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

D3.5
Report on Secure Information Sharing in the Cloud

Editors: Daniel Bernau (SAP)
Andreas Fischer (SAP)
Anselme Kemgne Tueno (SAP)
Reviewers: Gery Ducatel (BT)
Andrew Byrne (EMC)

Abstract

This deliverable presents the contributions and findings that were produced by Tasks 1 to 4 under Work Package 3 (WP3). Within ESCUDO-CLOUD WP3 is focusing on formulating solutions for secure information sharing in the Cloud. This topic has been approached by Tasks 1 - 4 on Secure Sharing, Secure multi-user interactions and sharing, Support for collaborative queries, and Security testing. To provide a comprehensive overview this deliverable presents the latest novel achievements and recapitulates previous work that has already been presented in deliverables of WP3.

Type	Identifier	Dissemination	Date
Deliverable	D3.5	Public	2017.10.31

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	2017.10.05	Initial Release
0.2	2017.10.26	Second Release
1.0	2017.10.31	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	Daniel Bernau (SAP)
Chapter 1: Selective sharing	Daniel Bernau (SAP), Anselme Kemgne Tueno (SAP), Andreas Fischer (SAP), Sabrina De Capitani di Vimercati (UNIMI), Sara Foresti (UNIMI), Stefano Paraboschi (UNIBG), Pierangela Samarati (UNIMI)
Chapter 2: Secure multi-user interactions and sharing	Björn Tackmann (IBM), Christian Cachin (IBM)
Chapter 3: Support for collaborative queries	Enrico Bacis (UNIBG), Sabrina De Capitani di Vimercati (UNIMI), Sara Foresti (UNIMI), Giovanni Livraga (UNIMI), Stefano Paraboschi (UNIBG), Marco Rosa (UNIBG), Pierangela Samarati (UNIMI), Roberto Sassi (UNIMI)
Chapter 4: Security testing	Ahmed Taha (TUD), Nicolas Coppick (TUD), Ruben Trapero (TUD), Neeraj Suri (TUD)
Chapter 5: Conclusion	All Partners

Contents

Executive Summary **9**

1 Selective sharing **11**
1.1 ESCUDO-CLOUD Innovation 11
1.2 Selective Encryption 11
1.3 Access Control for the Shuffle Index 12
1.3.1 Data and Policy Updates 13
1.3.2 Analysis 18
1.4 Search over encrypted data 25
1.5 Oblivious Order Preserving Encryption 26
1.5.1 Problem Statement 26
1.5.2 Stateless Order-preserving Encryption 27
1.5.3 Stateful Order-preserving Encryption 28
1.5.4 Cryptographic Background 29
1.5.5 Overview of the protocol 30
1.5.6 Description of the protocol 31
1.5.7 Protocol for Integer Comparison 35
1.5.8 Implementation and Evaluation 36
1.6 Summary 38

2 Secure multi-user interactions and sharing **40**
2.1 ESCUDO-CLOUD innovation 40
2.2 VICOS 40
2.2.1 Concepts: The ADT integrity protocol (AIP) 41
2.2.2 System architecture and components 42
2.2.3 Conclusion 43
2.3 Stateful multi-client verifiable computation 44
2.3.1 Overview 44
2.3.2 Authenticated data types 45
2.3.3 A general-purpose instantiation of ADT 45
2.3.4 Computational fork-linearizable Byzantine emulation 46
2.3.5 A lock-step protocol for emulating shared data types 46
2.3.6 Conclusion 46
2.4 Summary 47

3 Support for collaborative queries **48**
3.1 ESCUDO-CLOUD Innovation . . . 48
3.2 Twins and Markers for Integrity Verification . . . 49
3.2.1 Slim Twins and Markers . . . 49
3.2.2 Twins and Markers Analysis . . . 50
3.3 Mix&Slice For Efficient Access Revocation . . . 53
3.3.1 Mix&Slice . . . 53
3.3.2 Access management . . . 54
3.3.3 Effectiveness of the approach . . . 56
3.3.4 Implementation . . . 58
3.4 Collaborative Queries in the Cloud with Access Restrictions . . . 65
3.4.1 Rationale and Running Example . . . 65
3.4.2 Authorization Model . . . 67
3.4.3 Relation Content Model . . . 68
3.4.4 Authorization Enforcement . . . 69
3.5 Summary . . . 71

4 Security testing **73**
4.1 ESCUDO-CLOUD Innovation . . . 73
4.2 Techniques and Approaches for Security Testing . . . 74
4.2.1 Security Testing Techniques Overview . . . 74
4.2.2 Testing Support Tools . . . 74
4.2.3 Testing Multi-Threaded Software and Systems . . . 74
4.2.4 Testing the Security Lifecycle: Compliance Monitoring . . . 75
4.3 C'MON: Monitoring the Compliance of Cloud Services to Contracted Properties 75
4.3.1 Terminology . . . 76
4.3.2 The C'MON Cloud Monitoring Approach . . . 77
4.3.3 SLO Classification . . . 77
4.3.4 SLO Measurement Definition . . . 78
4.3.5 C'MON Monitoring Framework . . . 83
4.3.6 Evaluation . . . 85
4.4 Summary . . . 91

5 Conclusion **92**

Bibliography **94**

List of Figures

1.1 An example of a supply chain . . . 12
1.2 An example of a relation with *acls* associated with its resources (a), relation for the primary index (b), and relation for the secondary index (c) . . . 14
1.3 Primary shuffle index for the relation in Figure 1.2(b) . . . 14
1.4 Secondary shuffle index for the relation in Figure 1.2(c) . . . 15
1.5 Relation of Figure 1.2(a) with *acl*s associated with its resources (a) and relation for the primary index (b) when tokens are stored in leaf nodes . . . 17
1.6 Costs (USD) of data storage, access, and transfer per month, depending on the size of the primary index leaf blocks and of the fixed size of the secondary index (30TiB) 24
1.7 Total monthly cost (USD) varying the size of the primary index leaf blocks . . . 24
1.8 Scenario for Oblivious Order-Preserving Encryption . . . 26
1.9 Example initialization . . . 32
1.10 Oblivious Order-Preserving Encryption Protocol . . . 33
1.11 Oblivious Comparison Protocol . . . 34
1.12 Encryption Cost of OOPE . . . 37
1.13 Cost of oblivious comparison . . . 37
1.14 OPE-tree costs . . . 38

2.1 Protocol messages in AIP. . . . 41
2.2 VICOS architecture . . . 43

3.1 Probability that the misbehavior of a CSP goes undetected when using only markers 51
3.2 Probability that CSP misbehavior goes undetected when using only twins . . . 52
3.3 An example of fragments evolution . . . 54
3.4 Revoke on resource R . . . 55
3.5 Access to resource R . . . 56
3.6 Throughput varying the number of threads . . . 60
3.7 Time for the execution of `get` requests on Swift . . . 62
3.8 Throughput for a workload combining `get` and `put_fragment` requests on Swift 63
3.9 Throughput for a workload combining `get` and `put_fragment` requests with Swift DLOs . . . 64
3.10 Configurations for physical blocks . . . 65
3.11 An example of a query plan (a) and of authorizations on relations HOSP and INS (b) 66
3.12 An extended query plan . . . 70

4.1 Quantitative SLO Classification . . . 78
4.2 Monitoring Framework Architecture . . . 84
4.3 OpenStack Cloud Platform Architecture . . . 85

4.4 Outages Distribution per Month 86
4.5 Monitoring Server CPU Total Time 90
4.6 Dashboard CPU Total Time 90
4.7 Dashboard File I/O Throughput 91

Executive Summary

This deliverable provides an integrated view of the contributions and findings that were produced by Tasks 1 to 4 under WP3. The aim of WP3 within ESCUDO-CLOUD is to formulate solutions for secure information sharing in the Cloud. These solutions were designed to offer means of selective access control on data stored at the Cloud Service Provider (CSP), integrity checks of outsourced data stored at the CSP, collaborative query execution over encrypted data stored at the CSP, and application-level verification of expected security guarantees. For this, the work was segmented across four tasks:

Selective sharing (M4-M34). Task 3.1 provides means for enabling selective sharing of protected data with other users in the Cloud. Selective Encryption is combined with the Shuffle Index. In addition, Selective Encryption is applied in a supply chain context to foster information sharing between the supply chain parties. Furthermore, T3.1 formulates multi-user encryption schemes that enable search over encrypted data in the Cloud. Privacy of participants in Use Case 2 is improved by the novel Oblivious Order Preserving Encryption Protocol (OOPE).

Secure multi-user interactions and sharing (M7-M34). The solutions in Task 3.2 guarantee integrity and correctness of the data stored in an untrusted Cloud, and the responses of computations using that data in the presence of write operations by multiple users. This is achieved through distributed consistency protocols that are built on top of authenticated data types.

Support for collaborative queries (M7-M34). Task 3.3 analyzes probabilistic approaches for providing integrity guarantees, in terms of correctness and completeness, of computation results in collaborative scenarios. The proposed solutions are then refined to reduce the costs of integrity verification, while not impacting integrity. Task 3.3 also addressed the problem of enforcing selective access restrictions to outsourced data in collaborative scenarios. The task specifically proposes a solution for efficiently and effectively revoking authorizations by re-encrypting a small fragment of revoked resources, with limited overhead for the resource owner. It also proposes a solution for specifying and enforcing authorizations enabling controlled data sharing for collaborative queries in the Cloud.

Security testing (M10-M34). Task 3.4 provides security testing mechanisms that help to guarantee the protection of the Cloud customer data. While typically it is not possible to actually access the CSPs to check the security mechanisms implementation as they do not permit access to binaries (and of course also not to the code), T3.4 grouped and compared the testing techniques considering varied levels of parameter access that allow the user to determine in what aspect of the service they should be applied. Moreover, extending testing to the security lifecycle T3.4 proposes a customer-side monitoring framework for monitoring the compliance of Cloud services to the contracted properties in the agreed service level agreements.

The remainder of this document is organized as follows. Chapter 1 provides a presentation of the work done on task T3.1. Chapter 2 highlights the work performed on T3.2. Chapter 3 summarizes T3.3. Chapter 4 states the findings around T3.4. Finally, Chapter 5 summarises overall conclusions on the findings of WP3 and future work.

1. Selective sharing

Work under this task is focused on data ownership and the selective data sharing. The goal is to enforce fine grained access restrictions under the presence of an untrusted Cloud Service Provider (CSP). Two deliverables especially addressed challenges in selective sharing, namely D3.1 on Techniques for Selective Access and D3.3 on Techniques for Selective and Secure Data Sharing. In the following, Section 1.1 will present an overview of the contributions within T3.1. Sections 1.2 to 1.6 present the work on these innovations.

1.1 ESCUDO-CLOUD Innovation

This task produced several advancements over the state-of-the-art.

- *Key hierarchies for encryption-enforced access control* and a *multi-user encryption scheme tailored for queries on supply chain databases*. By applying Selective Encryption to the data contained on RFID modules that are attached to goods moving between members of supply chain we realize visibility policies (i.e., regulate access to data associated with objects received from supply chain partners).
- We integrated *Selective Encryption to the Shuffle Index* introduced in WP2 to extend its ability to support selective sharing in contexts where access confidentiality needs to be guaranteed in addition to access control.
- While we succeeded in *increasing the security of Order Preserving Encryption (OPE) schemes*, OPE is still limited to two parties. To extend the applicability of OPE to multiple parties, *we combine tree based OPE with garbled circuits and homomorphic encryption*.
- We formulated a *searchable encryption scheme for multiple users* and integrated it into an industrial grade database for evaluation.

