Ref. Ares(2017)3290587 - 30/06/2017

Project title: Enforceable Security in the Cloud to Uphold Data Ownership
Project acronym: ESCUDO-CLOUD
Funding scheme: H2020-ICT-2014
Topic: ICT-07-2014
Project duration: January 2015 – December 2017

D4.3

Final Versions of Tools for Security With Multiple Providers

Editors: Enrico Bacis (UNIBG)
Sabrina De Capitani di Vimercati (UNIMI)
Sara Foresti (UNIMI)
Stefano Paraboschi (UNIBG)
Marco Rosa (UNIBG)
Pierangela Samarati (UNIMI)
Reviewers: Björn Tackmann (IBM)
Daniel Bernau (SAP)

Abstract

The cloud is today characterized by the presence of a rich variety of cloud providers offering storage and computational services. This can be seen as an opportunity for users to operate over multiple providers, thus enhancing security while maximizing efficiency of access.

In this deliverable, we present the design and implementation of the *distributed shuffle index*, a technique that takes advantage of multiple providers to protect data and access confidentiality. The index randomly partitions data among three independent cloud providers and ensures access confidentiality by dynamically re-allocating data at every access. Dynamic re-allocation is enforced by swapping data involved in an access across the providers, in such a way that accessing a given node implies re-allocating it to a different provider. This protection technique destroys the ability of providers to build knowledge by observing accesses. The use of three cloud providers ensures uncertainty of the result of the swapping operation, even in presence of collusion among them. Experiments executed with the implementation demonstrate the fast access, considering a variety of scenarios (local cloud, common commercial cloud providers, high-performance multi-provider cloud infrastructure).

Type	Identifier	Dissemination	Date
Deliverable	D4.3	Public	2017.06.30

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 644579. This work was supported in part by the Swiss State Secretariat for Education, Research and Innovation (SERI) under contract No 150087. The opinions expressed and arguments employed herein do not necessarily reflect the official views of the European Commission or the Swiss Government.

ESCUDO-CLOUD Consortium

1.	Università degli Studi di Milano	UNIMI	Italy
2.	British Telecom	BT	United Kingdom
3.	EMC Corporation	EMC	Ireland
4.	IBM Research GmbH	IBM	Switzerland
5.	SAP SE	SAP	Germany
6.	Technische Universität Darmstadt	TUD	Germany
7.	Università degli Studi di Bergamo	UNIBG	Italy
8.	Wellness Telecom	WT	Spain

Disclaimer: The information in this document is provided "as is", and no guarantee or warranty is given that the information is fit for any particular purpose. The below referenced consortium members shall have no liability for damages of any kind including without limitation direct, special, indirect, or consequential damages that may result from the use of these materials subject to any liability which is mandatory due to applicable law. Copyright 2017 by Università degli Studi di Milano, EMC Corporation, Università degli Studi di Bergamo.

Versions

Version	Date	Description
0.1	2017.05.15	Initial Release
0.2	2017.05.22	Second Release
1.0	2017.06.30	Final Version

List of Contributors

This document contains contributions from different ESCUDO-CLOUD partners. Contributors for the chapters of this deliverable are presented in the following table.

Chapter	Author(s)
Executive Summary	Sabrina De Capitani di Vimercati (UNIMI), Sara Foresti (UNIMI), Stefano Paraboschi (UNIBG)
Chapter 1: Distributed Shuffle Index in the Cloud: Implementation and Evaluation	Enrico Bacis (UNIBG), Sabrina De Capitani di Vimercati (UNIMI), Sara Foresti (UNIMI), Stefano Paraboschi (UNIBG), Marco Rosa (UNIBG), Pierangela Samarati (UNIMI)
Chapter 2: Industrial Platform Performance Evaluation	Alan Barnett (EMC), Andrew Byrne (EMC)
Chapter 3: Conclusions	Sabrina De Capitani di Vimercati (UNIMI), Sara Foresti (UNIMI), Stefano Paraboschi (UNIBG)

Contents

Executive Summary **7**

1 Distributed Shuffle Index in the Cloud: Implementation and Evaluation **9**
1.1 Introduction . . . 9
1.2 Basic Concepts . . . 10
1.3 Distributed Index Implementation . . . 11
1.3.1 Node Data Structure . . . 12
1.3.2 Cloud DataLayer . . . 13
1.3.3 Configuration File . . . 14
1.4 Experimental Results . . . 14
1.4.1 Access Pattern Confidentiality . . . 15
1.4.2 Performance . . . 16
1.5 Related Work . . . 17

2 Industrial Platform Performance Evaluation **19**
2.1 Industry Testbed . . . 19
2.1.1 INFINITE Architecture . . . 19
2.1.2 Elastic Hybrid Cloud . . . 20
2.1.3 Shuffle Index Deployment . . . 21
2.2 Experimental Results . . . 21
2.2.1 Access Pattern Confidentiality . . . 21
2.2.2 Performance . . . 22

3 Conclusions **24**

Bibliography **25**

List of Figures

1.1 An example of shuffle index distributed at cloud providers (color coded) 11
1.2 An example of search for $C2$ on the index in Figure 1.1 (a), swapping between accessed nodes (b), and structure of the paths after the access (c) 11
1.3 Structure of internal (a) and leaf (b) nodes 12
1.4 Diagram of the *DataLayer* classes of the cloud providers supported 13
1.5 An example of instantiated *DataLayer* stack 14
1.6 Frequency distribution of accesses to blocks with the Gaussian access pattern . . 15
1.7 Frequency distribution of accesses to blocks with the single-node access pattern . 15
1.8 Access times and their standard deviation σ 16

2.1 INFINITE testbed overview 20
2.2 Dell EMC Elastic Cloud Storage (ECS) 21
2.3 Frequency distribution of accesses to blocks with the Gaussian access pattern . . 22
2.4 Frequency distribution of accesses to blocks with the single-node access pattern . 22
2.5 ECS testbed access times and their standard deviation σ 23

Executive Summary

The availability of a rich market for cloud services, offering a wide variety of actors, represents an opportunity for users to rely on multiple cloud providers for the storage and management of their data. The adoption of multiple cloud providers can lead to considerable advantages: the increased competitive pressure that derives from the demonstration that a client can easily switch providers leads to reduced economic costs and limited access times. The use of multiple providers also leads to higher security guarantees. Classical protection techniques designed to operate with a single provider can then be enhanced to operate with a set of cloud providers, increasing the protection guarantees they provide to the users.

In this deliverable, we present an approach aimed at protecting not only content confidentiality but also access and pattern confidentiality. It is widely recognized that protecting the confidentiality of sensitive data content is not sufficient in most scenarios. Indeed, revealing to the cloud provider storing data (and, more in general, to an observer) the target of accesses may still reveal sensitive information about both the user accessing the data and the data content itself. Different solutions have been proposed to protect access confidentiality (e.g., PIR, ORAM-based approaches, shuffle index). These solutions consider a scenario characterized by the presence of a single cloud provider storing the whole dataset.

