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

D3.3
Report on Techniques for Selective and Secure Data Sharing

Editors: Sabrina De Capitani di Vimercati (UNIMI)
Sara Foresti (UNIMI)
Reviewers: David Bowden (EMC)
Fadi El-Moussa (BT)

Abstract

Users moving their data to the cloud are often interested in relying on the services offered by cloud providers for sharing their data with others. To fully benefit from the services offered by cloud providers for enabling collaborative, possibly concurrent, access to data, it is however necessary to provide users with solutions that ensure the confidentiality and integrity of the data and of accesses, while enabling efficient query evaluation. The presence of different users with different access privileges, and hence views over the data, requires attention in the definition of protection techniques. The techniques for confidentiality and integrity studied in WP2 need then to be revised to operate in a multi-user scenario. In this deliverable, we build on the techniques analyzed in WP2 and propose solutions aimed to: *i)* protect data and access confidentiality; *ii)* support efficient query evaluation over encrypted data; and *iii)* guarantee integrity and consistency of object stores.

Type	Identifier	Dissemination	Date
Deliverable	D3.3	Public	2016.12.30

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

Versions

Version	Date	Description
0.1	2016.11.30	Initial Release
0.2	2016.12.14	Second Release
1.0	2016.12.30	Final Release

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)
Chapter 1: Access Control for the Shuffle Index	Sabrina De Capitani di Vimercati (UNIMI), Sara Foresti (UNIMI), Stefano Paraboschi (UNIBG), Pierangela Samarati (UNIMI)
Chapter 2: Selective Data Sharing via Encrypted Query Processing	Daniel Bernau (SAP), Andreas Fischer (SAP), Anselme Kemgne Tueno (SAP)
Chapter 3: Integrity and Consistency for Cloud Object Storage	Marcus Brandenburger (IBM), Christian Cachin (IBM)
Chapter 4: Conclusions	Sabrina De Capitani di Vimercati (UNIMI), Sara Foresti (UNIMI)

Contents

List of Figures

List of Tables

Executive Summary

The possibility to share data with a community of users represents one of the reasons why users increasingly move their data to the cloud. In fact, cloud providers offer services for easily sharing data with others, without the need for the data owner to be online. The presence of multiple users (possibly characterized by different access privileges) however leaves different issues open. The solutions proposed in WP2 to protect data and access confidentiality and to guarantee their integrity then need to be revised, to take into proper account the presence of multiple users, who may be authorized to see only a portion of the data (and of query results). This deliverable aims at providing techniques for guaranteeing confidentiality and integrity in a multi-user scenario, while supporting efficient access to the data in the cloud.

The first goal of this deliverable is to guarantee access confidentiality in presence of multiple users with different access privileges. To this aim, this deliverable presents an extension of the shuffle index structure analyzed in WP2 for supporting access control enforcement. The proposed approach is based on the adoption of selective encryption on the tuples stored in the leaves of the shuffle index, and on the definition of two shuffle index structures (a primary and a secondary index) to avoid the disclosure of sensitive information about accesses to both the cloud provider and to authorized users.

The second goal of this deliverable is to realize secure execution of queries over sensitive, access restricted data on an outsourced database. Therefore, this deliverable reports on the design and evaluation of ENKI, a prototype for selective data sharing between multiple users realized on SAP HANA. It applies a newly introduced encryption scheme to execute the relational operations count distinct, set difference, and join while protecting data confidentiality. The formulated approach provides encryption based access control and techniques for query execution over encrypted, access restricted data on the database with only a few cases requiring computations on the client.

A third goal of this document is to document the advances made for protecting consistency and integrity of data stored in a cloud-object store. In previous work an abstract protocol has been introduced that enables a group of mutually trusting clients to detect violations of data-integrity and consistency by a malicious cloud storage service. This document reports on the refined design and implementation of a prototype for this task, called VICOS. Furthermore, benchmark results with an optimized implementation are shown.

The remainder of this deliverable is organized as follows. Chapter 1 illustrates a technique for enforcing access control restrictions over the data stored in a shuffle index, by properly adapting selective encryption. Chapter 2 presents a technique for query processing over encrypted data, where users have different privileges over the accessed data. Chapter 3 describes a solution for providing integrity and consistency guarantees for cloud object storage in a scenario where multiple users have possibly different views over the data stored in the cloud. Finally, Chapter 4 presents our conclusions.

1. Access Control for the Shuffle Index

The shuffle index studied in WP2 provides an index-based hierarchical organization of the data supporting efficient and effective access execution and provides access confidentiality with limited (compared to classical solutions) performance overhead. The shuffle index, while supporting accesses by multiple users [DFP+13], assumes all users to be entitled to access the complete data structure: data are encrypted with a key shared between the data owner and all users, and all users can retrieve and decrypt these data, hence accessing the plaintext content. Encryption is applied only to provide confidentiality (of content and access) with respect to the storing server. However, in many situations access privileges may need to be granted selectively, that is, different users should be authorized to view only a portion of the stored data. While existing solutions for enforcing authorizations in data outsourcing context in presence of honest-but-curious providers (e.g., *selective encryption* [DFJ+10]) have emerged, they cannot be simply applied in conjunction with the shuffle index, given the specific characteristics of the index and its access execution, as well as the need to ensure access confidentiality guarantees.

In this chapter, we provide an approach to support access control over the shuffle index to ensure that access to the data be granted only in respect of authorizations specified by the data owner. Our approach leverages the availability of selective encryption to provide a self-enforcing layer of protection over the data themselves. To allow for authorizations enforcement while maintaining access confidentiality guarantees, our approach makes use of two shuffle indexes: a primary index, storing and providing access to selectively encrypted data, and a secondary index, enabling enforcement of access control. We show that our proposal correctly enforces the access control policy established by the data owner and has limited performance overhead.

The remainder of this chapter is organized as follows. Section 1.1 discusses the state of the art. Section 1.2 presents the innovation provided by ESCUDO-CLOUD. Section 1.3 recalls the basic concepts of the shuffle index. Section 1.4 illustrates how to represent and enforce access control, and the organization of data and authorizations with the primary and secondary index. Section 1.5 illustrates how data access works to provide users the ability to access data while maintaining access confidentiality with respect to the providers and ensuring that users be able to access all and only those data for which they are authorized. Section 1.6 provides an analysis of our approach with respect to correctness for access control enforcement as well as access confidentiality and performance guarantees. Finally, Section 1.7 concludes the chapter.

1.1 State of the Art

The solutions proposed to protect data externally stored encrypt the data and support query evaluation through indexes (i.e., metadata complementing the outsourced encrypted dataset) or specific cryptographic techniques that support keyword-based searches (e.g., [HIML02, WCRL12]). These

approaches, however, do not guarantee the confidentiality of accesses and/or patterns of accesses which, as illustrated in Deliverable D2.1, are equally important.

Solutions for protecting access and pattern confidentiality are based on Private Information Retrieval (PIR) techniques or on dynamically allocated data structures, which change the physical location where data are stored at each access (e.g., [CMS99, DFP+11, DFP+13, DFP+15, DFP+16b, LC04, OS07, SS13, SvS+13, WSC08]). PIR solutions are computationally expensive and do not protect content confidentiality (e.g., [CMS99, OS07]). The Oblivious RAM (ORAM) dynamic structure, which has been extensively studied, guarantees content, access, and pattern confidentiality (e.g., [WSC08]). While preliminary proposals suffer from high computational and communication overheads, recent attempts have been proposed to make ORAM more practical (e.g., ObliviStore [SS13], Path ORAM [SvS+13],). Besides ORAM structure, also tree-based dynamically allocated structures, including the shuffle index studied in WP2 [DFP+15, PFL15], have been studied that provide a good trade-off between privacy and performance (e.g., [DFP+11, DFP+13, DFP+15, DFP+16b, LC04]). The shuffle index operates under the assumption that only one user owns and accesses the data. The shuffle index can however support concurrent accesses by different users [DFP+13]. Even if different users can access the shuffle index, a user can access either all the tuples in the leaves of the shuffle index or none of them. In this deliverable, we propose an approach to enable the data owner to enforce access control restrictions in such a way that each user can access a subset of the tuples in the leaves of the index structure.

The problem of enforcing access control restrictions over outsourced data has been recently considered in the literature. Existing solutions are based on the idea that the data themselves should enforce the access control policy. Current approaches follow two different strategies: selective encryption (e.g., [DFJ+10]), and attribute-based encryption (e.g., [GPSW06]). The solution presented in this deliverable extends selective encryption proposals since we combine the shuffle index with selective encryption to enable efficient access to the data through a tree-based index (while not revealing to users index values they are not authorized to access [DFJ+11]). The developed approach has been designed in such a way that neither the index structure nor accesses over it reveal to an observer (i.e., to the provider or to other users) sensitive data she cannot access and target of searches.

1.2 ESCUDO-CLOUD Innovation

The innovation brought by the proposed approach is the ability to support selective sharing in contexts where access confidentiality needs to be guaranteed. Indeed, existing approaches for protecting access confidentiality do not support access control and consider a simplifying all-or-nothing assumption, that is, they assume that users who can access the system can be given complete visibility to the complete data collection. This assumption is clearly limiting in selective sharing scenarios (focus of WP3) where users should be selectively authorized access only to specific portions of data, as dictated by the access control policy set by the data owner. Our solution provides selective sharing capability extending the shuffle index structure (developed in WP2 for providing access confidentiality) with access control. The access control solution complementing the shuffle index builds over the selective encryption approach previously developed in WP3. The proposed approach has the following characteristics.

- It is based on the adoption of two shuffle index structures, complementing each other. The primary index stores the data and provides access to the same in a selective way. The

secondary index is instead necessary for enforcing access control restrictions over the data stored in the primary index.

- The visit of the two (primary and secondary) index structures when searching for a target value does not expose relationships among the primary and secondary index values. The proposed approach has been designed considering the fact that accesses to the primary and to the secondary index are related, and hence could possibly expose access and pattern confidentiality.

- It correctly enforces the access control policy.

- It guarantees access and pattern confidentiality.

The work presented in this chapter has been published in [DFP+16a].

1.3 Shuffle Index

The *shuffle index* [DFP+15, PFL15] is a dynamically allocated data structure offering access and pattern confidentiality while supporting efficient key-based data organization and retrieval. A data collection organized in a shuffle index is a set of pairs $\langle$*index_value*, *resource*$\rangle$ with *index_value* a candidate key for the collection (i.e., no two resources share the same value for *index_value*) used for index definition, and *resource* the corresponding resource associated with the index value. For simplicity, we assume the data collection to be a relational table $\mathscr{R}$ defined over a simplified schema $\mathscr{R}(I,Resource)$, where I is the indexed attribute and *Resource* is the resource content. At the *abstract* level, a shuffle index for $\mathscr{R}$ over I is an *unchained B+-tree* (i.e., there are no links between the leaves) with fan-out F defined over attribute I, storing the tuples in $\mathscr{R}$ in its leaves. Each node stores up to $F-1$ ordered values $v_1, v_2, \ldots, v_q$, and has as many children as the number of values stored plus one. The first child of a node is the root of the subtree including all values $v < v_1$; its last child is the root of the subtree including all values $v \geq v_q$; its i-th child ($i = 2, \ldots, q$) is the root of the subtree including all values $v_{i-1} \leq v < v_i$. Actual resources are stored in the leaves of the tree in association with their index value. At the *logical* level, each node is associated with a logical identifier. Logical identifiers are used in internal nodes as pointers to their children and do not reflect the order relationship among the values stored in the nodes. At the *physical* level, each node is stored in *encrypted form* in a physical block and logical identifiers are translated into physical addresses at the storing server. For the sake of simplicity, we assume that the physical address of a block storing a node corresponds to the logical identifier of the node itself. The encrypted node is obtained by encrypting the concatenation of the node identifier, its content (values and pointers to children or resources), and a randomly generated nonce (*salt*). Formally, block b storing node n is defined as $E(k, salt||id||n)$, where E is a symmetric encryption function with key k and id is the identifier of node n. Encryption protects the confidentiality of nodes content and the structure of the tree, as well as the integrity of each node and of the structure overall. Figure 1.1(c-e) illustrates an example the abstract (c), logical (d), and physical (e) level, respectively, of a shuffle index storing the 19 tuples in Figure 1.1(a), indexed according to the values of attribute I. Actual tuples are stored in the leaves of the index structure, where, for simplicity, we however report only the index values.

To retrieve the tuple with a given index value in the shuffle index, the tree is traversed from the root following the pointers to the children until a leaf is reached. Since the shuffle index is

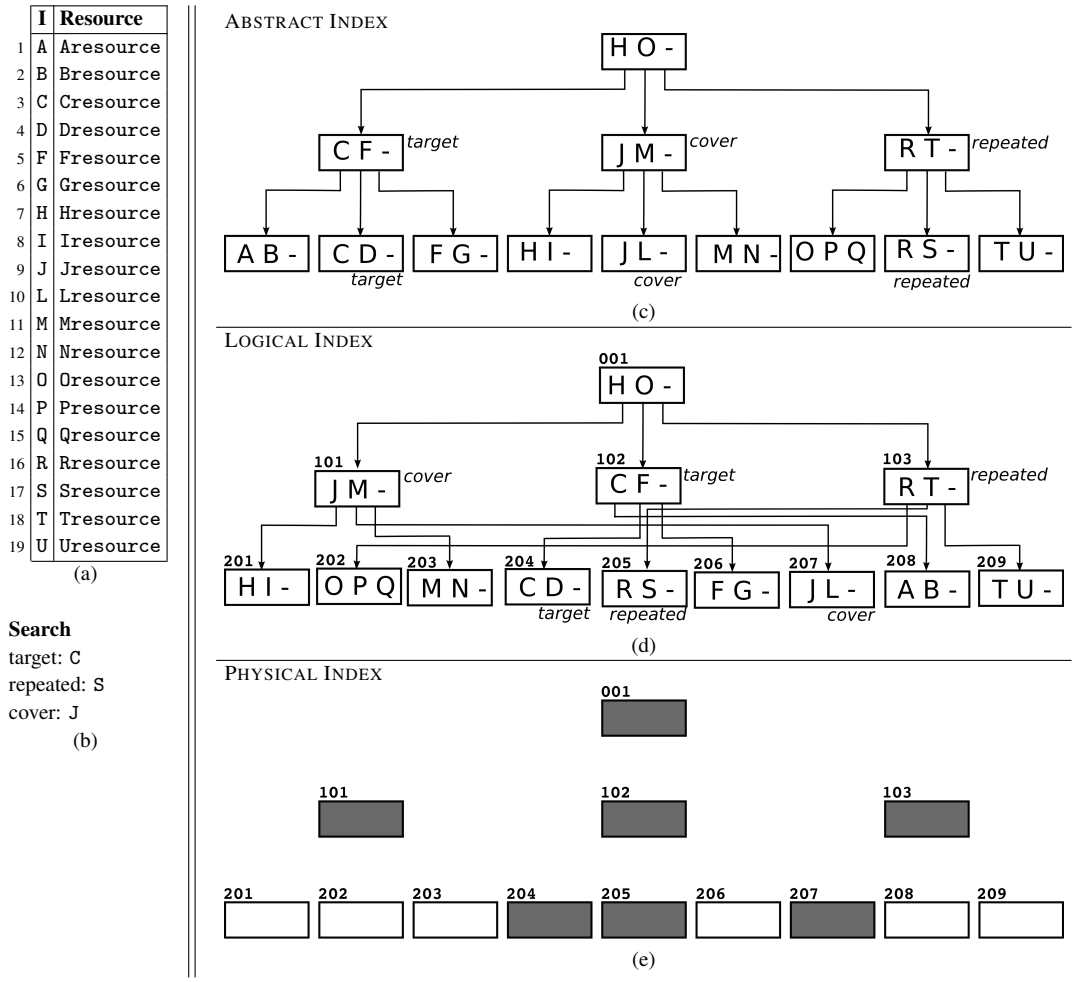

Figure 1.1: An example of a relation (a), an access over it (b), and of abstract (c), logical (d) and physical (e) shuffle index

stored at the server in encrypted form, such a process is iterative, with the client retrieving from the server (and decrypting) one node at a time to determine the child node to be read at the next level. To protect access and pattern confidentiality, in addition to storing nodes in encrypted form at the server, the shuffle index uses the following three techniques in access execution.

- *Cover searches*: in addition to the target value, additional values, called *covers*, are requested. Covers, chosen in such a way to be indistinguishable from the target and to operate on disjoint paths in the tree (also disjoint from the path of the target), provide uncertainty to the server on the actual target. If *num_cover* searches are used, the server will observe access to *num_cover*+1 distinct paths and corresponding leaf blocks, any of which could be the actual target.

- *Repeated access*: to avoid the server learning when two accesses refer to the same target since they would have a path in common, the shuffle index always produces such an observable by choosing, as one of the covers for an access, one of the values of the access just before it (if the current access is for the same target as the previous access, a new cover is used). In this way, the server always observes a repeated access, regardless of whether the two accesses refer to the same or to a different target.

- *Shuffling*: at every access, the nodes involved in the access are shuffled (i.e., allocated to different logical identifiers and corresponding physical blocks), re-encrypted (with a different random salt and including the new identifier of the block) and re-stored at the server. Shuffling provides dynamic reallocation of all the accessed nodes, thus destroying the otherwise static correspondence between physical blocks and their content. This prevents the server from accumulating knowledge on the data allocation as at any access such an allocation is refreshed.

To illustrate, consider the shuffle index in Figure 1.1(c-e) and the search in Figure 1.1(b) for the tuple with index value `C`, assuming `S` as repeated access and `J` as fresh cover. The access entails reading (i.e., retrieving from the server) the nodes annotated in the figure, with the server only observing downloads of the corresponding encrypted blocks in Figure 1.1(e) but not able to learn anything on the block content or on the roles (target, repeated, cover) of the blocks. Shuffling could produce, after the access, a re-allocation of the accessed nodes. For instance, 205→204, 204→207, 207→205 (where X→Y denotes the fact that the content of node X is moved to Y).

1.4 Primary and Secondary Indexes for Access Control

Providing access control means enabling data owners to regulate access to their data and selectively authorize different users with different views over the data. Figure 1.2(a) illustrates possible authorizations on the data of Figure 1.1(a), considering three users (u_1, u_2, u_3). The figure reports, for each tuple r in the dataset, the corresponding $acl(r)$, that is the set of users authorized to access it. (Note that authorizations do not explicitly report the access privileges, which is considered to be 'read', since we assume access by users to be read-only, with write operations reserved to the owner.) When clear from the context, with a slight abuse of notation, in the following we will denote the access control list of a tuple r as either $acl(r)$ or $acl(r[I])$, with $r[I]$ its index value. For instance, $acl(\texttt{A})=\{u_1,u_2,u_3\}$, while $acl(\texttt{B})=\{u_1,u_2\}$.

Before diving into our solution, we note that there could be two natural and straightforward approaches to enforce authorizations in the shuffle index, each of which would, however, have limitations and drawbacks. A first natural approach would be to simply associate a key k_i with each user u_i and produce different replicas of the data. Each tuple would be replicated as many times as the number of users authorized to access it. Each copy would be encrypted with the key of the user for which it is produced. For instance, with reference to Figure 1.2(a) three copies would be created for index value `A` and the corresponding resource `Aresource`, encrypted with keys k_1, k_2, and k_3, respectively. Different shuffle indexes would then be defined, one for each user, organizing and supporting accesses to the tuples that the user is authorized to access. Such an approach, besides bearing obvious data management problems (as replicas would need to be maintained consistent) would affect the protection offered by the shuffle index. In fact, it would organize each shuffle index only on a limited portion of the data (for each user, only those tuples that she can access, that is, less than half of the original tuples for each user in our example) with consequent limitations in the choice of covers. An alternative solution could then be to maintain the shuffle index as a single structure (so to build it on the complete dataset), and avoid replicas by producing only one encrypted copy for each tuple. Replicas can be avoided by considering different encryption keys not only for individual users but also for user sets (i.e., *acl*s), with a user u_i knowing her encryption key k_i as well as those of the *acl*s in which she is included. Each resource would then be encrypted only once and the encryption key with which it is encrypted

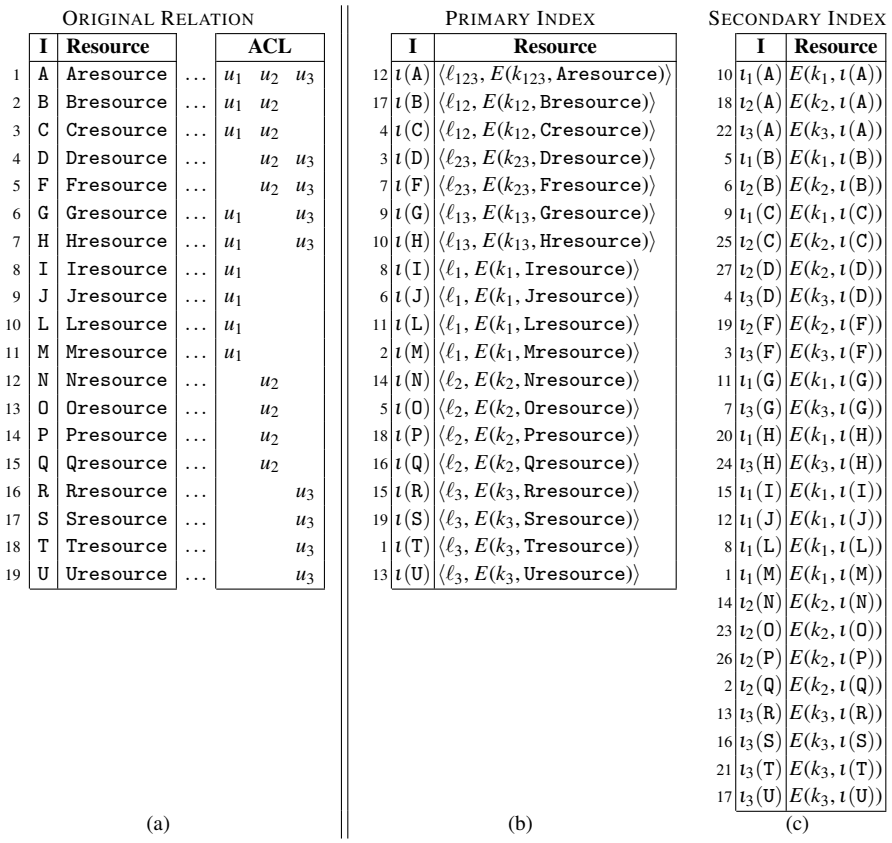

Figure 1.2: Relation of Figure 1.1(a) with *acl*s associated with its resources (a), relation for the primary index (b), relation for the secondary index (c)

known only to its authorized users. For instance, with reference to Figure 1.2(a), `Aresource` would be encrypted with key k_{123} known to all users while `Bresource` would be encrypted with key k_{12} known to u_1 and u_2 only. While such selective encryption correctly enforces access to the encrypted resources, it leaves the problem of ensuring protection (and controlling the possible exposure) of the index values with which the shuffle index is organized. As a matter of fact, on one hand, leaving such index values accessible to all users for traversing the tree would disclose to every user the complete set of index values, even those of the tuples she is not authorized to access. On the other hand, such index values cannot be encrypted with the same encryption key used for the corresponding resources, as otherwise the ability to traverse the tree by users would be affected.

Starting from these observations, we build our approach essentially providing selective encryption while protecting index values themselves against unauthorized users without affecting their ability to retrieve those tuples they are authorized to access. Our approach is based on the definition of two different indexes. A *primary index*, defined over an encoded version of the original index values, and a *secondary index*, providing a mapping enabling users to retrieve the value to look for in the primary index. Both indexes make use of an encoding of the values to be indexed to make them intelligible only to authorized users. We then start by defining an encoding function as follows.

Definition 1.4.1 (Encoding Function)**.** *Let $\mathscr{R}$(I,Resource) be a relation with I defined over domain $\mathscr{D}$. A function $\iota : \mathscr{D} \rightarrow \mathscr{E}$ is an* encoding function *for I iff ι is:* i) *non-invertible;* ii) *non order-*

preserving; iii) *injective.*

Intuitively, an encoding function maps the domain of index values I onto another domain of values $\mathscr{E}$, avoiding collisions (i.e., $\forall v_x, v_y \in I$ with $v_x \neq v_y$, $\iota(v_x) \neq \iota(v_y)$), and in such a way that the original ordering among values is destroyed. Also, non-invertibility ensures the impossibility of deriving the inverse function (from encoded to original values). For instance, an encoding function can be realized as a keyed cryptographic hash function operating on the domain of attribute I.

The second building block of our solution is the application of selective encryption, namely encryption of each resource with a key known only to authorized users. To apply selective encryption, we then define a key set for the encryption policy as follows.

Definition 1.4.2 (Encryption Policy Keys)**.** *Let $\mathscr{R}(I,Resource)$ be a relation, $\mathscr{U}$ be a set of users, and, $\forall r \in \mathscr{R}$, $acl(r) \subseteq \mathscr{U}$ be the acl of r. The set $\mathscr{K}$ of* encryption policy keys *for $\mathscr{R}$ is a set $\mathscr{K} = \{k_i \mid u_i \in \mathscr{U}\} \cup \{k_{i_1,\ldots,i_n} \mid \exists r \in \mathscr{R}, \{u_{i_1},\ldots,u_{i_n}\} = acl(r)\}$ of encryption keys. Each key $k_X \in \mathscr{K}$ has a* public label ℓ_X*. Each user $u_i \in \mathscr{U}$ knows the set $\mathscr{K}_i = \{k_i\} \cup \{k_X \mid k_X \in \mathscr{K} \wedge i \in X\}$ of keys.*

Definition 1.4.2 defines all the keys needed (and the knowledge of users on them) to apply selective encryption, meaning to encrypt the data selectively so that only authorized users can access them while optimizing key management and avoiding data replication. The public label associated with a key allows referring to the key without disclosing its value. Note that knowledge by a user of all the keys of the access control lists to which she belongs does not require direct distribution of the keys to the user, since hierarchical organization of keys and use of publicly available tokens enabling key derivation can provide such a knowledge to the user [DFJ+10].

We are now ready to define the first index used by our approach. This first index, called *primary*, is the one storing the actual data on which accesses should operate (i.e., tuples in $\mathscr{R}$). To provide selective access as well as enable all users to traverse the index without leaking to them information (index values and resources) they are not authorized to access, the index combines value encoding and selective encryption. Formally, the primary index is defined as follows.