1.2 Selective Encryption

D3.1 evaluated Selective Encryption as a mean to enforce access control in Cloud scenarios. Selective Encryption represents an effective solution for enforcing access control restrictions over data stored at an external, possibly not fully trusted, Cloud provider. In principle, Selective Encryption is realized by encrypting different data with different encryption keys, and in distributing keys in such a way that the key used to protect data is known to only the users authorized. Consequently, by utilising Selective Encryption data is effectively self-enforcing access control. Authorization policies can thus be translated into equivalent encryption policies, where equivalence is realized by the use of token-based key derivation. The foundations of Selective Encryption with key hierarchies in D3.1 led to two derived contributions by ESCUDO-CLOUD.

First, a Selective Encryption scheme for the authentication of (authorized) members within a supply chain was formulated and evaluated. These members move physical objects, to which an RFID module containing handling information of the other members is attached, through the supply chain. Specifying access control rules for the tuples for such a data handling scenario can be very delicate. Imagine a supplier p_2 selling a product o_2 to buyers p_3 and p_4 (Figure 1.1). If buyer p_3 has access to all scheduled orders for o_2, she can infer the volume of future business with p_4. This can be very sensitive, in case supplier p_2 has to cancel some orders due to a temporary capacity reduction (e.g., a machine failure). Buyer p_3 could then infer whether p_4's orders are treated preferentially. While this decision can be based on local information, in the case of bridging only one supply chain stage, it becomes difficult in case of a tier-2 supplier (i.e., a supplier's supplier). As another example, imagine a supplier p_1 selling product o_1 to p_2 which is then used by p_2 to produce o_2. Again, assume that p_2 later sells object o_2 to players p_3 and p_4. If either buyer p_3 or p_4 contacts supplier p_1 requesting data, p_1 cannot decide which object was shipped to which buyer. If supplier p_1 would grant access to all items, buyer p_3 could infer again the volume of business of p_4. An emerging access rule is to share data with partners about shared objects, that is, objects a group of partners have possessed.

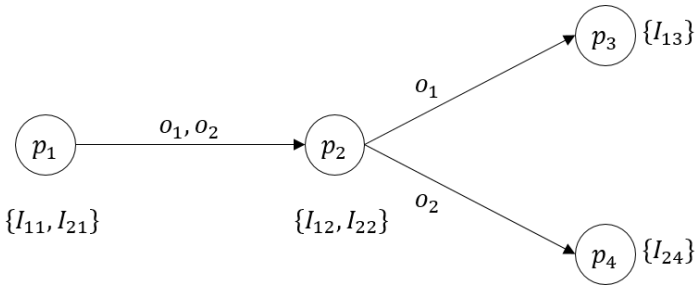


Figure 1.1: An example of a supply chain

We advanced the state-of-the-art by formulating an approach based on Selective Encryption for enforcing visibility policies in supply chain scenarios, where data about objects are stored at an external CSP. To the best of our knowledge, our authentication protocol is among the first addressing secure interaction among participants of an RFID-enhanced supply chain. Furthermore, we have shown that it can significantly reduce the administration burden by requiring only a small set of policy rules. The formulated approach enables companies to profitably use the sharing services offered by the Cloud for physical object data, while enforcing restrictions on the visibility of sensitive data. Our technique has the advantage of flexibility and enables the enforcement of arbitrary visibility policies over objects' data.

Second, leveraging Selective Encryption solutions developed in this task, we have extended the application of Selective Encryption to a shuffle index data structure. The proposed solution offers selective sharing capability, while providing access and pattern confidentiality. This work was presented to full extend in D3.3. This proposal has then been extended to support policy updates, as illustrated in Section 1.3

Both works resulted in publications, namely [DFP+16, DFP+17] and [KKBF17].

1.3 Access Control for the Shuffle Index

The shuffle index [DFP+15] approach has been studied in WP2 as a solution for providing access and pattern confidentiality. Access confidentiality is necessary for protecting data since breaches to

access confidentiality may leak information on access profiles, and, in the end, even on the data themselves [IKK14].

The shuffle index is a dynamically allocated data structure organized, at the logical level, as an unchained $B+$-tree. The shuffle index provides access and pattern confidentiality by combining three protection techniques: cover searches, repeated searches, and shuffling.

The shuffle index proposal (illustrated in D2.1) provides users knowing the encryption key access to the entire outsourced data collection. To address scenarios where users should be authorized for a different view over the data, D3.3 presented an extension of the shuffle index for fine grained access control enforcement. The proposed solution leverages selective encryption for access control enforcement on the data stored in a shuffle index. To protect the confidentiality of index values, as well as of data content, our proposal is based on the combined use of two shuffle indexes: a *primary index*, storing selectively encrypted data indexed according to an additional indexing attribute whose values are computed and known to the data owner only; and a *secondary index*, enforcing the access control policy by selectively providing users access to primary index values. Consider, as an example, the relation and access control policy in Figure 1.2(a). The primary and secondary indexes enforcing such a policy are illustrated in Figures 1.2(b-c) and graphically reported in Figures 1.3 and 1.4, respectively.

D3.3 presents the details of the proposed approach for enforcing access control restrictions over the shuffle index. The following subsections extend the proposal in D3.3, illustrating an approach for the management of insertion, deletion, and update of resources, as well as for granting and revoking authorizations. An analysis of the correctness and performance/economic overhead of the solution will also be presented.

1.3.1 Data and Policy Updates

Changes in the authorization policy can be of three types: *1)* insertion/deletion/update of a resource; *2)* insertion/deletion of a user; and *3)* grant/revoke of an authorization. We note that the insertion/deletion of a user has an impact on the policy only when the user is involved in authorizations. In the following, we then focus on the insertion, removal, and update of resources and on grant and revoke of authorizations. For simplicity, we assume that each operation refers to a single resource r and a single user u (extensions to sets of tuples and users are immediate).

Removal, insertion, and update of a resource

An observer can recognize operations that remove, insert, or update a resource from read-only accesses whenever they require a change in the structure of the primary and/or secondary indexes. To make them indistinguishable from read-accesses, we adopt the approach in [DFP+15]. This solution prevents the removal of nodes from the structure by marking removed tuples as non valid, while it adopts probabilistic splits to make the insertion of new tuples indistinguishable from read accesses. Intuitively, to prevent an observer from discriminating insert operations when a node is split, nodes are possibly split also when visited by a read access, according to the result of a random function.

- *Removal*. The removal of a resource r requests the removal of the encoded value $\iota(r[I])$ as well as all values $\iota_i(r)$, $\forall u_i \in acl(r)$ from the leaves of the primary and secondary index, respectively. These values are therefore searched in the primary and secondary index and the corresponding resources are marked as 'non-valid' (e.g., encrypted with a key known

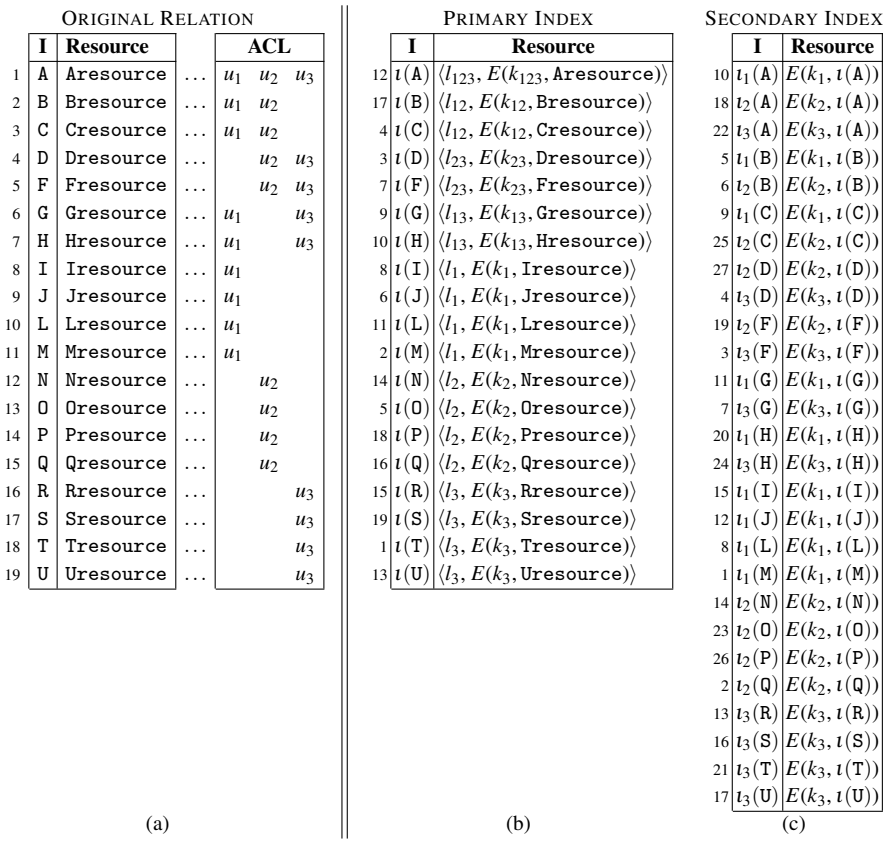


Figure 1.2: An example of a relation with *acls* associated with its resources (a), relation for the primary index (b), and relation for the secondary index (c)

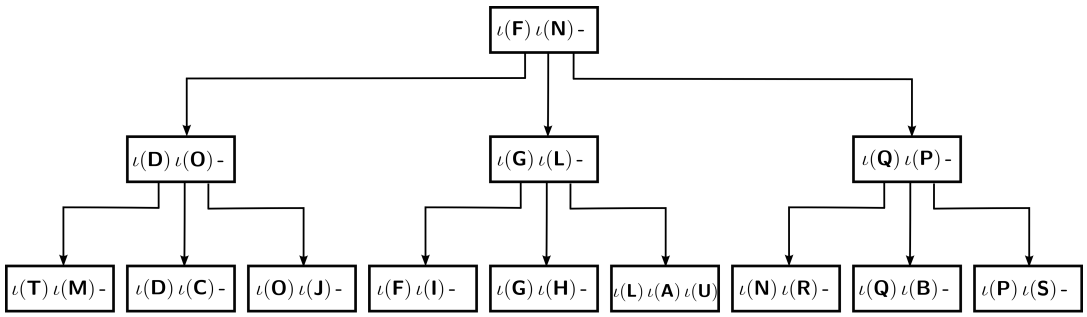


Figure 1.3: Primary shuffle index for the relation in Figure 1.2(b)

to the data owner only, or overwritten with a dummy content) [DFP+15]. For instance, the removal of resource `Cresource` with index value `C` requires to set as non-valid the resources with index values $\iota(\texttt{C})$ (primary index) and $\iota_1(\texttt{C})$, $\iota_2(\texttt{C})$ (secondary index). Note that, for the working of the system, the index values in the secondary index do not need to be removed or set as non-valid. Indeed, the user will discover the absence of the resource of interest when visiting the primary index.

