Distributed Shuffle Index in the Cloud: Implementation and Evaluation

Enrico Bacis*, Sabrina De Capitani di Vimercati†, Sara Foresti†,
Stefano Paraboschi*, Marco Rosa*, Pierangela Samarati†
* *Università degli Studi di Bergamo, 24044 Dalmine - Italy Email:* firstname.lastname@*unibg.it*
†*Università degli Studi di Milano, 26013 Crema - Italy Email:* firstname.lastname@*unimi.it*

***Abstract*—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 paper, 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.**

***Keywords*-Distributed shuffle index, access confidentiality, OpenStack, Amazon S3, Google Cloud Storage, EMC ECS.**

I. Introduction

Moving data to the cloud provides unprecedented 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 more under the direct control of their owner. The problem of protecting data confidentiality has been widely addressed by the research community (e.g., [1]). Protecting data confidentiality may however 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, with high probability, 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., [2]). 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 [3]. The shuffle index can also operate in a distributed scenario with even higher protection guarantees [4]. 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.

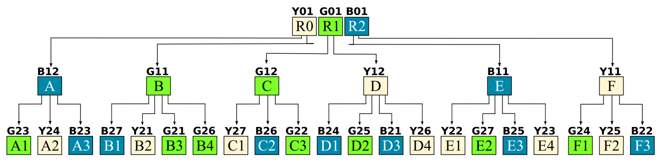
In this paper, we study the design and architectural issues to be addressed to implement the distributed shuffle index and to integrate it with real-world cloud providers (i.e., Amazon S3, OpenStack Swift, Google Cloud Storage, and EMC Elastic Cloud Storage). The experimental evaluation on our implementation confirms the ability of the distributed shuffle index to protect access confidentiality, and the limited performance overhead to obtain such protection guarantees.

The remainder of this paper is organized as follows. Section II illustrates the distributed shuffle index structure and the protection techniques it adopts to provide access confidentiality. Section III describes the implementation of the distributed shuffle index for its integration with real-world cloud providers. Section IV presents the results of our experimental evaluation. Section V discusses related works. Finally, Section VI concludes the paper.

II. Basic Concepts

The shuffle index [3] 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 [4] 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 but the root, which has three times the capacity of internal nodes because it is split in 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



Y01 G01 B01
R0 R1 R2
B12 G11 G12 Y12 B11 Y11
A B C D E F
G23 Y24 B23 B27 Y21 G21 G26 Y27 B26 G22 B24 G25 B21 Y26 Y22 G27 B25 Y23 G24 Y25 B22
A1 A2 A3 B1 B2 B3 B4 C1 C2 C3 D1 D2 D3 D4 E1 E2 E3 E4 F1 F2 F3

Figure 1. An example of shuffle index distributed at cloud providers (color coded)

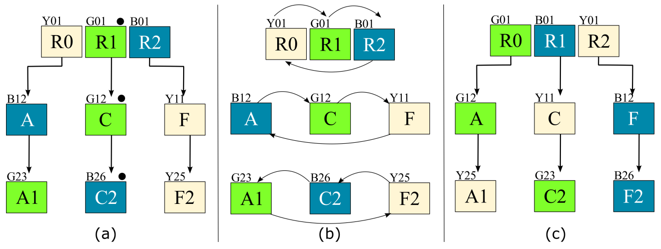


Figure 2. An example of search for $C2$ on the index in Figure 1 (a), swapping between accessed nodes (b), and structure of the paths after the access (c)

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 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 sequence 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. Figure 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 2(b) illustrates swapping and Figure 2(c) represents the status of accessed nodes after the search.

III. 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 real 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 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 IV). 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.

A. Node Data Structure

Given a collection of pairs of the form ⟨*key*, *value*⟩, we represent *key* as 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

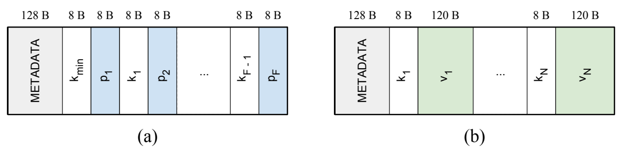


Figure 3. Structure of internal (a) and leaf (b) nodes

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 ⟨*key*, *value*⟩ 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 ⟨*key*, *value*⟩ 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 (i.e., *value* can accommodate 60 characters).

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 [3]. 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 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 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 $(4096-128)/128 = 31$ ⟨*key*, *value*⟩ pairs (Figure 3(b)). Leaves with fewer ⟨*key*, *value*⟩ 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 ⟨*key*, *value*⟩ pair and for metadata permits to store an integer number of ⟨*key*, *value*⟩ pairs in a memory page and guarantees that they are 64-bit aligned, which simplifies extraction and processing.

B. *Cloud DataLayer*

The architecture of our prototype has to support the interaction with distinct 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.

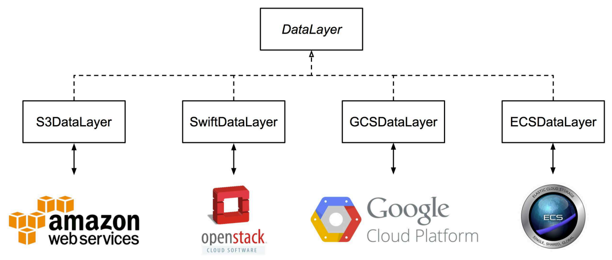


Figure 4. Diagram of the *DataLayer* classes of the cloud providers supported

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 the 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 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. EMC ECS provides two different APIs 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 the constructor to initialize the connection.

