}$ is set to 1 if, and only if, one of the files associated with u's children contains the keyword w_i. This can be efficiently computed by starting at the leaves, and then for the internal nodes computing d_u as the bitwise OR of the values of its two children. To search for a keyword w_i, simply start at the root and continue recursively until either a node is achieved in which $d_{u,i} = 0$ (no file associated with the children nodes contain w_i) or a leaf is achieved for which the associated file contains w_i. One reason why this data structure is useful is that it supports both keyword-based operations (following the paths from the root to the leaves), which are used for searching, and file-based operations (following paths from the leaves to the root), which are used to handle updates. Another useful property is that the search in each children can continue using a different processor. The idea for encrypting the data structure is as follows: for each keyword w_i there is a distinct key that is used to encrypt the bits $d_{u,i}$ (for all u). The encrypted bit $d_{u,i}$ is then stored at one of two hash tables associated with node u, at a pseudorandom position. Whether it is stored in the first or second hash table depends on the output of a random oracle. The other table will contain a random value in the respective position. In order to perform an update, the server performs a structure update on the keyword red-black tree, which involves the necessary rotations that are performed during an update of a red-black tree (in order to maintain a a logarithmic height). Note that only the file identifier is required for performing this operation. The server then sends to the client the part of the tree that needs to be updated, and the clients answers with a token that allows the server to update the values at those positions.

Using these building blocks, the scheme was proved to be secure in the random oracle model. The updates do not leak any information apart from what can be inferred from the previous search tokens (in contrast with the

scheme by Kamara et al. [5] for instance) and can be efficiently performed since all information about a file f can be found and updated in $O(\log |\mathbf{f}|)$ time, but require one and a half rounds of interaction. The total search time is almost optimal (loose by a factor $O(\log |\mathbf{f}|)$), but it is easily parallelizable, and if $\omega(\log |\mathbf{f}|)$ processors are used, its clock search time is smaller than the optimal sequential search time. If a large enough number of processors is available, the resulting clock search time is of $O(\log |\mathbf{f}|)$. One drawback of this scheme is that the data structure has size $O(m \cdot |\mathbf{f}|)$ and the constants are quite high.

4.5. Dictionary Entry per Combination of File and Keyword

As large databases are the main motivation for outsourcing storage, Cash et al. [8] proposed a (dynamic) SSE scheme based on a new approach that was designed with scalability to very-large databases (in the order of billions of file/keyword pairs) in mind. The new approach for designing (dynamic) SSE schemes is based on the idea of storing each occurring combination (file f, keyword w) as an entry in a generic dictionary data structure. Their scheme associates a pseudorandom label with each file/keyword pair, and then stores the encrypted file identifier with that label in a generic dictionary data structure. The labels are computed in such a way that the client, given a keyword w to be searched, can compute a short, keyword-specific key K_w that allows the server to perform the search by first recovering the necessary labels, then retrieving the encrypted file identifiers from the dictionary and decrypting them. This is done by using a pseudorandom function with the key K_w to create the labels and then applying it to a counter in order to generate the labels for each (file f, keyword w) pair. The search in this scheme is fully parallelizable, which is a key parameter for allowing the scalability of SSE schemes.

To allow additions to the database, the clients need to be able to compute the labels for the added data. This in turn requires either the storage of counters by the client or communication that is proportional to the total number of keywords ever added or deleted. Deletions are handled via a pseudorandom revocation list kept by the server and used by the server to filter out the results. Space can only be reclaimed via periodical re-encryption of the complete database.

SSE schemes often store data at random locations, thus resulting in a lack of locality, which impacts the I/O performance. Hence, to achieve high scalability – scaling for databases containing billions of file/keyword pairs – modifications to improve the I/O performance are needed, on top of providing a basic scheme using a dictionary.

Databases typically contain a large variability in the number of matches for different keywords. Searchable encryption schemes need to consider this, in order to improve scalability. One technique used to reduce the number of dictionary retrievals is packing the related results together. We differentiate between keywords with small, medium and large sets of associated files. For small sets, the file identifiers are stored directly (in a packed form) in the dictionary. For medium sets, blocks of pointers are stored in the dictionary and they point to blocks of file identifiers that are stored in random positions of an array. For large sets, there are two levels of indirection: the dictionary stores block of pointers

that point to block of pointers (stored in the array) that point to blocks of file identifiers.

This scheme [8] is secure against non-adaptive adversaries in the standard model and against adaptive adversaries in the random oracle model, has minimal leakage, optimal server index size (i.e., its size is of the order of the number of file/keyword pairs), optimal search time (i.e., of the order of the number of files matching the keyword) and allows fully parallel searching. One disadvantage of this scheme is that either expensive communication, or storage in the client side (to keep track of counters used in the updates) is required. Another disadvantage is that additional storage (linear in the number of deletes) is required on the server side in order to store the revocation list and the space corresponding to the delete items can only be reclaimed if the complete database is re-encrypted. Hence this scheme is suitable only for applications where deletions are relatively rare.

4.6. Hierarchical Structure of Logarithmic Levels

Stefanov et al. [29] 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 $1, \ldots, \ell - 1$ 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.