- *Insertion.* The insertion of a new resource r with $acl(r)$ requires the insertion of a new tuple p in the primary index, having $p[I]{=}\iota(r[I])$ and containing the original resource in encrypted form, that is, $p[Resource]{=}\langle l_{i_1,\ldots,i_n}, E(k_{i_1,\ldots,i_n}, r[Resource])\rangle$, with $acl(r){=}\{u_{i_1},\ldots,u_{i_n}\}$. Anal-

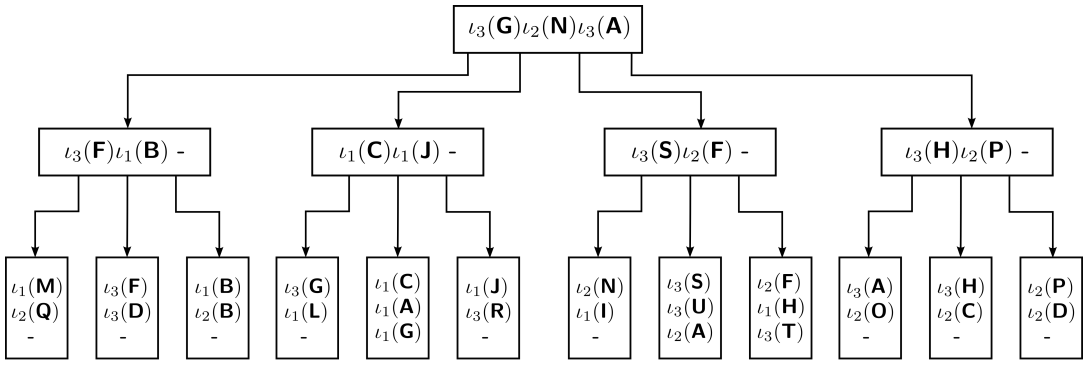


Figure 1.4: Secondary shuffle index for the relation in Figure 1.2(c)

ogously, for each user u_i in $acl(r)$ a new tuple s is inserted in the secondary index, having $s[I] = \iota_i(r[I])$ and $s[Resource] = E(k_i, \iota(r[I]))$. For instance, the insertion of a new resource `Zresource` with index value `Z` and $acl(\texttt{Z})=\{u_2,u_3\}$ requires the insertion of tuple p with $p[Resource]=\langle l_{23}, E(k_{23},\texttt{Zresource})\rangle$ and index value $p[I]=\iota(\texttt{Z})$ in the primary index, and of two tuples s_1 and s_2 in the secondary index, with $s_1[I]=\iota_2(\texttt{Z})$, $s_1[Resource]=\langle E(k_2,\iota(\texttt{Z}))\rangle$, and $s_2[I]=\iota_3(\texttt{Z})$, $s_2[Resource]=\langle E(k_3,\iota(\texttt{Z}))\rangle$.

- *Update*. The update of a resource may have an impact on the primary and secondary indexes only when it requires a change in the index value. In fact, if the index value does not change, it is sufficient to search for the resource to be updated, and to modify its encrypted representation in the primary index during the access operation. On the contrary, when the index value associated with the resource needs to be updated, it is necessary to modify the primary index to move the resource to the correct leaf, and the secondary index to enable users to retrieve the resource when searching for its new index value. Such an update can be seen (and realized) as the removal of a resource followed by the insertion of the same resource with a new value for the index attribute.

Grant and revoke

Grant and revoke operations require the insertion and removal of tuples in the primary and/or secondary index, which are again performed according to the approach described in [DFP+15].

- *Grant*. A request to grant user u_i access to resource r requires a change only in the secondary index to allow user u to retrieve the encoded value $\iota(r[I])$. The data owner then inserts a new tuple s in the secondary index with $s[I]=\iota_i(r[I])$ and $s[Resource]=E(k_i,\iota(r[I]))$. Also, the data owner re-encrypts resource r in the primary index, using an encryption key that also u_i can derive (i.e., the key associated with the new access control list $acl(r)$ of the resource). For instance, to grant user u_3 access to resource `Cresource`, the data owner inserts a tuple with index value $\iota_3(\texttt{C})$ and content $\langle E(k_3,\iota(\texttt{C}))\rangle$ in the secondary index. Also, she re-encrypts the content of the tuple with index value $\iota(\texttt{C})$ in the primary index, using key k_{123}.

- *Revoke*. To prevent users from distinguishing between the removal of a resource that she is authorized to access and the revoke of her access privilege for the same resource, the encoding $\iota(r[I])$ of the index value of the revoked resource must be changed. The primary and the secondary indexes must then be updated accordingly. The data owner then computes a new

encoded value $\iota(r[I], salt)$ for the resource by concatenating a random *salt* with the original index value $\iota(r[I], salt)$=h(k_o,$r[I]||salt$). The data owner removes from the primary index the tuple p_{old} storing the resource before the policy update, and inserts a new tuple p_{new} storing the resource after the policy update. The new tuple p_{new} has index value $p_{new}[I]$=$\iota(r[I], salt)$ and stores resource r, encrypted with a key known to authorized users only (i.e., all users X in $acl(r)$ but u_i), that is, $p_{new}[Resource]$=$\langle l_X$, $E(k_X$,$r[Resource])\rangle$. Since the encoded value associated with the revoked resource has been updated, the data owner must modify the secondary index, to enable authorized users to retrieve the resource. For each authorized user u_j, the data owner updates the tuple s in the secondary index with $s[I]$=$\iota_j(r[I])$, setting its content to $s[Resource]$=$E(k_j$,$\iota(r[I], salt))$. For instance, assume that user u_1 is revoked access to resource `Cresource`. The data owner first re-computes the encoded value for `C`, $\iota(\texttt{C}, salt)$, removes from the primary index tuple p_{old} with $p_{old}[I]$=$\iota(\texttt{C})$, and inserts a new tuple p_{new} with the same content as the removed one, but encrypted with k_2, $p_{new}[Resource]$=$\langle l_2$, $E(k_2$, `Cresource`) $\rangle$ and index value $p_{new}[I]$=$\iota(\texttt{C}, salt)$. The data owner then updates the secondary index: she removes tuple s_1 with index value $s_1[I]$=$\iota_1(\texttt{C})$, and updates the content of tuple s_2 with index value $s_2[I]$=$\iota_2(\texttt{C})$, setting $s_2[Resource]$=$E(k_2$,$\iota(\texttt{C}, salt))$.

We observe that, to guarantee that the server cannot recognize read accesses from modifications to the dataset or the access policy, every access to the primary index must be preceded by an access to the secondary index (and every access to the secondary index must be followed by an access to the primary index). Whenever the access to the primary (or secondary) index is not necessary for an update operation, the data owner performs a read access searching for a random value.

Token management for efficient revocation

The solution described above for the management of revoke operations, although effective, implies a high communication cost for the data owner, especially if the revoked resource can be accessed by many users. Indeed, the data owner needs to access the secondary index as many times as the number of users in the *acl* of the revoked resource. An alternative approach that limits the overhead of revoke management consists in using a different encryption key for each resource. Each resource is then associated with a set of tokens, enabling authorized users to derive the encryption key [AFB05]. To revoke a user access to a resource, the data owner re-encrypts the resource and modifies its tokens, without the need to update the secondary index.

Formally, each resource r_i is encrypted with an encryption key k_i randomly chosen by the data owner and used exclusively for it. The set of keys used for the encryption policy is then defined as follows.

Definition 1.3.1 (Encryption Policy Keys with Tokens) *Let $\mathscr{R}(I,Resource)$ be a relation, U be a set of users, and, $\forall r \in \mathscr{R}$,* acl$(r) \subseteq U$ *be the acl of r. The set $\mathscr{K}$ of* encryption policy keys *for $\mathscr{R}$ is a set $\mathscr{K} = \{k_i \mid u_i \in U\} \cup \{k_j \mid r_j \in \mathscr{R}\}$ of encryption keys. Each key $k_i \in \mathscr{K}$ has a* public label l_i. *Each user $u_i \in U$ knows the set $\mathscr{K}_i = \{k_i\} \cup \{k_j \mid k_j \in \mathscr{K} \wedge u_i \in \text{acl}(r_j)\}$ of keys.*

Each resource r_i is complemented, in the primary index, with a *token* $t_{j,i}$ for each user u_j. Token $t_{j,i}$ enables u_j to derive k_i from her key k_j, if $u_j \in acl(r_i)$; $t_{j,i}$ is a random string of the same length as real tokens, otherwise. Token $t_{j,i}$ enabling key derivation is defined as $t_{j,i}$=$k_i \oplus$h(k_j,l_i), where l_i is a publicly available label associated with k_i, $\oplus$ is the bitwise xor operator, and h is a deterministic cryptographic function. Given the set $U = \{u_1, \ldots, u_n\}$ of users, the tuple p_i representing a resource r_i in the primary index has content $p_i[Resource]$=$\langle l_i$, $E(k_i$,$r[Resource])$,$t_{1,i}, \ldots, t_{n,i}\rangle$.

ORIGINAL RELATION

	I	Resource		ACL		
1	A	Aresource	...	u_1	u_2	u_3
2	B	Bresource	...	u_1	u_2	
3	C	Cresource	...	u_1	u_2	
4	D	Dresource	...		u_2	u_3
5	F	Fresource	...		u_2	u_3
6	G	Gresource	...	u_1		u_3
7	H	Hresource	...	u_1		u_3
8	I	Iresource	...	u_1		
9	J	Jresource	...	u_1		
10	L	Lresource	...	u_1		
11	M	Mresource	...	u_1		
12	N	Nresource	...		u_2	
13	O	Oresource	...		u_2	
14	P	Presource	...		u_2	
15	Q	Qresource	...		u_2	
16	R	Rresource	...			u_3
17	S	Sresource	...			u_3
18	T	Tresource	...			u_3
19	U	Uresource	...			u_3

(a)