The solution we report here is the *distributed shuffle index*, which has first been introduced in Chapter 3 of Deliverable D4.2 [TTS+16]. It relies on the presence of three independent cloud providers to improve the protection guarantees offered by the single-provider shuffle index (described in Deliverable D2.1 [PFL15]). The proposed approach is based on the combination of *data distribution* and *swapping*. Data distribution consists of allocating the data structure over three different cloud providers, each of which will then observe only a portion of the data structure and will have a limited visibility of the accesses to the same. Data swapping implies changing the physical location of accessed data by swapping them among the three involved providers. Swapping enforces the requirement that, whenever a block is accessed, the data retrieved from it (i.e., stored in the block before the access) *should not* be stored at the same block after the access. The use of three providers (for distributed data allocation) together with swapping (forcing data re-allocation across providers) provides protection guarantees also in case providers collude. In fact, each provider may even not know that data have been distributed and observes accesses to a portion of the shuffle index, while thinking that it is observing the whole dataset.

The goal of this deliverable is to illustrate the implementation of the *distributed shuffle index*. We illustrate the architectural and implementation challenges faced to realize a tool able to: *i)* build a shuffle index over a dataset; *ii)* distribute the shuffle index over three different providers; *iii)* access the distributed shuffle index structure returning the expected result to the requesting user.

The remainder of this deliverable is organized as follows. Chapter 1 introduces the considered problem and scenario (Section 1.1), recalls the working of the distributed shuffle index and

presents the rational of the approach (Section 1.2); then, the main implementation choices are presented (Section 1.3) and a first series of experimental results are reported (Section 1.4), concluding the chapter with an analysis of related work (Section 1.5). Chapter 2 reports on the implementation of the distributed shuffle index in the high-performance EMC platform. The chapter first describes the main features of the industry testbed (Section 2.1) and then reports on the results of the experiments (Section 2.2). The low latency of this infrastructure offers good performance in the execution of accesses, comparable to what is observed using a local installation, demonstrating the ability of the distributed shuffle index to be used in industrial environments. Finally, Chapter 3 presents our conclusions.

1. Distributed Shuffle Index in the Cloud: Implementation and Evaluation

The distributed shuffle index strengthens the guarantees of access confidentiality provided by the shuffle index through the distribution of data among three cloud providers. In this chapter, we analyze architectural and design issues and describe an implementation of the distributed shuffle index integrated with different cloud providers (i.e., Amazon S3, OpenStack Swift, Google Cloud Storage, and EMC Elastic Cloud Storage). The experimental results obtained with our implementation confirm the protection guarantees provided by the distributed shuffle index and its limited performance overhead, demonstrating its practical applicability in cloud scenarios. The software is open-source and is available on the Github repository associated with the ESCUDO-CLOUD project (*https://github.com/escudocloud/distributed-shuffleindex*).

1.1 Introduction

Moving data to the cloud provides significant advantages, as testified by the growing success of companies offering cloud services. To fully benefit from these advantages, users need guarantees on the protection of their data, which are no longer under the direct control of the data owner. The problem of protecting data confidentiality has been widely addressed by the research community (e.g., [DFS12]). However, protecting data confidentiality may not be sufficient. Indeed, by observing accesses, a cloud provider could infer sensitive information about the user performing the access and the possibly sensitive content of the outsourced dataset. Consider, as an example, a user searching for the treatments for a rare disease in a publicly available medical database. A cloud provider observing this access operation can infer that probably the user (or a person close to her) suffers from the searched disease. Also, by observing a long enough sequence of accesses, a cloud provider can partially reconstruct the content of the dataset (e.g., [IKK14]). In fact, it can trace the frequency with which each piece of information is accessed and exploit it to infer its content (which would in turn also disclose the targets of accesses). It is then necessary to protect also the confidentiality of accesses (i.e., the fact that an access aims at a specific piece of information) and of patterns of accesses (i.e., the fact that two accesses aim at the same target). Among the approaches recently proposed for protecting access and pattern confidentiality, the *shuffle index* organizes the outsourced data in a tree-based index structure and efficiently supports searches for index values [DFP+15]. The shuffle index can also operate in a distributed scenario with even higher protection guarantees [DFP+17]. The distributed shuffle index partitions the data over three independent cloud providers. The key idea to provide access confidentiality is to guarantee *uniform visibility* at each cloud provider, which operates as if it was the only one serving the client, and to enforce *dynamic reallocation* of data at a different cloud provider at each access, which breaches the otherwise static correspondence between a piece of information and the physical block (and cloud provider) in which it is stored.

In this chapter, we study the design and architectural issues to be addressed to implement the distributed shuffle index and to integrate it with industrial cloud providers (i.e., Amazon S3, OpenStack Swift, Google Cloud Storage, and EMC Elastic Cloud Storage). The experimental evaluation of our implementation confirms the ability of the distributed shuffle index to protect access confidentiality, and the limited performance overhead to obtain such protection guarantees. The next chapter will analyze the deployment of the tool to the high-performance cloud infrastructure made available to the project by partner EMC.

The remainder of this chapter is organized as follows. Section 1.2 illustrates the distributed shuffle index structure and the protection techniques it adopts to provide access confidentiality. Section 1.3 describes the implementation of the distributed shuffle index for its integration with industrial cloud providers. Section 1.4 presents the results of our experimental evaluation. Section 1.5 discusses related work.

1.2 Basic Concepts

The shuffle index [DFP+15] is an indexing structure that enables efficient key-based data retrieval, while guaranteeing content, access, and pattern confidentiality. For outsourcing, we assume data to be indexed over a candidate key K. The outsourced data collection is a set of pairs $\langle key, value \rangle$, with *key* the value of candidate K, and *value* the resource associated with the candidate key value.

A distributed shuffle index [DFP+17] is an *unchained B+-tree* (i.e., a $B+$-tree with no link between leaves) with fan-out F defined over candidate key K, storing the resources in its leaves, where internal nodes and leaves are distributed among three cloud providers. Each node stores at most $F-1$ key values, whereas the root has three times the capacity of internal nodes because it is split into three nodes, each allocated at a different cloud provider. Each node is associated with a logical identifier, and it is allocated at one of the cloud providers. Logical identifiers are used as pointers to children in internal nodes. Note that logical identifiers do not reflect the order among keys, and the allocation of nodes to cloud providers does not depend on the tree topology (i.e., children and/or siblings of a node can be allocated to a different cloud provider). However, the structure is evenly distributed among the three cloud providers, both globally and among the children of each node (i.e., each cloud provider stores one third of the nodes in the shuffle index and nearly one third of the children of each node). The content of a node is protected by encrypting it together with its identifier and a randomly generated nonce; the encrypted node is then stored at one cloud provider. To guarantee authenticity and integrity of the content of nodes and of the whole index structure, each encrypted node is concatenated with the result of a HMAC function applied to the encrypted node and its identifier. Figure 1.1 illustrates an example of a shuffle index where we report logical identifiers on the top of each node, and we denote nodes allocated at different cloud providers with different colors: blue for P_B (darker color in b/w printouts); green for P_G (medium color in b/w printouts); and yellow for P_Y (lighter color in b/w printouts). For simplicity, logical identifiers start with a letter denoting the cloud provider where the corresponding block is stored (B for P_B, G for P_G, and Y for P_Y), and their first digit denotes the level in the tree. Also, we denote the content of each node with a label.