Definition 1.4.3 (Primary Index – Data)**.** *Let $\mathscr{R}(I,Resource)$ be a relation, I be the indexing attribute, ι be an encoding function for I computable only by the data owner, and $\mathscr{K}$ be the set of encryption policy keys for $\mathscr{R}$. A* primary index *for $\mathscr{R}$ over I is a shuffle index over relation $\mathscr{P}(I,Resource)$ having a tuple p for each tuple $r \in \mathscr{R}$ such that $p[I] = \iota(r[I])$ and $p[Resource] = \langle \ell_{i_1,\ldots,i_n}, E(k_{i_1,\ldots,i_n}, r[Resource]) \rangle$, with E a symmetric encryption function, $acl(r) = \{u_{i_1},\ldots,u_{i_n}\}$, and $k_{i_1,\ldots,i_n} \in \mathscr{K}$*

The primary index stores original data in encrypted form, encrypting each tuple with the key corresponding to its *acl* (i.e., known only to the users authorized to read the tuple). The inclusion in $r[Resource]$ of the label enables authorized users to know the key to be used for the decryption of the resource. The primary index is built on encoded values computable only by the data owner. For instance, the encoding function can be implemented through a cryptographic hash function, using a key k_o known only to the data owner (i.e., the encoded value $\iota(v)$ for a tuple r with index value v can be computed as $hash(v,k_o)$). Note that, although each resource singularly taken appears encrypted in the leaves of the primary index, all the nodes are (also) encrypted with a key k known to every user in the system. This second encryption layer is necessary to enable shuffling (Section 1.3).

Building the index on the encoded values provides protection of the original index values, and their order relationship, against users and storing server that observe the index on the encoded values. In fact, the encoding is non-invertible (hence the encoded values do not leak any information

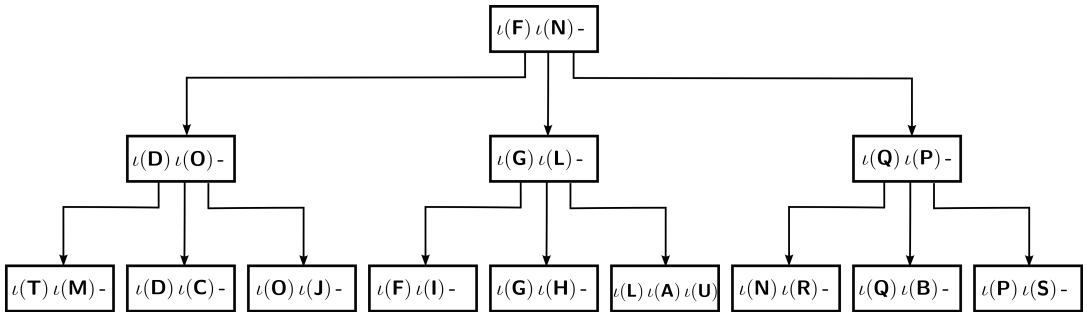

Figure 1.3: Primary shuffle index for the relation in Figure 1.2(b)

on the original values), and destroys the original ordering (hence the order relationship between encoded value does not leak anything on the order relationship among the original values).

Figure 1.2(b) illustrates a primary index $\mathscr{P}$ for our running example. The ordering among the encoded values is reported with numbers on the left of the table. Figure 1.3 illustrates the tree structure for such primary index. Note how the different order among the values to be indexed causes a different content within the leaves and a different ordering among them, with respect to the shuffle index in Figure 1.1(a) built over the original (non-encoded) index values.

While the index on the encoded values provides the ability to traverse the tree to look for the resource associated with an encoded value, to retrieve a given resource (i.e., the resource corresponding to an original value for the indexing attribute) one would need to know the encoding of such value. For instance, resource `Aresource` would be stored in association with index value $\iota(\texttt{A})$. The encoding (i.e., the fact that $\iota(\texttt{A})$ corresponds to `A`) is however known only to the data owner.

The second index of our approach allows the data owner to selectively disclose to users the mapping of encoding ι, releasing to every user the mapping for (*all and only*) those values she is authorized to access. Such knowledge is provided to each user u_i encrypted with the user key k_i (so to make it non intelligible to other users and to the server) and is indexed with a user-based encoding, so to provide a distinct mapping for every user u_i, which can be computed only by u_i. The second index of our approach is therefore a *secondary* index providing user-based mapping as follows.

Definition 1.4.4 (Secondary Index – User-based Mapping)**.** *Let $\mathscr{R}$(I,Resource) be a relation, I be the indexing attribute, $\mathscr{P}$ be a primary index for $\mathscr{R}$ over I with encoding function ι, $\mathscr{U}$ be a set of users, $\{\iota_i \mid u_i \in \mathscr{U}\}$ be a set of encoding functions for I such that ι_i is computable only by user u_i and by the data owner, and $\mathscr{K}$ be the set of encryption policy keys for $\mathscr{R}$. A* secondary index *for $\mathscr{R}$ and $\mathscr{P}$ is a shuffle index over relation $\mathscr{S}$(I,Resource) having a tuple s for each pair $\langle r,u_i\rangle$, $r\in\mathscr{R}$ and $u_i\in acl(r)$, such that $s[I]=\iota_i(r[I])$ and $s[Resource]=E(k_i,\iota(r[I]))$, with E a symmetric encryption function and $k_i\in\mathscr{K}$.*

For instance, the encoding function of each user u_i can be implemented as a cryptographic hash function, using a key k_i known to user u_i only (i.e., $\iota_i(v)$=*hash*(v,k_i)). Figure 1.2(c) illustrates a secondary index for our running example. Again, the number on the left of the table is the ordering among the index values of the secondary index. Notice how, once again, the encoding does not convey any information on the ordering of the original index values. Note that the secondary index has a larger number of tuples than the original index, since the encoding of an original index value is encrypted as many times as the number of users who can access it. For instance,

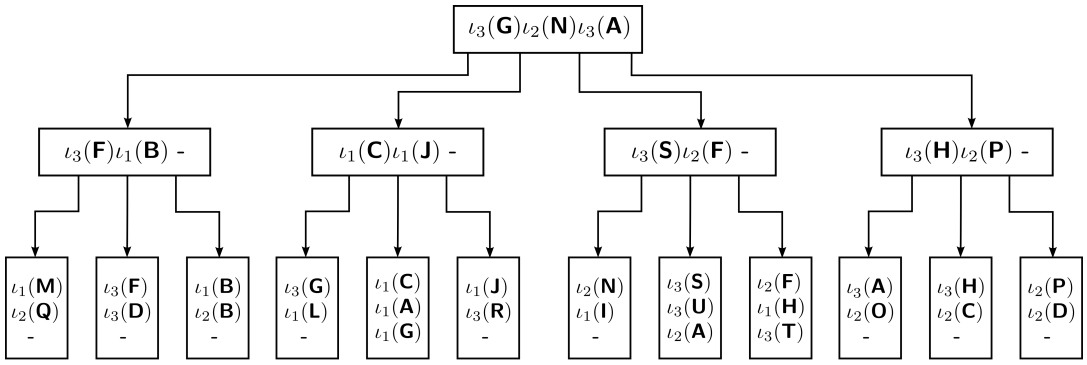

Figure 1.4: Secondary shuffle index for the relation in Figure 1.2(c)

in our example, there are three instances of $\iota(\texttt{A})$. Figure 1.4 illustrates the tree structure for the index in Figure 1.2(c). We note however that the secondary index is very slim as the resources are simply the encryption, with the key of a user, of the owner encoding. While in our examples, for simplicity, we maintain the same topology, the structure of the secondary index is independent from the structure of the primary index, meaning that they may have different fan-out and height.

Note that the property of the encoding function of destroying the ordering among original index values is particularly important to guarantee protection. In fact, users will know all encoded values computed by the data owner (i.e., the co-domain of function ι), but will know the actual mapping (i.e., the actual value v corresponding to $\iota(v)$) only for the values they are allowed to access. Figure 1.5(a-b) illustrates a possible logical organization for the primary and secondary index of our example, where for simplicity of illustration we assume the logical organization to reflect (at this initial time) the abstract organization of the tree. We distinguish blocks of the primary and secondary index by adding prefix P and S, respectively, to their identifiers. The coloring represents the visibility of users u_1. Encoded values with grey background are those which remain non intelligible to u_1 as they are encoded with the mapping of other users (for the secondary index) or their owner encoding is not disclosed to u_1 (for the primary index).

Since encoding does not preserve ordering, encoded values non intelligible to a user will remain protected, as no inference can be drawn on them from their presence or order relationships with respect to other encoded values which are intelligible to her. For instance, consider the primary index in Figure 1.5(b). User u_1, being authorized for B will know that $\iota(\texttt{B})$ is the corresponding encoding. At the same time, however, $\iota(\texttt{Q})$, stored in the same node, remains non intelligible to her. User u_1 simply observes the presence of another encoded value but will be able to infer neither its corresponding original value nor its order relationship with respect to B.

1.5 Access Execution

We now illustrate how the two indexes described in the previous section are jointly used for accessing a tuple of interest. To retrieve a tuple in $\mathscr{R}$ with value v for I, a user u_i would need to perform the following steps:

1. compute the user-based mapping $\iota_i(v)$=$hash(v,k_i)$;
2. search $\iota_i(v)$ in the secondary index $\mathscr{S}$, retrieving the corresponding encoded value $\iota(v)$;

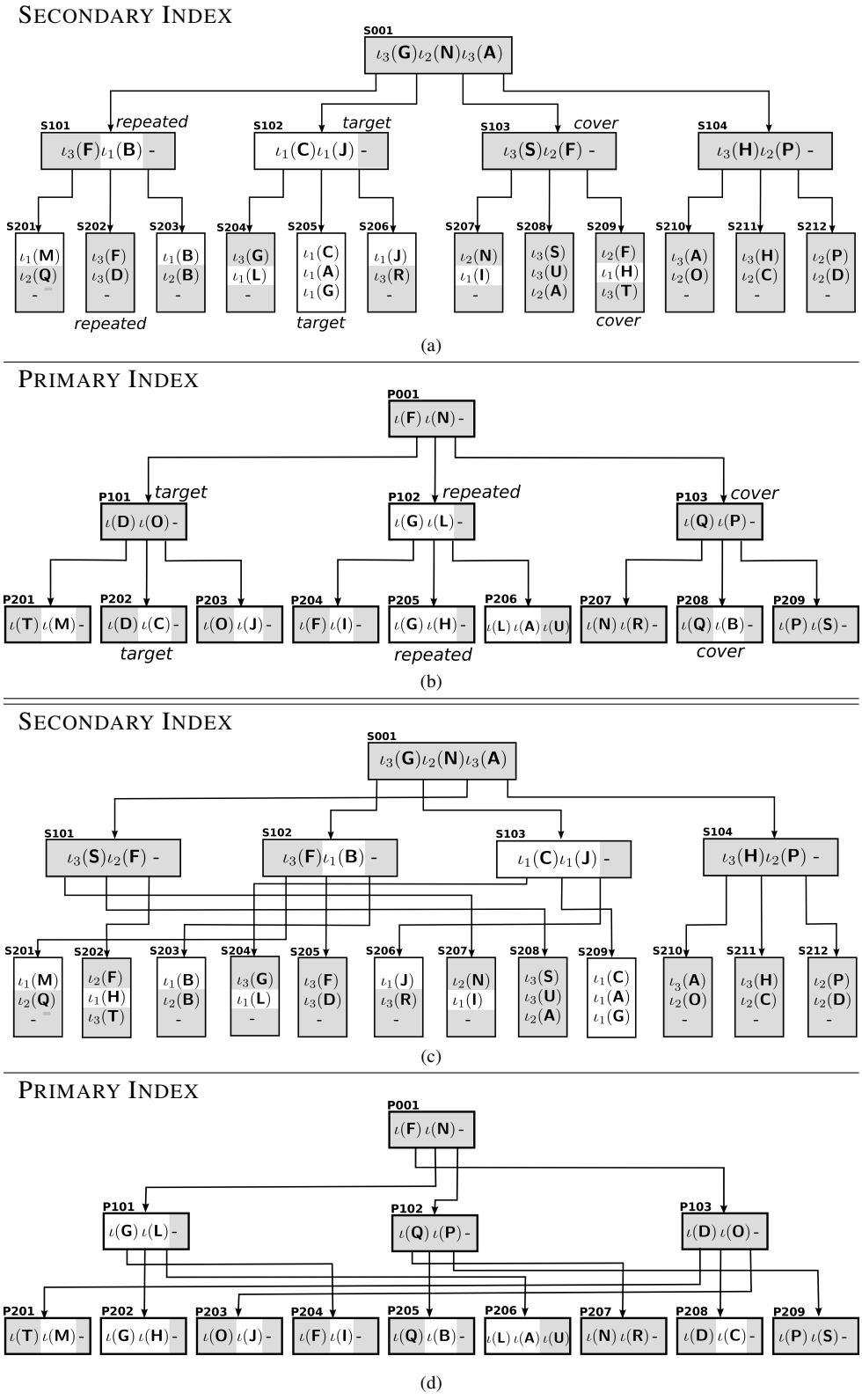

Figure 1.5: Secondary and primary index before (a-b) and after (c-d) the access by u_1 over C. Secondary index: i) cover: $\iota_2(\mathtt{F})$, ii) repeated access: [S001,S101,S202], iii) shuffling: S101→S102, S102→S103, S103→S101, S202→S205, S205→S209, S209→S202. Primary index: i) cover: $\iota(\mathtt{Q})$, ii) repeated access: [P001,P102,P205], iii) shuffling: P101→P103, P102→P101, P103→P102, P202→P208, P205→P202, P208→P205. The gray background denotes encoded values non intelligible to u_1

3. search $\iota(v)$ in the primary index $\mathscr{P}$, retrieving the corresponding target tuple.

As an example, consider the indexes in Figure 1.5(a-b) and suppose that user u_1 searches index value C. User u_1 computes ι_1(C)=*hash*(C,k_1) and then searches it in the secondary index in Figure 1.5(a). The search returns block S205, from which ι(C) is retrieved. Hence, u_1 searches ι(C) in the primary index in Figure 1.5(b). The search returns block P202, from which u_1 can retrieve resource Cresource.

Note that the steps above assume the searched value to be present in the index. If the value is not present in the secondary index, its user-based mapping does not appear in the block returned by step 2. In such a case, the process will continue providing a random value for the search in step 3, so to provide to the server the same observation as a successful search. Note also that the search for a value that is present in the dataset but for which the searching user is not authorized, present to the searching user the same observable as the search for a missing value (hence not disclosing anything to the user about values she is not authorized to access).

The steps above simply illustrate how to retrieve a target value. However, both the primary and the secondary index are shuffle indexes and accesses should not simply aim at the target value but should also be protected with the techniques (cover, repeated searches, and shuffling) devoted to protect access confidentiality. The application of these techniques on the two indexes is completely independent, meaning that the choice of covers, repeated searches, and shuffling can be completely independent in the two indexes. The only dependency among the two indexes is the fact that - clearly - the target to be searched in the primary index is the tuple retrieved by the search on the secondary index.

Covers, repeated searches, and shuffling on the primary and secondary index work essentially in the same way as they work in the shuffle index in absence of authorizations (Section 1.3). However, the nature of these indexes requires minor adjustments in their application, as follows.

- *Cover searches*. For both the secondary and the primary indexes, cover searches should be chosen from the set of *encoded values*, in contrast to the set of original values. The reason for this is that every user has limited knowledge on the set of original index values while she can have complete knowledge of the encoded values in the indexes (i.e., of the complete co-domains of all the encodings of all the users and the complete co-domain of the encoding of the owner). Since the encoding is non-invertible, this knowledge does not leak any information and allows the widest possible choice to the user.

- *Repeated accesses*. Repeated accesses for the primary and secondary indexes should refer to blocks, instead of specific values. The reason for this is that two subsequent accesses can be performed by two different users and therefore considering repeated searches referred to values would leak to the second user the target of the search of the previous user. Although such a leakage would be only on encoded values, we avoid it simply by assuming repeated accesses to be referred to blocks (and not to values) and to consider all accessed blocks, not only the target. At every access we then store at the server the identifiers of the blocks (target, covers, or repeated accesses) accessed during the last search. The knowledge of such identifiers is sufficient for a user to repeat an access to one of the paths visited by the search just before hers without revealing to the user the target of the previous search (which might have been performed by others).

- *Shuffling*. Shuffling works just like in the original proposal. We note that when shuffling, a user may also move content which is not intelligible to her. However, she will not be

able to change the content for which she is not authorized (since she would not know the encryption key and tampering would be detected). Note that since all physical blocks stored at the server are encrypted (with a key shared between all users and the data owner) and encryption of the block as a whole is refreshed at every shuffle, the server cannot detect whether the content of a block (or part of it) has changed or not. Hence, the fact that a user can operate only on a portion of the block does not prevent correct execution of the shuffling operation.

Figure 1.6 illustrates the algorithm, executed at the client side, searching for a value in the primary index and in the secondary index. The algorithm operates as discussed in Section 1.5 and relies on function **Search** to access the primary and secondary index structures.

Function **Search** receives as input the shuffle index $\mathscr{T}$ on which it should operate, the index value *target_value* target of the access, and the number *num_cover* of covers to be adopted. It returns the tuple r with index value *target_value* (if any). The function randomly chooses *num_cover*+1 values in the domain of the (primary or secondary) index and it downloads from the server the identifiers of the blocks visited by the previous search (lines 1-3). It then visits the shuffle index level by level, starting from the root. At each level *level*, the function determines the identifiers of the nodes along the path to the target, covers, and repeated access (lines 5-8). If the block along the path to the target has been accessed by the previous search, it is repeated (an additional cover is used). The function downloads from the server and decrypts the blocks of interest (line 13) and shuffles their content (line 16). To guarantee the correctness of the search and of the index structure, the function updates the references to children of the nodes accessed at level *level*-1 (which are the parents of the nodes shuffled at level *level*), variables *target*, *repeated*, and *cover*$[1,\ldots,$*num_cover*$]$ (lines 17-21). The nodes at level *level*-1 are then encrypted and written at the server. The identifiers of the nodes accessed at level *level* are then used to update *repeated_search*[*level*] (line 23). Once the leaf node where *target_value* is possibly stored has been reached, the function extracts and returns the tuple with index value equal to *target_value* (lines 25–27).

Given the request by user u_i to search for value *target_value*, the algorithm computes the user-based mapping $\iota_i(\textit{target\_value})$ and invokes function **Search** to search for such a value in the secondary index (lines 1–4). It decrypts the tuple retrieved by **Search**, obtaining encoded value $\iota(\textit{target\_value})$ for *target_value* (line 5). If such a value is not NULL (i.e., there is a tuple that u_i can access with index value equal to *target_value*), the algorithm invokes function **Search** over the primary index, looking for $\iota(\textit{target\_value})$. It then computes/retrieves the encryption key necessary to decrypt the retrieved resource and decrypts it. It returns the plaintext resource to the user (lines 7–11). If the result of function **Search** over the secondary index is NULL, the algorithm runs a fake search over the primary index (not to disclose any information to other users and to the server about u_i's privileges) and returns an empty resource to the user (lines 12-14).

Figure 1.5(a-b) illustrates an example of access execution for search of value C by user u_1, assuming $\iota_2(\mathtt{F})$ as cover and path [S001,S101,S202] as repeated access for the secondary index, and $\iota(\mathtt{Q})$ as cover and path [P001,P102,P205] as a repeated access for the primary index. Accessed nodes are, besides the root, those annotated (as target, cover, or repeated) in the figure. Figure 1.5(c-d) illustrates the new structure of the indexes that would result assuming shuffling: for the secondary index as S101→S102, S102→S103, S103→S101, S202→S205, S205→S209, and S209→S202; for the primary index as P101→P103, P102→P101, P103→P102, P202→P208, P205→P202, P208→P205.

```
/* 𝒫, 𝒮 : primary and secondary index */
/* num_cover : number of cover searches */
/* u_i,k_i : user performing the access and her key */
/* hash : non-invertible cryptographic hash function */
INPUT    target_value : value to be searched in the shuffle index
OUTPUT   resource with index value target_value
MAIN
 1: /* Phase 1: compute the user-based mapping ι_i(target_value) */
 2: target_idx := hash(target_value, k_i)
 3: /* Phase 2: search ι_i(target_value) in the secondary index */
 4: s := Search(𝒮,target_idx,num_cover)
 5: target_idx := decrypt s[Resource] with k_i /* encoded value ι(target_value) */
 6: /* Phase 3: search ι(target_value) in the primary index */
 7: if target_idx ≠ NULL then
 8:    p := Search(𝒫,target_idx,num_cover)
 9:    k := retrieve key k with label ℓ, where p[Resource]=⟨ℓ,content⟩
10:    result := decrypt content with k
11:    return(result)
12: else target_idx := randomly choose a value for ι(target_value)
13:    Search(𝒫,target_idx,num_cover)
14:    return(NULL)
SEARCH(𝒯,target_value,num_cover)
 1: repeated_search[0,...,𝒯.height] := download and decrypt the blocks of accesses for 𝒯
 2: randomly choose cover_value[1...num_cover+1] for target_value in the co-domain of hash
 3: repeated := repeated_search[0] /* identifier of the root block */
 4: for level:=1...𝒯.height do
 5:    /* identify the blocks to read from the server */
 6:    target := identifier of the node at level level along the path to target_value
 7:    cover[i] := id of the node at level level along the path to cover_value[i], i=1...num_cover+1
 8:    repeated := block identifier in repeated_search[level] that is a descendant of repeated
 9:    if target is the identifier of a node in repeated_search[level] then
10:       repeated := target, num_cover := num_cover−1
11:    ToGet := {target,repeated} ∪ cover[1...num_cover] /* ids of the blocks to be downloaded */
12:    /* read blocks */
13:    Nodes := download and decrypt the blocks with identifier in ToGet
14:    /* shuffle nodes */
15:    let π be a permutation of the identifiers of nodes in Nodes
16:    shuffle nodes in Nodes according to π
17:    update pointers to children of the parents of nodes in Nodes according to π
18:    encrypt and write at the server nodes accessed at iteration level − 1
19:    target := π(target)
20:    cover[i] := π(cover[i]), i=1...num_cover+1
21:    repeated := π(repeated)
22:    /* update the repeated search at level level */
23:    repeated_search[level] := ToGet
24: encrypt and write at the server nodes accessed at iteration 𝒯.height and repeated_search
25: let n∈Nodes the node with n.id=target
26: let r∈n be the tuple such that r[I]=target_value
27: return(r)
```

Figure 1.6: Shuffle index access algorithm

1.6 Analysis

We discuss the protection guarantees (i.e., the correct enforcement of authorizations and the protection of access and pattern confidentiality) and the performance of our approach.

Access control enforcement. To demonstrate that the primary and secondary indexes described in Section 1.4 guarantee the correct enforcement of the access control policy, we need to prove that each user u_i can access all and only the resources and index values in $\mathscr{R}$ she is authorized to

access. Formally, $\forall u_i \in \mathscr{U}$: *i)* u_i can access resource $r[Resource]$ iff $u_i \in acl(r)$; *ii)* u_i can see an index value v iff $\exists r \in \mathscr{R}$ s.t. $r[I]=v$ and $u_i \in acl(r)$.

Consider a user u_i s.t. $acl(r)=\{u_{i_1},\ldots,u_{i_n}\}$ and $u_i \in \{u_{i_1},\ldots,u_{i_n}\}$. We need to show that u_i can retrieve the plaintext content of tuple r. A user u_i can retrieve and decrypt r iff: *i)* u_i can compute $\iota_i(r[I])$; *ii)* $\exists! s \in \mathscr{S}$ s.t. $s[I]=\iota_i(r[I])$ and $s[Resource]=E(k_i,\iota(r[I]))$; *iii)* $\exists! p \in \mathscr{P}$ s.t. $p[I]=\iota(r[I])$ and $p[Resource]=\langle \ell_{i_1,\ldots,i_n}, E(k_{i_1,\ldots,i_n}, r[Resource])\rangle$; and *iv)* u_i can visit $\mathscr{S}$ and $\mathscr{P}$.
User u_i can compute $\iota_i(r[I])$ since it is defined as $hash(r[I],k_i)$ and u_i knows key k_i, by Definition 1.4.2. Tuple s exists and belongs to $\mathscr{S}$ by Definition 1.4.4. Tuple p exists and belongs to $\mathscr{P}$ by Definition 1.4.3. User u_i can decrypt the content of $s[Resource]$ as she knows $k_i \in \mathscr{K}_i$, and the content of $p[Resource]$ as she knows $k_{i_1,\ldots,i_n} \in \mathscr{K}_i$ because $u_i \in acl(r)$, by Definition 1.4.2. Any authorized user, including u_i, can visit both $\mathscr{S}$ and $\mathscr{P}$ since she knows both the encryption key k used by the data owner to encrypt nodes content to enable shuffling, and the co-domain of the encoding functions.

Consider now a user u_i s.t. $acl(r)=\{u_{i_1},\ldots,u_{i_n}\}$ and $u_i \notin \{u_{i_1},\ldots,u_{i_n}\}$. We need to show that u_i can access neither the plaintext content of $r[Resources]$, nor index value $r[I]$. It is immediate to see that u_i cannot access the plaintext content of the tuple since it is encrypted with a key k_X (Definition 1.4.3) that u_i does not know. In fact, by Definition 1.4.3, since u_i does not belong to $acl(r)$, she does not know the corresponding encryption key. User u_i cannot compute or guess index value $r[I]$ because $r[I]$ is never represented in internal or leaf nodes of the primary and secondary indexes; it is instead represented via its encoded value (i.e., $\iota(r[I])$ in the primary index and $\iota_j(r[I])$, $\forall u_j \in acl(r)$, in the secondary index). Since the encoding function is, by Definition 1.4.1, non-invertible, u_i cannot exploit her knowledge of encoded values to retrieve the corresponding original index values. Also, the traversal of the primary (and secondary) index does not reveal u_i anything about the original index values. In fact, by Definition 1.4.1, the encoding function does not preserve the order relationship among values. Hence, similar encoded values (e.g., represented in the same leaf) may not correspond to similar original values (and vice versa).

Access confidentiality. We first consider the storing server as our observer and analyze the protection offered by our proposal for the novel aspects introduced with respect to the shuffle index proposal in [DFP+15]. Like in the original proposal, we focus the analysis on the leaves of the shuffle index. In fact, nodes at a higher level are subject to a greater number of accesses, due to the multiple paths that pass through them, and are then involved in a larger number of shuffling operations, which increase their protection. A search operation on the primary and secondary index operates as in the original proposal. Hence, it enjoys the protection guarantees given by the combined adoption of covers, repeated searches, and shuffling. In the considered scenario, however, we operate with two indexes and each search for a value entails an access to the secondary index followed by an access to the primary index. The targets of the two accesses are related as they are the encoding of the same original index value. However, both indexes protect the target of accesses (as well as patterns thereof) and the covers and repeated searches adopted for the two indexes are different. This practice prevents the server from identifying any correspondence between the values in the leaves of the two indexes.
We now consider a user as our observer. A user can observe the blocks accessed by another user in a previous access (for repeated accesses), but she cannot identify the target of the access. In fact, this set of blocks includes the target, covers, and repeated accesses. Furthermore, each leaf stores multiple encoded values, which correspond to index values that are not close to each other since the encoding function is not order-preserving. A user can also possibly trace shuffling operations, but this would require her to download the whole index at each access.