PRIMARY INDEX

	I	Resource			
12	ι(A)	$\langle l_A$	$E(k_A$, Aresource$)$	$k_A \oplus h(k_1, l_A)$	$k_A \oplus h(k_2, l_A)$	$k_A \oplus h(k_3, l_A)\rangle$
17	ι(B)	$\langle l_B$	$E(k_B$, Bresource$)$	$k_B \oplus h(k_1, l_B)$	$k_B \oplus h(k_2, l_B)$	$fake_{3,B}\rangle$
4	ι(C)	$\langle l_C$	$E(k_C$, Cresource$)$	$k_C \oplus h(k_1, l_C)$	$k_C \oplus h(k_2, l_C)$	$fake_{3,C}\rangle$
3	ι(D)	$\langle l_D$	$E(k_D$, Dresource$)$	$fake_{1,D}$	$k_D \oplus h(k_2, l_D)$	$k_D \oplus h(k_3, l_D)\rangle$
7	ι(F)	$\langle l_F$	$E(k_F$, Fresource$)$	$fake_{1,F}$	$k_F \oplus h(k_2, l_F)$	$k_F \oplus h(k_3, l_F)\rangle$
9	ι(G)	$\langle l_G$	$E(k_G$, Gresource$)$	$k_G \oplus h(k_1, l_G)$	$fake_{2,G}$	$k_G \oplus h(k_3, l_G)\rangle$
10	ι(H)	$\langle l_H$	$E(k_H$, Hresource$)$	$k_H \oplus h(k_1, l_H)$	$fake_{2,H}$	$k_H \oplus h(k_3, l_H)\rangle$
8	ι(I)	$\langle l_I$	$E(k_I$, Iresource$)$	$k_I \oplus h(k_1, l_I)$	$fake_{2,I}$	$fake_{3,I}\rangle$
6	ι(J)	$\langle l_J$	$E(k_J$, Jresource$)$	$k_J \oplus h(k_1, l_J)$	$fake_{2,J}$	$fake_{3,J}\rangle$
11	ι(L)	$\langle l_L$	$E(k_L$, Lresource$)$	$k_L \oplus h(k_1, l_L)$	$fake_{2,L}$	$fake_{3,L}\rangle$
2	ι(M)	$\langle l_M$	$E(k_M$, Mresource$)$	$k_M \oplus h(k_1, l_M)$	$fake_{2,M}$,	$fake_{3,M}\rangle$
14	ι(N)	$\langle l_N$	$E(k_N$, Nresource$)$	$fake_{1,N}$	$k_N \oplus h(k_2, l_N)$	$fake_{3,N}\rangle$
5	ι(O)	$\langle l_O$	$E(k_O$, Oresource$)$	$fake_{1,O}$	$k_O \oplus h(k_2, l_O)$	$fake_{3,O}\rangle$
18	ι(P)	$\langle l_P$	$E(k_P$, Presource$)$	$fake_{1,P}$	$k_P \oplus h(k_2, l_P)$	$fake_{3,P}\rangle$
16	ι(Q)	$\langle l_Q$	$E(k_Q$, Qresource$)$	$fake_{1,Q}$	$k_Q \oplus h(k_2, l_Q)$	$fake_{3,Q}\rangle$
15	ι(R)	$\langle l_R$	$E(k_R$, Rresource$)$	$fake_{1,R}$	$fake_{2,R}$	$k_R \oplus h(k_3, l_R)\rangle$
19	ι(S)	$\langle l_S$	$E(k_S$, Sresource$)$	$fake_{1,S}$	$fake_{2,S}$	$k_S \oplus h(k_3, l_S)\rangle$
1	ι(T)	$\langle l_T$	$E(k_T$, Tresource$)$	$fake_{1,T}$	$fake_{2,T}$	$k_T \oplus h(k_3, l_T)\rangle$
13	ι(U)	$\langle l_U$	$E(k_U$, Uresource$)$	$fake_{1,U}$	$fake_{2,U}$	$k_U \oplus h(k_3, l_U)\rangle$

(b)

Figure 1.5: Relation of Figure 1.2(a) with *acl*s associated with its resources (a) and relation for the primary index (b) when tokens are stored in leaf nodes

The primary index structure is then formally defined as follows.

Definition 1.3.2 (Primary Index with Tokens) *Let $\mathscr{R}$(I,Resource) be a relation, I be the indexing attribute, ι be an encoding function for I, $U = \{u_1, \ldots, u_n\}$ be a set of users, 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_i for each tuple $r_i \in \mathscr{R}$ such that $p_i[I] = \iota(r_i[I])$ and $p_i[Resource] = \langle l_i, E(k_i, r_i[Resource]) t_{1,i}, \ldots, t_{n,i}\rangle$ such that if $u_j \in \text{acl}(r_i)$ $t_{j,i} = k_i \oplus h(k_j, l_i)$; $t_{j,i}$ is a fake token, otherwise, for each $j = 1, \ldots, n$.*

Figure 1.5 illustrates the primary index, with tokens stored together with resources, for the relation and access control policy in Figure 1.2(a). Note that the secondary index is not affected by this approach for token management.

This revised structure of the primary index has an impact on the operations aimed at updating the outsourced resource collection and the authorization policy. While tuple insertion, tuple update, and grant operations are marginally affected by the change in the primary index structure and operate as illustrated above, the removal of tuples and the revoke of authorizations need to be revised as follows.

- *Removal.* To remove resource r, the data owner substitutes the encrypted representation of r in the primary index with a random string of the same length and invalidates all the tokens, substituting each of them with a fake token. When user u searches for the index value $r[I]$ of a removed resource r, she will not be able to use the corresponding token. Using this approach, the data owner does not need to visit the secondary index to invalidate the encoded representation of $r[I]$ for each user u in $acl(r)$. For instance, considering the example in Figure 1.5, to remove resource Cresource, the data owner only needs to search for ι(C) in the primary index, download the corresponding tuple, generate three fake tokens (one for

each user) and a random string of the same length as the encrypted resource, and upload the new tuple.

- *Revoke.* To revoke user u_i access to resource r_j, the data owner randomly generates a new encryption key k_{new} for r_j and re-encrypts the resource with the new key. The data owner generates a token for each user authorized for r_j (i.e., $u \in acl(r_j)$), enabling her to compute the new encryption key k_{new}, and a fake token for the other users. The content of the tuple in the primary index storing r_j is then updated to $p[Resource] = \langle l_{new}, E(k_{new}, r_j[Resource]), t_{1,new}, \ldots, t_{n,new} \rangle$, where $t_{i,new}$ is a token enabling u_i to derive k_{new} from k_i if $u_i \in acl(r_j)$; it is a fake token, otherwise. Note that the data owner does not need to update any of the tuples in the secondary index (only the primary index is modified for the enforcement of revoke operations). For instance, with reference to the example in Figure 1.5, let us assume that the data owner wants to revoke to user u_1 access to `Cresource`. First, the data owner searches for $\iota(\mathtt{C})$ in the primary index, and downloads the corresponding tuple $\langle l_\mathtt{C}, E(k_\mathtt{C}, \mathtt{Cresource}), k_\mathtt{C} \oplus h(k_1, l_\mathtt{C}), k_\mathtt{C} \oplus h(k_2, l_\mathtt{C}), fake_{3,\mathtt{C}} \rangle$. She then decrypts $E(k_\mathtt{C}, \mathtt{Cresource})$ using key $k_\mathtt{C}$, obtaining plaintext resource `Cresource`. She generates a new encryption key $k'_\mathtt{C}$ with label $\ell'_\mathtt{C}$, and re-encrypts `Cresource` with $k'_\mathtt{C}$. The data owner computes a token $t'_{2,\mathtt{C}}$ enabling user u_2 to derive $k'_\mathtt{C}$ from her own key k_2, and generates two fake tokens for u_1 and u_3. The data owner finally updates the tuple in the primary index as $\langle \ell'_\mathtt{C}, E(k'_\mathtt{C}, \mathtt{Cresource}), fake_{1,\mathtt{C}}, k'_\mathtt{C} \oplus h(k_2, \ell'_\mathtt{C}), fake_{3,\mathtt{C}} \rangle$. This approach guarantees that a user cannot distinguish between the removal of a resource r she is authorized to access and the revocation of her privilege over it. In fact, after the update of the tuple representing r in the primary index, she will be associated with a fake token that cannot be used for decryption.

Our solution for token management has two advantages. First, it permits the data owner to manage revoke operations in a simpler and less expensive way. In fact, the data owner does not need to modify the secondary index, but only the primary index is affected. Second, this solution facilitates key and token management: each resource can be encrypted with a fresh key (which is not shared with other resources), and tokens are stored together with resources. Hence, searches over the primary/secondary index do not require to also access a token catalog, which should be properly protected. Also, key derivation requires only one derivation step.

1.3.2 Analysis

In this section, we demonstrate the correct enforcement of the (dynamic) authorization policy defined by the data owner, we discuss the protection of access and pattern confidentiality provided by our approach, and we analyze the performance and economic overhead it causes.

Correctness

The primary and secondary indexes described in D3.3 guarantee the correct enforcement of the access control policy if each user u_i can access all and only the resources and index values in $\mathscr{R}$ she is authorized to access, as formally stated by the following theorem.

Theorem 1.3.1 *Let $\mathscr{R}(I,Resource)$ be a relation, U be a set of users,* $\mathrm{acl}(r) \subseteq U$ *be the acl of r, $\forall r \in \mathscr{R}$. The encryption policy keys, the primary index $\mathscr{P}(I,Resource)$ for $\mathscr{R}$ over I, and the secondary index $\mathscr{S}(I,Resource)$ for $\mathscr{R}$ and $\mathscr{P}$* correctly enforce $\mathrm{acl}(r)$, *$\forall r \in \mathscr{R}$, iff $\forall u_i \in U$, the following conditions hold:* i) *u_i can access resource r*[Resource] *iff* $u_i \in \mathrm{acl}(r)$*;* ii) *u_i can see an*

index value v iff $\exists r \in \mathscr{R}$ s.t. $r[I]=v$ and $u_i \in \text{acl}(r)$.
The insertion, removal, and update of the tuples and the grant and revoke of authorization preserve the correctness of policy enforcement.

PROOF. 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:

1. u_i can compute $\iota_i(r[I])$;
2. $\exists! s \in \mathscr{S}$ s.t. $s[I]=\iota_i(r[I])$ and $s[Resource]=E(k_i,\iota(r[I]))$;
3. $\exists! p \in \mathscr{P}$ s.t. $p[I]=\iota(r[I])$ and $p[Resource]=\langle l_{i_1,\ldots,i_n}, E(k_{i_1,\ldots,i_n}, r[Resource])\rangle$;
4. u_i can visit $\mathscr{S}$ and $\mathscr{P}$.

User u_i can compute $\iota_i(r[I])$ since it is defined as $h(k_i, r[I])$ and u_i knows key k_i, by Definition 1.4.2 in D3.3. Tuple s exists and belongs to $\mathscr{S}$ by Definition 1.4.4 in D3.3. Tuple p exists and belongs to $\mathscr{P}$ by Definition 1.4.3 in D3.3. 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 in D3.3. 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 the content of nodes to enable shuffling, and the co-domain of the encoding functions.
Note that the observations above hold also when a new resource r is inserted into the data collection. Indeed, as illustrated in Section 1.3.1, the data owner inserts in the primary and in the secondary index the same tuples that she would have inserted at initialization time for r.
The ability of user u_i to retrieve and decrypt r is also not affected by grant and revoke operations. When user u_i is granted access to r, the data owner inserts a new tuple s in the secondary index having $s[I]=\iota_i(r[I])$ and $s[Resource]=E(k_i,\iota(r[I]))$, thus enabling the user to retrieve $\iota(r[I])$. Since the data owner also re-encrypts the content of tuple p in the primary index having $p[I]=\iota(r[I])$ with the key of the new $acl(r)$, which also includes u_i, user u_i can decrypt $p[Resource]$ and access r.
Let us now consider the case where another user u_j, $j \neq i$, is granted access to a resource r that u_i can access. This grant operation does not affect the ability of u_i to retrieve and decrypt r. Indeed, no tuple is removed from the primary and secondary index, and u_i can derive the key used to re-encrypt r since $u_i \in acl(r)$. Similarly, if u_j, $j \neq i$, is revoked access to a resource r that u_i can access, the ability of u_i to access r is not affected. In fact, tuple s with $s[I]=\iota_i(r[I])$ is not modified or removed from the secondary index. Also, the key used to re-encrypt r can still be derived by u_i, since $u_i \in acl(r)$ also after the revoke operation.
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 immediately clear that u_i cannot access the plaintext content of $r[Resources]$ since it is encrypted with a key k_X (Definition 1.4.3 in D3.3) that u_i does not know. In fact, by Definition 1.4.3 in D3.3, 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 in D3.3, 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 in

D3.3, 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).
Let us now analyze the case where user u_i is revoked access to r. After the revoke operation, u_i can access neither the plaintext content of r, nor index value $r[I]$ anymore. Since the resource is re-encrypted with a key k_X that u_i does not know, she cannot access the plaintext content of the resource. In fact, after the revoke operation, u_i does not belong to $acl(r)$ anymore. Furthermore, she cannot identify the new encoded value $\iota(r[I], salt)$ of the resource, since tuple s with $s[I]{=}\iota_i(r[I])$ has been removed from the secondary index and *salt* is a random nonce. ■

Also the alternative approach for token management previously discussed guarantees the correct enforcement of the access control policy, as stated by the following theorem.