4.7. Blind Storage

Naveed et al. [9] introduced a basic primitive called blind storage, which allows the client to store a collection of files on the server in such a way that all the information about them is kept secret from the server until they are accessed, including the number of stored files and the lengths of each file. When a file is accessed, the server learns about its existence and size, but not its name or contents. The server can also notice if the same file is accessed multiple times.

They build a blind storage scheme by storing each file as a collection of blocks kept in pseudorandom locations. There is an upper bound N on the number of data blocks that can be stored. Given a file f with n blocks, αn locations of the set $\{1, \ldots, N\}$ are chosen using a pseudorandom number generation 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. 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 achieve by either storing these information on the client (which is practical if the data collection 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 schemes, the original files and the added files are treated differently 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). The index corresponding to the added files is done using a much simpler scheme which supports efficient updates.

One advantage of this scheme is that the server does not need to perform any computation, but only to provide interfaces for uploading and downloading files, which makes the scheme much more transparent for using in cloud environments. Additionally, its proof of security is in the standard model, which is a consequence of the fact that the server does not carry out any decryption. A significant disadvantage however 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 time – which is often the case in practice.

4.8. Extensions to More Complex Queries and Models

The methodologies described above focused on the case of single-keyword searches. Cash et al. [11] showed how to extend the data structures of SSE schemes that allow single-keyword searches in order to permit more expressive queries such as conjunctive search and general Boolean queries (via the OXT protocol of [11]). The information stored in these data structures is expanded from simple document identifiers to also include protocol-specific values (of the

OXT protocol). The central idea of the OXT protocol is to start the search with the least frequent keyword using the basic search scheme of the single-keyword SSE scheme and then use the specific values of the OXT protocol in order to filter out the documents that do not match the remaining keywords. In order to do that the protocol uses a pre-computed two-party protocol based on the decisional Diffie-Hellman assumption about discrete-log related hard computational problems. Using this methodology it is possible to allow more expressive queries while maintaining the search performance. However, the price to pay is the larger leakage profile.

Jarecki et al. [12] similarly showed how to extend those data structures in order to allow more complex multi-client SSE settings. In these settings, the client doing the searches is not necessarily the data owner, but only gets search tokens from the data owner in order to perform the authorized queries that he wants. The authors present solutions for both the case in which the data owner can and cannot learn the searched terms. Their solution is essentially an extension of the OXT protocol.

5. Privacy Issues

There are obviously trade-offs that have to be done for searchable encryption to achieve functionality. This is captured in the security proof of the schemes by the leakage function. A desirable leakage profile for a SSE scheme would be to leak only the outcome of the search (i.e., the identifiers of the documents that contain the queried keyword), which is known as the access pattern, as trying to hide this information requires the use of expensive techniques. However, normally one has to make a bigger compromise: the current efficient approaches use deterministic search tokens, which leads to the leakage of the search pattern (i.e., whether two queries are for the same keyword or not). In addition to access and search patterns, many schemes also leak some general information, such as number of files, number of keywords, number of file/keyword pairs, etc. However, this kind of information is a reasonably acceptable form of leakage.

The main problem with leakage occurs in dynamic SSE schemes since many schemes leak additional information during the add/delete operations. One dangerous form of such leakage is leaking the search tokens corresponding to the keywords contained in the added/deleted file (even for the keywords that were not searched in past) [5]. This renders the scheme inappropriate for databases in which most of the data is added incrementally (the scheme would be no more secure than using a deterministic encryption scheme if the database is initially empty and the files are incrementally added by the client). Obviously, if the deterministic search tokens are still valid in the future (which is the case in all current schemes except [29]), then the server can test them against the added files in order to learn if the added file contains the keywords that were searched in the past; we are not aware of a solution to this issue.

Extending an SSE scheme that allows single-keyword searches in order to allow more complex queries [11] also implies an extended leakage function. In this case, it is not completely clear how dangerous this additional leakage can

be for the users. In the specific case of the OXT protocol [11], care should be taken to always use the least frequent keyword as the first keyword in the query, so that the additional leakage due to the OTX protocol is as limited as possible.

6. Efficiency

In terms of efficiency, one essential parameter is the search time complexity: schemes which have a search time which is linear in the number of documents are impractical in most scenarios. Therefore, it is essential to have sub-linear search time, and ideally optimal search time (i.e., search time which is proportional to the number of documents that contain the queried keyword). Schemes which have an asymptotically optimal search time, but have a linear search time for the first search of a keyword (such as [10]) are not useful in all practical scenarios. Having a poly-logarithmic (in the number of files) overhead over the optimal search time [7, 29] can also be problematic in the case of databases with large number of small files.

Another important parameter is the possibility to parallel the search. Schemes supporting this feature (e.g., [7, 8]) are particularly amenable for usage in a cloud environment. Additionally, the scheme should ideally be designed so that it maximizes the I/O performance [8] by improving the locality of the data structures used for searching.