Performance evaluation. The performance of the system is assessed as the average response time experienced by an authorized client when submitting an access request. System configurations providing a primary index and a secondary index with fixed heights and different fan-outs exhibit similar average response times for the client request. Moreover, varying the number of authorized users and the size of the access control lists do not significantly influence the performance of the system as long as the fan-out of the secondary index is chosen to be reasonably large. Our experiments show that the latency of the network is the factor with the greatest impact in a large-bandwidth LAN/WAN scenario. To assess the performance of our algorithm, we configured the primary index and the secondary index as 3-layer unchained $B+$-trees with fan-out 512, both of them built on a numerical candidate key of fixed-length to allow the indexing of more than 200K different values. The size of the blocks (nodes) of each index was 8KiB. The hardware used in the experiments included a client machine with an Intel Core i5-2520M CPU at 2.5GHz, L3-3MiB, 8GiB RAM DDR3 1066, running an Arch Linux OS. The server machine run an Intel Core i7-920 CPU at 2.6GHz, L3-8MiB, 12GiB, RAM DDR3 1066, 120GiB SSD disk running an Ubuntu OS. The network environment was configured through the `NetEm` suite for Linux operating systems to emulate a typical WAN interactive traffic with a round-trip time modeled as a normal distribution with mean of 100ms and standard deviation of 2.5ms. The performance figures obtained for accessing the secondary and the primary index exhibit an average value equal to 750ms, which compares favorably with the response time of 630ms experienced by the client when accessing two plain encrypted indexes (i.e., without shuffling).

1.7 Conclusions

We have developed an approach to enrich the shuffle index studied in WP2 with access control, building on the techniques for enforcing selective access analyzed in WP3. The enriched shuffle index illustrated in this chapter provides guarantees of access confidentiality while enabling data owners to regulate access to their data selectively granting visibility to users. Also, like the original shuffle index proposal, it has limited performance overhead.

2. Selective Data Sharing via Encrypted Query Processing

A data owner outsourcing the database of a multi user application wants to prevent information leaks caused by outside attackers exploiting software vulnerabilities or by curious personnel. Query processing over encrypted data solves this problem for a single user, but provides only limited functionality in the face of access restrictions for multiple users and keys. ENKI is a system for securely executing queries over sensitive, access restricted data on an outsourced database. It introduces an encryption based access control model and techniques for query execution over encrypted, access restricted data on the database with only a few cases requiring computations on the client. A prototype of ENKI supports all queries seen in three real world use cases and executes queries from TPC-C benchmark with a modest overhead compared to single user mode.

2.1 State of the Art

Queries over encrypted data. Techniques to handle certain relational operations are provided by [SWP00] for key word search and [AKSX04, BCLO12, PLZ13, KS14] for order preserving encryption. Providing data confidentiality using tuple-wise encryption and providing query processing using indexes organized in buckets is proposed in [HIML02, DDJ+03]. Ciriani et al. satisfy privacy-constraints by (partial) encryption and fragmentation of data and rely on the application logic to process a query [CDF+09]. Popa et al. introduce adjustable query-based symmetric encryption to process all relational operation on server-side [PRZB11]. The execution of complex queries e.g. TPC-H benchmark at the costs of client-server splits is discussed in [TKMZ13]. We see no principle obstacles to extend ENKI accordingly. Query processing with multiple keys without sharing data is presented in [PRZB11] and using searchable encryption in [YBDD09, PZ13, ARCI13]. Ferretti et al. introduces a proxy concept to handle multiple user in the CryptDB setting, but do not present an experimental evaluation or a security analysis to proof their claims [FCM13].

Joins over encrypted data. Hacigumus et al. require extensive query rewriting to compute joins [HIML02] while Agrawal et al. propose an interactive approach [AES03]. Symmetric, transitive proxy re-encryption of deterministic encryption schemes are provided in [PH78, PRZB11]. Furukawa et al. provide a non-transitive, non-symmetric approach to compute a join such that a probabilistic encryption is degraded to be deterministic in the single user mode [FI13]. The encryption scheme presented in [ARCI13] also handles join operations, but no confidentiality guarantees are provided.

Access Control. Encryption enforced access control for outsourced data is proposed in [DFJ+07], but it protects the data only the face of an untrusted service provider. Rizvi et al. introduce authorization views enabling specification of access policies using SQL queries on the application level

[RMSR04]. However, this only enforces access control for users.
Key Management. The key management strategies introduced in [AFB05, DDF+05] can extend our access control model. ENKI can benefit from these strategies by a reduced number of keys a user has to store.

2.2 ESCUDO-CLOUD Innovation

We design, implement, and evaluate ENKI, a system that securely processes relational operations over encrypted, access restricted relations. Its approach is to encrypt data with different access rights with different keys and to introduce techniques to handle query processing over data encrypted with multiple encryption keys. The support of query processing over access controlled encrypted data presents two major challenges: The first challenge is the mapping of any complex access control structure required in a multi user scenario to an encryption enforced access control model which still allows query execution. The second challenge is to efficiently execute a range of queries while minimizing the revealed information and the amount of computations on the client. We tackle these challenges using two ideas: First, we introduce a new model for encryption based access control in Section 2.5 which defines access control restrictions on the level of attribute values and applies encryption as a relational operation to enforce the access restrictions on a relational algebra.
Second, we present different techniques to support the execution of relational operations in multi user mode. These include a rewriting strategy to adapt relational operations over data encrypted with different keys in Section 2.6.1, a new privacy-preserving method for join, set difference, and count distinct in multi user mode in Section 2.6.2, and a post-processing step on the client saving computational effort on the client while preserving the confidentiality of the data in Section 2.6.3. Our results are subsumed in a multi user algorithm introduced in Section 2.6.4.
Previous work only focuses on the access control mechanism [DFJ+07] or the key management [AFB05, DDF+05]. Thus, the achieved ESCUDO-CLOUD innovations are:

- Our formulated system builds on previous work in encrypted query processing for a single user as described in [HIML02, DDJ+03, AES03, CDF+09, PRZB11], but is the first system that efficiently supports queries over data encrypted with different keys. Existing approaches only support query processing with multiple keys for searchable encryption which allows to check if an encrypted value matches a token [YBDD09, PZ13, ARCI13] or if there is no shared data [PRZB11].

- We overcome the limitations of current approaches for multiple users offer either limited functionality [YBDD09, PRZB11, PZ13, ARCI13] or expose confidential information to the database server [Ora10].

Part of the work of this chapter was performed jointly with UNIMI and has been published in [HKD15].

2.3 Introduction

Outsourcing an application's database backend offers efficient resource management and low maintenance costs, but exposes outsourced data to a service provider. To ensure data confiden-

tiality, data owners have to prevent unauthorized access while data is stored or processed. Storing data on an untrusted database requires protection measures against curious personnel working for the service provider or outside attackers exploiting software vulnerabilities on the database server. In addition, data owners also have to control data access for their own personnel.
An emerging solution to the problem of untrusted databases is encrypted query processing [SWP00, HIML02, DDJ+03, AES03, AKSX04, CDF+09, YBDD09, BCLO12, PRZB11, PZ13] where queries are executed over encrypted data.
To grant or restrict shared data access to personnel processing unencrypted query results, data owners have to implement additional fine-grained access control mechanisms.
Implementing such a multi user mode using encrypted query processing for a single user operating with one key [SWP00, HIML02, DDJ+03, AES03, AKSX04, CDF+09, BCLO12, PRZB11] combined with an additional authorization step at the application server like [RMSR04] can be compromised: Assume that a user working for the data owner and a service provider's employee collude. If the user knows the decryption key of the data and the employee provides the encrypted data stored in the database, they are able to decrypt all data bypassing the access control mechanisms.
We have implemented ENKI for a SAP HANA database extending HANA's JDBC driver and a client application. Our solution supports most relational operations and aggregation functions. The evaluation of different query types seen in three use cases and in the TPC-C benchmark shows that this range is suitable for real world applications. Our performance evaluation shows that ENKI consumes an average overhead of 36.98% (which is a time penalty of 0.6181 ms on average) for the query execution of queries from the TPC-C benchmark in a two user scenario compared to the single user mode and that the overhead increases modestly in a more complex scenario.

We have implemented ENKI for a SAP HANA database extending HANA's JDBC driver and a client application. Our solution supports most relational operations and aggregation functions. The evaluation of different query types seen in three use cases and in the TPC-C benchmark shows that this range is suitable for real world applications. Our performance evaluation shows that ENKI consumes an average overhead of 36.98% (which is a time penalty of 0.6181 ms on average) for the query execution of queries from the TPC-C benchmark in a two user scenario compared to the single user mode and that the overhead increases modestly in a more complex scenario.

2.4 Overview

Problem Statement. Alice and Bob share a database with two tables *R* and *S*. Assume that user Alice has private access to one tuple and shares access to four other tuples with user Bob in both tables *R* and *S* respectively.
We encrypt tuples of table *R* only accessible for Alice with key *r_a* and tuples of table *S* only accessible for Alice with key *s_a*. Tuples of table *R* accessible for Alice and Bob are encrypted with key *r_ab* and of table *S* with key *s_ab*. Alice knows keys *r_a*, *r_ab*, *s_a*, and *s_ab* and Bob knows keys *r_ab* and *s_ab*.
Assume Alice issues an equal join on tables *R* and *S*. Therefore, the database executes a cartesian product on all tuples of *R* and *S* that Alice is allowed to access and proxy re-encrypts these tuples to check the equal condition. Current proxy re-encryption protocols for deterministic encryption schemes [PRZB11],[PH78] cannot adhere the access restrictions while applied to the tuples: they reveal private information. To illustrate the problem, consider the proxy re-encryption of keys *a* and *ab* to a new key *c* denoted as $a \sim c$ and $ab \sim c$. Existing protocols are symmetric and transitive

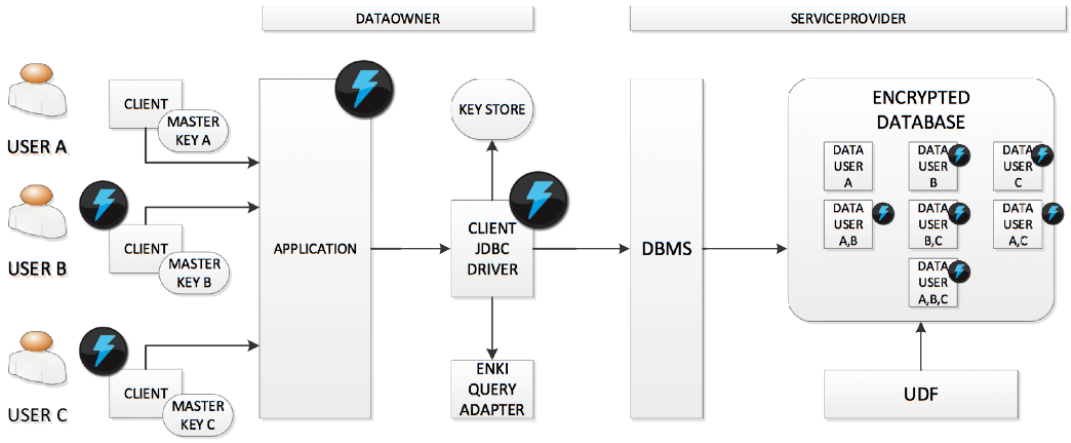

Figure 2.1: ENKI's architecture and threat model

such that a proxy re-encryption $a \sim c \sim ab$ exists. Therefore, Bob can proxy re-encrypt all data encrypted with Alice's key a to their shared key ab. This circumvents the defined access restrictions as the proxy re-encryption reveals information exclusively accessible by Alice.

Architecture. Figure 2.1 shows ENKI's overall architecture and the involved entities. These involved entities are the data owner who also maintains the application and the service provider who operates the database. There are also different users (denoted in Figure 2.1 as User A, User B, User C) which are personnel working for the data owner.

A user accesses the database backed application with a client which issues queries via JDBC driver to the database backend. We extended the JDBC driver with ENKI Query Adapter to rewrite an incoming query with minimal effort to be processable in the multi user mode and modified the clients to post-process the returned query results.

To execute a rewritten query on a database where all tuples are encrypted, the predicates of this query must be encrypted too. Based on access policies, users are acquainted with the necessary encryption keys. These keys are stored encrypted in a key store. If a user logs in, she hands over her masterkey to decrypt her encryption keys stored in the key store. Using the stored and decrypted encryption keys, ENKI Query Adapter encrypts the rewritten query. The database management system (DBMS) receives the rewritten, encrypted query and executes it on the encrypted database. The encrypted query result is returned to the JDBC driver where it is decrypted by ENKI Query Adapter before it is post-processed on the client. Note that keys stored in the key store cannot be decrypted if their respective users are logged out.

DBMS and database stay unmodified. User-defined functions (UDFs) perform cryptographic operations like our new privacy-preserving proxy re-encryption introduced in Subsection 2.6.2.

Threat. Our threat model assumes that an attacker has compromised application and database server. The attacks are depicted with flashes in Figure 2.1. We assume that the attacker is passive: she can read all information stored on the database, but does not manipulate the stored data or issued queries. The attacker learns the encryption keys of all compromised users at the time

Access Control Matrix of Relation R

User	t_1	t_2	t_3	t_4	t_5
Alice	0	1	1	1	1
Bob	1	1	0	1	0

Figure 2.2: Access Control Matrix for a relation R with five tuples $t_1, \ldots, t_5$ and two users Alice and Bob.

of the attack. Acquainted with their masterkeys and their encryption keys, the attacker is able to read all data of these compromised users stored on the database. In particular, the attacker can access data shared with other uncompromised users. ENKI offers confidentiality guarantees for non-compromised users during such an attack: the attacker cannot learn their private data i.e. data which are not shared with compromised users.

2.5 Encryption-Based Access Control on a Relational Algebra

This section presents a new model how to specify access rights on attribute values of relations and how to enforce them cryptographically.

2.5.1 Access Restrictions on Relations

We define access restrictions on attribute values of a relation using an access control matrix. Note that an access control matrix may serve as a base for more enhanced access models exploiting role-based access control [SHV01].

Let $\mathscr{A}$ be an access control matrix where the rows correspond to subjects s and the columns correspond to objects o. Figure 2.2 illustrates an access control matrix with two users, Alice and Bob, as subjects and a relation R containing five tuples, $t_1, \ldots, t_5$, as objects. We denote S as the set of all subjects with $|S| = n$ and O as the set of all objects.

A data owner grants access for an object o to a subject s by setting the entry in the access matrix $\mathscr{A}[s,o]$ to 1. If no access is granted, the entry is set to 0. A column of an access control matrix is a representation of the set of subjects which have access to a an object o. We denote this as *qualified set* QS_o of object o. We assume that each object can be accessed by at least one subject such that there are no zero columns and no empty qualified sets. Consider $QS_{t_4} = \{1,1\}$ in Figure 2.2. This is the qualified set of object t_4 denoting that user Alice and user Bob have access to tuple t_4.

We further name $\mathscr{P}^*(S)$ the power set of all subjects S without the empty set. We denote each of these subsets as $p_i \in \mathscr{P}^*(S)$ for all $i = 1, \ldots, 2^n - 1$ and call it a *user group*. From the access control matrix depicted in Figure 2.2, we derive three user groups:

$$p_1 = \{Alice\} := A$$
$$p_2 = \{Bob\} := B$$
$$p_3 = \{Alice, Bob\} := AB.$$

We define a mapping which assigns each user to the user groups she participates in. For each user s, there is a set of p_j with $j = \{1, \ldots, 2^n - 1\}$ of all user groups the user participates in. This

User Group Mapping

User	User Group
Alice	A
Alice	AB
Bob	B
Bob	AB

Virtual Relation Mapping

User Group	Relation	Virtual Relation
A	R	R_A
B	R	R_B
AB	R	R_AB

Figure 2.3: On the left, User Group Mapping which relates user to usergroup. On the right, Virtual Relation Mapping which relates a pair of relation and user group to a virtual relation.

mapping is called *user group mapping* and can be stored as a relation with the attributes user and user group. Figure 2.3 shows the user group mapping for our example. It has two attributes: User and User Group. It shows that user Alice is member of user groups A and AB and user Bob is member of user groups B and AB.

A qualified set of an object maps to one and only one user group which contains the same set of subjects. So, we group all objects accessible by the same user group and define an *object set* as

$$O(p_i) = \{o | o \in O \wedge QS_o = p_i\}$$

for a user group $p_i \in \mathscr{P}^*(S)$. This is the set of all objects assigned to the same user group. In Figure 2.2, it is

$$O(p_1) = \{t_3, t_5\}$$
$$O(p_2) = \{t_1\}$$
$$O(p_3) = \{t_2, t_4\}.$$

Note that all $O(p_i)$ form a partition over O as each two object sets are pairwise disjoint and the union of all object sets (which are non-empty by definition) is equal to the set of all objects O. We use this resulting partition to divide the underlying relation. This is to store each object set in a separate relation which we call *virtual relation*. A virtual relation indicates that one user group can access all of its tuples. This saves the annotation of a tuple with access information as its insertion in a virtual relation implies that this tuple can be accessed by a certain user group. For n users, each relation is partitioned in a maximum of $2^n - 1$ virtual relations. The total number of tuples does not change as each tuple of a relation is stored in one and only one virtual relation.

We define a mapping which assigns each user group and relation to the virtual relation containing the tuples this user group is granted access to. This mapping is called *virtual relation mapping* and can be stored as a relation with the attributes User Group, Relation, and Virtual Relation. Figure 2.3 shows the virtual relation mapping for the user groups A, B, and AB and the relation R. The pair user group A and relation R is mapped to virtual relation R_A, the pair user group B and relation R is mapped to virtual relation R_B, and the pair of user group AB and relation R is mapped to virtual relation R_{AB}. The data owner specifies and maintains the user group mapping and the virtual relation mapping.

2.5.2 Encryption as Relational Operation

We now define encryption as a relational operation and show how to enforce the previously defined access restrictions.

Definition 2.5.1. *Consider a relation $R = R(A_1,\ldots,A_n)$ with $A_1,\ldots,A_n$ attributes. It contains tuples $t_k = (t_{1_k},\ldots,t_{n_k})$ with $1,\ldots,n$ the number of attributes and $k = 1,\ldots,j$ the cardinality. An* encryption $\kappa_z(A_i)$ *of attribute A_i with key z is defined as*

$$\kappa_z(A_i) := \{\kappa_z(t_{i_k}) | t_{i_k} \in A_i \text{ for all } k = 1,\ldots,j\}$$

with t_{i_k} the attribute values of A_i for all $k = 1,\ldots,j$. The encryption of relation $R = R(A_1,\ldots,A_n)$ is the encryption of its attributes $A_1,\ldots,A_n$ and their attribute values $t_{1_k},\ldots,t_{n_k}$ for all $k = 1,\ldots,j$ as

$$\begin{aligned}\kappa_z(R) &:= R(\kappa_z(A_1),\ldots,\kappa_z(A_n)) \\ &= \{\kappa_z(t_{1_k}),\ldots,\kappa_z(t_{n_k}) | t_{i_k} \in A_i \text{ for all } i = 1,\ldots,n \text{ and for all } k = 1,\ldots,j\}.\end{aligned}$$

Focusing on the query processing over encrypted data, we rely on the adjustable query-based encryption introduced in [PRZB11] which presents onion encryption layers for different classes of computation and thereby allows the execution of any relational operations over one attribute. We formulate the onion encryption layers as the composition of encryption operations over an attribute A_i. It is

$$\begin{aligned}&\text{Onion DET: } \kappa_z^{\text{RND}}(\kappa_z^{\text{DET}}(\kappa_z^{\text{JOIN}}(A_i))) \\ &\text{Onion OPE: } \kappa_z^{\text{RND}}(\kappa_z^{\text{OPE}}(A_i)) \\ &\text{Onion HOM: } \kappa_z^{\text{HOM}}(A_i).\end{aligned}$$

Applying the adjustable query-based encryption strategy allows to dynamically adjust layers of encryption on the DBMS server to support relational operations. In the following sections, we only refer to an encryption scheme as relational operation κ_z with key z relying on the efficient support of encrypted query processing, but omit the details of the onion encryption layer, encryption scheme, or encryption key.

We now use encryption to enforce the access restrictions on tuples of a relation. Consider a relation $R = R(A_1,\ldots,A_n)$ and three user groups A, B, and AB. The data owner splits R into virtual relations R_A, R_B, and R_{AB} such that

$$R_A(A_1,\ldots,A_n) = R_B(A_1,\ldots,A_n) = R_{AB}(A_1,\ldots,A_n).$$

The data owner generates encryption keys for each user group and encrypts the respective virtual relation with this key. She generates key *r_a* for user group A and encrypts R_A as

$$\kappa_{r\_a}(R_A) = \{\kappa_{r\_a}(t) | t \in R_A\}.$$

She generates keys *r_b* and *r_ab* and encrypts R_B and R_{AB} for user groups B and AB respectively. The data owner issues the respective keys to the member of each user group.

2.6 Query Processing over an Encrypted Relational Algebra

Encrypted query processing over a relational operation can be efficiently supported for single user mode [PRZB11]. In particular, this holds for the following primitive and derived relational operations: *selection*, *projection*, *rename*, *cartesian product*, *set union*, *set difference*, *equi join*, and aggregate functions *group by*, *count (distinct)*, *sum*, *average*, *maximum*, *minimum*, and *sort*. The introduction of access restrictions on relations interferes with encrypted query processing in three ways:

First, a relational operation is now executed over (potentially) multiple virtual relations depending on the access rights of the user rather than on one relation. To tackle this problem, we introduce query rewriting strategies in Subsection 2.6.1. Applying these rewriting strategies does not change the application logic: we point out that the user only has to submit the original, unchanged query and its user id. The ENKI Query Adapter rewrites the query and adapts it for encrypted query processing.

Second, proxy re-encryption of virtual relations are necessary to support *count distinct*, *equi-join*, or *set difference* operations on the server. These might lead to a data compromise or a malfunction as a proxy re-encryption might falsely grant or revoke access rights. We present a new privacy-preserving encryption scheme to support proxy re-encryption in a multi user setting in Subsection 2.6.2. It offers proxy re-encryption of attributes or relations encrypted with different keys while preserving the access rights.

Third, some relational operations can only be executed on server-side with significant computational effort, huge storage capacities, or diminishing security. For such cases, we present a client-server split, requiring small data traffic and minimal computational effort on the client while preserving security in Subsection 2.6.3.

All introduced techniques are combined in a multi user algorithm to handle the presence of multiple users described in Subsection 2.6.4. The multi user algorithm takes as input a user id and a query consisting of a combination of relational operations, processes it over virtual relations, and returns the result. It can handle an arbitrary set of users.

To explain the three techniques, we use the small example introduced in Section 2.4 where we part the table R in virtual relations R_A, R_B, and R_{AB} and encrypt them as $\kappa_{r\_a}(R_A)$, $\kappa_{r\_b}(R_B)$, and $\kappa_{r\_ab}(R_{AB})$ with keys *r_a*, *r_b*, *r_ab*. Table S is treated accordingly.

2.6.1 Rewriting Strategies

We introduce rewriting strategies for the relational operations *selection*, *projection*, *rename*, aggregate function *count*, *set union*, and *cartesian product* over encrypted virtual relations. Applying these rewriting strategies allows the straightforward execution of these relational operations over encrypted data. The rest of this subsection presents the rewriting of these relational operations in detail.

Selection. Consider a predicate θ (e.g. $=, <, \leq, >, \geq$) and α, β attributes, constants, or terms of attributes, constants, and data operations. A *selection* $\sigma_{\alpha\theta\beta}(R)$ on relation R issued by user Alice is executed on the encrypted virtual relations $\kappa_{r\_a}(R_A)$ and $\kappa_{r\_ab}(R_{AB})$. The condition $\alpha\theta\beta$ has to be applied on both virtual relations. Therefore, $\alpha\theta\beta$ is encrypted with key *r_a* as $\kappa_{r\_a}(\alpha)\theta\kappa_{r\_a}(\beta)$

and with key *r_ab* as $\kappa_{r\_ab}(\alpha)\theta\kappa_{r\_ab}(\beta)$. It is

$$\begin{aligned}(\sigma_{\alpha\theta\beta}(R), Alice) =& \sigma_{\kappa_{r\_a}(\alpha)\theta\kappa_{r\_a}(\beta)}\kappa_{r\_a}(R_A) \wedge \sigma_{\kappa_{r\_ab}(\alpha)\theta\kappa_{r\_ab}(\beta)}\kappa_{r\_ab}(R_{AB}) \\ =& \{\kappa_{r\_a}(t) | t \in R_A \wedge \kappa_{r\_a}(\alpha)\theta\kappa_{r\_a}(\beta)\} \cup \{\kappa_{r\_ab}(t) | t \in R_{AB} \wedge \kappa_{r\_ab}(\alpha)\theta\kappa_{r\_ab}(\beta)\}.\end{aligned}$$

Projection. Let R' be a relation with

$$R'(A_{i(1)}, \ldots, A_{i(k)}) \subseteq R(A_1, \ldots, A_n)$$

and R'_A and R'_{AB} the respective virtual relations. A *projection* $\pi_\beta(R)$ with attribute list

$$\beta = (A_{i(1)}, \ldots, A_{i(k)}) \subseteq (A_1, \ldots, A_n)$$

on relation R issued by user Alice is executed over the virtual relations $\kappa_{r\_a}(R_A)$ and $\kappa_{r\_ab}(R_{AB})$. Therefore, the attribute list β is encrypted with key *r_a* as

$$\kappa_{r\_a}(\beta) = \kappa_{r\_a}(A_{i(1)}), \ldots, \kappa_{r\_a}(A_{i(k)})$$

and also encrypted with key *r_ab* as

$$\kappa_{r\_ab}(\beta) = \kappa_{r\_ab}(A_{i(1)}), \ldots, \kappa_{r\_ab}(A_{i(k)}).$$

It is

$$\begin{aligned}(\pi_\beta(R), Alice) =& \pi_{\kappa_{r\_a}(\beta)}(\kappa_{r\_a}(R_A)) \cup \pi_{\kappa_{r\_ab}(\beta)}(\kappa_{r\_ab}(R_{AB})) \\ =& \{\kappa_{r\_a}(t) | t \in R'_A\} \cup \{\kappa_{r\_ab}(t) | t \in R'_{AB}\}.\end{aligned}$$

Rename. A *rename* ρ of an attribute $A_i \in R$ to Q issued by Alice is executed on the encrypted virtual relations $\kappa_{r\_a}(R_A)$ and $\kappa_{r\_ab}(R_{AB})$. The new attribute name Q is encrypted with key *r_a* as $\kappa_{r\_a}(Q)$ and with key *r_ab* as $\kappa_{r\_ab}(Q)$ respectively. It replaces the encrypted original attribute name A_i in the virtual relations. It is

$$(\rho_{Q \leftarrow A_i}(R), Alice) = \rho_{\kappa_{r\_a}(Q) \leftarrow \kappa_{r\_a}(A_i)}(\kappa_{r\_a}(R_A)) \cup \rho_{\kappa_{r\_ab}(Q) \leftarrow \kappa_{r\_ab}(A_i)}(\kappa_{r\_ab}(R_{AB})).$$

A *rename* is not persisted.