Theorem 1.3.2 *Let $\mathscr{R}$(I,Resource) be a relation, U be a set of users,* acl$(r)\subseteq U$ *be the acl of r, $\forall r\in\mathscr{R}$. The encryption policy keys, the primary index $\mathscr{P}$(I,Resource) for $\mathscr{R}$ over I, and the secondary index $\mathscr{S}$(I,Resource) for $\mathscr{R}$ and $\mathscr{P}$* correctly enforce acl(r), *$\forall r\in\mathscr{R}$, iff $\forall u_i \in U$, the following conditions hold:* 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).
The insertion, removal, and update of the tuples and the grant and revoke of authorizations preserve the correctness of policy enforcement.

PROOF. Let us first consider the initial configuration outsourced by the data owner. The main difference with respect to the base scenario discussed above centers around key management. Indeed, each resource r_j is encrypted with a different key k_j. In the primary index, the tuple p_j storing r_j also includes a token $t_{i,j}$ for each user u_i. Such a token either enables u_i to derive k_j, if $u_i\in acl(r_j)$, or it is a fake token, which does not enable any key derivation. Hence, user u_i s.t. $u_i\in acl(r_j)$ can retrieve and decrypt r_j, while user $u_i\notin acl(r_j)$ can access neither the plaintext content $r_j[Resources]$, nor the index value $r_j[I]$ of the resource.
It is necessary to prove that the insertion and removal of tuples and authorizations does not affect the correctness of policy enforcement. The insertion of a new resource r maintains the correct enforcement of the access control policy because it implies the insertion in the primary and secondary index, of the same tuples that would have been inserted for r at initialization time. Similarly, the removal of a resource r does not affect the enforcement of the access control policy. In fact, it implies the substitution of the encrypted representation of r in the primary index with a random string of the same length, and of each token with a fake one.
When user u_i is granted access to r_j, the data owner inserts a new tuple s in the secondary index having $s[I]{=}\iota_i(r[I])$ and $s[Resource]{=}E(k_i,\iota(r_j[I]))$, thus enabling the user to retrieve $\iota(r_j[I])$. Since the data owner also modifies the token $t_{i,j}$ in the primary index to enable user u_i to compute the encryption key k_j used to protect $p_j[Resource]$, grant operations do not affect the correct enforcement of the access control policy.
When user u_i is revoked access to r_j, the data owner modifies the tuple p_j in the primary index storing it, substituting $t_{i,j}$ with a fake token. The data owner also re-encrypts r_j with a new key k_{new} and updates the tokens $t_{k,j}$ of non-revoked users u_k, enabling them to derive k_{new}. Therefore, any user $u\in acl(r_j)$ can still access r_j in plaintext. On the contrary, the token $t_{z,j}$ for a non-authorized user u_z (including u_i) is fake. Then, non-authorized users cannot decrypt $p_j[Resource]$. ■

Access confidentiality

We now discuss the confidentiality guarantees provided by the proposed approach. To analyze access and pattern confidentiality, we consider two possible observers: the storing server and a user of the system. Indeed, the storing server is the party with the highest potential for observations among the parties that are not authorized to access the content of resources, since all accesses are executed by it. Authorized users have instead plaintext visibility over a subset of the resources.

Server. 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]. In our analysis, we assume that the server knows: the number of blocks (nodes) in the primary and secondary index; the height of the two tree structures; the identifier of each block and its level in the tree; and the identifier of read and written blocks for each access operation. (This knowledge can be acquired by observing a sufficiently long sequence of accesses to the indexes.) Despite the ability of the server to observe all the accesses by the users and its knowledge of the shuffle index, it cannot identify the resource target of an access (access confidentiality) and it cannot infer whether two searcher aim at the same or at a different resource (patter confidentiality).

Similar to 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 or an insert 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.

User. We now consider a user as our observer, who has knowledge of: the plaintext content of a subset of the resources, the encryption key used by the data owner for shuffling (and hence has potential visibility of the encrypted content of all the blocks at the server), and the information necessary for executing a search over the primary and secondary indexes (i.e., her encoding function, the identifier of the blocks visited by the most recent access). However, an authorized user cannot identify the target of searches performed by other users and she cannot infer the changes in the access control policy over resources that she cannot access.

To infer the target of a search operation executed by a different user, the attacker could exploit her knowledge on the identifiers of the blocks visited by a previous access. However, for repeated accesses, we keep track of the identifiers of the blocks visited along the path to the target, to covers, and of repeated accesses. Furthermore, each leaf node stores multiple encoded values, which correspond to index values that are not close to each other since the encoding function is not order-preserving. Hence, the attacker cannot gain any information about the target of the last access. To reconstruct the content of the outsourced relation, and hence identify the target of accesses performed by other users, the attacker could also exploit her knowledge of the encryption key used by the data owner to wrap blocks. In fact, by downloading all the accessed blocks for each access, she could nullify their shuffling. However, this would require the user to download the whole (primary and secondary) index after each access, which seems impracticable.

Since the removal of a resource (or of a privilege) does not cause the deletion of the corresponding tuple in the primary and secondary indexes, but these tuples are re-encrypted with a new key, only users authorized for the resource can detect the change. However, a user cannot distinguish between the removal of a resource that she is authorized to access and the revocation of her privilege over the same resource. Similarly, thanks to the adoption of probabilistic splits, the insertion of a new resource or the grant of a privilege for a resource can be detected only by users authorized for the resource. Also in this case a user cannot distinguish between the insertion of a new resource for which she is authorized and the grant of her privilege for the same resource. We note however that a user u_i who can access a resource r that is subject to a grant or a revoke operation for a different user u_j can detect such a policy change, even if she cannot identify u_j. Indeed, r is re-encrypted to enforce the grant or the revoke operation. We note that a user can identify these changes in the set of outsourced resources and in the access control policy only if she keeps a copy of the resources that she is authorized to access and compares her copy with the one stored at the server.

Performance and economic evaluation

To analyze the overhead caused by the adoption of our approach, we evaluated the performance and economic costs of the adoption of our protection techniques.

Performance evaluation. The performance of the system is measured as the average response time experienced by an authorized client when submitting an access request. To assess the performance of our access 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. We did not vary the fan-out of our indexes since 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. Also, 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. We set the size of the internal and leaf blocks (nodes) to 8 KiB and 16 KiB, respectively, for both the primary and the secondary index and we fixed *num_cover* to 1 (i.e., two additional searches are executed for each access request, one is the cover and one is a repeated search [DFP+15]). The hardware used in the experiments included a client machine with an Intel Core i5–2520M CPU at 2.5 GHz, L3–3 MiB, 8 GiB RAM DDR3 1066, running an Arch Linux OS. The server machine runs an Intel Core i7–920 CPU at 2.6 GHz, L3–8 MiB, 12 GiB, RAM DDR3 1066, 120 GB 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 100 ms and standard deviation of 2.5 ms. Our experiments show that the latency of the network is the factor with the greatest impact in a large-bandwidth LAN/WAN scenario. This result confirms the performance analysis in [DFP+15], where we also showed the cost of CPU and disk. The performance figures obtained for accessing the secondary and the primary indexes looking for a value exhibit an average value equal to 750 ms, which compares favorably with the response time of 630 ms experienced by the client when accessing two plain encrypted indexes (i.e., without shuffling). These results are coherent with the fact that, at coarse-level, our approach is based on two consecutive accesses to two shuffle indexes (i.e., the overall response time of our solution is comparable to the response time experienced by two accesses to two shuffle indexes [DFP+15] with the same height and size of the blocks). We can then conclude that the support for access control does not add significant overhead and does not affect the performance of the shuffle index.

To analyze the performance of our primary and secondary index structures also in case the data collection and/or users privileges change, we implemented the approach proposed in [DFP+15] to make the insertion and removal of tuples in a shuffle index indistinguishable from read accesses. The performance overhead caused by the support of insertion and removal operations is ≈2.7 ms·(*num_cover*+1)=5.4 ms on average for each (read, insert, delete) access operation.

Economic evaluation. The price lists of most CSPs comprise three cost components[1]:

1. $Cost_{\texttt{storage}}$, monthly cost of the stored data (2.3 USD/TB per month, for less than 50 TB, 2.2 USD/TB per month, for between 50 TB and 500 TB, and 2.1 USD/TB per month, for more than 500 TB);

2. $Cost_{\texttt{access}}$, cost of the access requests ($Cost_{\texttt{PUT}}$=5 USD per million `PUT` requests, and $Cost_{\texttt{GET}}$=0.4 USD per million `GET` requests);

3. $Cost_{\texttt{out}}$, cost of the data transferred out of the server ($Cost_{\texttt{perTib}}$=90 USD/TB, for transferring up to 10 TB; $Cost_{\texttt{perTib}}$=85 USD/TB for transferring between 11 TB and 50 TB; $Cost_{\texttt{perTib}}$=70 USD/TB, for transferring between 50 TB and 150 TB; $Cost_{\texttt{perTib}}$=50 USD/TB, for transferring between 150 TB and 500 TB).

Sending data to the server or deleting data is free of charge.

The total monthly cost $Cost_{\texttt{total}}$ for outsourcing the management of a set of resources using our shuffle index is computed as $Cost_{\texttt{total}} = Cost_{\texttt{storage}} + Cost_{\texttt{access}} + Cost_{\texttt{out}}$. In particular, the storage cost ($Cost_{\texttt{storage}}$) and the access cost ($Cost_{\texttt{out}}$) will split up in several tiers depending on the amount of storage used during a month and the total number of accesses to the secondary and the primary indexes (as this entails the amount of bandwidth used for every network data transfers out of the CSP). We now present an example to better detail how these three cost components are computed.

Consider a dataset organized in a secondary and a primary index, both with height *height*=3, fan-out F=512, and *num_cover*=1. Assume that the set $\mathscr{U}$ of users includes 50 subjects and that access control lists include (on average) $0.3 \cdot |\mathscr{U}|$ users each. We assume internal nodes to be stored in blocks of 8 KiB, leaf nodes of the secondary index to be stored in blocks of 16 KiB, and leaf nodes of the primary index to be stored in blocks of size varying in the set $\{16\,\text{KiB}, 32\,\text{KiB}, 64\,\text{KiB}, 128\,\text{KiB}, 256\,\text{KiB}, 512\,\text{KiB}, 1\,\text{MiB}\}$ (to better accommodate resources). The size of the primary index will therefore depend on the size of its leaf blocks and will vary in the set $\{2\text{TiB}, 4\text{TiB}, 8\,\text{TiB}, 16\,\text{TiB}, 32\,\text{TiB}, 64\,\text{TiB}, 128\,\text{TiB}\}$. The size of the secondary index will depend on the number and cardinality of the access control lists and, in our running example, will amount to approximately 30 TiB.