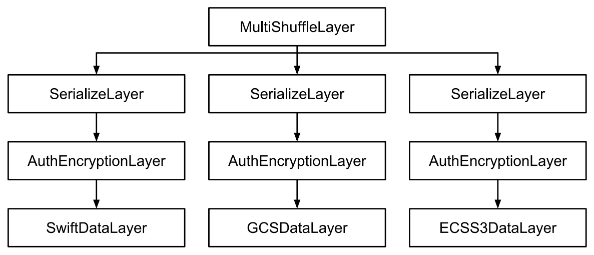


Figure 5. An example of instantiated *DataLayer* stack

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 III-A. 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 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.

C. 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. 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.

IV. 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 implementation. 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.

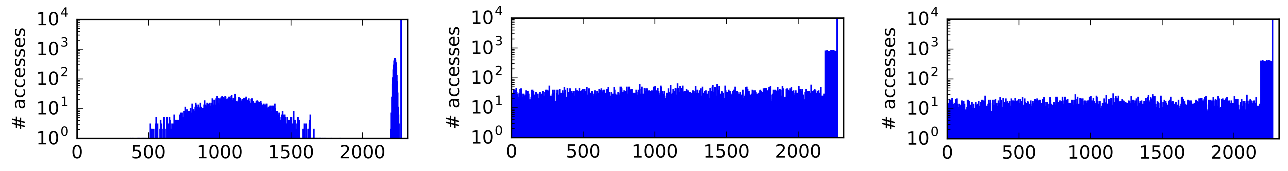
A. Access Pattern Confidentiality

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 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.

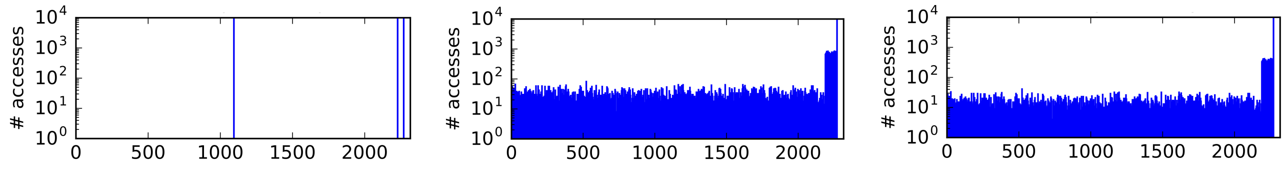
Gaussian access pattern. We considered first a sequence of accesses to keys that follows a Gaussian frequency distribution. Figure 6 illustrates the frequency distribution of read accesses to the blocks stored at one of the three cloud providers (the results obtained for the other two cloud providers 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 6(b)): the observer can only distinguish between leaves, internal nodes, and the root. As already pointed out in Section II, 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 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



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

Figure 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 7. Frequency distribution of accesses to blocks with the single-node access pattern

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

Figure 8. Access times and their standard deviation σ

Figure 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 7(b) is similar to the one in Figure 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 6(c) and Figure 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).

B. 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 [3]. 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 we performed 100 accesses over it. Figure 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 one time (not five as one would expect) slower than 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.

V. Related Work

Moving data to the cloud requires the secure management and storage of data (e.g., [1], [5]–[7]) as well as the protection of accesses to data (e.g., [3], [8]–[11]). 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., [12]) even when operating in distributed scenarios where several non-communicating cloud providers keep a copy of the data (e.g., [13]). Dynamic data structures rely on the Oblivious RAM (ORAM) structure (e.g., [10], [11]) or on tree-based structures (e.g., [3], [8], [9]). 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 [14]) and security. ORAM-based solutions have also been extended to operate in distributed scenarios to reduce communication costs for the client (e.g., [15]). 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 [4], [16]. The advantage of this solution is that it guarantees access confidentiality also in case of collusion among cloud providers. This paper 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 infrastructures (e.g., [17]). These solutions mainly aim to protect content confidentiality to the cloud providers in outsourcing scenario, but do not protect access confidentiality.

VI. 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). Our prototype demonstrates 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.

Acknowledgment

This work was supported by the EC within the H2020 under grant agreement 644579 (ESCUDO-CLOUD) and within the FP7 under grant agreement 312797 (ABC4EU).

References

[1] 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.

[2] M. Islam, M. Kuzu, and M. Kantarcioglu, “Inference attack against encrypted range queries on outsourced databases,” in *Proc. of CODASPY*, San Antonio, TX, March 2014.

[3] 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*, vol. 11, no. 4, pp. 1–55, October 2015, article 19.

[4] 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.

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

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

[7] C. Wang, N. Cao, K. Ren, and W. Lou, “Enabling secure and efficient ranked keyword search over outsourced cloud data,” *IEEE TPDS*, vol. 23, no. 8, pp. 1467–1479, August 2012.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

[14] K. Gai, M. Qiu, H. Zhao, L. Tao, and Z. Zong, “Dynamic energy-aware cloudlet-based mobile cloud computing model for green computing,” *Journal of Network and Computer Applications*, vol. 59, pp. 46–54, January 2016.

[15] E. Stefanov and E. Shi, “Multi-cloud oblivious storage,” in *Proc. of CCS*, Berlin, Germany, November 2013.

[16] S. De Capitani di Vimercati, S. Foresti, S. Paraboschi, G. Pelosi, and P. Samarati, “Distributed shuffling for preserving access confidentiality,” in *Proc. of ESORICS*, Egham, UK, September 2013.

[17] 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.