Another main parameter is the size of the data structures that need to be stored by the server (and possibly by the client). Ideally, the data structure kept by the server should have optimal size (i.e., size of the order of the number of file/keyword pairs). The need for additional storage (linear in the number of deletes) in order to store a revocation list (e.g., [8]) can be troublesome in the case of highly dynamic file collections. Not recovering the space corresponding to the delete items until the database is completely re-encrypted [8] can limit the applicability to scenarios where deletions are quite infrequent. Storing a small amount of information on the client side (such as one counter per keyword [8] or the search history [10]) in order to improve the performance can be a good solution in some scenarios, but is not universally applicable. Additionally, the number of rounds of interaction between the client and the server should be kept as small as possible in order to minimize network delay.

Despite the clear advantages of SE, there are certain drawbacks that prevents cloud service providers from adopting such techniques and keep them from becoming mainstream. More precisely, as described in [30] searching the stored documents takes linear time in the size of the database, and/or uses heavy arithmetic operations. Furthermore, the existing schemes do not consider adaptive attackers; a search-query will reveal information even about documents stored in the future. If they do consider this, it is at a significant cost to the performance of updates. Apart from that, SE schemes allow for Boolean searches but have not shown how to efficiently support phrase searching. Sub-word matching, exact matches, regular expressions, natural language searches and proximity-based queries are all functionalities that modern search engines employ and

users expect to have. However, such functionality is absent from current SE approaches.

7. Openstack

OpenStack is an open-source cloud computing software platform that was first released in 2010 and currently developed under the guidance of the OpenStack Foundation, a non-profit corporation entity. This project is supported by more than 200 companies around the world, including key industry players. One important criterium for the success of such attempt is the ability of introducing search capacities for the encrypted data with minimal modification on the server side, in order to facilitate its adoption by the OpenStack community. Currently, OpenStack only has native support for protection of data at rest, which allows limited actions for volume encryption, ephemeral disk encryption and object storage encryption.

Implementation of a searchable encryption scheme for the OpenStack Database components would significantly boost the security of OpenStack cloud deployments. A first use case for implementing searchable encryption in OpenStack is encrypted access to OpenStack service configuration data.

7.1. Architectural Overview

OpenStack is a free and open source cloud management platform, which allows to set up, operate and maintain large-scale cloud computing deployments. It is one of the largest open source cloud management platforms, supported by more than 500 companies[2]. Since its first release in 2010, OpenStack has had a rapid community-driven evolution and is currently at its eighth release.

On a higher level, OpenStack is a collection of independent components that communicate with each other through public APIs and collectively form a robust cloud computing platform. Some of the core OpenStack services are the *dashboard* which serves as a graphical user interface for the *compute component*, the *image store* and a *object store*. The three latter components authenticate through an *authentication* component.

The current release of OpenStack ("Newton") comprises five components which correspond to the above logical structure:

- *OpenStack Compute (code-name Nova)* is a core component of OpenStack and focuses on providing on-demand virtual servers. Nova offers several services, spawned on different nodes in an OpenStack deployment depending on the purpose of the node. The services are *nova-api, nova-compute, nova-volume, nova-network* and *nova-schedule*. Additional services, which are not part of *Nova* but are however used by it are a queue service (currently RabbitMQ is used, however any other queue system can be used instead) as well as a SQL database connection service (MySQL and PostgreSQL are supported for production, sqlite3 for testing purposes).

[2]List of supporting organization: `http://www.openstack.org/foundation/companies/`

- *OpenStack Networking (code-name Neutron)* is a core project implementing support for a range of networking models that fulfill the needs of various applications and user groups. While basic models include flat networks with VLANs for tenant isolation, *Neutron* can be extended to take advantage of the Software-Defined Networking model and create massively scalable multi-tenant virtualized networks. The extension framework also allows to deploy and manage software implementations of additional network services, e.g. load balancing, firewalls, virtual private networks, etc.

- *OpenStack Dashboard (code-name Horizon)* is a Django-based dashboard which serves as a user and administrator interface to OpenStack. The dashboad is deployed through `mod_wsgi` in Apache and is separated into a reusable python component and a presentation layer. Keystone also uses an easily replaceable data store which keeps information from other OpenStack components.

- *OpenStack Image Service (code-name Glance)* is VM image repository that stores and versions the images that are made available to the users initially or modified through subsequent runtime updates.

- *OpenStack Object Storage (code-name Swift)* is an object store with a distributed architecture which aims to avoid single points of failure and facilitate horizontal scalability. It is limited to the storage and retrieval of files and does not support mounting directories as in the case of a fileserver.

- *OpenStack Identity (code-name Keystone)* is a unified point of integration for the OpenStack policy, token and catalog authentication. Keystone has a pluggable architecture to support multiple integrations, and currently LDAP, SQL and Key-Value Store backends are supported.

- *OpenStack Block Storage (code-name Cinder)* manages the creation and operation of block devices on servers, enabling tenants to fulfil their storage requirements. The block storage system is appropriate for performance-sensitive scenarios (e.g. database storage, expandable file systems, access to raw block-level storage, etc.). Besides the native block storage implementation, the OpenStack Block Storage currently provides support for other storage platforms.