The monthly cost $Cost_{\texttt{storage}}$ for the usage of Cloud storage crosses more than one tier. Figure 1.6 shows that such a cost varies from 73.6USD to 352.6USD, depending on the size of the primary index leaf blocks and the fixed size of the secondary index.

The cost $Cost_{\texttt{access}}$ of the accesses takes into account the number and type of operations requested to the CSP and is computed as the product of the number η of accesses by the cost $C_{\texttt{a}}$ of a single access , that is, $Cost_{\texttt{access}} = \eta \cdot C_{\texttt{a}}$, where $C_{\texttt{a}} = 2 \cdot height \cdot (num\_cover + 2) \cdot (Cost_{\texttt{GET}} + Cost_{\texttt{PUT}})$ USD per millions of accesses. Indeed, for each search operation on the secondary and primary index, we access $2 \cdot height \cdot (num\_cover + 2)$ blocks because we visit the target path, *num_cover* cover paths, and one repeated access. Each path includes *height* nodes, both in the

[1] Referenced Amazon S3 prices, February 2017

	Primary index leaf block size (KiB)						
Cost (USD)	**16**	**32**	**64**	**128**	**256**	**512**	**1024**
$Cost_{\texttt{storage}}$	73.60	78.20	87.40	105.80	141.40	211.80	352.60
$Cost_{\texttt{access}}$	97.20	97.20	97.20	97.20	97.20	97.20	97.20
$Cost_{\texttt{out}}$	16.09	20.11	28.16	44.25	76.44	140.80	269.50
$Cost_{\texttt{total}}$	*186.89*	*195.51*	*212.76*	*247.25*	*315.04*	*449.81*	*719.36*

Figure 1.6: Costs (USD) of data storage, access, and transfer per month, depending on the size of the primary index leaf blocks and of the fixed size of the secondary index (30TiB)

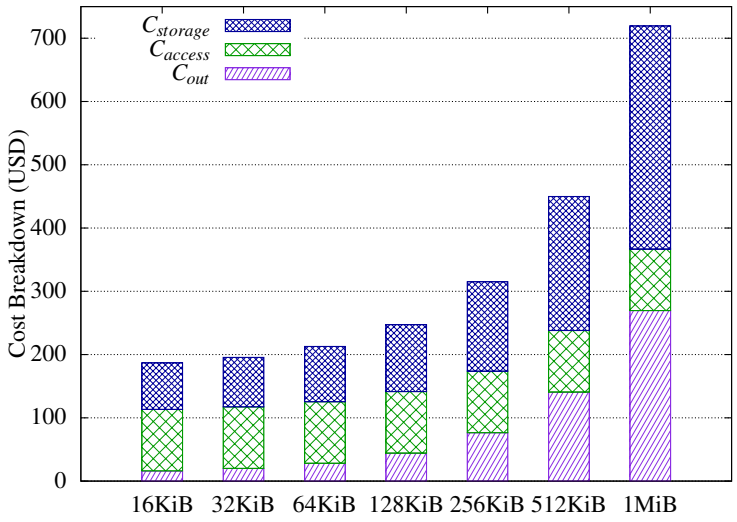


Figure 1.7: Total monthly cost (USD) varying the size of the primary index leaf blocks

primary and in the secondary index. Each block is first downloaded and then uploaded after shuffling, hence we pay a `GET` and a `PUT` request for each accessed block. In our example, $C_{\texttt{access}}$=97.2 USD per millions of accesses.

The cost $Cost_{\texttt{out}}$ of bandwidth usage is computed as the product of the number η of accesses by the cost $C_{\texttt{b}}$ of transferring out of the CSP the volume of data implied by each access, that is, $Cost_{\texttt{out}}=\eta \cdot C_{\texttt{b}}$, where $C_{\texttt{b}}=(num\_cover+2)\cdot(2\cdot height-2)\cdot$ `InternalBlockSize` + `PrimLeafSize`+`SecLeafSize`$)\cdot Cost_{\texttt{perTiB}})\cdot 10^6$ USD per million of accesses. In fact, $C_{\texttt{b}}$ depends on the number and size of downloaded/uploaded blocks and can cross one or more tiers ($Cost_{\texttt{perTiB}}$), depending on the actual number of access requests. In our example, assuming the highest cost value (i.e., $Cost_{\texttt{perTiB}}$=90 USD), the size of the primary index leaf block will mostly influence the amount of $Cost_{\texttt{out}}$. Figure 1.6 shows that $Cost_{\texttt{out}}$ varies from 16.09 USD to 269.56 USD, per millions of accesses.

Figure 1.6 illustrates the total monthly cost $Cost_{\texttt{total}}$ of our example, which varies from 186.89 USD to 719.36 USD. The total monthly cost as well as its components are graphically illustrated in Figure 1.7. It is interesting to note that $Cost_{\texttt{access}}$ (which is constant as it does not depend on the size of blocks, but only on the height of the shuffle index) dominates $Cost_{\texttt{out}}$ when the size of the leaf blocks of the primary index is less than 512 KiB, while the storage cost $Cost_{\texttt{storage}}$ remains the main contributor to the aggregate cost when the leaf blocks of the primary index has size up to 1 MiB.

The additional cost of a solution featuring the access control mechanism described in this paper compared with the adoption of one shuffle index with no access control restrictions, quickly decreases as the ratio between the size of leaf blocks on the primary index and the size of any other block increases. Considering the configuration illustrated above, the overhead of our solution ranges from 70%, when internal blocks have size 8 KiB and leaf blocks on the primary and secondary index have size 16 KB, down to 11%, when leaf blocks on the primary index have size 1 MiB.

As a concluding remark, we note that our approach for updating the access control policy and for inserting/removing data into/from the primary and secondary index never releases any physical storage on the CSP. This implies that the overall cost of our approach may include a cost of $\sim$2 USD (from 2.1 to 2.3 USD) per month for storing each TiB of data that is no more needed by the data owner. This additional cost highly depends on the kind of data stored at the storing server since it influences the size of the leaf blocks of the primary index. We note however that the impact of this additional cost of storage on the total monthly cost quickly decreases as the size of the primary index or the number of accesses increases. Indeed, it represents from 1.07%, in case of leaf blocks of 16 KiB, to 0.28%, in case of leaf blocks of 1024 KiB, of the total cost.

1.4 Search over encrypted data

In D3.3 we presented 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 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, a new privacy-preserving method for join, set difference, and count distinct in multi-user mode, and a post-processing step on the client saving computational effort on the client while preserving the confidentiality of the data. Our results are subsumed in a multi-user algorithm.

Previous work only focused on the access control mechanism [DFJ$^+$07] or the key management [AFB05, DDF$^+$05]. Thus, our improvements over the state-of-the-art are: First, our formulated system builds on previous work in encrypted query processing for a single user as described in [HIML02, DDJ$^+$03, AES03, CDF$^+$09, PRZB11b], 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 [PRZB11b].
Second, we overcome the limitations of current approaches for multiple users offer either limited functionality [YBDD09, PRZB11b, PZ13, ARCI13] or expose confidential information to the database server [Ora10].

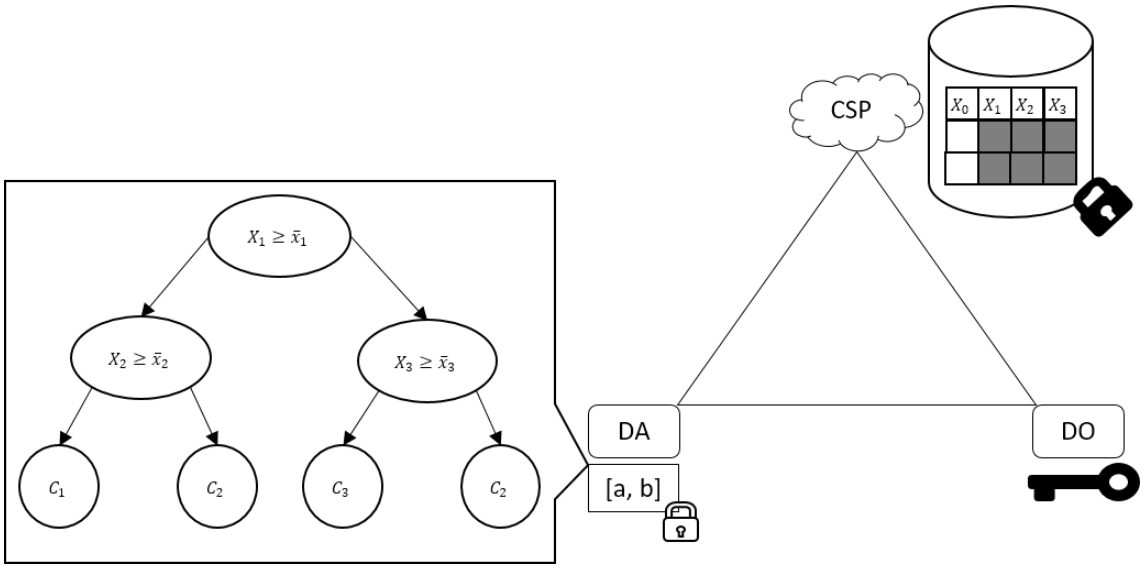


Figure 1.8: Scenario for Oblivious Order-Preserving Encryption

1.5 Oblivious Order Preserving Encryption

OOPE is a cryptographic protocol that uses secure multiparty computation technique to compute order-preserving encryption. The work on OOPE was initially featured in W3.2 and since then made several advancements that we will illustrate in this section. First, we will illustrate the problem motivation, and provide overviews of order-preserving encryption and the relevant cryptographic background. We will close this section with a discussion on implementation and evaluation details.

1.5.1 Problem Statement

Order-preserving encryption (OPE) allows for the encryption of data, while still enabling efficient range queries on the encrypted data. Moreover it does not require any change to the database management system, because comparison operates on ciphertexts. This makes OPE schemes very suitable for data outsourcing in Cloud computing scenarios, since it can be retrofitted to existing applications.

However, all OPE schemes are symmetric limiting the use case to one client and one server. This is due to the fact that a public-key encryption would allow a binary search on the ciphertext. Imagine a scenario where a Data Owner (DO) encrypts its data with an OPE, stores the encrypted data in a Cloud database held by a CSP, and retains the encryption key. Later, a Data Analyst (DA) wants to perform some analysis on the encrypted data. To this end, it holds a private machine learning model involving comparisons. For instance, we assume the model to be a decision tree as pictured in Figure 1.8, where the $\bar{x}_i$ are the thresholds and (X_1, X_2, X_3) is the input vector (that maps to corresponding columns in the DO's database) to be classified. In order to use the model for classification the DA transforms the decision tree into range queries, e.g., for class c_1 we have the query $(X_1 < \bar{x}_1) \wedge (X_2 < \bar{x}_2)$. More precisely the DA wants to execute queries such as equations 1.1 and 1.2, where we assume X_0 to be public.

$$\text{SELECT COUNT}(*) \text{ WHERE } X_1 < \bar{x}_1 \text{ AND } X_2 < \bar{x}_2 \tag{1.1}$$

$$\text{SELECT } X_0 \text{ WHERE } X_1 < \bar{x}_1 \text{ AND } X_2 < \bar{x}_2 \tag{1.2}$$

However, as the database is encrypted (i.e. columns X_1, X_2 and X_3 are OPE encrypted) the DA needs ciphertexts of the thresholds $\bar{x}_i$. For simplicity we assume that there is no update during the analysis, i.e., the DO does not insert new values in the database.