To search for a key, a path in the tree is visited starting from the root and following, at each visited node, the pointer to the child along the path to the target, until a leaf is reached. Since nodes are stored in encrypted form, such a process requires an interaction between the client and the cloud provider(s) for each level of the tree. Even if encryption protects data in storage, it is not sufficient to provide access and pattern confidentiality. In fact, by observing a long enough se-

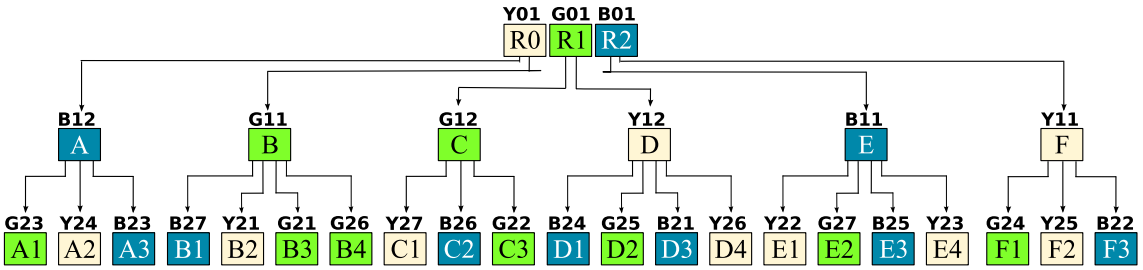


Figure 1.1: An example of shuffle index distributed at cloud providers (color coded)

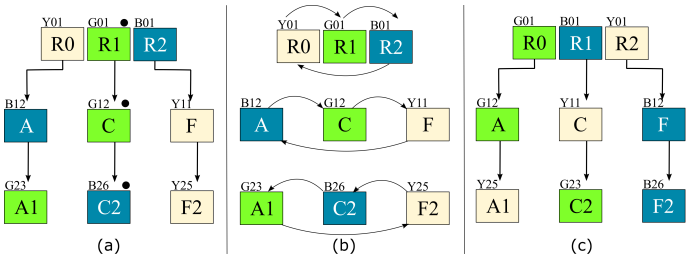


Figure 1.2: An example of search for $C2$ on the index in Figure 1.1 (a), swapping between accessed nodes (b), and structure of the paths after the access (c)

quence of accesses, the cloud providers could reconstruct the topology of the tree, identify repeated accesses, and possibly infer sensitive data content. To protect access and pattern confidentiality, the distributed shuffle index combines the following protection techniques in access execution.

- *Distributed covers*: at each level, download a block from each of the cloud providers. One of the accessed blocks is along the path to the target, the other two ensure uniform visibility at all the cloud providers.

- *Swapping*: at each level, the accessed nodes are re-allocated to a different block at a different cloud provider. To prevent cloud providers from tracking swap operations, nodes are re-encrypted with a different random nonce at each access. Swapping destroys the otherwise static relationship between the node content and the physical block where it is stored, preventing the cloud provider from accumulating information on frequency of accesses.

Being iterative, the search process discloses the level of each node in the tree, which is however not considered sensitive.

To illustrate, consider a search for value $C2$ in the distributed shuffle index in Figure 1.1. Figure 1.2(a) illustrates the accessed nodes (• denotes the target path). It is immediate to see that we download a node from each cloud provider at each level. Figure 1.2(b) illustrates swapping and Figure 1.2(c) represents the status of accessed nodes after the search.

1.3 Distributed Index Implementation

To demonstrate its practical applicability in a cloud scenario, we implemented the distributed shuffle index. Our implementation supports the creation, storage, and access to a distributed shuffle index stored at industrial cloud providers: *Amazon S3*, *OpenStack Swift*, *Google Cloud Storage*, and *EMC Elastic Cloud Storage* (ECS). The prototype has been realized in Python due to the wide availability of production-ready libraries that support the interaction with multiple cloud

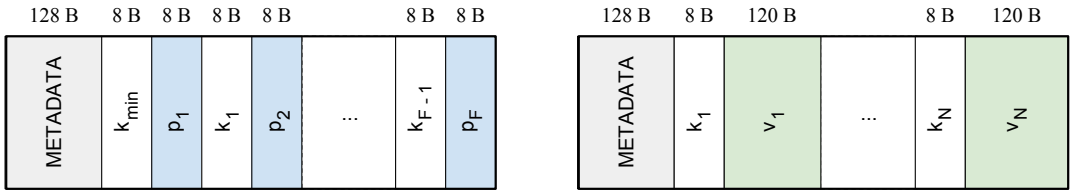


Figure 1.3: Structure of internal (a) and leaf (b) nodes

providers. An obstacle to the use of Python is the performance penalty, but this is not a critical aspect since, for the shuffle index, the bottleneck is represented by network latency (Section 1.4). In the remainder of this section, we first illustrate the data structures used to represent the nodes in the shuffle index, and then discuss the architecture and implementation of our prototype.

1.3.1 Node Data Structure

Given a collection of pairs of the form $\langle key, value \rangle$, we represent *key* as a 64-bit integer and *value* as a string. The use of an integer to represent *key* is a common choice for indexes, since it offers compactness and simplifies the verification of the relative order between elements, without the need of conversions. A 64-bit domain can support large data collections. Strings are the most common type in data management applications, and their use to represent *value* permits to easily manage resources of other data types. Since all the nodes in the shuffle index must have the same size (otherwise the cloud provider could easily identify each of them and hence track swap operations), we need to fix also the length of strings representing resources. We then represent *value* as a 120-byte string. This choice permits to represent each $\langle key, value \rangle$ pair in 128 bytes, which simplifies their storage and retrieval. Since resources do not have a fixed size, we manage larger resources by splitting them among multiple $\langle key, value \rangle$ pairs, and pad smaller resources. To support non-ASCII characters, we use UTF-16 encoding, in which each character is represented with 2 bytes, to represent our resources.

Besides storing keys and resources or pointers to children, each node should store additional metadata, that is, the node identifier, and the HMAC of the encrypted node content [DFP+15]. We use a 64-bit unsigned integer to represent the node identifier and reserve the first 128 bytes of each node to store its metadata.

Figure 1.3 illustrates the structure of internal and leaf nodes. Each internal node stores, besides its metadata, up to $F-1$ ordered keys $k_1, \ldots, k_q$ and $q+1$ pointers to children (nodes with fewer keys/pointers are padded). For efficiency reasons, each internal node also stores an additional key k_{min} representing the minimum key value in the subtree rooted at the node (Figure 1.3(a)). We consider a memory page of 4096 bytes and, since keys and node identifiers occupy 8 bytes, the fan-out of our distributed shuffle index is $F = (4096-128)/16 = 248$. Each leaf node can instead store up to $(4096-128)/128 = 31$ $\langle key, value \rangle$ pairs (Figure 1.3(b)). Leaves with fewer $\langle key, value \rangle$ pairs are padded. A distributed shuffle index with height 3 can then accommodate a dataset of ~685 MB. Our choice of using 128 bytes for each $\langle key, value \rangle$ pair and for metadata permits to store an integer number of $\langle key, value \rangle$ pairs in a memory page and guarantees that they are 64-bit aligned, which simplifies extraction and processing.