- *OpenStack Telemetry (code-name Ceilometer)* service aggregates usage and performance data across OpenStack services and provides support for billing and a global resource utilization map. This is necessary as service provides often require to collect accurate information about the utilization of computing, storage and networking resources within a certain infrastructure cloud deployment.

- *OpenStack Orchestration (code-name Heat)* In order to support scalable, large-scale cluster deployment, OpenStack uses a template-based orchestration engine which allows automated deployment of infrastructure. The

orchestration engine is used both for pre- and post-deployment actions and configuration changes, as well as for auto-scaling of key infrastructure elements based on the information provided by the telemetry service.

- *OpenStack Database (code-name Trove)* service provides a native OpenStack relational database which can be used for infrastructure management tasks, such as a deployment, patching, backing up, restoring and monitoring infrastructure components.

- *OpenStack Bare-Metal Provisioning (code-name Ironic)* service aims to provision bare metal (i.e. non-virtualized) computing resources similar to the current application of PXE and IPMI protocols.

All of the above described components interact through a set of REST application programming interfaces (APIs) and form the fabric of a cloud computing infrastructure deployment.

The OpenStack documentation [3] describes in details each of the above named components and their interaction.

7.2. Storage Protection Mechanims

There are currently several mechanisms for protection of data in OpenStack, both for data at rest and data in transit. While data in transit can be protected using common mechanism such as TLS and IPSec, we instead focus on the storage protection mechanisms found in OpenStack. When it comes to confidentiality of data at rest, the available functionality is limited to basic symmetric encryption capabilities. Thus, OpenStack tenants have the following complementary options: volume (i.e. block storage) encryption, ephemeral disk encryption and object storage encryption.

The *volume encryption* functionality in OpenStack supports per-tennant creation and usage of encrypted volumes, as well as encrypted backups and is exposed to a key management service. Some proposed approaches for volume encryption allow to transparently mount volumes to guest virtual machines with the encryption and decryption being handled by the disk encryption subsystem of the cloud host. However, this functionality is not currently integrated in the official OpenStack release.

The *ephemeral disk encryption* feature allows encryption of the temporary work space used by each individual virtual host operating system. This prevents plain-text residual information from earlier tenants to be left on the physical disks of the cloud hosts.

Finally, *object storage encryption* is currently limited to disk-level encryption per node. The encryption functionality for the Swift object storage is currently under development.

[3] OpenStack Documentation Page `http://docs.openstack.org`

7.3. Searchable Encryption in OpenStack

Searchable encryption has the potential to considerable expand the use of encryption of data at rest within OpenStack and directly contribute to the proliferation of security-hardened OpenStack deployments. Furthermore, a contribution to the implementation of a searchable encryption scheme for the block storage in OpenStack would be welcomed by the OpenStack community and give **significant** visibility among the users and contributors of the project.

A feasible target for implementing searchable encryption functionality is the OpenStack configuration database (code-name Trove). The database contains sensitive configuration data and is accessed for operational purposes by various components of the OpenStack deployment. Disclosure of such sensitive configuration information can lead to a complete and irreversible compromise of the cloud deployment. Implementing searchable encryption functionality for the configuration database would allow the system components to identify and retrieve encrypted entries in the configuration database without having to decrypt the entire set of stored data. This would help protect the confidentiality of the data with a minimal communication overhead.

8. Recommendation for Implementation

In the light of the issues discussed in the previous sections it is obvious that some kind of compromise has to be done as none of the state of art searchable encryption schemes achieves all the ideal attributes. Sub-linear time and support for dynamic databases are with great probability the most important points and therefore they should be supported by the scheme chosen to be integrated with OpenStack. Another important facet, as pointed out earlier, is the ability to add support for search over encrypted data while changing the server side as less as possible (in order to minimize the resistance against its adoption from the side of the OpenStack community). Taking these parameters into account, the scheme of Naveed et al. [9] stands out as the most appropriate for integration with OpenStack-based platforms as it views the cloud simply as a storage service, has optimal search time and supports dynamic databases. One additional advantage of this scheme is that it has a security proof in the standard security model, as opposed to most schemes which were only proven to be secure in the heuristic random oracle model. The disadvantage of the scheme is that the level of security for the added files is smaller than for the original files. However, we considered that this is the best trade-off possible given the current state of affairs in the field of searchable encryption. Therefore our choice was to implement the scheme of Naveed et al. [9] in order to check its performance for real applications and the possibility of integrating it with OpenStack.

9. Experimental Results

For the needs of the paper, we implemented the Searchable Encryption Scheme on top of a Blind Storage System as proposed by [9]. In order to be

GNU Toolchain (GCC compiler)	5.2.1
Boost	1.58.0
Crypto++	5.6.1
Curl++	0.7.3
Curl	7.43.0

(a) Build environment, libraries and versions

α (expansion factor)	4
κ (minimal number of blocks per file)	80
block size (bytes)	4096
total block number	2^{18}

(b) Runtime parameters

CPU	AMD A10-7850K Radeon R7
RAM	16 GB
OS	Ubuntu Desktop 15.10

(c) Client environment

CPU	Virtual CPU with 1 core (see client)
RAM	2 GB
OS	Debian 8 (Jessie)

(d) Server environment (actually not used)

Table 1: Experimental setup

comparable with their findings our implementation was built with the same tool chain and uses the same third party libraries as far as known. This is to say, the application is written in ISO C++ 2011 and uses the Boost [31], Crypto++ [32] and CurlPP [33, 34] libraries.