OPE is symmetric, therefore only the DO can encrypt and decrypt the data stored on the Cloud server. If the DA needs to obtain a ciphertext for a query, it can just send the plaintext threshold to the DO. However, if the model contains intellectual property which the DA wants to remain protected, then this free sharing of information is no longer possible.

This distinction between DO and DA occurs in many cases of collaborative data analysis, data mining and machine learning. In such scenarios, multiple parties need to jointly conduct data analysis tasks based on their private inputs. As concrete examples from the literature consider, e.g., supply chain management, collaborative forecasting, benchmarking, criminal investigation, smart metering, etc.) [DA01, AEDS03, ABL$^+$04, Ker12]. Although in these scenarios plaintext information sharing would be a viable alternative, participants are reluctant to share their information with others. This reluctance is quite rational and commonly observed in practice. It is due to the fact that the implications of the information are unknown or hard to assess. For example, sharing the information could weaken their negotiation position, impact customers' market information by revealing corporate performance and strategies or impact reputation [AEDS03, ABL$^+$04, CK08].

The goal is therefore to allow the data analysis to be performed efficiently without revealing any sensitive information in the query and without revealing the key to the OPE. We overcome the limitation of private range querying on order-preserving encrypted data by allowing the equivalent of a public-key encryption. The idea is to replace public-key encryption with a secure, interactive protocol. Non-interactive binary search on the ciphertext is no longer feasible, since every encryption requires the participation of the DO who can rate limit the DA.

Since neither the DA wants to reveal his query value nor the DO his encryption state (key), this is clearly an instance of a secure computation where two or more parties compute on their secret inputs without revealing anything but the result. In an ideal world the DA and DO would perform a two-party secure computation for the encryption of the query value and then the DA would send the encrypted value as part of an SQL query to the CSP. However, this two-party secure computation is linear in the encryption state (key) and hence the size of the database. The key insight is that we can construct an encryption with logarithmic complexity in the size of the database by involving the CSP - which is already hosting the encrypted data - in a three-party secure computation without sacrificing any security, since the CSP will learn the encrypted query value in any case. One may conjecture that in this construction the encryption key of the DO may be outsourced to secure hardware in the CSP simplifying the protocol to two parties, but that would prevent the DO from rate limiting the encryption and the binary search attack would be a threat again, even if the protocol were otherwise secure.

OPE can be classified into stateless schemes (see Section 1.5.2) and stateful schemes (see Section 1.5.3). Our work is concerned with stateful schemes and hence we introduce some of their algorithms below. However, we review stateless schemes and their security definitions first in order to distinguish them from stateful schemes.

1.5.2 Stateless Order-preserving Encryption

OPE ensures that the order relation of the ciphertexts is the same as the order of the corresponding plaintexts. This enables efficient searching on the ciphertexts using binary search or performing range queries without decrypting the ciphertexts. The concept of OPE was introduced in the database

community by Agrawal et al. [AKSX04]. The cryptographic study of Agrawal et al.'s scheme was first initiated by [BCLO09], which proposed an ideal security definition IND-OCPA[2] for OPE. The authors proved that under certain implicit assumptions IND-OCPA is infeasible to achieve. Their proposed scheme was first implemented in the CryptDB tool of Popa et al. [PRZB11b] and attacked by Naveed et al. [NKW15]. In [BCO11] Boldyreva et al. further improved the security and introduced modular OPE (MOPE). MOPE adds a secret modular offset to each plaintext value before it is encrypted. It improves the security of OPE, as it does not leak any information about plaintext location, but still does not provide ideal IND-OCPA security. Moreover Mavroforakis et al. showed that executing range queries via MOPE in a naive way allows the adversary to learn the secret offset and so negating any potential security gains. They address this vulnerability by introducing query execution algorithms for MOPE [MCO+15]. However, this algorithm assumes a uniform distribution of data and has already been attacked in [DDC16]. In a different strand of work Teranishi et al. improve the security of stateless OPE by randomly introducing larger gaps in the ciphertexts [TYM14]. However, they also fail to provide ideal security.

1.5.3 Stateful Order-preserving Encryption

Popa et al. were the first to observe that one can avoid the infeasibility of IND-OCPA [BCLO09] by giving up certain restrictions of OPE. As result of their observations they introduced mutable OPE [PLZ13]. Their first observation was that most OPE applications only require a less restrictive interface than that of encryption schemes. Their encryption scheme is therefore implemented as an interactive protocol running between a client that also owns the data to be encrypted and an honest-but-curious server that stores the data. Moreover, it is acceptable that a small number of ciphertexts of already-encrypted values change over time as new plaintexts are encrypted. With this relaxed definition their scheme was the first OPE scheme to achieve ideal security.

Popa et al.'s scheme (mOPE$_1$) [PLZ13]. The basic idea of Popa et al.'s scheme is to have the encoded values organized at the server in a binary search tree (*OPE-tree*). Specifically the server stores the state of the encryption scheme in a table (*OPE-table*). The state contains ciphertexts consisting of a deterministic AES ciphertext and the order (*OPE Encoding*) of the corresponding plaintext. To encrypt a new value x the server reconstructs the OPE-tree from the OPE-table and traverses it. In each step of the traversal the client receives the current node v of the search tree, decrypts and compares it with x. If x is smaller (resp. larger) then the client recursively proceeds with the left (resp. right) child node of v. An edge to the left (resp. to the right) is encoded as 0 (resp. 1). The OPE encoding of x is then the path from the root of the tree to x padded with $10\cdots0$ to the same length l. To ensure that the length of OPE encodings do not exceed the defined length l, the server must occasionally perform balancing operations. This updates some order in the OPE table (i.e. the OPE encodings of some already encrypted values mutate to another encoding).

Kerschbaum and Schröpfer's scheme (mOPE$_2$) [KS14]. The insertion cost of Popa et al's scheme is high, because the tree traversal must be interactive between the client and the server. To tackle this problem Kerschbaum and Schröpfer proposed in [KS14] another ideal secure, but significantly more efficient, OPE scheme. Both schemes use binary search and are mutable, but the main difference is that in the scheme of [KS14] the state is not stored on the server but on the client. Moreover the client chooses a range $\{0\cdots M\}$ for the order. For each plaintext x and the corresponding OPE encoding $y \in \{0\cdots M\}$ the client maintains a pair $\langle x,y\rangle$ in the state. To insert a

[2] IND-OCPA means indistinguishability under ordered chosen plaintext attacks and requires that OPE schemes must reveal no additional information about the plaintext values besides their order

new plaintext the client finds two pairs $\langle x_i, y_i \rangle$, $\langle x_{i+1}, y_{i+1} \rangle$ in the state such that $x_i \leq x < x_{i+1}$ and computes the OPE encoding as follows:

- if $x_i = x$ then the OPE encoding of x is $y = y_i$
- else
  - if $y_{i+1} - y_i = 1$ then
    - update the state (Algorithm 2 in [KS14])[3].
  - the OPE encoding of x is $y = y_i + \lceil \frac{y_{i+1} - y_i}{2} \rceil$.

The encryption algorithm is keyless and the only secret information is the state which grows with the number of encryptions of distinct plaintexts. The client uses a dictionary to keep the state small and hence does not need to store a copy of the data.

Kerschbaum's scheme (mOPE$_3$) [Ker15]. Deterministic OPE schemes [AKSX04, BCLO09, PLZ13, KS14, TYM14] are vulnerable to many attacks like: frequency analysis, sorting attack, cumulative attack [NKW15, GSB+16]. To increase the security of OPE Kerschbaum first introduced in [Ker15] a new security definition called *indistinguishability under frequency-analyzing ordered chosen plaintext attack* (IND-FAOCPA) that is stronger than IND-OCPA. Second he proposed a novel OPE scheme mOPE$_3$ that is secure under this new security definition. The basic idea of this scheme is to randomize ciphertexts such that no frequency information from repeated ciphertexts leaks. It borrows the ideas of [KS14] with a modification that re-encrypts the same plaintext with a different ciphertext. First the state of the client and server remain the same as in mOPE$_2$. The order also ranges from 0 to M as in mOPE$_2$. The algorithm traverses the OPE-tree by going to the left or to the right depending on the comparison between the new plaintext and nodes of the tree. However, if the value being encrypted is equal to some value in the tree then the algorithm traverses the tree depending on the outcome of a random coin (i.e. the algorithm randomly chooses between left and right). Finally, if there is no more nodes to traverse, the algorithm rebalances the tree if necessary and then computes the ciphertext similarly to $y = y_i + \lceil \frac{y_{i+1} - y_i}{2} \rceil$.

In subsequent independent analysis [GSB+16] this encryption scheme has been shown to be significantly more secure against attacks on OPE (albeit not perfectly secure).

1.5.4 Cryptographic Background

OOPE is a mix-technique secure multiparty computation protocol. We therefore review some relevant building blocks in this section.

Secure Multiparty Computation

Secure multiparty computation (SMC) is a cryptographic technique that allows several parties to compute a function on their private inputs without revealing any information other than the function's output. A classical example in the literature is the so called *Yao's millionaire's problem* introduced in [Yao82]. Two millionaires are interested in knowing which of them is richer without revealing their actual wealth. Formally we have a set of n parties $P_1, \cdots, P_n$, each with its own private input $x_1, \cdots, x_n$ and they want to compute the function $y = f(x_1, \cdots, x_n)$[4] without disclosing their private inputs.

[3] This potentially updates all OPE encoding y produced so far [KS14].

[4] The output of each party can also be private, in this case $y = (y_1, \cdots, y_n)$.

Security of SMC protocols is often defined by comparison to an *ideal model*. In that model, parties privately send their input to a trusted third party (TTP). Then the TTP computes the outcome of the function on their behalf, sends the corresponding result to each party and forgets about the private inputs. In the *real model*, parties emulate the ideal model by executing a cryptographic protocol to perform the computation. At the end, only the result should be revealed and nothing else. A SMC protocol is then said to be secure if the adversary can learn only the result of the computation and data that can be deduced from this result and known inputs [Gol04, CDN15, Fri10].

An important issue to consider when defining the security of SMC is the adversary's power. There exists many security models, but the *semi-honest* and the *malicious* adversary model are the most popular [Gol04, CDN15]. In the semi-honest (a.k.a *honest-but-curious*) model parties behave *passively* and follow the protocol specification. However, the adversary can obtain the internal state of corrupted parties and uses this to learn more information. In contrast, a malicious adversary is *active* and instructs corrupted parties to deviate from the protocol specification.