Count. The aggregate function ${}_\beta\gamma_{Count(A_i)}(R)$ on a relation R issued by Alice is executed on the virtual relations $\kappa_{r\_a}(R_A)$ and $\kappa_{r\_ab}(R_{AB})$. It is

$$({}_\beta\gamma_{Count(A_i)}(R), Alice) = {}_{\kappa_{r\_a}(\beta)}\gamma_{Count(\kappa_{r\_a}(A_i))}\kappa_{r\_a}(R_A) +_{\kappa_{r\_ab}(\beta)} \gamma_{Count(\kappa_{r\_ab}(A_i))}\kappa_{r\_ab}(R_{AB}).$$

with *Count* the aggregate function executed on server-side. It counts the numbers of attribute values of A_i for virtual relations $\kappa_{r\_a}(R_A)$ and $\kappa_{r\_ab}(R_{AB})$ separately and adds these partial results on the server. The output represents the number of attribute values of attribute A_i accessible by Alice.

Set Union. Let relations R and S have the same set of attributes. A set union $R \cup S$ issued by Alice is executed on the virtual relations $\kappa_{r\_a}(R_A)$, $\kappa_{r\_ab}(R_{AB})$, $\kappa_{s\_a}(S_A)$, and $\kappa_{r\_ab}(S_{AB})$. It is

$$\begin{aligned}(R \cup S, Alice) =& \{\kappa_{r\_a}(t) | t \in R_A\} \cup \{\kappa_{r\_ab}(t) | t \in R_{AB}\} \\ & \cup \{\kappa_{s\_a}(t) | t \in S_A\} \cup \{\kappa_{r\_ab}(t) | t \in S_{AB}\}.\end{aligned}$$

Cartesian Product. We denote a tuple r of relation R and a tuple s of relation S. A cartesian product $R \times S$ issued by Alice is executed on the virtual relations $\kappa_{r\_a}(R_A)$, $\kappa_{r\_ab}(R_{AB})$, $\kappa_{s\_a}(S_A)$, and $\kappa_{s\_ab}(S_{AB})$. It is

$$\begin{aligned}(R \times S, Alice) = &\{\kappa_{r\_a}(r)\kappa_{s\_a}(s) \vee \kappa_{r\_a}(r)\kappa_{s\_ab}(s) \vee \kappa_{r\_ab}(r)\kappa_{s\_a}(s) \vee \kappa_{r\_ab}(r)\kappa_{s\_ab}(s) \\ &| r \in (R_A \vee R_{AB}) \wedge s \in (S_A \vee S_{AB})\}.\end{aligned}$$

2.6.2 Proxy Re-Encryption as Relational Operation

Processing the unary operation *count distinct* or the binary operations *equi-join* and *set difference* over (deterministically) encrypted virtual relations requires that these virtual relations are encrypted with the same encryption key as these operations rely on comparisons which are only feasible if the same encryption key is used. Our goal is to proxy re-encrypt virtual relations on the database server so that all queried attributes share the same encryption key while preserving data confidentiality. To formalize this approach, we define proxy re-encryption as a relational operation.

Definition 2.6.1. *A* proxy re-encryption *alters an attribute $\kappa_z(A_i)$ encrypted with key z to enable its decryption with another key y. We define a proxy re-encryption $\chi_y(\kappa_z(A_i))$ of attribute A_i as*

$$\begin{aligned}\chi_y(\kappa_z(A_i)) :=&\{\chi_y(\kappa_z(t_{i_k}))|t_{i_k} \in A_i \text{ for all } k = 1,\ldots,j\} \\ =&\{\kappa_y(t_{i_k})|t_{i_k} \in A_i \text{ for all } k = 1,\ldots,j\} \\ =&\kappa_y(A_i)\end{aligned}$$

with t_{i_k} the attribute values of A_i for all $k = 1,\ldots,j$.
The proxy re-encryption of a relation $\kappa_z(R)$ is the proxy re-encryption of all attributes. It is

$$\begin{aligned}\chi_y(\kappa_z(R)) :&= R(\chi_y(\kappa_z(A_1)),\ldots,\chi_y(\kappa_z(A_n))) \\ &= R(\kappa_y(A_1),\ldots,\kappa_y(A_n)) \\ &= \kappa_y(R).\end{aligned}$$

Definition 2.6.2. *A proxy re-encryption is called* symmetric *if*

$$\chi_b(\kappa_a(A_i)) = \kappa_b(A_i) \Leftrightarrow \chi_a(\kappa_b(A_i)) = \kappa_a(A_i).$$

Definition 2.6.3. *A proxy re-encryption is called* transitive *if*

$$\chi_b(\kappa_a(A_i)) = \kappa_b(A_i) \wedge \chi_c(\kappa_b(A_i)) = \kappa_c(A_i) \Rightarrow \chi_c(\kappa_a(A_i)) = \kappa_c(A_i).$$

A symmetric and transitive proxy re-encryption scheme ensures privacy-preserving computations on the database server for the single user mode [PRZB11]. However, if you recall the problem statement in Section 2.4, it does not preserve data confidentiality in the face of multiple users as its application leads to a data compromise. This motivates the introduction of a new non-symmetric and non-transitive proxy re-encryption scheme called DETPRE as the cryptographic primitive for *count distinct*, *equi-joins*, and *set differences* in multi user mode.

Definition 2.6.4. *A deterministic proxy re-encryption scheme is a tuple of algorithms ParamGen, KeyGen, Enc, Token, Pre such that:*

- ***Parameter Generation.*** *The probabilistic polynomial time algorithm ParamGen takes as input the security parameter λ and outputs system parameters params:*

$$params \leftarrow ParamGen(1^{\lambda})$$

- ***Key Generation.*** *The probabilistic polynomial time algorithm KeyGen takes as input the security parameter λ and outputs a key k_i:*

$$k_i \leftarrow KeyGen(1^{\lambda})$$

- ***Encryption.*** *The deterministic polynomial time algorithm Enc takes as input a plaintext m and key k_i and outputs a ciphertext:*

$$C = Enc(m, k_i)$$

- ***Token.*** *The deterministic polynomial time algorithm Token takes as input two keys k_i, k_j and outputs a token to proxy re-encrypt k_i to k_j:*

$$T = Token(k_i, k_j)$$

- ***Proxy Re-Encryption.*** *The deterministic polynomial time algorithm Pre takes as input a ciphertext C and a token T and outputs a ciphertext C':*

$$C' = Pre(C, T)$$

We now present our deterministic proxy re-encryption scheme DETPRE by specifying each algorithm.

ParamGen. Given a security parameter λ, *ParamGen* works as follows: generate a prime p and two groups $\mathbb{G}_1, \mathbb{G}_2$ of order p, and a bilinear, non-degenerated, computable map $e : \mathbb{G}_1 \times \mathbb{G}_1 \longrightarrow \mathbb{G}_2$. Choose a generator $G \in \mathbb{G}_1$ uniformly at random.

KeyGen. Choose $k_i \in \mathbb{Z}_p$ uniformly at random.

Enc. To encrypt plaintext m with key k_i as ciphertext C compute

$$C = G^{mk_i} \in \mathbb{G}_1.$$

Token. To generate a token T that proxy re-encrypt a ciphertext encrypted with key k_i to be encrypted with key k_j, compute

$$T = G^{\frac{k_j}{k_i}} \in \mathbb{G}_1.$$

Pre. To proxy re-encrypt a ciphertext C encrypted with key k_i to a ciphertext C' encrypted with key k_j, compute

$$\begin{aligned} C' = e(C,T) =& e(G^{mk_i}, G^{\frac{k_j}{k_i}}) = e(G,G)^{mk_i \frac{k_j}{k_i}} = e(G,G)^{mk_j} \\ =& g^{mk_j} \in \mathbb{G}_2. \end{aligned}$$

DETPRE is single-hop meaning that a ciphertext can only be proxy re-encrypted once. This restricts its usability as the key of a once proxy re-encrypted ciphertext is persisted. Therefore, we propose the following strategy which allows to benefit from the application of DETPRE while maintaining its re-usablity:

1. We encrypt all attribute values using the algorithm *Enc*. These encrypted attribute values are called *base values*.

2. If a proxy re-encryption is required, we use the algorithm *Pre* and proxy re-encrypt the base values with a temporary key c. The proxy re-encrypted results are called *PreDet values*.

3. We store the *PreDet values* temporarily as a concatenation to the base values and use them to process a relational operation.

4. After the user logs out, the *PreDet values* are deleted.

We now describe our adversary model to informally explain the security guarantees our proxy re-encryption schemes provides. We consider a passive adversary, i.e., he can read all encrypted attribute values of all users, but does not modify them. We assume that the adversary has also compromised the application and its database proxy and observes executed operations. In particular, if a user is compromised during the attack, the adversary learns the masterkey, the encryption keys stored in the key store, and the used tokens of a user. Our goal is to prevent an adversary from using this information to learn private data of non-compromised users. Therefore, we assume a number of users distributed to n user groups with each user group endowed with an encryption key $d_1, \ldots, d_n$ which are kept private.

We allow the adversary to compromise all but one encryption key. The adversary could have learned these keys as a result of a collusion between service provider and personnel working for the data owner. Therefore, she has access to keys $\{d_1, \ldots, d_{n-1}\}$ but not to key d_n. It implies that the adversary can decrypt all database entries encrypted with keys $d_1, \ldots, d_{n-1}$. In particular, if d_n is the private key of a single user i.e. of a user group with only one member and this user also participates in additional user groups with compromised members, then the adversary can decrypt all tuples encrypted for these user groups but cannot decrypt the tuples encrypted with d_n.

The adversary can compute or learn tokens $Token(d_{i^*}, d_i)$ for all compromised keys $d_{i^*} \in \{d_1, \ldots, d_{n-1}\}$ to be proxy re-encrypted to an arbitrary key d_i. Thereby, he can proxy re-encrypt the database entries encrypted with the compromised encryption keys $d_1, \ldots, d_{n-1}$. The database entries encrypted with key d_n are not compromised and the adversary cannot access these database entries. He also cannot compute or learn tokens $Token(d_n, d_i)$ which proxy re-encrypt database entries encrypted with key d_n to an arbitrary key d_i. However, he can compute tokens $Token(d_i, d_n)$ such that a database entry encrypted with an arbitrary key d_i can be proxy re-encrypted to key d_n. Given all these information, the adversary should not be able to proxy re-encrypt an attribute value encrypted with encryption key d_n to another key. We refer to this property as non-reversion.

Next, we study a security game to formally define the described security guarantees and proof our claims based on a known hardness assumption. Let $\mathscr{A}$ be a probabilistic time adversary

modeled as described above. Let $\mathscr{C}$ be the challenger. Then consider the following security game for security parameter λ:

Setup. $\mathscr{C}$ takes security parameter λ, runs algorithm *ParamGen*, and returns the system parameters params to $\mathscr{A}$. $\mathscr{C}$ also runs algorithm *KeyGen* and outputs keys $d_1,\ldots,d_n$. $\mathscr{C}$ sends $d_1,\ldots,d_{n-1}$ to $\mathscr{A}$ and keeps d_n as a secret. $\mathscr{C}$ runs algorithm *Token* and outputs $Token(d_i,d_j)$ for all $i,j=1,\ldots n$ that allow a database entry encrypted with key d_i to be proxy re-encrypted to key d_j. $\mathscr{C}$ sends Token(d_{i^*},d_i) with $i^*=1,\ldots,n-1$ and $i=1,\ldots,n$ to $\mathscr{A}$.

Phase 1. $\mathscr{A}$ performs actions $q_1,\ldots,q_m$ where q_i is one of the following type:

Enc $\mathscr{A}$ chooses an arbitrary value s and runs algorithm *Enc* to encrypt it with key d_{i^*} with $i^*=1,\ldots,n-1$. This is as he knows the keys $d_1,\ldots,d_{n-1}$ of all compromised users. Although he does not know key d_n, he can encrypt an arbitrary value with key d_n as he can compute

$$Token(d_{i^*},d_n)d_{i^*} = (G^{\frac{d_n}{d_{i^*}}})^{d_i^*} = G^{d_n}$$

given a $Token(d_{i^*},d_n)$ and a key d_{i^*} to encrypt a chosen value s under the uncompromised key d_n as G^{md_n},

Pre $\mathscr{A}$ runs algorithm *Pre* to proxy re-encrypt a ciphertext $C=G^{d_im}$ with a token $Token(d_i,d_j)=G^{\frac{d_j}{d_i}}$ as

$$\begin{aligned} Pre(C,T) &= e(G^{d_im},G^{\frac{d_j}{d_i}}) = e(G^{d_im},G^{\frac{d_j}{d_i}}) \\ &= e(G,G)^{d_im\frac{d_j}{d_i}} = e(G,G)^{d_jm} = g^{d_jm}. \end{aligned}$$

Challenge. $\mathscr{A}$ chooses a key d and sends it to $\mathscr{C}$. $\mathscr{C}$ picks a random value r and encrypts it with key d_n as G^{rd_n}. It sends G^{rd_n} to $\mathscr{A}$ and ask her to proxy re-encrypt $C=G^{rd_n}$ to key d as C_d.

Phase 2. $\mathscr{A}$ performs further actions $q_{m+1},\ldots,q_n$ of the types described above.

Guess. $\mathscr{A}$ outputs its guess C'_d and wins the security game if and only if $C_d=C'_d$.

The advantage of $\mathscr{A}$ in the security game is defined as

$$Pr[C_d=C'_d]=\varepsilon.$$

Definition 2.6.5. DETPRE *is secure if for a probabilistic time adversary* $\mathscr{A}$

$$Pr[\mathscr{A}\ wins\ the\ Security\ Game\ of\ \text{DETPRE}]$$

is negligible in λ.

Assumption 1. *The l-Bilinear Diffie-Hellman Inversion Assumption (l-BDHI) holds if for any probabilistic polynomial time (PPT) adversary* $\mathscr{A}$ *the probability that* $\mathscr{A}$ *on input*

$$G^a, G^{a^2}, G^{a^3}, \ldots, G^{a^l}$$

outputs W such that $W=g^{\frac{1}{a}}$ *is negligible in security parameter* λ *[Ver13].*

Theorem 1. *If the L-BDHI assumption holds, then our proxy re-encryption is secure.*

Proof. Assuming that an adversary can solve the described security game correctly, we construct a polynomial time algorithm which can solve the underlying problem of the l-Bilinear Diffie-Hellman Inversion assumption. This algorithm receives an instance of the l-BDHI problem with

$$G^a, G^{a^2}, \ldots, G^{a^l} \in \mathbb{G}_1$$

and has to compute $e(G,G)^{\frac{1}{a}} = g^{\frac{1}{a}} \in \mathbb{G}_2$.

Setup. Receive an instance of the l-BDHI problem as

$$p, e, \mathbb{G}_1, \mathbb{G}_2, G, g, G^a, G^{a^2}, \ldots, G^{a^l}$$

Choose $d_i \in \mathbb{Z}_p$ uniformly at random. Run algorithm *Token* to compute $Token(d_i, d_j)$ with $i, j = 1, \ldots, n$. Send system parameters $p, \mathbb{G}_1, \mathbb{G}_2, e, G, g$, encryption keys $d_1, \ldots, d_{n-1}$, and tokens $Token(d_{i^*}, d_i)$ with $i^* = 1, \ldots, n-1$ and $i = 1, \ldots, n$ to $\mathscr{A}$.

Phase 1. $\mathscr{A}$ performs the following actions:

Enc. $\mathscr{A}$ runs algorithm *Enc* to encrypt arbitrary messages m with keys $d_1, \ldots, d_{n-1}$. To encrypt message m with encryption key d_n, which is not known to $\mathscr{A}$, the adversary exploits its knowledge of encryption keys $d_1, \ldots, d_{n-1}$ and tokens $Token(d_{i^*}, d_n)$ to compute

$$G^{\frac{d_n}{d_{i^*}} d_{i^*}} = G^{d_n}.$$

Using this result, $\mathscr{A}$ computes G^{md_n}.

Pre. *Adv* runs algorithm *Pre* to proxy re-encrypt ciphertext C encrypted with key d_{i^*} with $i^* = 1, \ldots, n-1$ to be encrypted with key d_i with $i = 1, \ldots, n$.

Challenge. $\mathscr{A}$ chooses a key $d \notin \{d_1, \ldots, d_n\}$ and sends it to $\mathscr{C}$. $\mathscr{C}$ picks a valid ciphertext as

$$C = \mathrm{Enc}(m,k) = G^{\tilde{m}a} = G^r$$

and sends $C = G^r$ to $\mathscr{A}$. $\mathscr{C}$ asks her to guess the proxy re-encryption of C to key d as

$$V = g^{\frac{r}{a} d}.$$

Phase 2. $\mathscr{A}$ performs further actions as described above.

Guess. $\mathscr{A}$ returns its guess for V as V' to $\mathscr{C}$. $\mathscr{C}$ computes

$$W = V'^{\frac{1}{rd}}$$

to solve the instance of the l-BDHI problem as

$$W = V'^{\frac{1}{rd}} = (g^{\frac{r}{a} d})^{\frac{1}{rd}} = g^{\frac{1}{a}}.$$

The probability that this algorithm solves the l-BDHI problem is the same as the advantage of the adversary in the security game. It is

$$Pr[V = V'] = \varepsilon.$$

If the l-BDHI assumption holds, this advantage can only be negligible. Therefore, the adversary can only achieve this attack with a negligible advantage. □ □

The remaining of this section presents the proxy re-encryption and rewriting strategies to execute the relational operations *count distinct*, *set difference*, and *equi-join*.

Count Distinct. The aggregate function count distinct

$$_{\beta}\gamma_{CountDistinct(A_i)}(R)$$

on a relation R issued by Alice is executed on the virtual relations $\kappa_{r\_a}(R_A)$ and $\kappa_{r\_ab}(R_{AB})$. As these virtual relations are encrypted with different keys, it is not possible to apply a count distinct. So, we adjust the key of both virtual relations to key c. It is

$$\begin{aligned}\chi_c(\kappa_{r\_a}(R_A)) &= R_A(\chi_c(\kappa_{r\_a}(A_1)),\ldots,\chi_c(\kappa_{r\_a}(A_n)))\\ &= R_A(\kappa_c(A_1),\ldots,\kappa_c(A_n))\\ &= \kappa_c(R_A)\end{aligned}$$

and $\chi_c(\kappa_{r\_ab}(R_{AB})) = \kappa_c(R_{AB})$ respectively. The aggregate function count distinct is then computed as

$$(_{\beta}\gamma_{CountDistinct(A_i)}(R), Alice) =_{\kappa_c(\beta)} \gamma_{CountDistinct(\kappa_c(A_i))}(\kappa_c(R_A) \cup \kappa_c(R_{AB})).$$

It counts the numbers of different attribute values of A_i of all relations accessible by user Alice.

Set Difference. Let relations R and S have the same set of attributes. A set difference $R \backslash S$ of relation S in relation R issued by Alice is executed on the virtual relations $\kappa_{r\_a}(R_A)$, $\kappa_{r\_ab}(R_{AB})$, $\kappa_{s\_a}(S_A)$, and $\kappa_{s\_ab}(S_{AB})$. As these virtual relations are encrypted with different keys, it is not possible to apply a set difference. Therefore, we adjust the keys of all virtual relations to key c. It is

$$\begin{aligned}\chi_c(\kappa_{r\_a}(R_A)) &= R_A(\chi_c(\kappa_{r\_a}(A_1)),\ldots,\chi_c(\kappa_{r\_a}(A_n)))\\ &= R_A(\kappa_c(A_1),\ldots,\kappa_c(A_n))\\ &= \kappa_c(R_A)\end{aligned}$$

and $\kappa_c(R_{AB})$, $\kappa_c(S_A)$, and $\kappa_c(S_{AB})$ respectively. Then, we apply the set difference on the proxy re-encrypted virtual relations. It is

$$R \backslash S = \{\kappa_c(t) | t \in (R_A \vee R_{AB}) \wedge t \notin (S_A \vee S_{AB})\}.$$

Equi-Join. An *equi-join* issued by user Alice between two relations

$$R = R(A_1, \ldots, A_n) \text{ and } S = S(B_1, \ldots, B_m)$$

on their respective attributes A_i and B_j is executed on the virtual relations $\kappa_{r\_a}(R_A)$, $\kappa_{r\_ab}(R_{AB})$, $\kappa_{s\_a}(S_A)$, and $\kappa_{s\_ab}(S_{AB})$. However, the attributes A_i and B_j are encrypted with different keys within the relations (A_i is encrypted with key *r_a* in relation R_A and with key *r_ab* in relation R_{AB} and B_j is encrypted with key *s_a* in relation S_A and with key *s_ab* in relation S_{AB}). To apply the condition $A_i = B_j$ on the accessible subsets of R and S, we have to adjust the key of all virtual relations and the condition with a shared encryption key. It is

$$\chi_c(\kappa_{r\_a}(R_A)) = \kappa_c(R_A)$$

and the keys of $\kappa_c(R_{AB})$, $\kappa_c(S_A)$, and $\kappa_c(S_{AB})$ are adjusted respectively.

We encrypt the condition $A_i = B_j$ as $\kappa_c(A_i) = \kappa_c(B_j)$. As now all involved relations are encrypted with the same key c, we can apply these encrypted condition as follows

$$(R \bowtie_{A_i=B_j} S, Alice) = \{\kappa_c(r)\kappa_c(s) | r \in (R_A \vee R_{AB}) \wedge s \in (S_A \vee S_{AB}) \wedge \kappa_c(r_i)\theta\kappa_c(s_j)\}.$$

2.6.3 Client-Server Split

Aggregate functions *count*, *count distinct*, *group by*, *sum*, *average*, *maximum*, *minimum*, and *sort* compute key figures over a whole relation. The encrypted processing of aggregation results is supported on the server in single user mode [PRZB11]. In 2.6.1 and 2.6.2, we introduced the server-side execution of *count* and *count distinct* in multi user mode.

Now, we explain the execution of the rest of these aggregate functions. Introducing virtual relations to specify access restrictions, aggregate functions cannot be executed over the whole relation as this relation is split in different virtual relations encrypted with different keys. In order to evaluate an aggregate function, a user has to process the aggregate function over all virtual relations she is allowed to access. These virtual relations are encrypted with different encryption keys. Typically, the evaluation of aggregation results requires that all invoked virtual relations are encrypted with a shared encryption key.

One possible solution is a proxy re-encryption on the server to compute the aggregation results. Such proxy re-encryption schemes must be suitable for the encryption scheme required by the aggregate function. Unfortunately, some can be hard to construct [PRZB11] while others require notable computational effort and execution time [GH11].

Another naive solution is to process the aggregate functions on the client. This generates significant data traffic as all data have to be transferred to the client. On the client, this data consumes storage capacity and computational power in order to evaluate the aggregate function.

This in mind, we opt for a client-server split where a significant amount of computational effort is executed over encrypted data and small encrypted partial result sets are issued to the client where they are decrypted and further processed to receive the final result. Therefore, we split the execution of these aggregate functions between server and client as follows:

- On the server: Computation of the encrypted results for each virtual relation. These are the *partial results*.
- On the client: Decryption of the partial results and computation of a function $\mathscr{F}_{Agg}$ which takes as input the unencrypted partial results and computes the *final result* depending on the underlying aggregate function.

To illustrate this approach, consider an aggregate function $F(A_i)$ which computes *maximum*, *minimum*, *average*, *sum*, or *sort* over an attribute A_i. Let $\beta = (A_1, \ldots, A_k)$ be an attribute list to *group* the results. If $\beta = \emptyset$, then there is no group-by function defined. An aggregate function ${}_{\beta}\gamma_{F(A_i)}(R)$ on a relation R issued by Alice is executed over the virtual relations R_A and R_{AB}. Therefore, the attribute list β is encrypted with key *r_a* as $\kappa_a(\beta)$ and with key *r_ab* as $\kappa_{r\_ab}(\beta)$. The function $F(A_i)$ is also encrypted with key *r_a* as $F(\kappa_a(A_i))$ and with key *ab* as $F(\kappa_{r\_ab}(A_i))$. We compute the partial result for virtual relation R_A on the server as

$$\kappa_{r\_a}(Res(R_A)) = {}_{\kappa_{r\_a}(\beta)}\ \gamma_{F(\kappa_{r\_a}(A_i))}\, \kappa_{r\_a}(R_A)$$

and the partial result for virtual relation R_{AB} as

$$\kappa_{r\_ab}(Res(R_{AB})) = {}_{\kappa_{r\_ab}(\beta)}\ \gamma_{F(\kappa_{r\_ab}(A_i))}\, \kappa_{r\_ab}(R_{AB})$$

The partial results $\kappa_{r\_a}(Res(R_A))$ and $\kappa_{r\_ab}(Res(R_{AB}))$ are sent to the client where they are decrypted. On the client, we compute the function $\mathscr{F}_{Agg}$ which takes as input the unencrypted partial results. It is

$$\mathscr{F}_{Agg} = Agg(Res(R_A), Res(R_{AB}))$$

the final result.

Depending on the underlying aggregate function, we define $\mathscr{F}_{Agg}$ differently. We describe this in the rest of this subsection.

Maximum/Minimum. It is

$$\begin{aligned} Res(R_A) &= Max(R_A) \\ Res(R_{AB}) &= Max(R_{AB}) \end{aligned}$$

and we compute $\mathscr{F}_{Agg} = \mathscr{F}_{Max}$ as

$$\mathscr{F}_{Max} = Max(Max(R_A), Max(R_{AB})).$$

This also holds for the computation of the minimum.

Sum. It is $Res(R_A) = Sum(R_A)$ and $Res(R_{AB}) = Sum(R_{AB})$ and we compute $\mathscr{F}_{Agg} = \mathscr{F}_{Sum}$ on the client as

$$\mathscr{F}_{Sum} = Sum(R_A) + Sum(R_{AB}).$$

Average. The aggregate function *average* is replaced by the aggregate functions *sum* and *count*. This provides the partial results

$$\begin{aligned} Res(R_A) &= \{Sum(R_A), Count(R_A)\} \\ Res(R_{AB}) &= \{Sum(R_{AB}), Count(R_{AB})\}. \end{aligned}$$

We compute $\mathscr{F}_{Agg} = \mathscr{F}_{Avg}$ on the client as

$$\mathscr{F}_{Avg} = \frac{Sum(R_A) + Sum(R_{AB})}{Count(R_A) + Count(R_{AB})}.$$

Sort. It is $Res(R_A) = Sort(R_A)$ and $Res(R_{AB}) = Sort(R_{AB})$, each a sorted list. We compute $\mathscr{F}_{Agg} = \mathscr{F}_{Sort}$ on the client as

$$\mathscr{F}_{Sort} = Merge\_sorted\_lists(Sort(R_A), Sort(R_{AB})).$$

Group By. The aggregate function *group by* provides as partial results the grouped results for virtual relations R_A and R_{AB}. There are

$$Res(R_A) = \{Agg(R_A) \text{ grouped by } R_A.A_i\}$$
$$Res(R_{AB}) = \{Agg(R_{AB}) \text{ grouped by } R_{AB}.A_i\}.$$

On the client, we process the partial results Res(R_A) and Res(R_{AB}) as follows. If $R_A.A_i = R_{AB}.A_i$, we merge these groups of R_A and R_{AB} and include it in the final result. If $R_A.A_i \neq R_{AB}.A_i$, we overtake the partial result in the final result.

2.6.4 Multi User Algorithm

We apply these introduced techniques and present a multi user algorithm allowing a user to execute a query (i.e. a combination of the presented relational operations) over a set of access restricted relations.