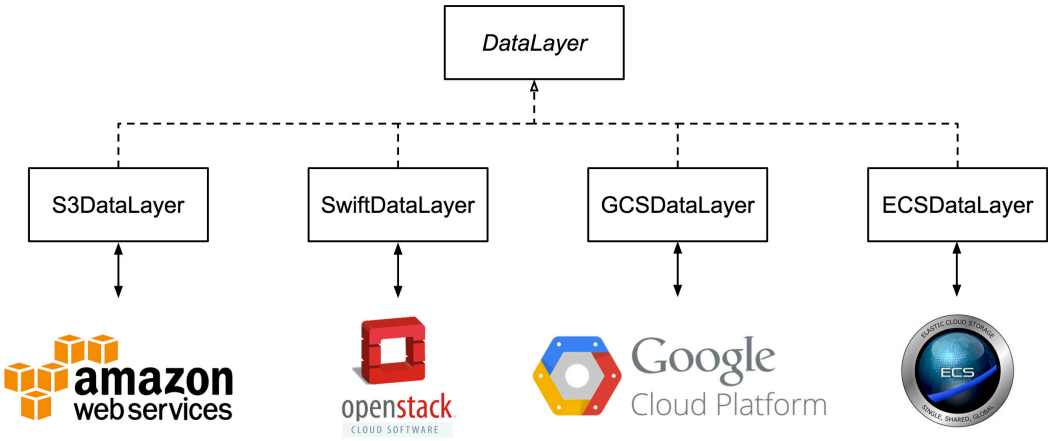


Figure 1.4: Diagram of the *DataLayer* classes of the cloud providers supported

1.3.2 Cloud DataLayer

The architecture of our prototype has to support the interaction with different cloud providers. It is a natural design choice to identify a high-level interface, *DataLayer*, that offers an abstract representation of a cloud storage service. This permits to isolate all aspects associated with the specific protocol used for the interaction with the storage service. The implementation of this abstract layer is simplified by the fact that the storage of a shuffle index needs the basic *put/get* primitives offered by cloud providers. To provide access confidentiality, the client is responsible for all computations, while the cloud provider acts only as storage server. We can use widely available object storage services. In this case, each object implements a node where the physical block identifier is represented through the object name and the encrypted node content is stored as the object content.

There are multiple public and private cloud providers that offer object storage services. Our implementation supports Amazon S3, OpenStack Swift, Google Cloud Storage, and EMC Elastic Cloud Storage (ECS). Each cloud provider exposes specific REST APIs, therefore we implemented a different *DataLayer* for each cloud provider (see Figure 1.4), as illustrated in the following.

- *S3DataLayer* permits the connection to Amazon Simple Storage Service (S3) using the *boto* library (`https://github.com/boto/boto`).
- *SwiftDataLayer* permits the connection to an OpenStack Swift storage service using the *python-swiftclient* library (`https://github.com/openstack/python-swiftclient`). This class can be used with both OpenStack-based public clouds and privately-hosted OpenStack instances.
- *GCSDataLayer* permits the connection to Google Cloud Storage using the *gcloud-python* library (`https://github.com/GoogleCloudPlatform/google-cloud-python`).
- *ECSDataLayer* permits the connection to EMC ECS. Two different APIs are provided by EMC ECS to interact with their storage service, based on S3 and Swift. We therefore specialized this *DataLayer* into two classes: *ECSS3DataLayer* and *ECSSwiftDataLayer*.

Note that our architecture can integrate additional cloud providers or even custom servers by simply adding a class that inherits the *DataLayer* interface, and implements *get* and *put* methods and

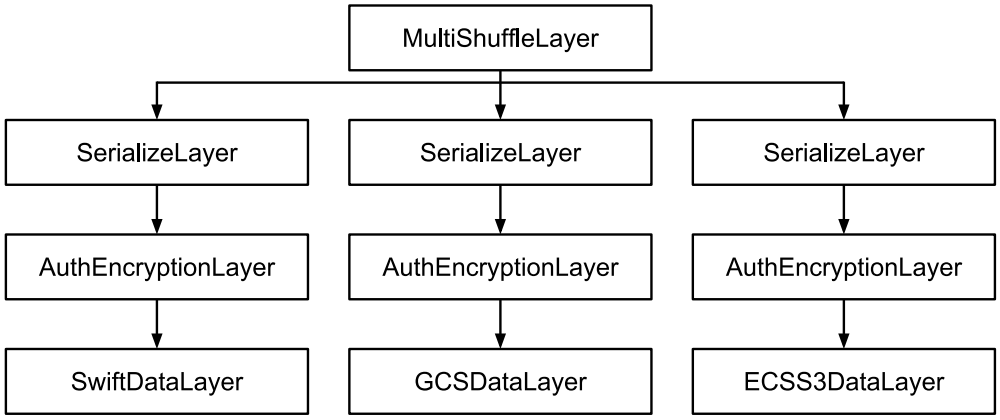


Figure 1.5: An example of instantiated *DataLayer* stack

the constructor to initialize the connection.

Besides the *DataLayer* classes illustrated above (which are in charge of realizing *get* and *put* primitives), two additional layers are necessary in the implementation of our prototype: *SerializeLayer* and *AuthEncryptionLayer* layers. The *SerializeLayer* is in charge of converting an instance of the object representing a node into a fixed-length string buffer, according to the node structure illustrated in Section 1.3.1. The *AuthEncryptionLayer* encrypts this buffer and adds the HMAC to the metadata.

Finally, the *MultiShuffleLayer* implements the distributed shuffle index logic and is the only layer with which the user needs to interact. It is independent from the chosen cloud providers. Its constructor receives three *DataLayer* instances as input, one for each of the cloud providers chosen for storing the distributed shuffle index. Figure 1.5 illustrates an example of an instantiated class stack, assuming to use OpenStack Swift, Google Cloud Storage, and EMC ECS for the distributed shuffle index.

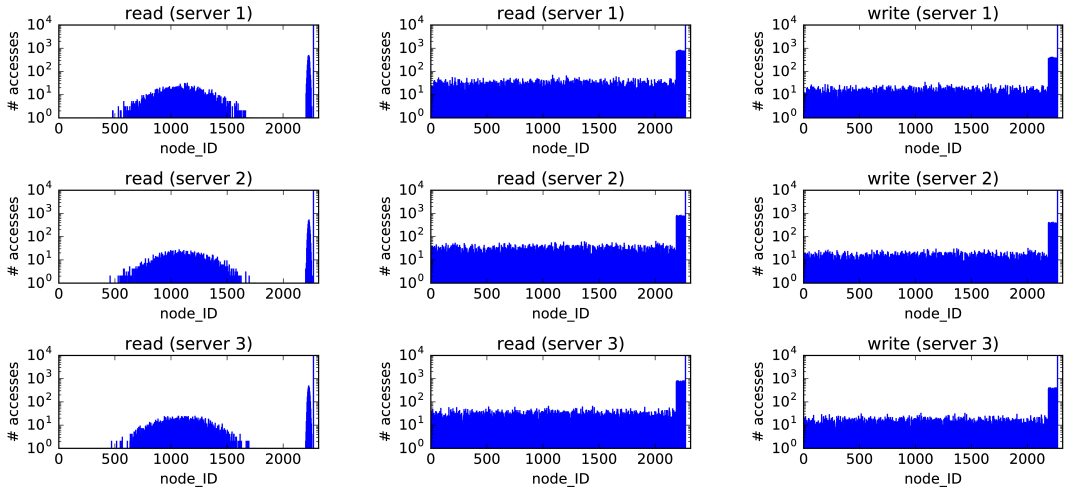
1.3.3 Configuration File

Our prototype receives a configuration file as input. The configuration file provides the parameters needed to initialize the connection (e.g., API keys, credentials) to the three cloud providers chosen by the user to store the distributed shuffle index (each using any of the four cloud APIs supported). The file has one section for each selected cloud provider. For each section, a *DataLayer* is instantiated. Each section has the following mandatory parameters:

- *type*: type of the *DataLayer* (i.e., *S3DataLayer*, *SwiftDataLayer*, *GCSDataLayer*, or *ECSDataLayer*);
- *rootnode*: identifier of the root node stored at the cloud provider;
- *configuration*: dictionary of parameters needed to initialize the connection to the cloud provider.