We used the CurlPP is a multi-protocol network library for all network IO. We used the Crypto++ library for all cryptographic primitives. The Boost library was used for two different aspects: to abstract from OS-dependent parts such as runtime configuration and user interaction. This part does not contribute to the performance measurement, because any interaction "with the outside world" (like reading runtime parameters and user input) only occurs during the start-up phase of the program and not during the actual processing phase. Moreover, the Boost library is used to split the files into tokens and to create lists of keywords that are stored in the index and can be searched for. We stress this aspect, because [9] do not state how the files were preprocessed and tokenized and our results are not comparable to theirs (see details below). For more details on the build environment see Table 1a.

We chose a comparable environment as [9] (see Table 1c) but had to change the runtime parameters to those depicted in Table 1b. The reasons are explained

in 9.1.

Originally, we planned to run our performance measurements in a somewhat realistic scenario with a real FTP server and virtualized network communication. For this purpose a virtual machine was set up on the same host as the client (see Table 1d). All network communication was sent through a virtual network between the client and the FTP server running within the virtual machine. However, we later modified the setup and replaced the network attached storage by a local storage (9.1).

We also used the Enron dataset [35] and selected random subsets of appropriate size for the experiments.

9.1. Preliminary remarks

In a first experiment we initialized the blind storage system with the parameters used by Naveed, Prabhakaran, and Gunter, i.e. 2^{22} blocks with $256\,bytes$ each or in other words $1\,GB$ of total storage space. The backend storage was provided by a FTP server inside a virtual machine. The build phase of the blind storage took about $590\,s$ of effective CPU time ($312\,s$ in user space and $278\,s$ in system space) but roughly 6 hours of real execution time. We repeated the same experiment with a much smaller number of blocks and traced all function calls by means of the profiling tool CallGrind as part of the instrumentation suite Valgrind [36]. This revealed that 95% of the running time was spent within the FTP client library. Each of the 2^{22} blocks is represented by a single file that needs to be transferred between the client and the server. No matter how the file transfer was scheduled (sequentially, n-parallel, reuse of TCP connections) the FTP transfer represented a serious bottle neck. The initialization and termination of a individual file transfer creates a non-negligible overhead especially if each file (or block) has only 256 bytes of payload. This still holds if the FTP connection as a whole is kept open and is reused for all transfers.

Naveed, Prabhakaran, and Gunter did not consider the IO time for their performance analysis, hence we decided to replace the FTP storage by a local storage[4]. After that the real execution time dropped down to $500\,s$ (instead of $6\,h$). However, we stress that for any realistic deployment this is a serious concern, because the whole point in having a blind storage system is to put it on some untrusted network storage. To ignore the time spent on network file transfer leads to seriously misleading numbers.

Originally, we also planned to use the same runtime parameters as Naveed, Prabhakaran, and Gunter, namely 2^{22} blocks with $256\,bytes$ each. However, this choice of parameters lead to a waste of disk space. In order to store 2^{22} blocks (or files) one has to use a hierarchical naming scheme similar to the one used internally by many proxy daemons. In our case the files representing the blocks were enumerated from `./00/00/00.bin` through `./3f/ff/ff.bin`. This directory structure already occupies disk space by itself. Moreover, a file size

[4]Essentially, all function calls to the FTP library were replaced by equivalent local system calls

of 256 bytes cannot be recommended, because most filesystems allocate files in chunks of $4\,kB$. On an EXT-4 filesystem the bare directory structure already used $0.5\,GB$ and the fully built blind storage scheme with 1 GB storage net capacity used $34\,GB$ of tangible disk space.

Hence, we tweaked the parameters and used 2^{18} blocks with 4kB each (see Table 1b) to better match the underlying filesystem's own parameters. With these settings the bare directory structure only used 8.2MB and the complete blind storage scheme allocated 1.1GB of real disc space, thus the overhead dropped down to 10%. Moreover, the real execution time of the built phase further declined to 125s whereby 12s were spent in user-space and 113s were spent in kernel-space.

We stress that we did not calculate whether the modified settings offer the same level of security and success probability. It is unlikely that they do, because the number of blocks were reduced, but a storage and processing overhead of 34 is not acceptable for any realistic scenario.

9.2. Methodology

Naveed, Prabhakaran, and Gunter state that they concentrated on client-side computation time and the reported numbers suggest that they somehow calculated out the costs for IO operations (especially because they used a remote DropBox as their backend). Moreover they report that the symmetric encryption (AES) accounts for a significant part of the runtime. We cannot support this statement if IO operations over a network are considered, but the statement becomes true if all network operations are replaced by local disk IO. In this case CallGrind [36] reports that 35 % of the runtime is spent inside the AES library.