It takes as input a user id and an unencrypted query and returns the final result of the query as output. The user id is an identifier unique for each user. A query is a combination of relational operations over one or more relations. The final result is the decrypted result of the query.

As before consider a relation R with attributes $A_1, \ldots, A_n$ and a relation S with attributes $B_1, \ldots, B_m$. The data owner splits relation R in virtual relations $R_1, \ldots, R_k$ encrypted with keys $v_1, \ldots, v_k$ respectively. She also splits relation S in virtual relations $S_1, \ldots, S_l$ which are encrypted with keys $w_1, \ldots, w_l$ respectively. The data owner handles n user. Each user is equipped with a user id. The data owner defines the user group mapping where each user id is related to its user groups. She also defines the virtual relation mapping where each pair of user group and relation is assigned to a virtual relation. Here, we focus on a user which is member in $i + j$ different user groups. For relation R, the user is member in user groups which are assigned to virtual relations $\kappa_{v_1}(R_1), \ldots, \kappa_{v_i}(R_i)$ and for relation S, the user is member in user groups which are assigned to virtual relations $\kappa_{w_1}(S_1), \ldots, \kappa_{w_j}(S_j)$.

With respect to the specific query, the multi user algorithm requires six steps:

1. Look Up: Determine the required virtual relations given query and user id and check if the required attributes are encrypted with the necessary encryption layer.

2. Proxy Re-Encryption: Initiate a proxy re-encryption if the query contains *count distinct*, *equi join*, or *set difference*.

3. Query Encryption: Encrypt all elements of the rewritten query like attributes, conditions, attribute list.

4. Query Rewriting: Adapt the query to be applied on virtual relations. Therefore, we present a query rewriting algorithm in Algorithm 1.

5. Server-side Execution: Process the rewritten query over encrypted data and return the encrypted results to the client.

6. Client-side Execution: Decrypt the returned results and do further processing if required.

We now discuss the details of these steps given an arbitrary query and a user id.

Look Up. ENKI Query Adapter checks the user group mapping to identify all user groups containing this user id. Given these user groups and the relation(s) contained in the query, it determines all virtual relations for each pair of user group and relation(s) in the virtual relation mapping.

Proxy Re-Encryption. If the query contains *equi-join*, a *set difference*, or a *count distinct*, an UDF adjusts the keys of all involved relations to a temporary key c.

Query Encryption. ENKI Query Adapter encrypts all attributes. If there exists a condition $a\theta b$ with a, b attributes, then it encrypts the attributes a and b of relations accessible by this user with the same key c. If a is an attribute and b is a constant, it encrypts $a\theta b$ with all keys $v_1,\ldots,v_i$. The attribute list $\beta = A_{i_{(1)}},\ldots,A_{i_{(k)}}$ of a projection or an aggregate function is encrypted with the keys $v_1,\ldots,v_i$ of the accessible relations $R_1,\ldots,R_i$ of this user. This step encrypts the aggregate function $F(A_i)$ with key $v_1,\ldots,v_i$.

Query Rewriting. ENKI Query Adapter includes a query rewriting algorithm that modifies the original query to be executable over the required virtual relations

$$\kappa_{v_1}(R_1),\ldots,\kappa_{v_i}R_i \text{ and } \kappa_{w_1}(S_1),\ldots,\kappa_{w_j}(S_j).$$

We describe the details of this algorithm in Algorithm 1. It takes as input the query Q which can contain one or more unary or binary operations over relation R (and relation S). It returns a rewritten query sQ to be executed on the server and in some cases also a rewritten query cQ to be executed on the client.

Server-side Execution. The server executes the rewritten, encrypted query sQ and returns the encrypted results to the client. If client-side processing is necessary, the server also returns a query cQ.

Client-side Execution. The client receives the encrypted results and decrypts them. If the client does not receive a query cQ, the query processing is finished. If the client receives a query cQ, it executes the query over the decrypted partial results receiving the final result.

2.7 Key Management and Dynamic Access Control Policies

ENKI enforces access policies through selective encryption leading to different keys for each user. However, access policies (and thereby keys) might change. This is that the data owner grants access rights to new users or revokes access rights from others. Adding or deleting users of a user group can be formalized as changes in a user hierarchy.

Definition 2.7.1. *Given the set of users $S = \{s_1,\ldots,s_n\}$ a* user hierarchy *$\mathscr{U}$ is a pair $(\mathscr{P}^*(S),\prec)$ where $\mathscr{P}^*(S)$ is the powerset without the empty set of S and $\prec$ is a partial order such that for all sets of users $p_i,p_j \in \mathscr{P}^*(S)$, $p_i \prec p_j$ if $p_j \subseteq p_i$ for all $i,j = \{1,\ldots 2^{n-1}\}$.*

Algorithm 1 Query Rewriting Algorithm

Require: $\kappa_{v_1}(R_1),\ldots,\kappa_{v_i}R_i$, $(\kappa_{w_1}(S_1),\ldots,\kappa_{w_j}(S_j))$: virtual relations
Q: query containing one or more relational operations Δ

1: **for all** $avg \in Q$ **do**
2: rewrite ${}_{\beta}\gamma_{Avg(A_i)}(R)$ as $Q_k =_{\beta} \gamma_{Sum,Count(A_i)}(R)$
3: generate cQ
4: **end for**
5: **for all** $\rho \in Q$ **do**
6: rewrite $\rho(R)$ as $Q_t = \rho(\kappa_{v_k}(R_k))$ for all $k,t = 1,\ldots,i$
7: **end for**
8: **for all** unary $\sigma \in Q$ **do**
9: rewrite $Q_t = \sigma(R)$ as $Q_t = \sigma(\kappa_{v_k}(R_k))$ for all $k,t = 1,\ldots,i$
10: **end for**
11: **for all** $\pi \in Q$ **do**
12: rewrite $\pi(R)$ as $Q_t = \pi(R_k)$ for all $k,t = 1,\ldots,i$
13: **end for**
14: **for all** $max \vee min \vee sum \in Q$ **do**
15: rewrite $\gamma_{F(R.A_i)}\Delta R$ as
$Q_t = \gamma_{F(R_k.A_i)}\Delta(\kappa_{v_k}(R_k))$ for all $k,t = 1,\ldots,i$
16: **if** $\exists cQ$ **then**
17: modify cQ
18: **else**
19: generate cQ
20: **end if**
21: **end for**
22: **for all** $\cup \vee \setminus \vee \times \vee \bowtie \in Q$ **do**
23: rewrite $\Delta(R,S)$ as $Q_t = \Delta(R_k,S_l)$ for all $k = 1,\ldots,i$, $l = 1,\ldots,j$, and $t = 1,\ldots,i*j$
24: **end for**
25: **for all** $sort \vee group \in Q$ **do**
26: rewrite ${}_{\beta}\gamma_{F(R.A_i)}\Delta(Q_t)$ as ${}_{\beta}\gamma_{F(R_k.A_i)}\Delta(Q_t)$
for all $t = 1,\ldots,i$ in case Q unary or $t = 1,\ldots,i*j$ in case Q binary
27: **if** $\exists cQ$ **then**
28: modify cQ
29: **else**
30: generate cQ
31: **end if**
32: **end for**
33: **if** Q unary **then**
34: rewrite ΔR as $sQ = \bigcup_{t=1,\ldots,i} Q_t$
35: **end if**
36: **if** Q binary **then**
37: rewrite $\Delta(R,S)$ as $sQ = \bigcup_{t=1,\ldots,i*j} Q_t$
38: **end if**
39: **for all** $count\ distinct \vee count \in Q$ **do**
40: rewrite ${}_{\beta}\gamma_{F(R_k.A_i)}(sQ)$ as $sQ =_{\beta} \gamma_{F(R_k.A_i)}sQ$
41: **end for**
42: **return** sQ, (cQ)

All user groups $p_i \in \mathscr{P}^*(S)$ with a non-empty object set such that

$$O(p_i) = \{o | o \in O \wedge QS_o = p_i\} \neq \emptyset$$

are called *busy user groups*. These are user groups granted access to a set of objects specified by an access policy. These busy user groups might also change when adding or deleting users.

Granting Access Right. Adding a new user s_{n+1} changes the original user hierarchy $\mathscr{U}$ by adding sets of users. These are a set with only one element s_{n+1} and sets $p_i \cup \{s_{n+1}\}$ for all $i = \{1, \ldots 2^n - 1\}$. The result is a new user hierarchy. Consider all busy user groups p_i^{orig} in the original user hierarchy. Then for each user group p_i^{orig}, the new hierarchy contains a user group $p_i^{new} = p_i^{orig}$ and a user group $p_i^{orig} \cup \{s_{n+1}\}$.

Consider two cases: First, a busy user group p_i^{orig} evolves to a non busy user group p_i^{new} in the new user hierarchy and user group $p_i^{orig} \cup \{s_{n+1}\}$ is busy in the new user hierarchy. This is that the new user s_{n+1} has access to all objects accessible by user group p_i^{orig}. It is

$$O(p_i^{orig}) = O(p_i^{orig} \cup \{s_{n+1}\}).$$

The data owner shares the key of user group p_i^{orig} with user s_{n+1}.

Second, a busy user group p_i^{orig} of the original hierarchy is still a busy user group p_i^{new} in the new user hierarchy and user group $p_i^{orig} \cup \{s_{n+1}\}$ is also busy. This is that the new user s_{n+1} has been granted access to a subset of objects accessible for user group p_i^{orig}. The object set $O(p_i^{orig})$ is split such that

$$O(p_i^{orig}) = O(p_i^{new}) \cup O(p_i^{orig} \cup \{s_{n+1}\})$$

with

$$O(p_i^{new}) \cap O(p_i^{orig} \cup \{s_{n+1}\}) = \emptyset.$$

The data owner downloads object set $O(p_i^{orig} \cup \{s_{n+1}\})$ and re-encrypts it with a new key.

Revoking Access Right. We differentiate three scenarios where access rights are revoked from a user. First, a user is revoked from all access rights. Second, a user is revoked from a user group. Third, a user is revoked from certain objects of a user group.

Consider the first scenario where all access rights of a user are revoked. The original user hierarchy changes as the set of users S is reduced by one element s_n. This is to reduce

$$\mathscr{P}^*(\mathscr{S}) = \mathscr{P}^*(\mathscr{S} \backslash s_n) \cup (\mathscr{P}(\mathscr{S} \backslash s_n) \cup s_n)$$

to $\mathscr{P}^*(\mathscr{S} \backslash s_n)$ resulting in a new user hierarchy. Consider all busy user groups $p_i^{orig} \cup \{s_n\}$ in the original user hierarchy. These user groups are deleted from the new hierarchy. Their object sets are then accessible by the user groups $p_i^{new} = p_i^{orig}$ and merged with the respective object sets as

$$O(p_i^{new}) = O(p_i^{orig} \cup \{s_n\}) \cup O(p_i^{orig}).$$

The data owner downloads all object sets $O(p_i^{orig} \cup \{s_n\})$ and re-encrypts them with the respective keys of user groups p_i^{orig}.

Consider the second scenario where the user is revoked from a user group. This does not change the user hierarchy, but changes the busy user groups. Consider the respective busy user

group $p_i^{orig} \cup \{s_n\}$ in the original user hierarchy. It is non busy in the new hierarchy. Its object set is then accessible by user group $p_i^{new} = p_i^{orig}$ and merged with the respective object set as

$$O(p_i^{new}) = O(p_i^{orig} \cup \{s_n\}) \cup O(p_i^{orig}).$$

The data owner downloads object set $O(p_i^{orig} \cup \{s_n\})$ and re-encrypts it with the key of user group p_i^{orig}.

Consider the third scenario where the user is revoked from access of certain objects accessible by a user group. Consider the respective busy user group $p_i^{orig} \cup s_n$ in the original hierarchy. Revoking user s_n from accessing certain objects requires to split $O(p_i^{orig} \cup \{s_n\})$ as

$$O(p_i^{orig} \cup \{s_n\}) = O(p_i^{new} \cup \{s_n\}) \cup O(p_i^{new})$$

with

$$O(p_i^{new} \cup \{s_n\}) \cap O(p_i^{new}) \neq \emptyset.$$

This is that user s_n is granted access to object set $O(p_i^{new} \cup \{s_n\})$ but cannot access object set $O(p_i^{new})$. This results in two busy user groups $(p_i^{new} \cup \{s_n\})$ and p_i^{new}. Note that $O(p_i^{new})$ might also contain additional tuples previously available for the user group p_i^{old}. The data owner downloads object set $O(p_i^{new})$ and re-encrypts it with the key of user group p_i^{old}.
The data owner updates the user group and virtual relation mapping according to the changes of user hierarchy and busy user groups to keep track of the changing users, user groups, and virtual relations. She also distributes the encryption keys to the respective users while updating their key stores when they are logged in.

2.8 Implementation

We implemented ENKI as the extension of an existing single user solution to support the multi user setting. We use a modified JDBC driver for the single user mode which receives unencrypted SQL queries, modifies their operator tree, performs the onion selection, and encrypts the results.

As described in Figure 2.1, ENKI is an additional modification of the JDBC driver to perform query rewriting for the multi user mode and provides a client add-on to execute the post-processing. We handle the client-side using a SQLite database. The queries are executed on an unmodified SAP HANA database [FML+12] where UDFs execute cryptographic operations. We implemented DETPRE in C using pbc and gmp libraries providing the mathematical operations underlying pairing based encryption [Lyn07, GG14].

2.9 Experimental Evaluation

We evaluate functionality and performance of ENKI on the TPC-C benchmark and three real world application scenarios described in Subsection 2.9.1. We explain our experimental setup in Subsection 2.9.2. In Subsection 2.9.3, we analyze which types of queries and access policies can be supported. In Subsection 2.9.4, we evaluate the performance overhead consumed by the necessary modifications of ENKI.

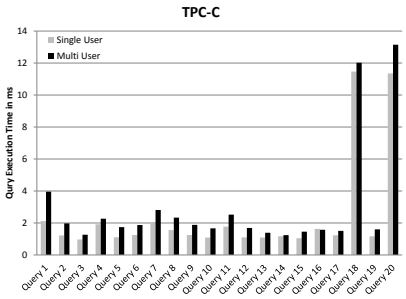

Figure 2.4: TPC-C: Query Execution Time for Single and Multi User Mode

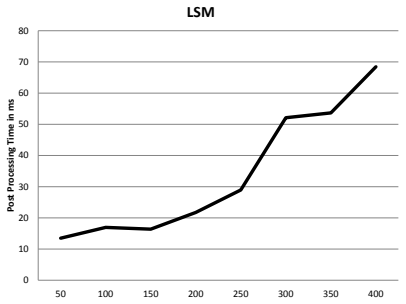

Figure 2.5: LSM: Post-Processing for Multi User Mode given $n = 50, \ldots, 400$ user groups

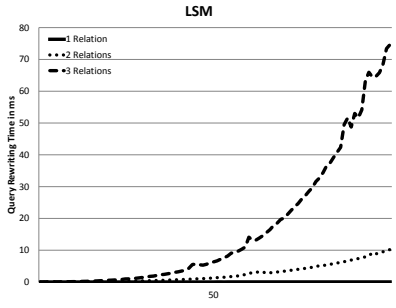

Figure 2.6: LSM: Query Rewriting for unary, binary, and tertiary relation given $n = 5, \ldots, 100$ user groups

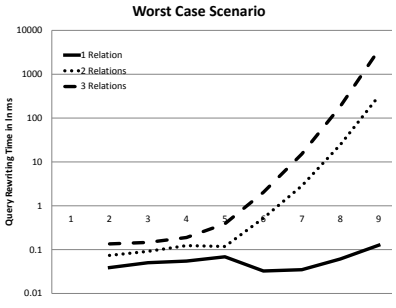

Figure 2.7: Worst Case: Query Rewriting for unary, binary, and tertiary relations given $n = 1, \ldots, 256$ user groups

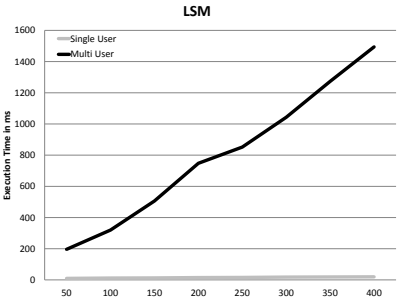

Figure 2.8: LSM: Query Execution Time for Single and Multi User Mode given $n = 50, \ldots, 400$ user groups

Query Type	IS-H	LSM	TPC-C	SFIN
Equal	x	x	x	x
Range	x		x	x
Equal Join	x		x	x
Range Join				
Aggregate	x	x	x	x

Table 2.1: Overview of the query types in the use cases

2.9.1 Application Scenarios

IS-H. IS-H is the healthcare management solution of SAP for patient management. In our observed query trace, we see 7 tables with 477 columns in total. As all tables contain personal information, we assume that all tables must be treated confidential. The users accessing this application are typically associated to different roles which are sets of organizational units. Patient information is associated with the organizational units of her encounters. To protect sensitive patient information, access policies prevent user from accessing medical details of patients if they are not associated to the set of organizational units of the patient.

LSM. LSM is an internal SAP solution which supports facility management to plan resources. Peers on a certain SAP management level include confidential planning information for their area. The peers are only allowed to access the data they committed themselves but not the data of other peers. Facility management has access to all data and calculates figures for future resource planning which are sensitive. The application contains of 25 tables and 173 columns.

TPC-C. TPC-C is an OLTP benchmark consisting of 9 tables and 92 columns. We assume that all tables and columns are sensitive and define an access policy for a two user scenario where each user has certain private data and other data is shared.

SFIN. Simplified Financials (SFIN) is part of SAP ERP application relying on SAP HANA as a database backend. In our use case, this application analyzes consumers' data sets consisting of 9 tables and 741 columns. We assume that all tables and columns are sensitive and define an access policy for a two user scenario where each user has certain private data and other data is shared.

2.9.2 Experimental Setup

Our experimental setup consists of a client running the modified JDBC proxy and a SQLite database and a server running an unmodified SAP HANA database. The client has 16 GB RAM and 2-core 2.8GH processor. The HANA database server has 252 GByte RAM and 8-core 2.6 GHz processor.

2.9.3 Functional Evaluation

To evaluate which queries and access policies ENKI can support, we analyzed the applications described in Subsection 2.9.1.

Queries. Table 2.1 shows the issued query types for each application. ENKI supports all observed queries including equal and order selections, equal joins, aggregations and combinations of these. In addition, ENKI also supports update, insert, and delete statements. Therefore, ENKI provides enhanced functionalities compared to existing solutions [YBDD09, PRZB11, PZ13].
ENKI cannot support the execution of range joins on the database server if a range join includes columns of different virtual relations encrypted with different keys. To our knowledge, there is no proxy re-encryption scheme for OPE encryption available. Therefore, ENKI would execute range joins only on client-side, however we consider this as acceptable as we did not observe a range join in any of our four applications.

Access Policies. ENKI supports the access policies specified for the IS-H and LSM application as its tuple-wise access restrictions on tables match well with the described requirements. This tuple-wise access enables the implementation of most of the access policies specified by authorization views [RMSR04]. An exception are those policies which only allow aggregated views on columns e.g. a user is only allowed to see the average of all attribute values of a column but not the unaggregated attribute values.

2.9.4 Performance Evaluation

We investigate two questions in order to evaluate the performance of ENKI:

- What is the performance penalty of our algorithm for multi user mode compared to single user mode?
- What is the performance impact of our proxy re-encryption scheme?

In the two experiments for the first question we assume that proxy re-encryption has already taken place.

TPC-C. We measure the execution time of the 20 select queries of the TPC-C benchmark and compare the query execution time of single and multi user mode. Table 2.1 shows the types of queries in TPC-C. For single user mode we execute the encrypted queries as [PRZB11], i.e. without any access policy. For the multi user mode we use the same access policy as in the examples in this paper: there are two users and three user groups. Each user has access to her private data and both user have access to shared data. In order to execute a multi user mode query, we need to rewrite, execute, and post-process. We measure the time of these steps and compare their total to the execution time of the single user mode. Figure 2.4 presents the results for the 20 TPC-C. The multi user mode incurs an average overhead of 36.98% (median overhead of 33.797%) compared to the single user mode. This is an absolute performance penalty of 0.6181 ms on average.
The total query execution time of the multi user mode is composed of three parts: query rewriting, query execution, and post-processing. On average query rewriting accounts for 3% of the total, query execution for 86%, and post-processing for 11%.

LSM. We measure the execution time of the queries of the LSM application over an increasing number of user groups. The LSM application specifies the following access policy: there are n users and n user groups. Each user group is unique for one user. $n-1$ users can only access their private data. One user has access to all data sets. This user performs unary relational operations

with aggregations on these data. With each additional user this user participates in an additional user group. We analyze the time of the three steps of the multi user mode algorithm over an increasing number of user groups. Given 50 user under this access policy, then a user incurs an average query execution time of 209.51 ms issuing a query over 50 virtual relations.

Query Rewriting. Figure 2.6 shows the time for rewriting a query over a linearly increasing number of virtual relations of $k = 5, \ldots, 100$. We measure the time for unary, binary, and tertiary relational operations. Figure 2.6 confirms that the effort is $\mathcal{O}(k)$ for unary operations, $\mathcal{O}(k^2)$ for binary operations, and $\mathcal{O}(k^3)$ for tertiary operations. Hence query rewriting depends on the specified access policy, since it defines the number of virtual relations and also on the number of operators in the query (i.e. unary, binary, or tertiary).
In order to further investigate the impact of the access policy, we bound the maximum number of user groups given n users where one user can participate in. For n users there are at $2^n - 1$ user groups. One user can participate in a maximum of 2^{n-1} user groups. For our measurement we use $1, \ldots 9$ users such that a user may participate in $1, \ldots, 256$ user groups. We depict the query rewriting time for unary, binary, and tertiary queries in Figure 2.7 on a logarithmic scale.
Theoretically, the time scales exponentially. However, we have neither observed such an access policy in a real world application nor have we found a case in the literature [KTZ11]. In accordance with current literature, we even assume that the number of actual user groups is smaller than the number of users [KTZ11]. This implies that the LSM application is indeed a representative scenario as the number of user groups equals the number of users.

Query Execution. Figure 2.8 presents the time to execute unary relational operations with aggregation functions over an increasing number of user groups $n = 50, \ldots, 400$. The query execution time in single user mode is nearly constant as the same query is only executed on a growing set of data. In contrast in multi user mode, each additional user adds one more user group. Hence the query expands by an additional subquery for the virtual relation. This effort is reflected by the execution time ranging from 0.196 s for 50 user groups to 1.5s for 400 user groups. We conclude that also the query execution time depends on the specified access policies.

Post-Processing. Figure 2.5 shows the effort to post-process unary relational operations including aggregation functions over an increasing number of user groups $n = 50, \ldots, 400$. These numbers contain the computational time for client-server-split as well as the necessary merge of the result sets on the client. The overhead for post-processing (which is not required in single user mode) is moderately growing up to 68.449 ms for 400 user groups. This is expected, since computation of *maximum*, *minimum*, and *sum* require $k - 1$ operations on the client and computation of *average* requires $2k - 1$ operations. The time to compute *sort* depends on the number of virtual relations k, but also on the maximum cardinality of all invoked virtual relations. It is $\mathcal{O}(m \log k)$ to merge k sorted lists with a total of m attribute values. The time to post-process the *group by* operation is a merge of m groups of k virtual relations which is to the time to merge k sorted lists. In addition, it is $\mathcal{O}(m - 1)$ to aggregate the partial results if similar groups exists. We conclude a similar dependency on the access policy as for query execution.
For the second performance question we conduct a third experiment.

Proxy Re-Encryption. We measure the execution time of our new encryption scheme DETPRE used to process *count distinct*, *set difference*, and *join* securely over data encrypted with different

	DETPRE	JOIN-ADJ
Encrypt	1.585956 ms	1.6058 ms
Token	0.0331148 ms	0.0014051 ms
PRE	1.019126 ms	0.000347 ms

Table 2.2: Microbenchmark of Encryption, Token Computation, and Proxy Re-Encryption of Det-Prey and JOIN-ADJ [PRZB11] over 10.000 Iterations

keys in the multi user mode and compare it to the encryption scheme Join-Adj used in the single user mode [PRZB11].
We present a micro benchmark in Table 2.2 which contains the time to compute the three algorithms of the scheme: encryption, token computation, and proxy re-encryption. The time to encrypt data is almost equal in both schemes with DETPRE consuming 1.5860 ms and Join-Adj consuming 1.6058 ms. The computation of the token consumes 0.03311 ms compared to Join-Adj with 0.0014 ms. The proxy re-encryption consumes 1.0191 ms in DETPRE and 0.0003 ms in Join-Adj. This proxy re-encryption time multiplies with the cardinalities of all columns which have to be proxy re-encrypted.
To minimize the time, it is possible to perform some computations in advance i.e. during the user logs in. However, it is not possible to substitute DETPRE with Join-Adj in the multi user case as proxy re-encryption in multi user mode which privacy-preserves data must be non-symmetric and non-transitive.

2.10 Conclusions

This paper presented ENKI, a system for securely executing relational operations over encrypted, access restricted data. ENKI introduces an encryption based access control model to enforce access restrictions encrypting data with different access rights with different encryption keys. ENKI uses query rewriting and post-processing to process relational operations over data encrypted with different encryption keys. It protects data confidentiality in case of the relational operations count distinct, set difference, or join by introducing a new privacy-preserving encryption scheme. The evaluation shows that its performance depends on the specified access policy and relational operations. It achieves modest overhead for the select queries of the TPC-C benchmark and the LSM use case.

3. Integrity and Consistency for Cloud Object Storage

Cloud services have turned remote computation into a commodity and enable convenient online collaboration. However, they require that clients fully trust the service provider, in terms of confidentiality, integrity, and availability. Towards reducing this dependency, we have introduced a protocol for *verification of integrity and consistency for cloud object storage (VICOS)*, which enables a group of mutually trusting clients to detect data-integrity and consistency violations for a cloud object-storage service [BCK15]. This protocol is aimed at services where multiple clients cooperate on data stored remotely on a potentially misbehaving service. VICOS enforces the consistency notion of fork-linearizability, supports wait-free client semantics for most operations, and reduces the computation and communication overhead compared to previous protocols such as SUNDR [CSS07], FAUST [CKS11], or Venus [SCC+10]. VICOS is based, in a generic way, on any *authenticated data structure*. Moreover, its operations cover the hierarchical name space of a cloud object store, supporting a real-world interface and not only a simplistic abstraction.

This chapter documents the progress of the project towards realizing, evaluating, and releasing the prototype implementation of VICOS. The toolkit is available as open-source code on GitHub at

```
https://github.com/ibm-research/vicos/
```

It has been documented in W3.1 [BC16].

3.1 State of the Art

3.1.1 Problem Description

For protecting remotely stored data, if there is only a single client, the client may locally keep a short *cryptographic hash value* of the outsourced data. Later, this can be used to verify the integrity of the data returned by the cloud storage service. However, with multiple disconnected clients, no common synchronization, and no communication among the clients, neither hashing nor digital signatures are sufficient by themselves. The reason is that a malicious or *Byzantine* server may violate the consistency of the data, for example, by reordering or omitting properly authenticated operations, so that the *views* of the storage state at different clients diverge. A malicious cloud server may pretend to one set of clients that some operations by others simply did not occur. In other words, *freshness* can be violated and the clients cannot detect such replay attacks until they communicate directly. The problem is particularly relevant in cryptographic online voting and for web certificate transparency [Lau14].