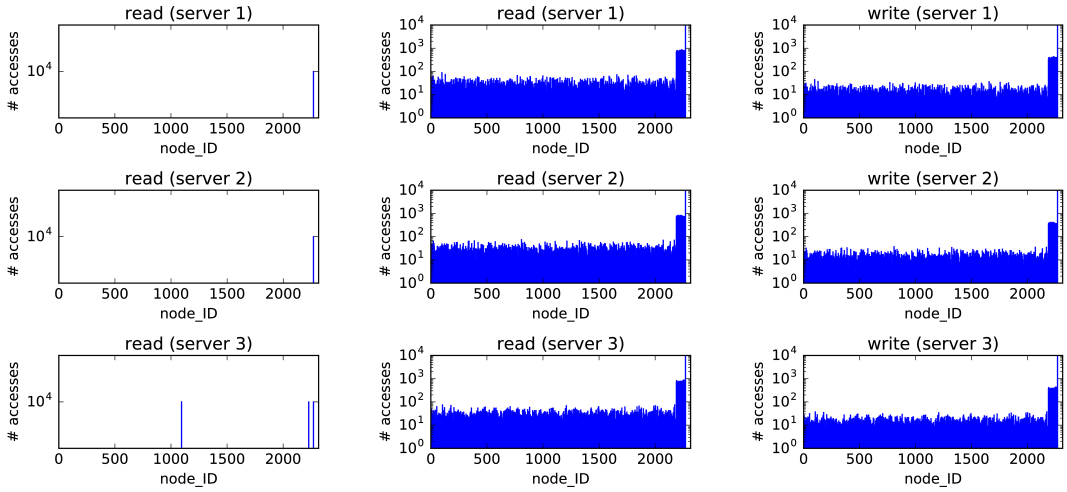
1.4 Experimental Results

To evaluate the confidentiality guarantees and the impact on performance of the protection techniques used by a distributed shuffle index, we performed a series of experiments using our im-



(a) plain encrypted index (b) shuffle index - read (c) shuffle index - write

Figure 1.6: Frequency distribution of accesses to blocks with the Gaussian access pattern



(a) plain encrypted index (b) shuffle index - read (c) shuffle index - write

Figure 1.7: Frequency distribution of accesses to blocks with the single-node access pattern

plementation. To analyze confidentiality guarantees, we compared the frequency of read/write accesses to physical blocks obtained when relying on our distributed shuffle index and when using a plain encrypted index with the same static structure (i.e., an encrypted $B+$-tree). The adoption of a plain encrypted index still requires the client to visit the nodes in the tree level by level, but it does not use distributed covers and swapping. To analyze performance, we compared the access times obtained with a plain encrypted index and with our distributed shuffle index, to measure the performance overhead caused by the adoption of distributed covers and swapping.

1.4.1 Access Pattern Confidentiality

The data structure used for this experiment is a 3-level $B+$-tree with fan-out $F = 248$. To compare the frequency distribution of accesses to blocks obtained with a shuffle index and with a plain encrypted index, we performed 10^4 accesses to the index structures, analyzing two different access

	Plain encrypted index	*Distributed shuffle index*
Local	$0.06169s$	$0.17074s$
(*OpenStack Swift*)	$\sigma = 0.00947s$	$\sigma = 0.01840s$
Remote	$0.56777s$	$1.07609s$
(*Amazon S3*)	$\sigma = 0.25588s$	$\sigma = 0.42817s$

Figure 1.8: Access times and their standard deviation σ

patterns: the Gaussian access pattern, generating a Gaussian distribution of frequency of accesses to keys, and the single-node access pattern, where all the accesses search for the same key.

The experiment was performed on a private cloud with three cloud providers based on OpenStack Swift, since the results do not depend on the chosen cloud providers. The use of a private cloud permits an in-depth analysis of the accessed data both at the client and at the cloud provider side, since all the log files are available.

Gaussian access pattern. We considered first a sequence of accesses to keys that follows a Gaussian frequency distribution. Figure 1.6 illustrates the frequency distribution of read accesses to the blocks stored at one of the three cloud providers (the results obtained for the other two cloud providers are similar) in the plain encrypted index (a) and in the distributed shuffle index (b). In the figure, we ordered the nodes in the structure by decreasing level in the tree (i.e., first leaves, then internal nodes, and finally the root). Note that the frequency distribution of read accesses to the (leaf) blocks in the plain encrypted index leaks the Gaussian distribution of accesses to keys. An observer knowing the frequency with which keys are accessed can then easily determine which leaf block stores which key. On the contrary, the frequency distribution of read accesses to the blocks in each level of the distributed shuffle index is almost flat (Figure 1.6(b)): the observer can only distinguish between leaves, internal nodes, and the root. As already pointed out in Section 1.2, the iterative process between the cloud providers and the client discloses the level of nodes.

Single-node access pattern. We then considered a sequence of accesses where all the searches aim at the same target key, and then visit the same path in the $B+$-tree. Figure 1.7 illustrates the frequency distribution of read accesses to the blocks stored at one of the three cloud providers in the plain encrypted index (a) and in the distributed shuffle index (b). Also in this scenario, the adoption of a plain encrypted index leaks to an observer the pattern of accesses: the spikes in Figure 1.7(a) correspond to the nodes along the path to the target. On the contrary, the frequency distribution of read accesses to the blocks in each level of the distributed shuffle index is almost flat. The frequency distribution of read accesses in Figure 1.7(b) is similar to the one in Figure 1.6(b), further demonstrating the ability of the distributed shuffle index to protect access confidentiality. This is also confirmed by the frequency distributions of write operations over the blocks in the distributed shuffle index, which are similar for the two patterns of accesses as illustrated in Figure 1.6(c) and Figure 1.7(c) (note that we did not evaluate the frequency of write operations for the plain encrypted index, because searches do not require any rewrite operation).

1.4.2 Performance

To analyze the performance of our distributed shuffle index, we considered two configurations: a private cloud based on OpenStack Swift (*local* configuration), and the public cloud infrastructure provided by Amazon S3 (*remote* configuration). In our experimental evaluation, we did not consider a specific configuration for EMC Elastic Cloud Storage and Google Cloud Storage since we

expect their behavior to be similar to our local and remote configurations, respectively. In fact, the most relevant factor influencing access times is network latency [DFP+15]. For our experiment, we used a client PC with Intel i7 3.4GHz processor, 16GB RAM, 240GB SSD running Ubuntu 16.04. The OpenStack Swift provider was running on a server PC with Intel i7 3.6GHz processor, 32GB RAM, 480GB SSD running Ubuntu 14.04.

For this set of experiments, we used a 2-level $B+$-tree with fan-out $F = 27$ and performed 100 accesses. Figure 1.8 compares the average access times obtained when adopting a plain encrypted index and our distributed shuffle index in the local and remote configurations. The figure also reports the standard deviation (σ). Note that the search for a value in the plain encrypted index implies one read access for each node along the path to the target. On the contrary, the search for a value in the distributed shuffle index requires one read and one write access for each node along the path to the target and for each of its distributed covers (i.e., it requires three read and three write accesses for each level of the shuffle index). We note however that an access to the distributed shuffle index is only double (not six times as one may expect) the time of an access to the plain encrypted index in the remote configuration. The reason is that we leverage parallel connections to the cloud providers to read and write the nodes, thus reducing the performance overhead necessary to obtain access confidentiality. As expected, access times are higher in the remote configuration than in the local configuration, because the network represents a bottleneck. It is interesting to note that access times in the remote configuration are one order of magnitude greater than the ones obtained in the local configuration, for both the plain encrypted index and the distributed shuffle index. Based on our results, we can conclude that the adoption of distributed covers and swapping protection techniques have a limited impact on access times. Hence, the distributed shuffle index represents a practical solution for providing access confidentiality.