However, it remains unclear how Naveed, Prabhakaran, and Gunter measured the "bare" computation time. One approach is to use an instrumentation suite such as Valgrind [36] and look at the time being spent in individual function calls. However, this raises the question of functions to be instrumented; moreover, this approach is highly implementation specific.

Another approach is to query the process scheduler of the operating system and look at the amount of the process spent in user-space and kernel-space. One could argue that the time spent in user-space is the "true" computation time (tokenization, index calculation, encryption) while the time spent in kernel-space is related to IO. However, this is misleading, since even after we replaced all network IO by local file IO and thus reducing the total execution time of the built phase of the blind storage from $6\,h$ to $125\,s$ the process spent 90% of its time in kernel-space ($113\,s$ vs. $12\,s$). However, as already stated 35% of the runtime was due to the AES operations. These observations are consistent, because a huge portion of the AES operation is memory management and thus contributes to the execution time in kernel-space.

In summary, we decided to take the total execution time from process creation through process termination as reported by the Linux `time`-command. Hence, we do not distinguish between different aspects of the execution time.

number of blocks (2^{18}) / total net size (GiB)	1	2	3	4	6	8
size of bare directory structure (MiB)	4.02	8.04	12.1	16.1	24.1	32.1
total gross size on disk (GiB)	1.01	2.02	3.04	4.05	6.07	8.09
overhead (%)	1.17	1.17	1.17	1.17	1.17	1.13
average total runtime (s)	62.8	125	189	250	375	501

Table 2: Parameters and results of test 1 (the block size was held constant at $4\,kiB$)

Total net size of files (MB)	1	2	4
Number of files	214	421	1267
Number of file-keyword-pairs	54945	111546	246657
Total execution time (s)	148	315	964

Table 3: Results of test 2: Indexing and uploading documents

With respect to practical deployment we argue this is a valid approach, because a typical end-user is not concerned with *what* contributes to the runtime. However, we would like to point out that all numbers are generated with a local storage backend, rather than a NAS backend.

9.3. Test 1: Building the Blind Storage System

Deviating from the numbers depicted in Table 1b the first series of tests built a blind storage system at different sizes by varying the number of blocks. The block size was kept at $4\,kB$ to match the underlying filesystem. The results are shown in Table 2.

Unsurprisingly, the runtime is linear in the total size of the Blind Storage scheme.

9.4. Test 2: Indexing and uploading documents

Due to our changed runtime parameters (2^{18} blocks 4kiB instead of 2^{22} blocks 256B), we were unable to upload up to 256MiB of files, as the scheme ran out of free first blocks prematurely. Recall that the largest test set of Naveed, Prabhakaran, and Gunter was 256MiB. We were able to upload at most 8MiB, as the Enron test set consists of tens of thousands of small ASCII files with each file only consisting of a few bytes but each file occupies a integral multiple of the block size. In our scenario this leads to significant fragmentation. However, our findings are consistent with Naveed, Prabhakaran, and Gunter. As our block size is 16 times larger, the possible maximum size of the total upload is expected to be smaller by the same factor. The results are depicted in Table 3.

9.5. Test 3: Searching

Similar to Naveed, Prabhakaran, and Gunter we also search for the keyword “the” in the uploaded Enron dataset. In addition to that, we also counted the

Total net size of files (MiB)	1	2	4
Total number of files	214	421	1267
Number of results	156	348	1020
Total execution time (ms)	97	186	435

Table 4: Results of test 3: Searching for the keyword "the"

time needed to handle the lazy delete strategy, as the lazy delete strategy is essential for the security of the scheme and comes with significant costs.

Without this strategy searching for a specific keyword is merely downloading the corresponding index file. This simplified task is extremely fast but the result of the search may be incorrect. Whenever a file is modified (or deleted) it is added to all index files for each keyword it contains after the modification (n.b.: in case of deletion the file is added to no index, because the file is essentially empty after deletion). This implies that each index may list a file multiple times if old versions of the same file also contained the same keyword. Moreover, a file might be listed although it does not contain the keyword anymore but an outdated version of the same file did. The latter also holds if the file was deleted. In order to get rid of such "phantom" entries, each entry in the index is accompanied with a hash of the file content. Upon searching for a keyword, the stored hash is compared to the hash of the current file content and if they differ the entry is considered outdated and removed from the search result. Moreover, the outdated entry is removed from the index and the revised index is written back to the server. In order to be able to compare the stored hashes with the current hash the first block of each file in the search result must be downloaded from the server (via the Blind Storage Scheme) and locally decrypted. Even worse, downloading a file also implies to upload a fresh re-encryption of the file such that the server cannot distinguish between a read and a write access.

In summary, skipping the lazy deletion strategy does not reflect what would actually happen if the scheme was deployed in practice. Hence, we argue that taking the time of the lazy deletion strategy into account is the right way to measure the performance.

The results are depicted in Table 4. Our findings indicate an execution time that is slower than [9] by a factor of approximately 90. This is expected because much time is spent in performing the consistency checks due to the lazy deletion strategy.