The strongest achievable notion of consistency in this multi-client model is captured by *fork-linearizability*, introduced by Mazières and Shasha [MS02]. A consistency and integrity verification protocol may guarantee this notion by adding condensed data about the causal evolution of the client's views into their interaction with the server. This ensures that if the server creates only a single discrepancy between the views of two clients, these clients may never observe operations of each other afterwards. In other words, if the server ever lies to some clients and these clients communicate later, they will immediately discover the violation. Hence, with only one check they can verify a large number of past operations.

The goal of this work is to demonstrate a complete practical system that supports the optimal consistency notion of fork-linearizability, provides wait-free semantics for all compatible client operations, and has smaller overhead than previous protocols.

3.1.2 Background

Many previous systems providing data integrity for storage systems rely on trusted components. Distributed file systems with cryptographic protection provide stronger notions of integrity and consistency than given by VICOS; there are many examples for this, from early research prototypes like FARSITE [ABC+02] or SiRiUS [GSMB03] to production file-systems today (e.g., IBM Spectrum Scale, `http://www-03.ibm.com/systems/storage/spectrum/scale/`). However, they rely on trusted directory services for freshness. Such a trusted coordinator is often missing or considered to be impractical. Iris [SvDO12] relies on a trusted gateway appliance, which mediates all requests between the clients and the untrusted cloud storage. Several recent systems ensure data integrity with the help of trusted hardware, such as CATS [YC07], which offers accountability based on an immutable public publishing medium, or A2M [CMSK07], which assumes an append-only memory. They all require some form of global synchronization, usually done by the trusted component, for critical metadata to ensure linearizability. In the absence of such communication, as assumed here, they cannot protect consistency and prevent replay attacks.

In CloudProof [PLM+11], an object-storage protection system with accountable and proof-based data integrity and consistency support, clients may verify the freshness of returned objects with the help of the data owner. Its auditing proceeds in epochs; operations are verified only at the end of each epoch and the detection may fail with some probability. Moreover, the clients need to communicate directly with the owner of an object for establishing integrity and consistency.

Cryptographic integrity guarantees are of increasing interest for many diverse domains: Verena [KFP+16], for example, is a recent enhancement for web applications that involve database queries and updates by multiple clients. It targets a patient database holding diagnostic data and treatment information. In contrast to VICOS, however, it relies on a trusted server that supplies hash values of data objects to clients during every operation.

The remainder of this section discusses related work without trusted components for synchronization. With only one client, the classic solution for memory checking by Blum et al. [BEG+94] provides data integrity through a hash tree and by storing its root at the client. Many systems have exemplified this approach for remote file systems and for cloud storage (e.g., Athos [GPTT08]).

With *authenticated data structures (ADS)* [NN00, MND+04], the single-writer, multi-reader model of remote storage can be authenticated, assuming there is a trusted and timely way to distribute authenticators from the writer to all readers. In practice, this approach is often taken for software distribution, where new releases are posted to a repository and authenticated by broadcasting hash values of the packages over a mailing list. AIP as introduced in [BCK15] represents

one way to generalize ADS for multiple writers.

In the multi-client model, Mazières et al. [MS02, LKMS04] have introduced the notion of fork-linearizability and implemented SUNDR, the first system to guarantee fork-linearizable views to all clients. It detects integrity and consistency violations among all clients that become aware of each other's operations. The SUNDR system uses messages of size $\Omega(n^2)$ for n clients [CSS07], which might be expensive. The SUNDR prototype [LKMS04] description also claims to handle multiple files and directory trees; however, the protocol description and guarantees are stated informally only, so that it remains unclear whether it achieves fork-linearizability under all circumstances.

Several systems have already expanded the guarantees of fork-linearizability to different applications [FZFF10] and improved the general efficiency of protocols for achieving it [CSS07]. Others have explored aborting operations [MDSS09] or introduced weak fork-linearizability in order to avoid blocking operations. In particular, SPORC [FZFF10], FAUST [CKS11], and Venus [SCC+10] sacrifice full linearizability to avoid aborts and blocking, respectively, and achieve weak fork-linearizability instead. The latter is a relaxation of fork-linearizability in which the most recent operation of a client may violate atomicity.

The SPORC system [FZFF10] is a groupware collaboration service whose operations may conflict with each other, but can be made to commute by applying a specific technique called "operational transformations." Through this mechanism, different execution orders still converge to the same state; still SPORC achieves only weak fork-linearizability.

Furthermore, VICOS also reduces the communication overhead compared to past systems considerably, since SUNDR, FAUST, and Venus all use messages of size $\Theta(n)$ or more with n clients, whereas the message size in VICOS does not depend on n.

The BST protocol [WSS09] supports an encrypted remote database hosted by an untrusted server that is accessed by multiple clients. Its consistency checking algorithm allows some commuting client operations to proceed concurrently; COP and ACOP [CO14] extend BST and also guarantee fork-linearizability for arbitrary services run by a Byzantine server, going beyond data storage services, and support wait-freedom for commuting operations. VICOS builds directly on COP, but improves the efficiency by avoiding the local state copies at clients and by reducing the computation and communication overhead. The main advantage is that clients can remain offline between executing operations without stalling the protocol.

3.1.3 The ADS Itegrity Protocol (AIP)

Brandenburger et al. [BCK15] introduce the *ADS integrity protocol (AIP)*, a generic protocol to verify the integrity and consistency for any authenticated data structure (ADS) operated by a remote untrusted server. It benefits from executing compatible operations concurrently. AIP extends and improves upon the *commutative-operation verification protocol (COP)* and its authenticated variant (ACOP) of Cachin and Ohrimenko [CO14]. For the completeness of this document, we recall AIP here in detail.

The processing of one operation in AIP is structured into an *active* and a *passive* phase. The active phase begins when the client invokes an operation and ends when the client completes it and outputs a response; this takes one message roundtrip between the client and the server. Different from past protocols, the client stays further involved with processing authentication data for this operation during the passive phase, which is decoupled from the execution of further operations.

More precisely, when client C_i invokes an operation $o \in \mathcal{O}$, it sends a signed INVOKE message

carrying o to the server S. The server assigns a global sequence number (t) to o and responds with a REPLY message containing a list of *pending* operations, the response, an authenticator, and auxiliary data needed by the client for verification. Operations are pending (for o) because they have been started by other clients and S has ordered them before o, but S has not yet finished processing them. We distinguish between *pending-other* operations, which have been invoked by other clients, and *pending-self* operations, which C_i has executed before o.

After receiving the REPLY message, the client checks its content. In particular, if the pending-other operations are compatible with o, then C_i verifies the pending-self operations including o with the help of the authenticator; if they are correct, C_i proceeds and outputs the response immediately. Along the way C_i verifies that all data received from S satisfies conditions to ensure fork-linearizability. An operation that terminates like this is called *successful*; alternatively, when the pending-other operations are not compatible with o, then o *aborts*. In this case, C_i returns the symbol ABORT. In any case, the client subsequently *commits* o and sends a signed COMMIT message to S (note that also aborted operations are committed in that sense). This step terminates the active phase of the operation. The client may now invoke the next operation or retry o if it was aborted.

Processing of o continues with the passive phase. At some (later) time, as soon as the operation immediately preceding o in the assigned order has terminated its own passive phase, S sends an UPDATE-AUTH message with auxiliary data and the authenticator of the preceding operation to C_i. When C_i receives this, it validates the message content and verifies the execution of o unless o had been aborted. Using the methods of the ADS, the client now computes and signs a new authenticator that it sends to S in a COMMIT-AUTH message. We say that C_i *authenticates* o at this time. When S receives this message, then it *applies* o by executing it on the state and stores the corresponding authenticator; this completes the passive phase of o.

Note that the server may receive COMMIT messages in an order that differs from the one of the globally assigned sequence numbers due to asynchrony. Still, the authentication steps in the passive phases of the different operations must proceed according to the assigned operation order. For this reason, the server maintains a second sequence number (b), which indicates the last authenticated operation that the server has applied to its state. Hence, S buffers the incoming COMMIT messages and runs the passive phases sequentially in the assigned order.

For ensuring consistency, every client needs to know about all operations that the server has executed. Therefore, when S responds to the invocation of an operation by C_i, it includes in the REPLY message a summary (including the corresponding signatures) of all those operations that C_i has missed since it last executed an operation. Prior to committing o, the client verifies these operations and thereby *clears* them.

Notation. The protocol is shown in Alg. 1–3 and formulated reactively. The clients and the server are state machines whose actions are triggered by events such as receiving messages. An ordered list with elements $e_1, e_2, \ldots, e_k$ is denoted by $E = \langle e_1, e_2, \ldots, e_k \rangle$; the element with index j may be accessed as $E[j]$. We also use maps that operate as associative arrays and store values under unique keys. A value v is stored in a map H by assigning it to a key k, denoted by $H[k] \leftarrow v$; for non-assigned keys, the map returns $\perp$. The symbol $\|$ denotes the concatenation of bit strings. The **assert** statement, parameterized by a condition, catches an error and immediately terminates the protocol when the condition is false. Clients use this to signal that the server misbehaved.

Data structures. This section describes the data structures maintained by every client and by the server. For simplicity, the pseudo code does not describe garbage collection, but we note where this is possible.

Every *client* C_i (Alg. 1) stores the sequence number of its last cleared operation in a variable c. The *hash chain* H represents the condensed view that C_i has of the sequence of all operations. It is computed over the sequence of all applied operations and the sequence of pending operations announced by S. Formally, H is a map indexed by operation sequence number; an entry $H[l]$ is equal to $hash(H[l-1]\|o\|l\|j)$ when the l-th operation o is executed by C_j, with $H[0] = \text{NULL}$. Variable Z is a map that represents the status (SUCCESS or ABORT) of every operation, according to the result of the test for compatibility. The client only needs the entries in H and Z with indices greater than c and may garbage-collect older entries. Finally, C_i uses a variable u that is set to o whenever C_i has invoked operation o but not yet completed it; otherwise u is $\perp$.

The *server* (Alg. 3) maintains the sequence number of the most recently invoked operation in a counter t. In addition to that, the counter b contains the sequence number of the most recently applied operation and governs the authentication of operations in the passive phase. Every invoked operation is stored in a map I and every committed operation in a map O; both maps are indexed by sequence number. The server only needs the entries in I with sequence numbers greater than b. An entry in O, at a sequence number b or greater, has to be stored until every client has committed some operation with a higher sequence number and may be removed later. Most importantly, the server keeps the state s of the ADS for the implemented functionality F, which reflects all successful operations up to sequence number b. In contrast to COP [CO14] and SPORC [FZFF10], where every client maintains a complete copy of the state, here only the server stores that state. Moreover, S stores the authenticator for every operation in a map A indexed by sequence number.

The protocol in detail. This section describes the ADS integrity protocol (AIP) as shown in Alg. 1–3. AIP is parameterized by an ADS and a functionality F that specifies its operations through $query_F$, $authexec_F$, and $refresh_F$. The client invokes AIP with an ADS-operation o by calling *aip-invoke*(o); it completes when AIP executes **return** at the end of the handler for the REPLY message. This ends the active phase of AIP, and the passive phase continues asynchronously in the background.

ACTIVE PHASE: When client C_i invokes an operation o, it computes an INVOKE-signature τ over o and i; this proves to other clients that C_i has invoked o. Then C_i stores o in u and sends an INVOKE message with o and τ to the server.

Upon receiving an INVOKE message with o, the server increments the sequence number t, assigns it to o, and assembles the REPLY message for C_i. First, S stores o and the accompanying signature in $I[t]$; the value t is also called the *position* of o. The pending operations for o, assigned to ω, are found in $I[b+1],\ldots,I[t]$, i.e., starting with the oldest non-applied operation, and include o. In order to compute the response and the auxiliary data for o from the correct state, the server must then extract the *successful* pending-self operations μ of C_i, using the following method:

function *separate-pending*(i, ω)
  $\mu \leftarrow \langle\rangle$; $\gamma \leftarrow \langle\rangle$
  for $k = 1, \ldots, \mathit{length}(\omega)$ **do**
    $(o', \cdot, j) \leftarrow \omega[k]$
    if $j = i$ **then**
      if $k = \mathit{length}(\omega) \vee$ status of o' is SUCCESS **then**
        $\mu \leftarrow \mu \circ \langle o' \rangle$ // see text how to get status of o'
    else if $j \neq i$ **then**
      $\gamma \leftarrow \gamma \circ \langle o' \rangle$
  return (μ, γ)

This method is common to the server and the clients. Note that μ includes the current operation (which appears at the end of ω) but not the aborted operations of C_i. The server finds the status of a pending-self operation o' of C_i in $O[b+k]$ (except for o itself, obviously) because C_i has already committed o' prior to invoking o and because the messages between C_i and S are FIFO-ordered. On the other hand, C_i retrieves the status of o' from $Z[b+k]$.

Then, S computes the response r and auxiliary data σ_o by calling $\mathit{query}_F(s, \mu)$ from the ADS for F; the response, therefore, takes into account the state reached after the successful pending-self operations of C_i but excludes any pending-other operations present in ω. However, the client will only execute o and output r when γ is compatible with o and, therefore, C_i is guaranteed a view in which the operations of γ occur after o. This will ensure fork-linearizability. The REPLY message to C_i also includes $A[b]$ containing the authenticator and its AUTH-signature, for the operation with sequence number b. The client passes these to $\mathit{authexec}_F$ of μ for verifying the correctness of the response. Furthermore, the REPLY message contains δ, the list of all operations that have been authenticated since C_i's last operation. In particular, when c is the sequence number from the INVOKE message, δ contains the operations at sequence numbers $c+1, \ldots, b$; when $c = b$, however, δ still contains $O[b]$.

After receiving the REPLY message from S, the client (1) processes and clears the authenticated operations in δ, (2) verifies the pending operations in ω, and (3) verifies that r is the correct response for o. These steps are explained next.

For verifying and processing δ and the last signed authenticator in α, client C_i calls a function *check-view* and verifies and/or extends the hash chain for every operation and verifies the corresponding COMMIT-signature. In particular, this ensures for any operation which has been pending for C_i and must be cleared, that the *same* operation was also authenticated by its originator. Finally, C_i also checks the AUTH-signature on the authenticator a, which is contained in α. If successful, all operations in δ are cleared and C_i's operation counter c is advanced to the position of the last operation in δ. (The check for $b = c$ ensures that δ contains at least one operation at position c.)

The client continues in *check-pending* by verifying that the pending operations are announced correctly: for every operation in ω, it determines the sequence number l, verifies the corresponding INVOKE-signature τ, and checks the hash chain entry $H[l]$. If there is no entry in H for l, then C_i computes the new entry from o, l, j, and $H[l-1]$; otherwise, C_i verifies that the existing entry matches the expected value. If this validation succeeds, it means the operation is consistent with a pending operation sent previously by S. After iterating through the pending operations, the client checks also that the last operation in ω is indeed its own current operation o.

Next, C_i invokes *separate-pending* to extract μ and γ from ω (see earlier). Then, C_i checks whether γ is compatible with u (the last invoked operation). If yes, C_i calls the ADS opera-

Algorithm 1 ADS integrity protocol (AIP) for client C_i

state

$c \in \mathbb{N}_0$: sequence number of last cleared operation, initially 0
$H : \mathbb{N}_0 \to \{0,1\}^*$: the hash chain, initially only $H[0] = \bot$
$Z : \mathbb{N}_0 \to \mathscr{Y}$: status map, initially empty
$u \in \mathscr{O} \cup \{\bot\}$: current operation or $\bot$ if none, initially $\bot$

function *aip-invoke*(o)

$u \leftarrow o$
$\tau \leftarrow sign_i(\text{INVOKE}\|o\|i)$
send message $[\text{INVOKE}, o, \tau, c]$ to S

upon receiving message $[\text{REPLY}, \delta, b, \alpha, \omega, t, r, \sigma_o]$ from S **do**

$(a, \psi) \leftarrow \alpha$
check-view(δ, b, a, ψ)
check-pending(ω)
$(\mu, \gamma) \leftarrow$ *separate-pending*(i, ω)
$t \leftarrow b +$ *length*(ω)
if $compatible_F(\gamma, u)$ **then**
    $(\cdot, \cdot, v) \leftarrow authexec_F(\mu, a, r, \sigma_o)$
    assert v
    $Z[t] \leftarrow \text{SUCCESS}$
else
    $r \leftarrow \text{ABORT}$
    $Z[t] \leftarrow \text{ABORT}$
$\phi \leftarrow sign_i(\text{COMMIT}\|t\|u\|i\|Z[t]\|H[t])$
send message $[\text{COMMIT}, u, t, Z[t], \phi]$ to S
$u \leftarrow \bot$
return r // response of operation *aip-invoke*(o)

upon recv. msg. $[\text{UPDATE-AUTH}, o, r, \sigma_o, \phi, q, \delta, \alpha]$ from S **do**

assert $verify_i(\phi, \text{COMMIT}\|q\|o\|i\|Z[q]\|H[q])$
$(o_\delta, \cdot, \cdot, \cdot, j) \leftarrow \delta$
$(a, \psi) \leftarrow \alpha$
assert $verify_j(\psi, \text{AUTH}\|o_\delta\|q-1\|H[q-1]\|a)$
if $Z[q] = \text{SUCCESS}$ **then**
    $(a', \sigma_o', v) \leftarrow authexec_F(o, a, r, \sigma_o)$
    assert v
else
    $(a', \sigma_0') \leftarrow (a, \bot)$
$\psi' \leftarrow sign_i(\text{AUTH}\|o\|q\|H[q]\|a')$
send message $[\text{COMMIT-AUTH}, a', \sigma_o', \psi']$ to S

Algorithm 2 ADS integrity protocol (AIP) for client C_i, continued

```
function extend-chain(o, l, j)
    if H[l] = ⊥ then
        H[l] ← hash(H[l−1]‖o‖l‖j)                          // extend by one
    else if H[l] ≠ hash(H[l−1]‖o‖l‖j) then
        return FALSE                                        // server replies are inconsistent
    return TRUE

function check-view(δ, b, a, ψ)
    assert length(δ) ≥ 1
    if b = c then d ← c−1 else d ← c
    for k = 1, ..., length(δ) do
        l ← d + k
        (o, z, φ, j) ← δ[k]
        assert extend-chain(o, l, j)
        assert verify_j(φ, COMMIT‖l‖o‖j‖z‖H[l])
    assert verify_j(ψ, AUTH‖o‖b‖H[b]‖a)                    // variables o and j keep their values
    c ← d + length(δ)                                       // all operations in δ have been cleared

function check-pending(ω)
    assert length(ω) ≥ 1
    for k = 1, ..., length(ω) do
        l ← c + k
        (o, τ, j) ← ω[k]
        assert extend-chain(o, l, j) ∧ verify_j(τ, INVOKE‖o‖j)
    assert o = u ∧ j = i                                    // variables o and j keep their values
```

tion $authexec_F(\mu, a, r, \sigma_o)$ for verifying that applying the operations in μ yields r as the response (recall that μ includes o at the end). The goal of this step is only to check the correctness of the response, and the authenticator and auxiliary data output by $authexec_F$ are ignored. Finally, C_i commits o by generating a COMMIT-signature over t, the sequence number of o, its status, and its hash chain entry, sends a COMMIT message (with t, the operation, and the signature) to S, and outputs the response r.

PASSIVE PHASE: The server stores the content of all incoming COMMIT messages in O and processes them in the order of their sequence numbers, indicated by b. When an operation with sequence number $b+1$ has been committed but not yet authenticated by the client and applied by S (i.e., **upon** $O[b+1] \neq \bot \wedge A[b+1] = \bot$), the server uses $query_F$ to compute the response r and to extract the auxiliary data σ_o from the current state s. It sends this in an UPDATE-AUTH message to C_i, also including the operation at position b (from $O[b]$) and its authenticator (taken from $A[b]$), which have been computed before. These values allow the client to verify the authenticity of the response for the operation at position $b+1$.

The client C_i then receives this UPDATE-AUTH message (for o and sequence number q), and first validates the message contents. In particular, C_i verifies that the authenticator a is covered by a valid AUTH-signature by client C_j with sequence number $q-1$, using C_i's hash chain entry $H[q-1]$.

Next, if o was not aborted, i.e., $Z[q] = \text{SUCCESS}$, the client invokes $authexec_F$ to verify that the auxiliary data and the response are correct, and to generate new auxiliary data s'_o and a new

Algorithm 3 ADS integrity protocol (AIP) for server S

state

$t \in \mathbb{N}_0$: seqno. of last invoked op., initially 0
$b \in \mathbb{N}_0$: seqno. of last applied op., initially 0
$I : \mathbb{N} \to \mathcal{O} \times \{0,1\}^* \times \mathbb{N}$: invoked ops., initially empty
$O : \mathbb{N} \to \mathcal{O} \times \{0,1\}^* \times \mathcal{Z} \times \{0,1\}^* \times \mathbb{N}$: committed ops., initially empty
$A : \mathbb{N}_0 \to \{0,1\}^* \times \{0,1\}^*$: authenticators, init. $A[0] = a_0$
$s \in \{0,1\}^*$: state of the service, initially $s = s_0$

upon receiving message $[\text{INVOKE}, o, \tau, c]$ from C_i **do**
$t \leftarrow t+1$
$I[t] \leftarrow (o, \tau, i)$
if $b = c$ **then** $\delta \leftarrow \langle O[b] \rangle$ **else** $\delta \leftarrow \langle O[c+1], \ldots, O[b] \rangle$
$\omega \leftarrow \langle I[b+1], I[b+2], \ldots, I[t] \rangle$ // all pending operations
$(\mu, \cdot) \leftarrow \textit{separate-pending}(i, \omega)$
$(r, \sigma_o) \leftarrow \textit{query}_F(s, \mu)$
send message $[\text{REPLY}, \delta, b, A[b], \omega, t, r, \sigma_o]$ to C_i

upon receiving message $[\text{COMMIT}, o, q, z, \phi]$ from C_i **do**
$O[q] \leftarrow (o, z, \phi, i)$

upon $O[b+1] \neq \bot \wedge A[b+1] = \bot$ **do**
$(o, z, \phi, j) \leftarrow O[b+1]$
if $z = \text{SUCCESS}$ **then**
$(r, \sigma_o) \leftarrow \textit{query}_F(s, o)$
else
$(r, \sigma_o) \leftarrow (\bot, \bot)$
send msg. $[\text{UPDATE-AUTH}, o, r, \sigma_o, \phi, b+1, O[b], A[b]]$ to C_j

upon receiving message $[\text{COMMIT-AUTH}, a, \sigma_o, \psi]$ from C_i **do**
$b \leftarrow b+1$
$A[b] \leftarrow (a, \psi)$
$(o, z, \cdot, \cdot) \leftarrow O[b]$
if $z = \text{SUCCESS}$ **then**
$s \leftarrow \textit{refresh}_F(s, o, \sigma_o)$

authenticator a', which vouches for the correctness of the state updates induced by o. Otherwise, C_i skips this step, as the authenticator does not change. Then C_i issues an AUTH-signature ψ' and sends it back to S together with a' and s'_o in a COMMIT-AUTH message.

As the last step in the passive phase, S increments b, stores the data in the COMMIT-AUTH message at $A[b]$, and if the operation did not abort, S applies it to s through $\textit{refresh}_F$.

Remarks. As in BST [WSS09] and in COP [CO14], operations that do not interfere with each other may proceed without blocking. More precisely, if some pending operation is not compatible with the current operation, the latter is aborted and must be retried later. Preventing clients from blocking is highly desirable but cannot always be guaranteed without introducing aborts [CSS07]. The potential for blocking has led other systems, including SPORC [FZFF10] and FAUST [CKS11], to adopt weaker and less desirable guarantees than fork-linearizability.

Obviously, it makes no sense for a client to retry its operation while the non-compatible operation is still pending. However, the client does not know when the contending operation commits. Additional communication between the server and the clients could be introduced to signal this. Alternatively, the client may employ a probabilistic waiting strategy and retry after a random delay.

In the following we assume that S is correct. The communication cost of AIP amounts to the five messages per operation. Every client eventually learns about all operations of all clients, as it must clear them and include them in its hash chain. However, this occurs only when the client executes an operation (in REPLY). At all other times between operations, the client may be offline and inactive. In a system with n clients that performs h operations in total, BST [WSS09] and COP [CO14] require $\Theta(nh)$ messages overall. AIP reduces this cost to $\Theta(h)$ messages, which means that each client only processes a small constant number of messages per operation.

The size of the INVOKE, COMMIT, UPDATE-AUTH, and COMMIT-AUTH messages does not depend on the number of clients and on the number operations they execute. The size of the REPLY message is influenced by the amount of contention, as it contains the pending operations. If one client is slow, the pending operations may grow with the number of further operations executed by other clients. Note that the oldest pending operation is the one at sequence number $b+1$; hence, all operations ordered afterwards are treated as pending, even when they already have been committed. The REPLY message can easily be compressed to constant size, however, by omitting the pending operations that have already been sent in a previous message to the same client.

The functionality-dependent cost, in terms of communicated state and auxiliary data, is directly related to the ADS for F. In practice, hierarchical authenticated search structures, such as hash trees and authenticated skip lists, permit small authenticators and auxiliary data [CW11].

3.2 ESCUDO-CLOUD Innovation

A prototype of VICOS that works with the key-value store interface of commodity cloud storage services has been implemented, and an evaluation demonstrates its advantage compared to existing systems. The specific innovation compared to the published work [BCK15] consists of a revised integration with the cloud-object storage and a completely new evaluation.

The remaining sections are structured as follows: The object-storage protocol VICOS is introduced in Section 3.3, and Section 3.4 describes the prototype. An evaluation has been performed and is described in Section 3.5.

3.3 Verification of Integrity and Consistency of Cloud Object Storage (VICOS)

We now describe the protocol for *verifying the integrity and consistency of cloud object storage*, abbreviated *VICOS*. It leverages AIP from the previous section and provides a fork-linearizable Byzantine emulation for a practical object-store service, in a manner that is transparent to the storage provider. We first define the operations of the cloud storage service and outline the architecture of VICOS. Next we instantiate AIP for verifying the integrity of a simple object store and show how VICOS extends this to practical cloud storage.

More precisely, VICOS consists of the following components (see Fig. 3.1):

1. A *cloud object store (COS)* service with a key-value store interface, as offered by commercial providers. It maintains the object data stored by the clients using VICOS.

2. An *AIP client* and an *AIP server*, which implement the protocol from the previous section for the functionality of an *authenticated dictionary (ADICT)* and authenticate the objects at the cloud object store. The AIP server runs remotely as a cloud service accessed by the AIP client. This is abbreviated as *AIP with ADICT*.

3. The *VICOS client* exposes a cloud object store interface to the client application and transparently performs integrity and consistency verification. During each operation, the client consults the cloud object store for the object data itself and the AIP server for integrity-specific metadata. In particular, AIP server running ADICT stores a cryptographic hash of every object.

Note that the cloud object store as well as the AIP server are in the untrusted domain; they may, in fact, collude together against the clients.