1.5 Related Work

Moving data to the cloud requires the secure management and storage of data (e.g., [DFS12,For11, JPS12,WCRL12]) as well as the protection of accesses to data (e.g., [DFP+11,DFP+15,DFM+16, SvS+13,DvF+16]). Current approaches addressing the problem of protecting access confidentiality are based on Private Information Retrieval (PIR) or on dynamically allocated data structures that change physical data allocation at each access. PIR solutions do not protect content confidentiality and require high access times (e.g., [OS07]) even when operating in distributed scenarios where several non-communicating cloud providers keep a copy of the data (e.g., [CMS99]). Dynamic data structures rely on the Oblivious RAM (ORAM) structure (e.g., [SvS+13, DvF+16]) or on tree-based structures (e.g., [DFP+11, DFP+15, DFM+16]). These solutions assume that the data are stored at one cloud provider. Cloud-based solutions often provide data distribution services, which can be used to improve both performance (at the price of properly balancing workload distribution [GQZ+16]) and security. ORAM-based solutions have also been extended to operate in distributed scenarios to reduce communication costs for the client (e.g., [SS13]). These solutions, however, require that storage providers do not communicate or do not collude with each other to guarantee confidentiality. The shuffle index proposal has also been extended to operate in a distributed scenario characterized by three cloud providers [DFP+17]. The advantage of this solution is that it guarantees access confidentiality also in case of collusion among cloud providers. This deliverable demonstrates the practical applicability of the distributed shuffle index in real cloud environments, confirming its limited performance overhead. Recently, different proposals have considered the integration of protection techniques with real-world cloud infras-

tructures (e.g., [BDF+16]). These solutions mainly aim to protect content confidentiality to the cloud providers in outsourcing scenarios, but do not protect access confidentiality.

2. Industrial Platform Performance Evaluation

Chapter 1 provided an overview and experimental evaluation of the distributed shuffle index implementation across a number of public cloud providers. This chapter analyzes the implementation in an industrial testbed across three datacenters to provide closer to *real-world* insights into the impact of the distributed shuffle index on performance. These results independently confirm the minimal overhead reported in the previous chapter. The following section introduces the industrial testbed environment and the deployment of the shuffle index on it, while Section 2.2 presents the experimental results.

2.1 Industry Testbed

INFINITE (INternational Future INdustrial Internet TEstbed) is an "Internet of Things" (IoT) innovation platform built specifically for the development of Industrial IoT products and solutions across a wide and diverse range of industries and sectors. It is a strategic initiative led by Dell EMC, Vodafone Ireland and partners and it is the first-of-its kind in Europe. It was publicly launched in June 2015 in Dublin, Ireland and encompasses all the significant technologies, domains and platforms that will form the foundation of Industrial IoT solutions and deployments which include: advanced analytics, cloud platforms, security, networks, mobile M2M (Machine to Machine), wireless and sensors.

The aim of the project was to make a state of art secure virtual cloud and network infrastructure available that provides highly scalable bandwidth and capacity with analytics capabilities, spanning multiple domains (wireless, wireline, virtualized, physical) at scale. The resulting infrastructure represents a real world environment encompassing Cloud, Analytics, Security and Mobile M2M domains. This enables Dell EMC and partners to execute use cases that demonstrate the potential of innovative solutions and products.

2.1.1 INFINITE Architecture

The testbed is composed of a full mobile network (2G to LTE) covering the island of Ireland and a core cloud infrastructure with a distributed Data Center architecture spanning three geographically diverse sites in Cork City, Ireland, interconnected via a full-mesh of dark fiber pairs. The testbed's three site architecture is detailed in Figure 2.1. The three sites are:

- EMC datacenter which contains a VBlock [1] (configured as a VMWare cluster) and analytics platforms
- Vodafone datacenter which contains compute, storage and networking resources

[1] https://www.emc.com/en-us/converged-infrastructure/converged-systems.htm

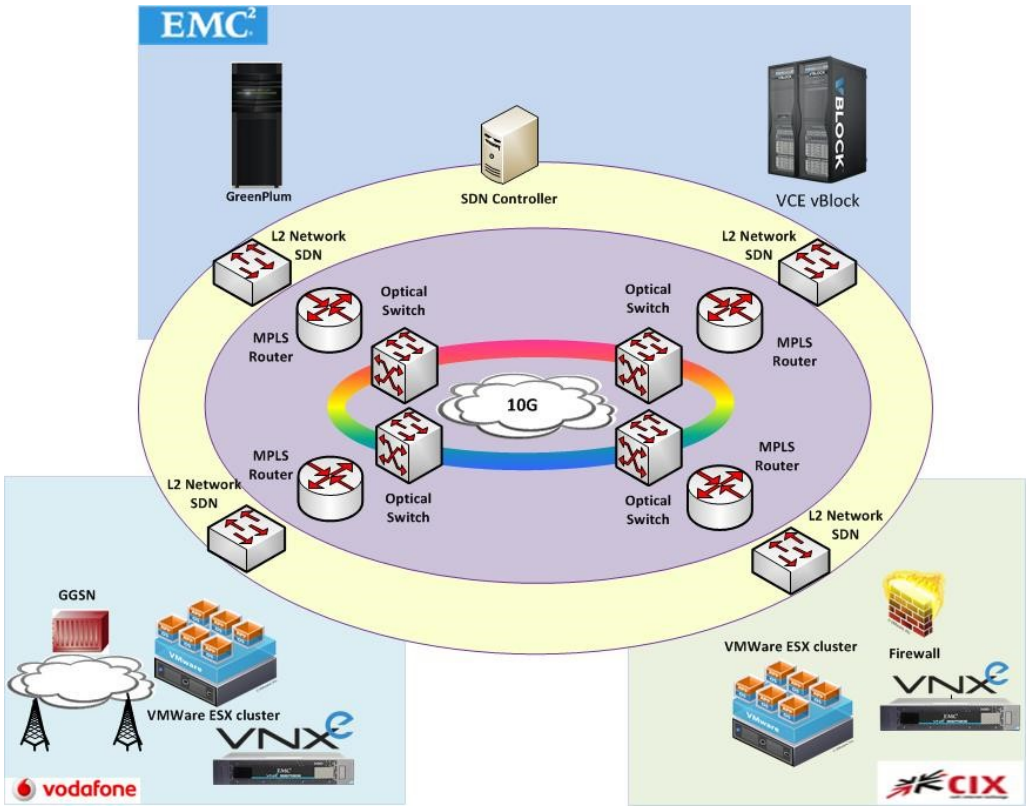


Figure 2.1: INFINITE testbed overview

- CIX datacenter which hosts compute, storage and networking resources. The main goal of this datacenter is to be an independent location where potential testbed participants can locate their platforms in a way that ensures isolation from the other testbed participants. It is also the single point of access to the Internet.

The core network is a MPLS dark fiber ring offering 10 Gbps capacity. Each site is connected to the dark fiber ring through a HP 5800 MPLS router and a DWDM Mux/Demux optical device. The core network extends a Layer 2 domain across the three sites by creating VPLS (virtual private LAN service) between the MPLS routers. MPLS technology enables the flexibility to add new sites in the future or change their connectivity.