9.6. Summary of the Implementation Report

We implemented the searchable encryption scheme via Blind Storage of Naveed, Prabhakaran, and Gunter in order to reproduce their performance results and to experiment with its performance in practical situations. Unfortunately, we cannot deny that – despite all improvements over previous schemes – this scheme still fails to be practically useful. Basically, this is due to two main reasons.

The first reason can be traced back to latencies within the network IO. The security of the scheme stems from the fact that an individual file is randomly scattered across many small chunks of data that needs to be transferred between the client and the server. In fact, this is the same problem that oblivious RAM (ORAM) also suffers from.

The second reason is that the scheme is highly wasteful in space. Both our test scenario and the scenario of Naveed, Prabhakaran, and Gunter used a blind storage with a net size of $1\,GB$, but with different block sizes. Due to a 16 times smaller block size Naveed, Prabhakaran, and Gunter were able to upload $256\,MB$ into the blind storage. But this blind storage requires $34\,GB$ of space on the underlying filesystem. Taking both factors together yields an overhead by a factor of 136. We used a much larger block size that better matched the underlying file system. Thus the gross usage was $1.01\,GiB$. However, we could only upload $8\,MiB$ into the blind storage system, which has yield a factor of 130. In both cases, both overhead factors are far from being practical, due to the need to allocating two magnitudes more space than required to store the data.

10. Conclusion

Cloud storage and computing is growing exponentially due to its cost effectiveness, but it brings with it new security concerns. While security solutions that protect the clients data from the cloud are urgently needed, they must impact the clients' functionality as little as possible. Dynamic searchable encryption schemes can be very useful in this context. Indeed we have seen a lot of progress in this area in the last couple of years [37, 38, 39, 40]. Considering the latest developments in the fiels, we first performed an extensive survey of the literature of searchable encryption in order to determine the schemes that look more appropriate for integration with cloud solutions. The scheme in [9] stand out since it views the cloud as a simple storage service. Hence we proceeded with an implementation of that scheme and performed experiments in order to determine its practicality for real applications. Unfortunately, despite all theoretical progress presented by that scheme, it is still not practical for deployment in most real-world applications. Perhaps, other state of the art dynamic searchable encryption schemes which involve more interaction on the cloud side can prove to be better suited for deployment in real systems, however we fear that they can also fail to be practical enough for deployment. Searchable encryption has a lot of potential for increasing the security of cloud solutions. It is a very interesting direction for future research, in order to obtain more practical solutions that can be deployed in real applications.

References

[1] D. X. Song, D. Wagner, A. Perrig, Practical techniques for searches on encrypted data, 2000, pp. 44–55.

[2] E.-J. Goh, Secure indexes, Cryptology ePrint Archive, Report 2003/216, http://eprint.iacr.org/2003/216 (2003).

[3] R. Curtmola, J. A. Garay, S. Kamara, R. Ostrovsky, Searchable symmetric encryption: improved definitions and efficient constructions, 2006, pp. 79–88.

[4] M. Chase, S. Kamara, Structured encryption and controlled disclosure, 2010, pp. 577–594.

[5] S. Kamara, C. Papamanthou, T. Roeder, Dynamic searchable symmetric encryption, 2012, pp. 965–976.

[6] K. Kurosawa, Y. Ohtaki, UC-secure searchable symmetric encryption, 2012, pp. 285–298.

[7] S. Kamara, C. Papamanthou, Parallel and dynamic searchable symmetric encryption, 2013, pp. 258–274. doi:10.1007/978-3-642-39884-1_22.

[8] D. Cash, J. Jaeger, S. Jarecki, C. S. Jutla, H. Krawczyk, M.-C. Rosu, M. Steiner, Dynamic searchable encryption in very-large databases: Data structures and implementation, 2014.

[9] M. Naveed, M. Prabhakaran, C. A. Gunter, Dynamic searchable encryption via blind storage, 2014, pp. 639–654. doi:10.1109/SP.2014.47.

[10] F. Hahn, F. Kerschbaum, Searchable encryption with secure and efficient updates, 2014, pp. 310–320.

[11] D. Cash, S. Jarecki, C. S. Jutla, H. Krawczyk, M.-C. Rosu, M. Steiner, Highly-scalable searchable symmetric encryption with support for boolean queries, 2013, pp. 353–373. doi:10.1007/978-3-642-40041-4_20.

[12] S. Jarecki, C. S. Jutla, H. Krawczyk, M.-C. Rosu, M. Steiner, Outsourced symmetric private information retrieval, 2013, pp. 875–888.

[13] D. Boneh, G. Di Crescenzo, R. Ostrovsky, G. Persiano, Public key encryption with keyword search, 2004, pp. 506–522.

[14] D. Boneh, B. Waters, Conjunctive, subset, and range queries on encrypted data, 2007, pp. 535–554.

[15] A. Michalas, N. Paladi, C. Gehrmann, Security aspects of e-health systems migration to the cloud, in: e-Health Networking, Applications and Services (Healthcom), 2014 IEEE 16th International Conference on, IEEE, 2014, pp. 212–218.

[16] N. Paladi, C. Gehrmann, A. Michalas, Providing user security guarantees in public infrastructure clouds, IEEE Transactions on Cloud Computing PP (99) (2016) 1–1. doi:10.1109/TCC.2016.2525991.