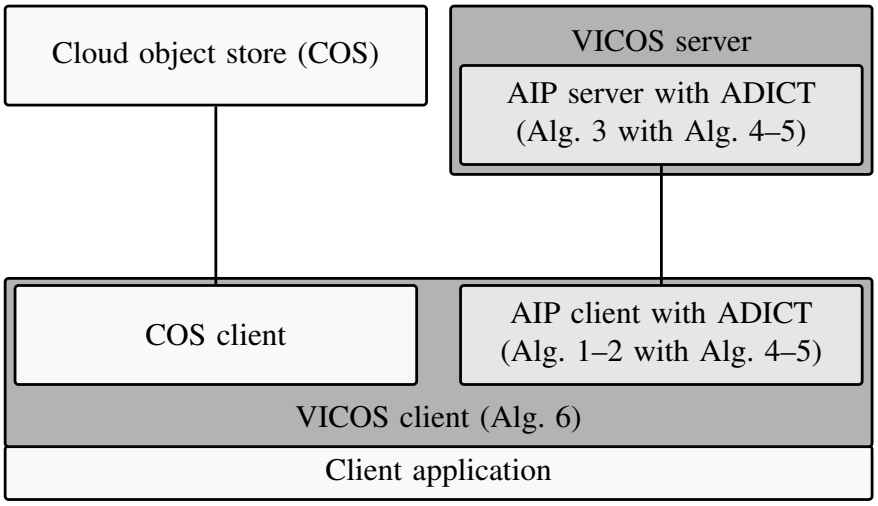

Figure 3.1: Architecture of VICOS: the two untrusted components of the cloud service are shown at the top, the trusted client is at the bottom.

3.3.1 Cloud Object Store (COS)

The *cloud object store (COS)* is modeled by a *key-value store (KVS)* and provides a "simple" storage service to multiple clients. The COS stores objects in a flat namespace, where each *object* is an arbitrary sequence of bytes (or a "blob," a binary large object), identified by a unique *name* or *key*. We assume that clients may only read and write entire objects, i.e., it is not possible to read from or write into the middle of an object, as in a file system.

Our formal notion of a KVS internally maintains a map M that stores the values in $\mathscr{V}$ under their respective keys taken from a universe $\mathscr{K}$. It provides four operations:

1. *kvs-put*(k,v): Stores a value $v \in \mathscr{V}$ under key $k \in \mathscr{K}$, that is, $M[k] \leftarrow v$.
2. *kvs-get*(k): Returns the value stored under key $k \in \mathscr{K}$, that is, $M[k]$.
3. *kvs-del*(k): Deletes the value stored under key $k \in \mathscr{K}$, that is, $M[k] \leftarrow \perp$.
4. *kvs-list*$()$: Returns a list of all keys for which a value is stored, that is, the list $\langle k \in \mathscr{K} \,|M[k] \neq \perp \rangle$.

This API forms the core of many real-world cloud storage services, such as Amazon S3 or OpenStack Swift. Typically there is a bound on the length of the keys, such as a few hundred bytes, but the stored values can be much larger and practically unbounded (on the order of several Gigabytes). For simplicity, we assume that the object store provides atomic semantics during concurrent access, being aware that cloud storage systems may only be eventually consistent [BG13] due to network partitions.

Many practical cloud object stores support a single-level hierarchical name space, formed by *containers* or *buckets*. We abstract this separation into the keys here; however, a production-grade system would introduce this separation again by applying the design of VICOS for every container.

3.3.2 Authenticated Dictionary Implementation (ADICT)

VICOS instantiates AIP with the functionality of a KVS that stores only *short values*. In order to distinguish it from the cloud object store, we refer to it as the *authenticated dictionary*, denoted by *ADICT*, with operations *adict-put*, *adict-get*, *adict-del*, and *adict-list*.

The implementation of ADICT uses the well-known approach of building a hash tree over its entries [BEG+94, NN00, CW11]; see Alg. 4–5 for the details of how ADICT is implemented within AIP. The AIP server stores the values in a map D and maintains a hash tree T, constructed over the list of key-value pairs stored in the map, according to a fixed sort order on the keys. That is, every leaf node of the hash tree is computed by hashing the node key, its value, and the key of the successor leaf node together. The next node has to be included in order to authenticate the *absence* of a key in response to a query for a non-existing key (e.g., [NN00]). The root of the hash tree serves as the authenticator for ADICT.

For the *adict-put*, *adict-get*, and *adict-del* operations, the server extracts those paths from T that are necessary to verify the correctness of the retrieved value and places them in s_o. For *adict-put* and *adict-del* operations, $query_{ADICT}$ also places these paths into s_o because the client needs them to construct the updated root hash. For *adict-list*, the complete hash tree is included in s_o. The asterisks (*) in Alg. 4–5 denote some additional data and steps necessary to verify the predecessor or successor leaves for authenticating an absent key (details of this are omitted here and can be found in the literature [CW11]).

The $compatible_{ADICT}(\mu, u)$ function of Alg. 4–5 defines the compatibility relation among the operations of the authenticated dictionary; VICOS supports the same KVS interface and inherits this notion of compatibility for the cloud-storage operations. For more general services like databases, one would invoke a transaction manager here. For ADICT, the compatibility between a first (pending) operation u and a second (current) operation o is given by Table 3.1. For instance, *adict-put* followed by *adict-get* for the same key or followed by *adict-list* are not compatible, whereas two *adict-list* and *adict-get* operations are always compatible.

The advantage of considering operation compatibility over commutativity (as used by previous work such as ACOP [CO14]) becomes apparent here: only 8 pairs among the 49 cases shown are not compatible, whereas 22 out of 49 cases do not commute and would be aborted with commutativity (the additional 14 cases are underlined in Table 3.1).

Algorithm 4 Authenticated dictionary implementation (ADICT) for AIP, Part 1

state

$D : \mathcal{K} \to \{0,1\}^*$: authenticated dictionary, initially empty

T: hash tree over D, initially empty

function $query_{ADICT}((D,T),o)$

if $o = adict\text{-}put(k,v) \vee o = adict\text{-}del(k)$ **then**

$r \leftarrow \perp$

$s_o \leftarrow$ sibling nodes on path (*) from k to root in T

else if $o = adict\text{-}get(k)$ **then**

$r \leftarrow D[k]$

$s_o \leftarrow$ sibling nodes on path (*) from k to root in T

else // $o = adict\text{-}list()$

$r \leftarrow \langle k \in \mathcal{K} | D[k] \neq \perp \rangle$

$s_o \leftarrow T$

return (r, s_o)

function $authexec_{ADICT}(o,a,r,s_o)$

if $o = adict\text{-}put(k,v) \vee o = adict\text{-}get(k) \vee o = adict\text{-}del(k)$ **then**

if s_o is not a valid path (*) from k to tree root a **then**

return $(\cdot,\cdot,\text{FALSE})$

if $o = adict\text{-}put(k,v)$ **then**

insert leaf node k with value v in the tree

$s'_o \leftarrow$ updated path from k to tree root

$a' \leftarrow$ updated hash-tree root

else if $o = adict\text{-}get(k)$ **then**

if path (*) not consistent with node k holding r **then**

return $(\cdot,\cdot,\text{FALSE})$

$s'_o \leftarrow \perp$

$a' \leftarrow a$

else if $o = adict\text{-}del(k)$ **then**

delete leaf node k from the tree

$s'_o \leftarrow$ updated paths from siblings of k to tree root

$a' \leftarrow$ updated hash-tree root

else // $o = adict\text{-}list()$

if r is not list of keys in leaves of tree with root a **then**

return $(\cdot,\cdot,\text{FALSE})$

$s'_o \leftarrow \perp$

$a' \leftarrow a$

return (a', s'_o, TRUE)

Algorithm 5 Authenticated dictionary implementation (ADICT) for AIP, Part 2

function $refresh_{ADICT}((D,T),o,s_o)$
  if $o = adict\text{-}put(k,v)$ **then**
    $D[k] \leftarrow v$
    update path in T from k to root, as taken from s_o
  else if $o = adict\text{-}del(k)$ **then**
    $D[k] \leftarrow \perp$
    update path in T from k to root, as taken from s_o
  return (D,T)

function $compatible_{ADICT}(\mu,u)$
  for $o \in \mu$ **do**
    if $\neg compatible_{ADICT}(u,o)$ **then** // See Table 3.1 for the $compatible_{ADICT}()$ relation on operations
      return FALSE
  return TRUE

	$adict\text{-}put(x,\cdot)$	$adict\text{-}put(y,\cdot)$	$adict\text{-}get(x)$	$adict\text{-}get(y)$	$adict\text{-}del(x)$	$adict\text{-}del(y)$	$adict\text{-}list$
$adict\text{-}put(x,\cdot)$	$\underline{\surd}$	$\surd$	—	$\surd$	$\underline{\surd}$	$\surd$	—
$adict\text{-}put(y,\cdot)$	$\surd$	$\underline{\surd}$	$\surd$	—	$\surd$	$\underline{\surd}$	—
$adict\text{-}get(x)$	$\underline{\surd}$	$\surd$	$\surd$	$\surd$	$\underline{\surd}$	$\surd$	$\surd$
$adict\text{-}get(y)$	$\surd$	$\underline{\surd}$	$\surd$	$\surd$	$\surd$	$\underline{\surd}$	$\surd$
$adict\text{-}del(x)$	$\underline{\surd}$	$\surd$	—	$\surd$	$\surd$	$\surd$	—
$adict\text{-}del(y)$	$\surd$	$\underline{\surd}$	$\surd$	—	$\surd$	$\surd$	—
$adict\text{-}list()$	$\underline{\surd}$	$\underline{\surd}$	$\surd$	$\surd$	$\underline{\surd}$	$\underline{\surd}$	$\surd$

Table 3.1: The $compatible_{ADICT}(\cdot,\cdot)$ relation for ADICT and the KVS interface, where $x,y \in \mathcal{K}$ denote distinct keys: if the operation in a row is pending, then a checkmark $\surd$ means that the operation in the column is compatible. Underlined checkmarks indicate cases where the operations do not commute.

3.3.3 VICOS Client Implementation

VICOS emulates the key-value store API of a cloud object store (COS) to the client and transparently adds integrity and consistency verification. As with AIP, consistency or data integrity violations committed by the server are detected through **assert**; any failing assertion triggers an alarm. It must be followed by a recovery action whose details go beyond the scope of this document. Analogously to AIP, VICOS may return ABORT; this means that the operation was not executed and the client should retry it.

Algorithm 6 Implementation of VICOS at the client.

```
function vicos-put(k, v)
    x ← a random nonce
    cos-put(k||x, v)
    h ← hash(v)
    r ← aip-invoke(adict-put(k, ⟨x, h⟩))
    if r = ABORT then                          // concurrent incompatible operation
        cos-del(k||x)
    return r

function vicos-get(k)
    r ← aip-invoke(adict-get(k))
    if r = ABORT then                          // concurrent incompatible operation
        return ABORT
    ⟨x, h⟩ ← r
    v ← cos-get(k||x)
    assert hash(v) = h
    return v

function vicos-del(k)
    r ← aip-invoke(adict-del(k))
    if r = ABORT then                          // concurrent incompatible operation
        return ABORT
    cos-del(k||∗)                              // deletes all keys with prefix k
    return r

function vicos-list()
    r ← aip-invoke(adict-list())
    if r = ABORT then                          // concurrent incompatible operation
        return ABORT
    return r
```

Algorithm 6 presents the pseudo code of the VICOS client. Basically, it protects every object in the COS by storing its cryptographic hash in the authenticated dictionary (ADICT). Operations on the object store trigger corresponding operations on COS and on ADICT, as provided by AIP for consistency enforcement.

In order to prevent race conditions, VICOS does not store the hash of an object under the object's key in COS directly, but *translates* every object key to a unique key for COS. Otherwise, two concurrent operations accessing the same object might interfere with each other and leave the system in an inconsistent state. More precisely, in a *vicos-put*(k, v) operation, the client chooses

a nonce x (a value guaranteed to be unique in the system, such as a random string) and stores v in COS using *cos-put*$(k\|x, v)$ with the translated key $k\|x$. Furthermore, it computes $h \leftarrow$ *hash*(v) and stores $\langle x, h\rangle$ in ADICT using *adict-put* at key k. The *cos-put* and *cos-get* operations actually stream long values. When *adict-put* aborts due to concurrent operations, the client deletes v again from COS using *cos-del*$(k\|x)$.

For a *vicos-get*(k) operation, the client first calls *adict-get*(k) and retrieves $\langle x, h\rangle$. Unless this operation aborts, the client translates the key and calls *cos-get*$(k\|x)$ to retrieve the value v. After v has been read (or streamed), the client compares its hash value to h, asserts that they match, and then outputs v.

Without key translation, two concurrent *vicos-put* operations o_1 and o_2 writing different values to the same key k might both succeed with *cos-put*(k, v_1) and *cos-put*(k, v_2), respectively, but the *adict-put* for o_2 might abort due to another concurrent operation. Then COS might store v_2 but ADICT stores the hash of v_1 and readers would observe a false integrity violation. Thanks to key translation, no versioning conflicts arise in the COS. Atomicity for multiple operations on the same object key follows from the properties of AIP with the ADICT implementation. The *vicos-del*(k) and *vicos-list*$()$ operations proceed analogously; but the latter does not access COS.

3.3.4 Correctness

It is easy to see that the implementation of VICOS satisfies the two properties of a fork-linearizable Byzantine emulation. First, when S is correct, then the clients proceed with their operations and all verification steps succeed. Hence, VICOS produces a linearizable execution. The linearization order is established by AIP running ADICT. Furthermore, when the clients execute sequentially, then by the corresponding property of AIP, no client ever receives ABORT from ADICT.

Second, consider the case of a malicious server controlling COS and the AIP server together. AIP ensures that the operations on ADICT (*adict-put*, *adict-get*, etc.) are fork-linearizable according to Section 3.3. The implementation of ADICT follows the known approach of memory checking with hash trees [BEG+94, NN00] and therefore authenticates the object hash values that VICOS writes to ADICT. According to the properties of the hash function, the object data is uniquely represented by its hash value. Since VICOS ensures the object data written to COS or returned to the client corresponds to the hash value stored in ADICT, it follows that all operations of VICOS are also fork-linearizable.

3.4 Prototype

We have implemented a prototype of VICOS in Java; it consists of a client-side library ("VICOS client") and the server code ("VICOS server"). The system can be integrated with applications that require cryptographic integrity and consistency guarantees for data in untrusted cloud storage services. It is available as open-source on GitHub (`https://github.com/ibm-research/vicos`).

3.4.1 VICOS Implementation

The client-side library uses the *BlobStore* interface of Apache *jclouds* (Version 2.0.0 -Snapshot, `https://jclouds.apache.org/`) for connecting to different cloud object stores. The server runs as a standalone web service, communicating with the client-side library using the *Akka* framework (Version 2.4.4, `http://www.akka.io`).

Akka is an event-driven framework which supports the actor model [HBS73]. It fits perfectly into the system model of VICOS and allows for rapid development. This simplifies the actual protocol implementation because of the high level of abstraction, especially relating to network operations and concurrency.

The client library as well as the server code are implemented as actors within the framework. Actors are independent units which can only communicate by exchanging messages. Every actor has a mailbox that buffers all incoming messages. By default, messages are processed in FIFO order by the actor. This allows the server protocol implementation to process all incoming messages sequentially and execute each protocol step atomically, that is, mutually exclusive with respect to all others. The implementation of VICOS, therefore, closely follows the high-level description of AIP. Note that the system performance is limited by the fact that it does not exploit modern multi-core architectures.

We developed the VICOS client library so that it may easily be integrated into existing applications to provide integrity protection. It uses the modular approach of AIP instantiated with an authenticated data structure (ADS). A developer only needs to implement the desired functionality, by defining the state (e.g., the internal KVS data structure), a set of operations, and their compatibility relation. For that reason, we have defined two interfaces: *state* and *operation processor*. Operations are described using Google's Protocol Buffers (`https://developers.google.com/protocol-buffers/`) executed by the operation processor. This modular concept allows us to reuse and extend the core implementation of the protocol.

The VICOS prototype provides a simple KVS functionality by implementing these interfaces. More precisely, the KVS state is a map supporting GET, PUT, DEL and LIST operations. The KVS operation processor provides the implementations of *query*, *authexec*, *refresh*, and *compatible* as described in Alg. 4–5. The client and the server protocol each contain an instance of the KVS operation processor. The client exposes a simple KVS interface, adding Java Exceptions for signaling integrity and consistency violations. Furthermore, the client library is completely asynchronous and supports processing the AIP passive phase in the background without blocking the client process.

The cryptographic signatures can be implemented in multiple ways. According to the security assumptions of the model, all clients trust each other, the server alone may act maliciously, and only clients issue digital signatures. Therefore, one can also adopt a simplified trust model with "signatures" provided by a message-authentication code (MAC). For many applications, where strong mutual trust exists among the clients, MACs suffice and will result in faster execution. On the other hand, this simplification renders the system more fragile and exposes it easily to attacks by clients.

In particular, VICOS uses HMAC-SHA1 with 128-bit keys provided by the Java Cryptography Extension as the default signature implementation. The code also supports RSA and DSA signatures with 2048-bit keys. A user can choose between these signature implementations via a configuration file. This approach also allows developers to change to a different signature implementation, or even implement their own according to new requirements. Particularly, this might be useful for porting the code to other platforms such as mobile devices, with less computation power.

The core implementation of VICOS consists of 3400 sloc, the server part is 400 sloc more, whereas the client part including the integration with the evaluation platform (see below) takes 800 sloc extra.

3.4.2 Practical Issues and Optimizations

While developing VICOS and experimenting with it, we obtained experience with Akka and gained insight into the protocol's operation. This has led to further optimizations described here.

Bounded pending list. An issue that we discovered in Akka while implementing VICOS is Akka's default maximum message size of only 128kB. In particular, this becomes a problem in VICOS when REPLY messages include a large partial state or a very long list of pending operations. By configuring Akka with larger message sizes (we used a maximum size of 128MB), the direct limitation disappears.

However, we experienced that large messages impact the overall performance negatively. When the number of pending operations increases, the resulting very large messages slow down the operations of VICOS. Therefore, we implemented a way to bound the length of the pending-operations list, that is, we introduced a maximum number of pending operations as a configurable value and modified the protocol. When this maximum is reached, the server buffers all new incoming requests (INVOKE messages) until enough other operations have completed and the number of pending operations goes below the limit. We tested with different maximum sizes for the pending list from 32 up to 1024 operations, and chose a limit of 128 for the evaluations.

A more robust solution of that issue would be to signal the clients to wait before sending more requests, instead of just buffering them at the server. Although limiting the number of pending operations under high server load increases the latency of client requests, it also increases the overall performance and stability of the system. In summary we found that the benefits of limiting the size of the pending list outweigh the drawbacks.

Message delivery order. During development we experienced slow performance caused by the FIFO order in which the protocol actors processed the arriving messages. Therefore, we implemented priority mailboxes for the server and the client actor and defined a priority rule to prefer COMMIT, UPDATE-AUTH, COMMIT-AUTH messages over INVOKE and REPLY messages. This has the immediate benefit that the server processes UPDATE-AUTH messages and thereby completes the PASSIVE PHASE of already authenticated operations *before* it starts working on new INVOKE messages. This preference shortens the list of pending operations directly. Certainly, this may increase the response time for new operations again, but eventually, it prevents more operations from aborting due to conflicts under high load.

3.5 Evaluation

This section reports on performance measurements with the VICOS prototype. We study the general overhead of integrity protection, the scalability of the protocol, and the effect of (in-)compatible operations.

3.5.1 Experimental Setup

The experiments use cloud servers and an OpenStack Swift-based object storage service (`http://swift.openstack.org`) of a major cloud provider with about two dozen data centers worldwide (Softlayer — an IBM Company, `http://www.softlayer.com/object-storage`).

The VICOS server runs on a dedicated "baremetal" cloud server with a 3.5GHz Intel Xeon-Haswell (E3-1270-V3-Quadcore) CPU, 8GB DDR3 RAM, and a 1Gbps network connection. The

clients run on six baremetal servers in total, each server with 2x 2GHz Intel Xeon-SandyBridge (E5-2620-HexCore) CPUs, 16GB DD3 RAM, and a 1Gbps network connection. All clients are hosted in the same data center. All machines run Ubuntu 14.04-64 Linux and Oracle Java (JRE 8, build 1.8.0_77-b03).

To simulate a realistic environment, we conduct experiments in two settings as shown in Table 3.2: A *datacenter* setting, with all components in the same data center ("Amsterdam"), and a *wide-area* setting, where the VICOS server and COS are located together in one data center ("Milan"), and the clients at a remote site ("Amsterdam").

Setting	Clients	VICOS server	Cloud object storage	Latency [ms]
Datacenter	Amsterdam	Amsterdam	Amsterdam	< 1
Wide-area	Amsterdam	Milan	Milan	~ 10

Table 3.2: Evaluation setting

The datacenter setting establishes a best-case baseline due to the very low network latencies (< 1ms). This deployment is not very realistic in terms of the security model because the clients and the storage service are co-located.

The wide-area setting exhibits a moderate network latency (round-trip delay time of 20ms) between the two data centers and models the typical case of geographically distributed clients accessing a cloud service with its point-of-access on the same continent but in different countries.

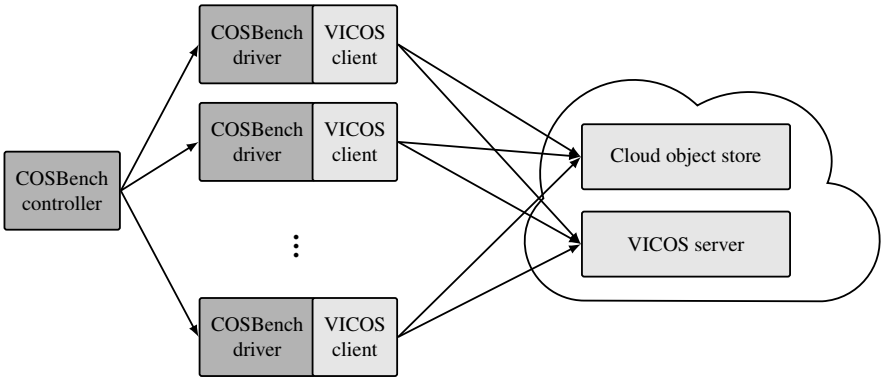

Figure 3.2: The experimental setup, with one COSBench controller and many COSBench drivers, accessing the cloud storage service through the VICOS client.

The evaluation is driven by COSBench (Version 0.4.2 – `https://github.com/intel-cloud/cosbench`), an extensible tool for benchmarking cloud object stores. We have created an adapter to drive VICOS from COSBench, as shown in Fig. 3.2. COSBench uses a distributed architecture, consisting of multiple *drivers*, which generate the workload and simulate many clients invoking concurrent operations on a cloud object store, and one *controller*, which controls the drivers, selects the workload parameters, collects results, and outputs aggregated statistics. In particular, the COSBench setup for VICOS reports the *operation latency*, defined as the time that an operation takes from invocation to completion, and the aggregated *throughput*, defined as the data rate between all clients and the cloud storage service.

COSBench invokes "read" and "write" operations, implemented by *vicos-get* and *vicos-put*,

respectively. Every reported data point involves read and write operations taken over a period of at least 30s after a 30s warm-up. In the experiments, two configurations are measured:

1. The *native* object storage service as a baseline, with direct unprotected access from COSBench to cloud storage, but accessing the cloud storage through the *jclouds* interface; and
2. *VICOS*, running all operations from COSBench through *jclouds* and the verification protocol.

3.5.2 Results

Cryptography microbenchmark

In a first experiment we study how different signature implementations affect the computation and network overhead of VICOS. We implemented digital signatures using RSA and DSA with 2048bit keys, and additionally HMAC-SHA1 with 128bit keys. The cryptographic algorithms are provided by the SunJCE version 1.8 provider.

We measured the time it takes on a client to sign and verify an invoke message using the three signature implementations. Figure 3.3 shows that the RSA signing takes around 5ms, while the verification takes around 220us. DSA takes around 4ms for signing and around 1.5ms for verification, whereas HMAC signing and verification only takes less than 20us. Additionally, the resulting signature sizes have a direct effect on the message sizes and the network load. RSA signatures are 256byte, while DSA signatures are only 40byte. HMAC reduces the signature size to 20byte.

We use HMAC-SHA1 as the default implementation of signatures in the remainder of the evaluation. Its operations are much faster than for RSA and DSA signatures, reducing the computation overhead at the client. Moreover, the smaller signature size of HMAC-SHA1 also reduces the network overhead of VICOS.

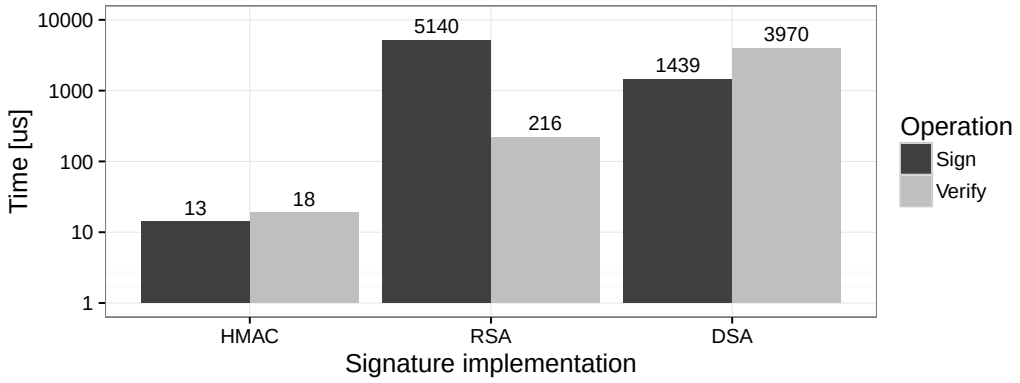

Figure 3.3: The average time for digital-signature operations (HMAC, RSA, and DSA); note that the y-axis uses log-scale.

Object size

In this experiment we study how the object size affects the latency and the throughput of VICOS. We define a workload with a single client executing read and write operations for objects of size 1 kB, 10 kB, 100 kB, 1 MB.