2.1.2 Elastic Hybrid Cloud

Dell EMC Elastic Cloud Storage (ECS) is a turnkey software defined cloud-scale object storage platform. Traditional infrastructure often isolates data, making it difficult and costly to manage. This type of environment does not scale efficiently nor economically. As illustrated in Figure 2.2, ECS enables organizations to consolidate multiple storage systems and content archives into a single, globally accessible and efficient repository. For this reason ECS was selected in UC4 to provide *elastic* storage resources to the reference architecture.

When considering an implementation of the shuffle index, ECS was also identified as a suitable candidate, enabling access to the shuffle index via multiple APIs (SWIFT, S3). Furthermore the solution offered functionality comparable to cloud platforms already evaluated against in Section 1.4. As such ECS was deemed an appropriate tool for simulating multiple CSPs on the testbed environment.

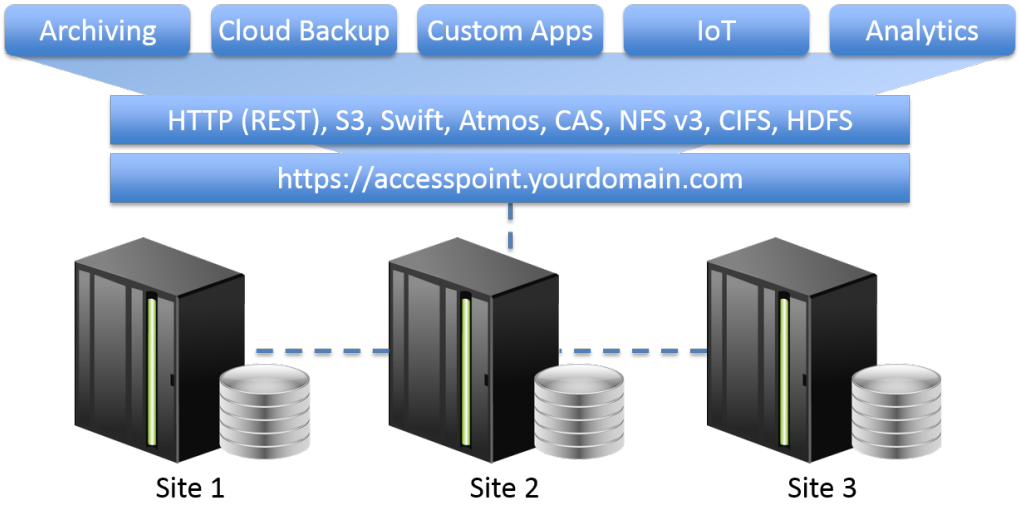


Figure 2.2: Dell EMC Elastic Cloud Storage (ECS)

2.1.3 Shuffle Index Deployment

The deployment for this experiment consisted of four virtual machines. The shuffle index was deployed on a virtual machine running Ubuntu 16.04, this VM was allocated 8 GB of RAM and 4 virtual CPU cores operating at 4.20 GHz and 60 GB of disk space. The software-only version of ECS was deployed to the testbed in three installations configured on CentOS 7 VMs. Each of these VMs was allocated 16 GB RAM, 4 virtual CPU cores operating at 4.20 GHz and 120 GB of disk space (100 GB data disk, 20 GB OS boot disk). Dell EMC Elastic Cloud Storage was set up as three single-node ECS configurations rather than one multi-node distributed ECS cluster; this was in order to satisfy the simulation of three separate and isolated CSPs which are not intended to be 'aware' of each other in the context of this experiment. Once operational, each single-node configuration of ECS was migrated to a different physical location of the testbed. The ECS nodes were accessed by the Shuffle Index via the S3 API.

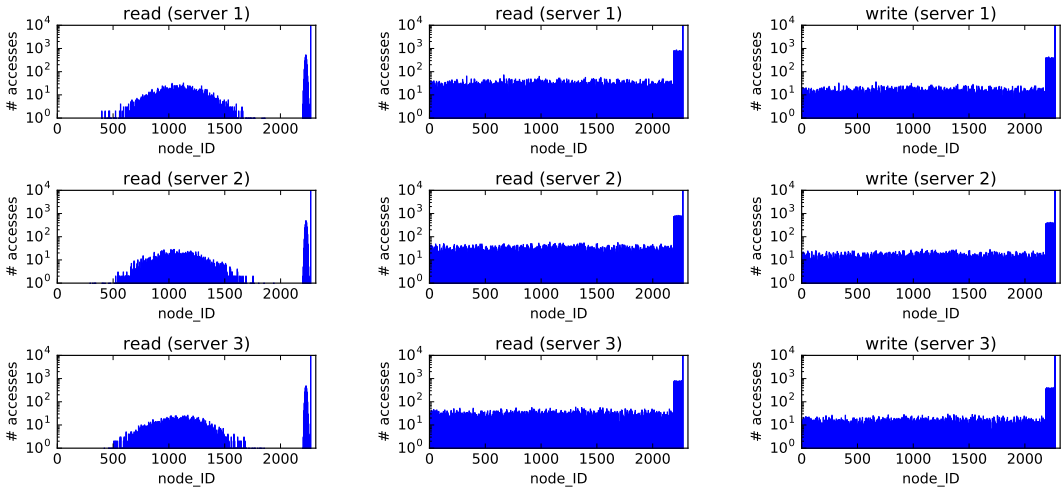
2.2 Experimental Results

The results presented in this section follow the experimental process established in Section 1.4, evaluating the confidentiality guarantees and the performance of the distributed shuffle index in the context of the deployment detailed in Section 2.1.3.

2.2.1 Access Pattern Confidentiality

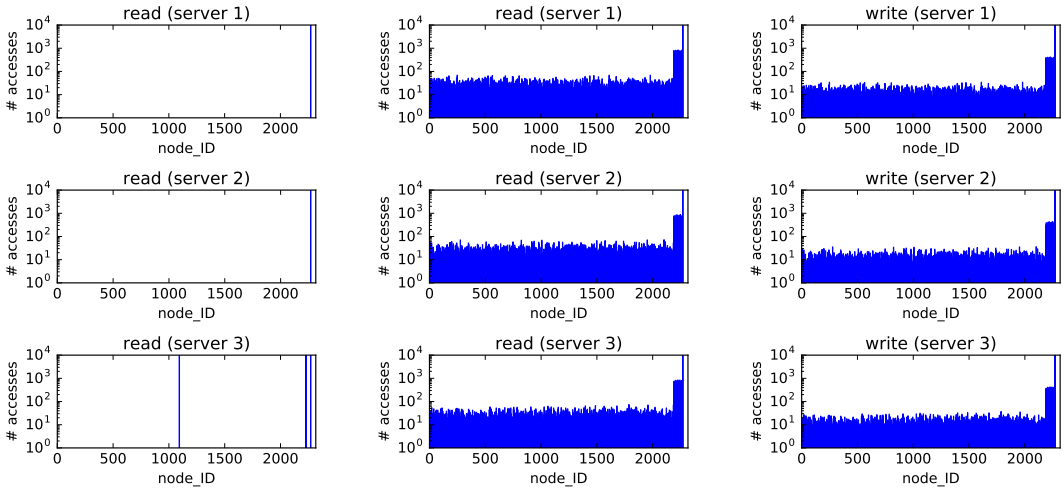
This experiment utilizes a B+-tree consisting of 2 levels with a fan-out F = 27. Similarly to the experiment described in Section 1.4 the shuffle index and a plain encrypted index were accessed 10^4 times using the Gaussian access pattern, and the single-node access pattern. The experiment was carried out using the Dell EMC Elastic Cloud Storage platform deployment distributed across the INIFINITE testbed described in Section 2.1.3.