[17] N. Paladi, A. Michalas, C. Gehrmann, Domain based storage protection with secure access control for the cloud, in: Proceedings of the 2014 International Workshop on Security in Cloud Computing, ASIACCS '14, ACM, New York, NY, USA, 2014.

[18] M. Blaze, G. Bleumer, M. Strauss, Divertible protocols and atomic proxy cryptography, 1998, pp. 127–144.

[19] N. Paladi, A. Michalas, "One of our hosts in another country": Challenges of data geolocation in cloud storage, in: Wireless Communications, Vehicular Technology, Information Theory and Aerospace Electronic Systems (VITAE), 2014 4th International Conference on, 2014, pp. 1–6.

[20] C. Gentry, A fully homomorphic encryption scheme, Ph.D. thesis, Stanford, CA, USA, aAI3382729 (2009).

[21] T. Dimitriou, A. Michalas, Multi-party trust computation in decentralized environments, in: 2012 5th International Conference on New Technologies, Mobility and Security (NTMS), 2012, pp. 1–5. doi:10.1109/NTMS.2012.6208686.

[22] T. Dimitriou, A. Michalas, Multi-party trust computation in decentralized environments in the presence of malicious adversaries, Ad Hoc Networks 15 (2014) 53–66. doi:10.1016/j.adhoc.2013.04.013.
URL http://dx.doi.org/10.1016/j.adhoc.2013.04.013

[23] R. Ostrovsky, Efficient computation on oblivious RAMs, 1990, pp. 514–523.

[24] O. Goldreich, R. Ostrovsky, Software protection and simulation on oblivious rams, J. ACM 43 (3) (1996) 431–473. doi:10.1145/233551.233553.
URL http://doi.acm.org/10.1145/233551.233553

[25] B. H. Bloom, Space/time trade-offs in hash coding with allowable errors, Commun. ACM 13 (7) (1970) 422–426. doi:10.1145/362686.362692.
URL http://doi.acm.org/10.1145/362686.362692

[26] Y.-C. Chang, M. Mitzenmacher, Privacy preserving keyword searches on remote encrypted data, 2005, pp. 442–455.

[27] R. Canetti, Universally composable security: A new paradigm for cryptographic protocols, 2001, pp. 136–145.

[28] P. van Liesdonk, S. Sedghi, J. Doumen, P. Hartel, W. Jonker, Computationally efficient searchable symmetric encryption, in: W. Jonker, M. Petkovi (Eds.), Secure Data Management, Vol. 6358 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2010, pp. 87–100. doi:10.1007/978-3-642-15546-8_7.
URL http://dx.doi.org/10.1007/978-3-642-15546-8_7

[29] E. Stefanov, C. Papamanthou, E. Shi, Practical dynamic searchable encryption with small leakage, 2014.

[30] P. van Liesdonk, S. Sedghi, J. Doumen, P. Hartel, W. Jonker, Computationally Efficient Searchable Symmetric Encryption, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 87–100. doi:10.1007/978-3-642-15546-8_7.
URL https://doi.org/10.1007/978-3-642-15546-8_7

[31] Boost c++ libraries (8 2014).
URL http://www.boost.org/

[32] Crypto++ library (11 2015).
URL https://www.cryptopp.com/

[33] J.-P. Barrette-LaPierre, curlpp (5 2015).
URL http://www.curlpp.org/

[34] curl and libcurl (12 2015).
URL http://curl.haxx.se/

[35] Enron email dataset (5 2015).
URL https://www.cs.cmu.edu/~enron/

[36] J. Seward, C. Armour-Brown, C. Borntrger, J. Fitzhardinge, T. Hughes, P. Jovanovic, D. Jevtic, F. Krohm, C. Love, M. Johnson, P. Mackerras, D. Müller, N. Nethercote, P. Pavlu, I. Raisr, B. V. Assche, R. Walsh, P. Waroquiers, J. Weidendorfer, Valgrind (9 2015).
URL http://valgrind.org/

[37] Y. Verginadis, A. Michalas, P. Gouvas, G. Schiefer, G. Hbsch, I. Paraskakis, Paasword: A holistic data privacy and security by design framework for cloud services, in: Proceedings of the 5th International Conference on Cloud Computing and Services Science, 2015, pp. 206–213. doi:10.5220/0005489302060213.

[38] A. Michalas, R. Dowsley, Towards trusted ehealth services in the cloud, in: 2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC), 2015, pp. 618–623. doi:10.1109/UCC.2015.108.

[39] A. Michalas, K. Y. Yigzaw, Locless: Do you really care your cloud files are?, in: 2016 IEEE/ACM 9th International Conference on Utility and Cloud Computing (UCC), 2015, pp. 618–623.

[40] Y. Verginadis, A. Michalas, P. Gouvas, G. Schiefer, G. Hübsch, I. Paraskakis, Paasword: A holistic data privacy and security by design framework for cloud services, 2017, pp. 1–16. doi:10.1007/s10723-017-9394-2.
URL http://dx.doi.org/10.1007/s10723-017-9394-2