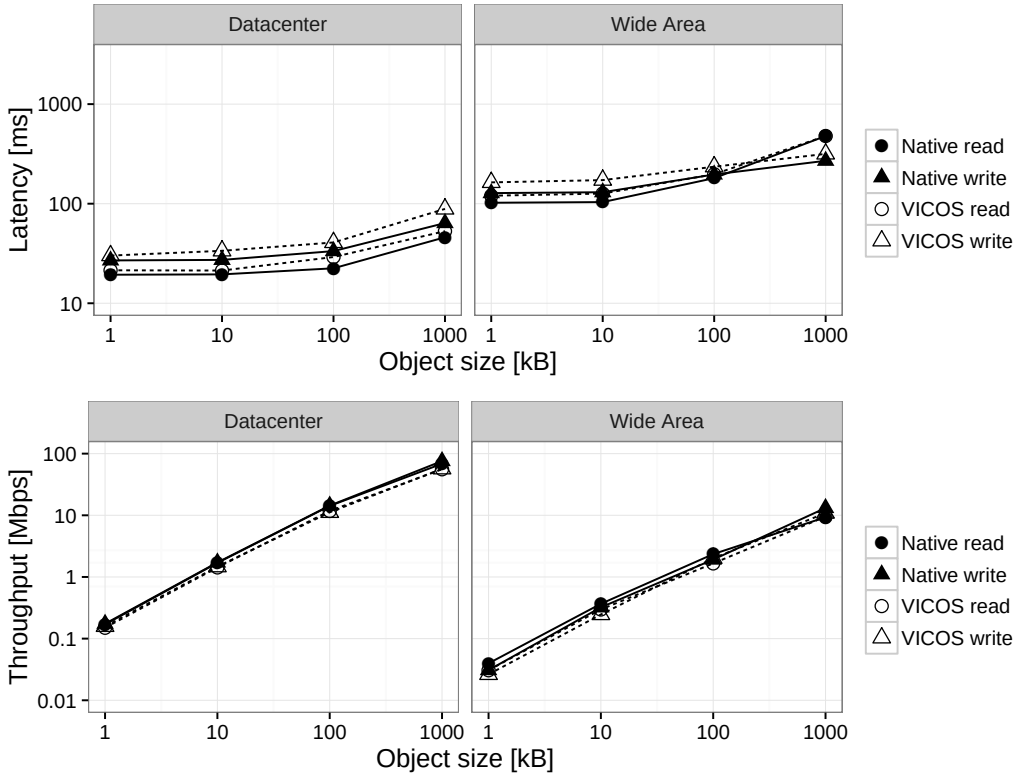

Figure 3.4: The effect of different object sizes: Latency and throughput of read and write operations with one client.

Fig. 3.4 shows that the latency and throughput of VICOS behave very similarly to those of the native system. As expected, we observe that VICOS introduces an overhead that incurs a small cost compared to unprotected access to storage. In particular, for the datacenter setting, VICOS increases latency by an average of 16.2% for *read*, and 24.0% for *write*; it decreases throughput by an average of 15.8% for *read*, and 17.7% for *write*. We also expect the overhead to decrease with bigger objects. However, we could not find this effect in the datacenter setting: here the overhead remains practically constant, from the small to the large objects. In the wide-area setting, the overhead is approximately the same for the smaller objects (1kB and 10kB), but it indeed decreases as the object size grows and disappears at the largest object size (1000kB).

Interestingly, in the wide-area setting, the relative performance in terms of throughput is reversed between *read* and *write* for 1000kB objects. Whereas *read* has lower latency and achieves better throughput than *write* in all other experiments, this relation is reversed in the right-most data points of Fig. 3.4. This may be caused by caching data on the cloud object store, which improves read performance for smaller objects, but disappears when larger objects are stored and accessed less frequently than in the datacenter setting.

Number of clients

We also study the scalability of VICOS by increasing the number of clients. The workload uses up to 128 clients (spread uniformly over the six COSBench drivers), and 64 objects with a fixed size of 10kB. One half of the objects are designated for read operations and the other half for write operations, respectively. This division prevents concurrency conflicts among the client operations.

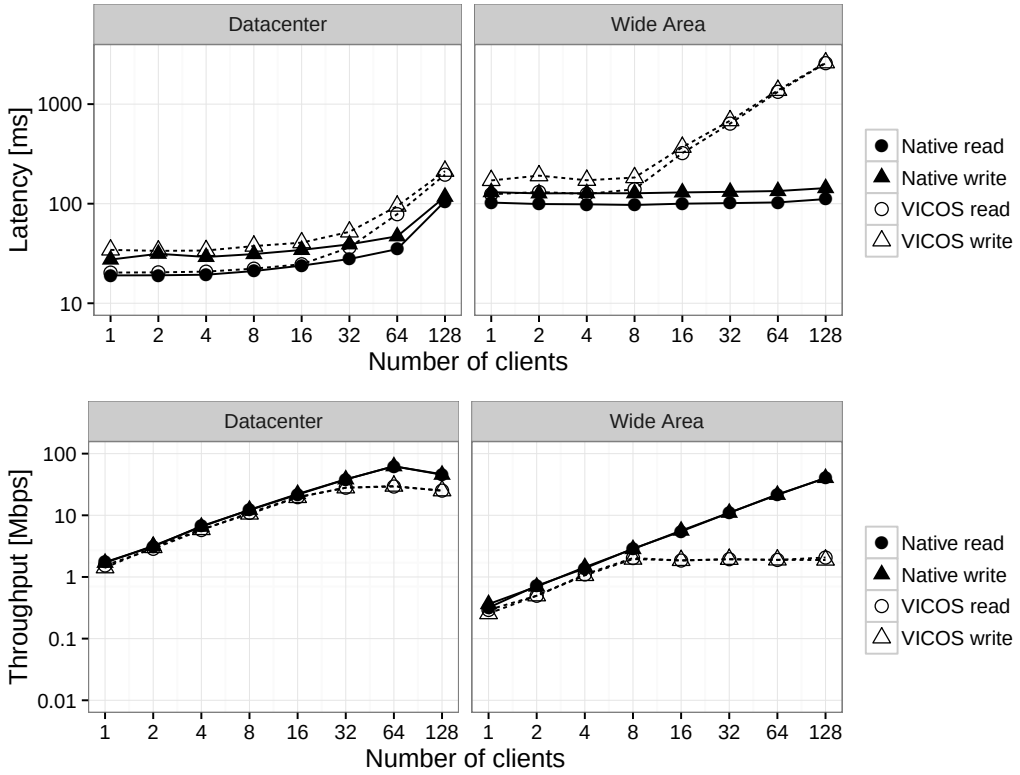

Figure 3.5: Scalability with the number of clients: Latency and throughput of read and write operations with 10kB objects.

For practical reasons, we restrict the size of the pending list in the protocol to 128 operations. We do not use a large number of clients because of the underlying assumption that clients trust each other, which might not be realistic in much larger groups.

As Fig. 3.5 shows, the native system throughput scales linearly until the network is saturated with 64 clients in the data center setting. VICOS follows the same behavior but reaches saturation already with 32 clients. In contrast, in the wide-area setting, VICOS becomes saturated with 8 clients, from where throughput remains almost constant and latency grows. No saturation is evident with the native configuration and up to 128 clients.

The reason for the slower operation in the wide-area experiment is that all requests of the clients are handled by the VICOS server sequentially and thus it becomes a bottleneck of the system. Due to the higher latency in the wide-area setting, operations remain longer in the pending queue. This means more work for the clients and the server. Since the active and passive protocol phases are asynchronous, clients may invoke the next operation already before they have completed a previous operation. Hence, they reach limit of the bounded pending queue and the throughput of VICOS remains limited at the rate imposed by the server's operation.

Concurrent operations

Finally, we investigate the effect of conflicting concurrent operations. VICOS aborts an operation if it is not compatible with one of the pending operations (according to the definition of "compatible" [BCK15]). In that case the client has to retry later. Protocols like BST [WSS09] and ACOP [CO14] are more cautious and abort as soon as two operations do *not commute*, which oc-

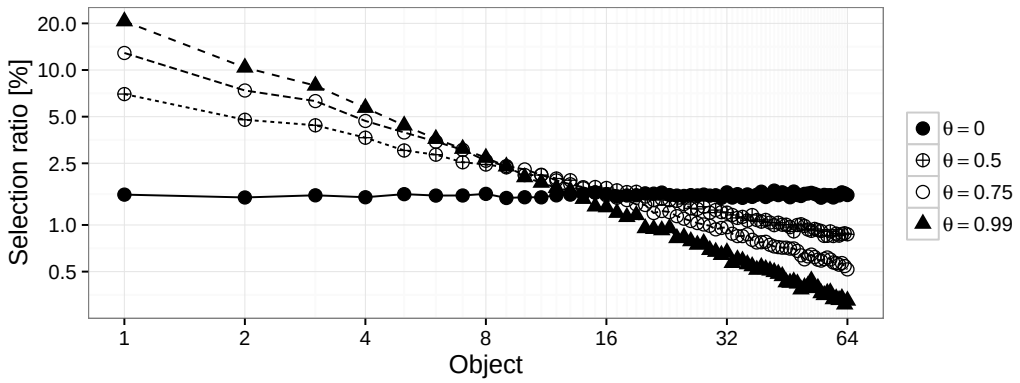

Figure 3.6: The effect of different values for the Zipf distribution factor θ of the COSBench object selector: Selection rate for each object with 10000 selections, 64 objects, and varying θ.

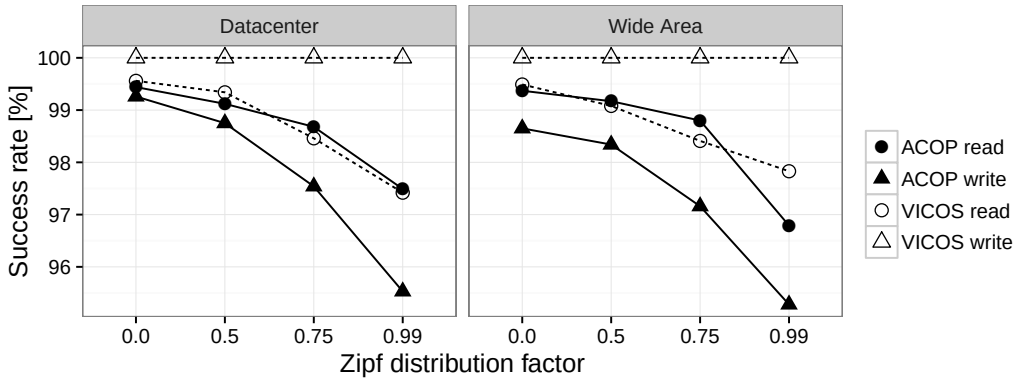

Figure 3.7: The effect of conflicting concurrent operations: Success rate of read and write operations with 10kB objects, sixteen clients, and varying Zipf distribution factors θ.

curs more often. Hence, we define ACOP as our baseline for this experiment. The implementation throws an abort exception when a conflict occurs; that causes COSBench to report the operation as failed and to continue immediately with the next operation. At the end of each experiment COSBench reports the overall operation success rate. We expect a higher success rate for VICOS compared to ACOP as already discussed in Section 3.3.2.

To evaluate this behavior we created a workload with sixteen clients each invoking read and write operations over 64 objects with a fixed size of 10kB. Object accesses are chosen according to "Zipf's law", which approximates many types of data series found in natural and social structures. The Zipf distribution is based on a ranking of the elements in a universe and postulates that the frequency of any element is inversely proportional to its rank in the frequency table. Thus, the most "popular" object will occur approximately twice as often as the second most popular one, three times as often as the third most popular and so on. Zipf distributions are often observed when users access websites [AH02]. Figure 3.6 shows the object access rate using four different values for the Zipf factor θ. For $\theta = 0.99$ the access rate for the first object (with the biggest contention) is about 20%, and the least often accessed object is selected only with probability about 0.3%. With $\theta = 0$ the Zipf distribution corresponds to uniformly random access over the objects.

Since COSBench does not support Zipf distributions by default, we implemented the algorithm of Gray et al. [GSE+94] for generating a Zipf-like access distribution, as also used in

YCSB (`https://github.com/brianfrankcooper/YCSB`). With this workload we cause operations to conflict by progressively increasing $\theta \in \{0, 0.5, 0.75, 0.99\}$. The higher the Zipf factor θ, the more clients concurrently access the same objects and invoke non-compatible operations, causing aborts.

Figure 3.7 shows the operation success rates for VICOS and for ACOP, using the four different Zipf factors. Recall from from Table 3.1 that a *put* operation in the KVS interface is always compatible with every preceding operation and never aborts. Therefore, the *write* operations show 100% success rate for VICOS. For the commuting operations in ACOP, on the other hand, *writes* are progressively more often aborted with increasing θ. The behavior of *reads* is similar for VICOS and ACOP because preceding *writes* cause aborts equally often.

The advantage of VICOS over protocols considering only operation commutativity becomes evident here, in that *write* operations always succeed, and the overall abort rate is significantly reduced.

Conclusion

The performance evaluation shows that VICOS achieves its goal of adding consistency and integrity protection while remaining almost transparent to clients using cloud object stores. The cost added over the raw performance is most visible in the datacenter setting, which is not a realistic deployment for the intended applications. Still this overhead remains limited to about 20% for accesses with high throughput (Section 3.5.2). With many clients performing operations concurrently, the extra cost may become noticeable (Section 3.5.2), and further work is needed for decreasing this. It should be added that the VICOS prototype is currently a proof of concept and not product-level code.

3.6 Conclusions

VICOS is a complete system for protecting the integrity and consistency of data outsourced to untrusted commodity cloud object stores. VICOS works with commodity cloud storage services and ensures the best possible consistency notion of fork-linearizability. It supports wait-free client operations and does not require any additional trusted components.

There are several challenges that this work does not yet address, which remain open for the future. An interesting question, for instance, is how to recover from an integrity violation. Since we assume only a single untrusted server and that client data resides at the cloud storage service, orthogonal techniques are needed for resilience of the data itself.

Another interesting challenge would be to consider malicious clients, as one further step towards a more realistic system. For small groups of clients our system model makes sense, but for groups with hundreds of clients it seems difficult to maintain this assumption. The situation is especially interesting when a client colludes with the malicious server.

Finally, the approach of AIP also be applied to services beyond cloud storage; for example, cloud and NoSQL databases, interactions in a social network, or certificate and key management services.

4. Conclusions

This deliverable presents the techniques developed by WP3 for ensuring collaborative access to data, while ensuring the confidentiality and integrity of data and of accesses to them.

Chapter 1 focuses on the protection of the confidentiality of accesses. The innovation of ESCUDO-CLOUD is represented by a novel solution for enforcing access control over the shuffle index. The approach presented in this chapter extends the original shuffle index structure studied in WP2, to enable access control enforcement on the data stored in the leaves of the index. To this purpose, it builds on the techniques for selective access illustrated in Deliverable D3.1.

Chapter 2 presents ENKI, a novel system for securely executing relational operations on encrypted, access restricted data. The innovation of ESCUDO-CLOUD is represented by a new model for encryption based access control (which defines access control restrictions at the level of attribute values) and by different techniques to support the execution of relational operations in multi user mode. To achieve this, we extend the work on key management presented in Deliverable D3.1, on the definition of a multi-party model for selective encryption in a supply chain scenario. We illustrate that our new encryption scheme, which supports several relational operations, is applicable to multiple application scenarios on SAP HANA.

Chapter 3 illustrates the advances made in protecting consistency and integrity of data stored in a cloud-object store. The innovation of ESCUDO-CLOUD is represented by the realization of a prototype of VICOS that works with the key-value store interface of commodity cloud storage services. In this chapter, we illustrate the design of VICOS and present benchmark results obtained for its evaluation.

Bibliography

[ABC+02] A. Adya, W.J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J.R. Douceur, J. Howell, J.R. Lorch, M. Theimer, and R.P. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proc. of OSDI*, Boston, MA, USA, December 2002.

[AES03] R. Agrawal, A. Evfimievski, and R. Srikant. Information sharing across private databases. In *Proc. of SIGMOD*, San Diego, CA, USA, June 2003.

[AFB05] M.J. Atallah, K.B. Frikken, and M. Blanton. Dynamic and efficient key management for access hierarchies. In *Proc. of CCS*, Alexandria, VA, USA, November 2005.

[AH02] L.A. Adamic and B.A. Huberman. Zipf's law and the internet. *Glottometrics*, 3(1):143–150, 2002.

[AKSX04] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Order preserving encryption for numeric data. In *Proc. of SIGMOD*, Paris, France, June 2004.

[ARCI13] M.R. Asghar, G. Russello, B. Crispo, and M. Ion. Supporting complex queries and access policies for multi-user encrypted databases. In *Proc. of CCSW*, Berlin, Germany, November 2013.

[BC16] M. Brandenburger and C. Cachin. First version of tools for concurrent data access. Work document W3.1, ESCUDO-CLOUD, 2016.

[BCK15] M. Brandenburger, C. Cachin, and N. Knežević. Don't trust the cloud, verify: Integrity and consistency for cloud object stores. In *Proc. of SYSTOR 2015*, Haifa, Israel, May 2015.

[BCLO12] A. Boldyreva, N. Chenette, Y. Lee, and A. O'Neill. Order-preserving symmetric encryption. *IACR Cryptology ePrint Archive*, 2012.

[BEG+94] M. Blum, W.S. Evans, P. Gemmell, S. Kannan, and M. Naor. Checking the correctness of memories. *Algorithmica*, 12(2/3):225–244, 1994.

[BG13] P. Bailis and A. Ghodsi. Eventual consistency today: Limitations, extensions, and beyond. *ACM Queue*, 11(3):20, 2013.

[CDF+09] V. Ciriani, S. De Capitani Di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati. Keep a few: Outsourcing data while maintaining confidentiality. In *Proc. of ESORICS*, Saint-Malo, France, September 2009.

[CKS11] C. Cachin, I. Keidar, and A. Shraer. Fail-aware untrusted storage. *SIAM Journal on Computing*, 40(2):493–533, 2011.

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

[CMSK07] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested append-only memory: Making adversaries stick to their word. In *Proc.of SOSP*, Stevenson, WA, USA, October 2007.

[CO14] C. Cachin and O. Ohrimenko. Verifying the consistency of remote untrusted services with commutative operations. In Marcos K. Aguilera, Leonardo Querzoni, and Marc Shapiro, editors, *Proc. of OPODIS*, Cortina d'Ampezzo, Italy, December 2014.

[CSS07] C. Cachin, A. Shelat, and A. Shraer. Efficient fork-linearizable access to untrusted shared memory. In *Proc. of PODC*, Portland, OR, USA, August 2007.

[CW11] S.A. Crosby and D.S. Wallach. Authenticated dictionaries: Real-world costs and trade-offs. *ACM TISSEC*, 14(2), 2011.

[DDF+05] E. Damiani, S. De Capitani Di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati. Key management for multi-user encrypted databases. In *Proc. of StorageSS*, Fairfax, VA, USA, November 2005.

[DDJ+03] E. Damiani, S. De Capitani di Vimercati, S. Jajodia, S. Paraboschi, and P. Samarati. Balancing confidentiality and efficiency in untrusted relational DBMSs. In *Proc. of CCS*, Washington, DC, October 2003.

[DFJ+07] S. De Capitani Di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati. Over-encryption: Management of access control evolution on outsourced data. In *VLDB*, Vienna, Austria, September 2007.

[DFJ+10] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati. Encryption policies for regulating access to outsourced data. *ACM TODS*, 35(2):12:1–12:46, 2010.

[DFJ+11] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati. Private data indexes for selective access to outsourced data. In *Proc. of WPES*, Chicago, IL, October 2011.

[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+13] S. De Capitani di Vimercati, S. Foresti, S. Paraboschi, G. Pelosi, and P. Samarati. Supporting concurrency and multiple indexes in private access to outsourced data. *JCS*, 21(3):425–461, 2013.

[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):19:1–19:55, 2015.

[DFP+16a] S. De Capitani di Vimercati, S. Foresti, S. Paraboschi, G. Pelosi, and P. Samarati. Access control for the shuffle index. In *Proc. of DBSec*, Trento, Italy, July 2016.

[DFP+16b] S. De Capitani di Vimercati, S. Foresti, S. Paraboschi, G. Pelosi, and P. Samarati. Three-server swapping for access confidentiality. *IEEE TCC*, 2016. pre-print.

[FCM13] L. Ferretti, M. Colajanni, and M. Marchetti. Access control enforcement on query-aware encrypted cloud databases. In *Proc. of CLOUDCOM*, Bristol, UK, December 2013.

[FI13] J. Furukawa and T. Isshiki. Controlled joining on encrypted relational database. In *Proc. of Pairing*, Cologne, Germany, May 2013.

[FML+12] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees. The sap hana database – an architecture overview. *IEEE DATA Engineering Bulletins*, 35(1):28–33, 2012.

[FZFF10] A.J. Feldman, W.P. Zeller, M.J. Freedman, and E.W. Felten. SPORC: Group collaboration using untrusted cloud resources. In *Proc. of OSDI*, Vancouver, BC, Canada, October 2010.

[GG14] T. Granlund and GMP Developement Team. *Manual of The GNU Multiple Precision Arithmetic Library*, 6.0.0 edition, March 2014.

[GH11] C. Gentry and S. Halevi. Implementing Gentry's fully-homomorphic encryption scheme. In *Proc. of EUROCRYPT*, Tallinn, Estonia, May 2011.

[GPSW06] V. Goyal, O. Pandey, A. Sahai, and B. Waters. Attribute-based encryption for fine-grained access control of encrypted data. In *Proc. of CCS*, Alexandria, VA, October-November 2006.

[GPTT08] M.T. Goodrich, C. Papamanthou, R. Tamassia, and N. Triandopoulos. Athos: Efficient authentication of outsourced file systems. In *Proc. of ISC*, Taipei, Taiwan, September 2008.

[GSE+94] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P.J. Weinberger. Quickly generating billion-record synthetic databases. In *Proc. of SIGMOD*, Minneapolis, MN, USA, May 1994.

[GSMB03] E.-J. Goh, H. Shacham, N. Modadugu, and D. Boneh. SiRiUS: Securing remote untrusted storage. In *Proc. of NDSS*, San Diego, CA, USA, February 2003.

[HBS73] C. Hewitt, P. Bishop, and R. Steiger. A universal modular ACTOR formalism for artificial intelligence. In *Proc. of IJCAI*, Stanford, USA, August 1973.

[HIML02] H. Hacigümüş, B. Iyer, S. Mehrotra, and C. Li. Executing SQL over encrypted data in the database-service-provider model. In *Proc. of SIGMOD*, Madison, WI, June 2002.

[HKD15] I. Hang, F. Kerschbaum, and E. Damiani. ENKI: Access control for encrypted query processing. In *Proc. of SIGMOD*, Melbourne, Australia, May 2015.

[KFP+16] N. Karapanos, A. Filios, R.A. Popa, , and S. Capkun. Verena: End-to-end integrity protection for web applications. In *Proc. of IEEE S&P*, San Jose, CA, USA, May 2016.

[KS14] F. Kerschbaum and A. Schröpfer. Optimal average-complexity ideal-security order-preserving encryption. In *Proc. of CCS*, Scottsdale, AZ, USA, November 2014.

[KTZ11] M. Komlenovic, M.V. Tripunitara, and T. Zitouni. An empirical assessment of approaches to distributed enforcement in role-based access control (RBAC). In *Proc. of CODASPY*, San Antonio, TX, USA, February 2011.

[Lau14] B. Laurie. Certificate transparency. *Communications of the ACM*, 57(10):40–46, September 2014.

[LC04] P. Lin and K.S. Candan. Hiding traversal of tree structured data from untrusted data stores. In *Proc. of WOSIS*, Porto, Portugal, April 2004.

[LKMS04] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *Proc.of OSDI*, San Francisco, CA, December 2004.

[Lyn07] B. Lynn. *PBC Library Manual*. Stanford University, 2007.

[MDSS09] M. Majuntke, D. Dobre, M. Serafini, and N. Suri. Abortable fork-linearizable storage. In *Proc. of OPODIS*, Nimes, France, December 2009.

[MND+04] C. Martel, G. Nuckolls, P. Devanbu, M. Gertz, A. Kwong, and S.G. Stubblebine. A general model for authenticated data structures. *Algorithmica*, 39(1):21–41, 2004.

[MS02] D. Mazières and D. Shasha. Building secure file systems out of byzantine storage. In *Proc. of PODC*, Monterey, California, July 2002.

[NN00] M. Naor and K. Nissim. Certificate revocation and certificate update. *IEEE J. Selected Areas in Communications*, 18(4):561–570, April 2000.

[Ora10] Oracle Inc. Transparent data encryption: New technologies and best practices for database encryption. http://www.oracle.com/us/products/database/sans-tde-wp-178238.pdf, 2010.

[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.

[PH78] S. Pohlig and M. Hellman. An improved algorithm for computing logarithms over GF(p) and its cryptographic significance (corresp.). *IEEE Trans. Inf. Theor.*, 24(1):106–110, 1978.

[PLM+11] R.A. Popa, J.R. Lorch, D. Molnar, H. J. Wang, and L. Zhuang. Enabling security in cloud storage SLAs with CloudProof. In *Proc. of USENIX*, Portland, OR, USA, June 2011.

[PLZ13] R.A. Popa, F.H. Li, and N. Zeldovich. An ideal-security protocol for order-preserving encoding. In *Proc. of IEEE S&P*, San Francisco, CA, USA, May 2013.

[PRZB11] R.A. Popa, C.M.S. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Proctecting confidentiality with encrypted query processing. In *Proc. of SOSP*, Cascais, Portugal, October 2011.

[PZ13] R.A. Popa and N. Zeldovich. Multi-key searchable encryption. *IACR Cryptology ePrint Archive*, 2013.

[RMSR04] S. Rizvi, A. Mendelzon, S. Sudarshan, and P. Roy. Extending query rewriting techniques for fine-grained access control. In *Proc. of SIGMOD*, Paris, France, June 2004.

[SCC+10] A. Shraer, C. Cachin, A. Cidon, I. Keidar, Y. Michalevsky, and D. Shaket. Venus: Verification for untrusted cloud storage. In *Proc. of CCSW*, Chicago, IL, USA, October 2010.

[SHV01] G. Saunders, M. Hitchens, and V. Varadharajan. Role-based access control and the access control matrix. *SIGOPS Oper. Syst. Rev.*, 35(4):6–20, October 2001.

[SS13] E. Stefanov and E. Shi. ObliviStore: High performance oblivious cloud storage. In *Proc. of IEEE S&P*, San Francisco, CA, May 2013.

[SvDO12] E. Stefanov, A. van Dijk, M.and Juels, and A. Oprea. Iris: A scalable cloud file system with efficient integrity checks. In *Proc. of ACSAC*, Orlando, FL, USA, December 2012.

[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.

[SWP00] D.X. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *Proc. of IEEE S&P*, Berkeley, CA, USA, May 2000.

[TKMZ13] S. Tu, M.F. Kaashoek, S. Madden, and N. Zeldovich. Processing analytical queries over encrypted data. *PVLDB*, 6:289–300, March 2013.

[Ver13] F. Vercauteren. Final report on main computational assumptions in cryptography. Deliverable ICT-2007-216676, European Network of Excellence in Cryptography II, January 2013.

[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, 2012.

[WSC08] P. Williams, R. Sion, and B. Carbunar. Building castles out of mud: Practical access pattern privacy and correctness on untrusted storage. In *Proc. of CCS*, Alexandria, VA, October 2008.

[WSS09] P. Williams, R. Sion, and D. Shasha. The blind stone tablet: Outsourcing durability to untrusted parties. In *Proc. of NDSS*, San Diego, CA, USA, February 2009.

[YBDD09] Y. Yang, F. Bao, X. Ding, and R.H. Deng. Multiuser private queries over encrypted databases. *Int. J. Appl. Cryptol.*, 1(4):309–319, 2009.

[YC07] A.R. Yumerefendi and J.S. Chase. Strong accountability for network storage. *ACM Transactions on Storage*, 3(3), 2007.