Gaussian access pattern. As described in Section 1.4 the access to keys using this pattern is based on a Gaussian frequency distribution. It is evident when viewing Figure 2.3 that the results gathered are largely akin to those of Figure 1.6; the distributed environment did not have a



(a) plain encrypted index (b) shuffle index - read (c) shuffle index - write

Figure 2.3: Frequency distribution of accesses to blocks with the Gaussian access pattern



(a) plain encrypted index (b) shuffle index - read (c) shuffle index - write

Figure 2.4: Frequency distribution of accesses to blocks with the single-node access pattern

discernable impact on the Gaussian access pattern.

Single-node access pattern. This access pattern, as detailed in Section 1.4, aims all searches at one target key, and traverses the B+- tree using the same path. It is immediately apparent that the graphed data in Figure 2.4 essentially mirrors the data in Figure 1.7. As it was the case with the Gaussian access pattern, the single-node access pattern was not noticeably impacted by the distributed environment.

2.2.2 Performance

This performance analysis of the shuffle index utilizes a remote configuration for the ECS - INFINITE testbed deployment described in Section 2.1.3. The data structure utilized for this experiment is a 2 level B+-tree with fan-out F = 27 and 100 performed accesses. Figure 2.5 draws a comparison between the distributed shuffle index and a plain encrypted index parameters which

	Plain encrypted index	*Distributed shuffle index*
Remote (ECS)	0.04210 s σ = 0.01322	0.19674 s σ = 0.08075

Figure 2.5: ECS testbed access times and their standard deviation σ

were discussed in Section 1.4, denoting the average access time as well as standard deviation for each.

Operating in a physically distributed network commonly incurs some performance penalties stemming from factors such as network latency; this, however, is not the case in this experiment. Despite this experiment being conducted in a distributed environment, when the access times of Figure 2.5 are compared with the remote access times of Figure 1.8 those of the remote ECS configuration appear to have a discrepancy in that the aforementioned impact to performance has not manifested. The explanation for this is the 10 Gbps connection speeds provided by the INFINITE testbed; this enables the rapid transmission of data during shuffle operations, which effectively counteracts the potential negative impacts of the physical distribution of the experiment's components. To contextualise this information: when operating in an environment such as an industrial high-speed, closed area network the network latency bottleneck encountered by shuffle index is mitigated, making this an ideal application scenario. An additional observation taken from comparing the data in Figures 1.8 and 2.5 is that the mitigation of the network bottleneck resulted in an increased disparity between the access times and standard deviation of the plain encrypted index, and those of the shuffle index; though both indexes benefited from the testbed environment's high speed connection, the plain encrypted index benefited more so.

3. Conclusions

We presented the architectural and design issues addressed for the implementation of the distributed shuffle index integrated with different cloud providers (i.e., Amazon S3, OpenStack Swift, Google Cloud Storage, and EMC Elastic Cloud Storage). A series of experiments has been run comparing the performance obtained with a local infrastructure and using the services of common commercial cloud providers. Another series of experiments has used the high-performance cooperative cloud infrastructure supported by EMC. These experiments demonstrate the availability of effective and practical protection solutions that can easily be integrated with real-world cloud infrastructures. Our experimental evaluation demonstrates the effectiveness of the distributed shuffle index in providing access confidentiality, and its limited performance overhead.

Bibliography

[BDF+16] E. Bacis, S. De Capitani di Vimercati, S. Foresti, D. Guttadoro, S. Paraboschi, M. Rosa, P. Samarati, and A. Saullo. Managing data sharing in OpenStack Swift with over-encryption. In *Proc. of WISCS*, Vienna, Austria, October 2016.

[CMS99] C. Cachin, S. Micali, and M. Stadler. Computationally private information retrieval with polylogarithmic communication. In *Proc. of EUROCRYPT*, Prague, Czech Republic, May 1999.

[DFM+16] S. De Capitani di Vimercati, S. Foresti, R. Moretti, S. Paraboschi, G. Pelosi, and P. Samarati. A dynamic tree-based data structure for access privacy in the cloud. In *Proc. of CloudCom*, Luxembourg, December 2016.

[DFP+11] S. De Capitani di Vimercati, S. Foresti, S. Paraboschi, G. Pelosi, and P. Samarati. Efficient and private access to outsourced data. In *Proc. of ICDCS*, Minneapolis, MN, June 2011.

[DFP+15] S. De Capitani di Vimercati, S. Foresti, S. Paraboschi, G. Pelosi, and P. Samarati. Shuffle index: Efficient and private access to outsourced data. *ACM TOS*, 11(4):1–55, October 2015. Article 19.

[DFP+17] S. De Capitani di Vimercati, S. Foresti, S. Paraboschi, G. Pelosi, and P. Samarati. Three-server swapping for access confidentiality. *IEEE TCC*, 2017. pre-print.

[DFS12] S. De Capitani di Vimercati, S. Foresti, and P. Samarati. Managing and accessing data in the cloud: Privacy risks and approaches. In *Proc. of CRiSIS*, Cork, Ireland, October 2012.

[DvF+16] S. Devadas, M. van Dijk, C. W. Fletcher, L. Ren, E. Shi, and D. Wichs. Onion ORAM: A constant bandwidth blowup oblivious RAM. In *Proc. of TCC*, Tel Aviv, Israel, January 2016.

[For11] S. Foresti. *Preserving Privacy in Data Outsourcing*. Springer, 2011.

[GQZ+16] K. Gai, M. Qiu, H. Zhao, L. Tao, and Ziliang Zong. Dynamic energy-aware cloudlet-based mobile cloud computing model for green computing. *Journal of Network and Computer Applications*, 59:46–54, January 2016.

[IKK14] M.S. Islam, M. Kuzu, and M. Kantarcioglu. Inference attack against encrypted range queries on outsourced databases. In *Proc. of CODASPY*, San Antonio, TX, March 2014.

[JPS12] R. Jhawar, V. Piuri, and P. Samarati. Supporting security requirements for resource management in cloud computing. In *Proc. of CSE*, Paphos, Cyprus, December 2012.

[OS07] R. Ostrovsky and W. E. Skeith III. A survey of single-database private information retrieval: Techniques and applications. In *Proc. of PKC*, Beijing, China, April 2007.

[PFL15] S. Paraboschi, S. Foresti, and G Livraga. Report on data protection techniques. Deliverable D2.1, ESCUDO-CLOUD, December 2015.

[SS13] E. Stefanov and E. Shi. Multi-cloud oblivious storage. In *Proc. of CCS*, Berlin, Germany, November 2013.

[SvS+13] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path ORAM: An extremely simple Oblivious RAM protocol. In *Proc. of CCS*, Berlin, Germany, November 2013.

[TTS+16] A. Tata, R. Trapero, N. Suri, S. Foresti, and A. Sajjad. First report on multi cloud and federated cloud. Deliverable D4.2, ESCUDO-CLOUD, December 2016.

[WCRL12] C. Wang, N. Cao, K. Ren, and W. Lou. Enabling secure and efficient ranked keyword search over outsourced cloud data. *IEEE TPDS*, 23(8):1467–1479, August 2012.