Yao's Garbled Circuit

Yao's initial protocol for secure two-party computation uses a technique called garbled circuits (GC). In GC protocols, a party called Generator garbles the Boolean circuit of the function to be computed and sends it with the keys corresponding to its input to a second party called Evaluator. Then both parties engage in an Oblivious Transfer (OT) protocol, that allows the Evaluator to learn the keys corresponding to its input without revealing any information on the actual input to the Generator. Finally the Evaluator evaluates the garbled circuit and outputs the result. For a detailed, technical description of circuit garbling and its implementation see [Yao82, LP09a, PSSW09, LP09b, EFLL12].

Homomorphic Encryption

A homomorphic encryption scheme is an encryption scheme that allows computations on ciphertexts by generating an encrypted result whose decryption matches the result of operations on the corresponding plaintexts. With fully homomorphic encryption schemes [Gen09] one can compute any efficiently computable function on ciphertexts. However, with the current state of the art, the computational overhead is still too high for practical applications. More Efficient alternatives are additive homomorphic encryption schemes, e.g.: Paillier [Pai99, DJ01]. They allow specific arithmetic operations on plaintexts, by applying an efficient operation on the ciphertexts. Let $E(x)$ denote the probabilistic encryption of a plaintext x. Then the following addition property holds $E(x)E(y) = E(x+y)$, i.e., by multiplying two ciphertexts one obtains a ciphertext of the sum. In our protocol we will use the public-key encryption scheme of Paillier [Pai99].

1.5.5 Overview of the protocol

In theory generic SMC allows to compute any efficiently computable function. However, any generic SMC is at least linear in the input size, which in this case is the number of encrypted values in the database. The idea of our solution is to exploit the inherent, i.e. implied by input and output, leakage of Popa et. al.'s OPE scheme making our oblivious OPE sublinear in the database size. Furthermore, we exploit the advantage of (homomorphic) encryption allowing a unique, persistent OPE state stored at the CSP while being able to generate secure inputs for the SMC protocol and the advantage of garbled circuits allowing efficient, yet provably secure comparison. Our oblivious

OPE is therefore a mixed-technique, secure multi-party computation protocol between the DO, the DA and the CSP in the semi-honest model.

In detail, our protocol proceeds as follows: The DO outsources its OPE state to the CSP. As already described, the state consists of an OPE-table of ciphertext, order pairs. However, in oblivious OPE the ciphertext is created using an additively homomorphic public-key encryption scheme instead of standard symmetric encryption. When the DA traverses the state in order to encrypt a query plaintext, the CSP creates secret shares[5] using the homomorphic property. One secret share is sent to the DA and one to the DO. The DA and DO then engage in a secure two-party computation using Yao's Garbled Circuits in order to compare the reconstruction of the secret shares (done in the garbled circuit) to the query plaintext of the DA. The result of this comparison is again secret shared between DA and DO, i.e., neither will know whether the query plaintext is above or below the current node in the traversal. Both parties – DA and DO – send their secret shares of the comparison result to the CSP which then can determine the next node in the traversal. These steps continue until the query plaintext has been sorted into the OPE-table and the CSP has an order-preserving encoding that can be sent to the DA. A significant complication arises from this order-preserving encoding, since it must not reveal the result of the comparison protocols to the DA (although it may be correlated to the results). In the next section we provide a detailed, step-by-step description of the construction.

1.5.6 Description of the protocol

Let $\mathbb{D} = \{x_1, \cdots, x_n\}$ be the finite data set of the DO, and $h = \log_2 n$. Let $[\![x]\!]$ denote the ciphertext of x under Paillier's scheme with public key pk and corresponding private key sk that only the DO knows. Let $\mathbb{P} = \{0 \cdots 2^l - 1\}$ (e.g. $l = 32$) and $\mathbb{O} = \{0 \cdots M\}$ (M positive integer) be plaintext and order[6] range resp., i.e.: $\mathbb{D} \subseteq \mathbb{P}$.

Initialization

We assume the DO's dataset is fixed and does not change, i.e. the DO does not insert new values, during the OOPE protocol. This is a realistic assumption since this could enable the DO to learn values of the tree that come from the DA, if the insertion path of a new DO's value contains DA's values. Let $\mathbb{D} = \{x_1 \cdots x_n\}$ be the unordered DO's dataset and $h = \log_2 n$. The DO chooses a range $0 \cdots M$ such that $\log_2 M > h$, runs mOPE$_2$ to generate the ordered set $\langle x_{j_1}, y_{j_1} \rangle \leq \cdots \leq \langle x_{j_n}, y_{j_n} \rangle$. Finally, using Paillier scheme it generates the OPE-table $\langle [\![x_{j_1}]\!], y_{j_1} \rangle, \cdots, \langle [\![x_{j_n}]\!], y_{j_n} \rangle$ and sends it to the CSP.

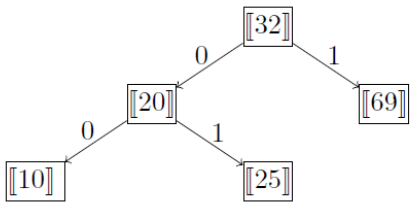
For example, assume $\mathbb{D} = \{10, 20, 25, 32, 69\}$ is the data set, $M = 28$ and the insertion order is $32, 20, 25, 69, 10$. Then the ciphertexts after the initialization step are $\langle [\![32]\!], 14 \rangle$, $\langle [\![20]\!], 7 \rangle$, $\langle [\![25]\!], 11 \rangle$, $\langle [\![69]\!], 21 \rangle$, $\langle [\![10]\!], 4 \rangle$. The OPE-tree, the OPE-table and the DO state are depicted in Figure 1.9.

The Protocol Algorithm.

The protocol (Figure 1.10) is executed between three parties. First the CSP retrieves the root of the tree and sets it as current node. Then the protocol loops h $(= \log_2 n)$ times. In each iteration of the loop the CSP increments the counter and the parties run an oblivious comparison protocol (Figure

[5] Secret shares are random values that add up to the plaintext.

[6] We will use *order* and *OPE encoding* interchangeably.



(a) OPE-tree $\mathcal{T}$

$[\![x]\!]$	y
$[\![10]\!]$	4
$[\![20]\!]$	7
$[\![25]\!]$	11
$[\![32]\!]$	14
$[\![69]\!]$	21

(b) OPE-table T

$\langle x, y \rangle$
$\langle 32, 14 \rangle$
$\langle 20, 7 \rangle$
$\langle 25, 11 \rangle$
$\langle 69, 21 \rangle$
$\langle 10, 4 \rangle$

(c) Client State

Figure 1.9: Example initialization

1.11) whose result enables the CSP to traverse the tree (Algorithm TRAVERSE). If the inputs are equal or the next node is empty then the traversal stops. However, the CSP uses the current node as input to the next comparison until the counter reaches the value h. After the loop the result is either the order of the current node in case of equality or it is computed by the CSP using Algorithm ENCRYPT. In the last step, the DA computes $[\![\bar{x}]\!]$ using DO's public key pk and sends it to the CSP. This is only necessary if the DA wants to encrypt several values, as the encryption depends on the state. We assumed that there is no update from the DO during the analysis and that all DA's values are removed from the state thereafter. Alternatively, the DA could generate an unique identifier (UID) for each element that is being inserted and send this UID instead. So if the corresponding node is later involved in a comparison step, the result is computed by the DA alone.

Oblivious Comparison Protocol

The oblivious comparison (Figure 1.11) is a protocol between the three parties as well, with input $([\![x]\!], \bar{x}, sk)$ for the CSP, the DA and the DO respectively. First the CSP randomizes its input, with a random integer $r \in \{0 \cdots 2^{l+k}\}$[7], to $[\![x+r]\!] \leftarrow [\![x]\!] \cdot [\![r]\!]$, by first computing $[\![r]\!]$ with DO's public key, such that the DO will not be able to identify the position in the tree, and it sends $[\![x+r]\!]$ to the DO and r to the DA. Then the DO with input $(b_o, b'_o, x+r)$ and the DA with input $(b_a, b'_a, \bar{x}+r)$ engage in a garbled circuit protocol for comparison as described in Section 1.5.7. For simplicity, the garbled circuit is implemented in Figure 1.11 as ideal functionality. In reality the DO generates the garbled circuit and the DA evaluates it. The DA and the DO receive $(b_e \oplus b_a \oplus b_o, b_g \oplus b'_a \oplus b'_o)$ as output of this computation and resp. send $(b_a, b'_a, b_e \oplus b_o, b_g \oplus b'_o)$ and $(b_o, b'_o, b_e \oplus b_a, b_g \oplus b'_a)$ to the CSP. Finally the CSP evaluates Equation 1.3 and outputs $\langle b_e, b_g \rangle$. This will be used to traverse the OPE-tree.

$$\begin{cases} b_e & = b_e \oplus b_o \oplus b_o = b_e \oplus b_a \oplus b_a \\ b_g & = b_g \oplus b'_o \oplus b'_o = b_g \oplus b'_a \oplus b'_a \end{cases} \tag{1.3}$$

Tree Traversal Algorithm

The tree traversal (Algorithm TRAVERSE) runs only at the CSP. Depending on the output of the oblivious comparison the CSP either goes to the left or to the the right. If the comparison step returns equality there is no need to traverse the current node and the protocol returns the corresponding ciphertext.

[7] Where k is the security parameter that determines the statistical leakage, e.g. $k = 32$ [DT08].

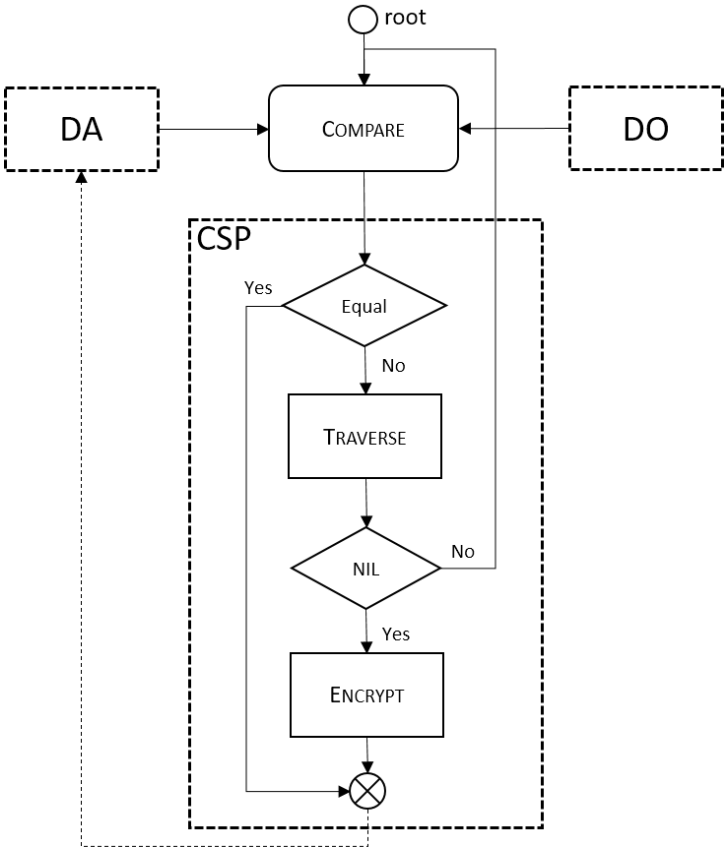


Figure 1.10: Oblivious Order-Preserving Encryption Protocol

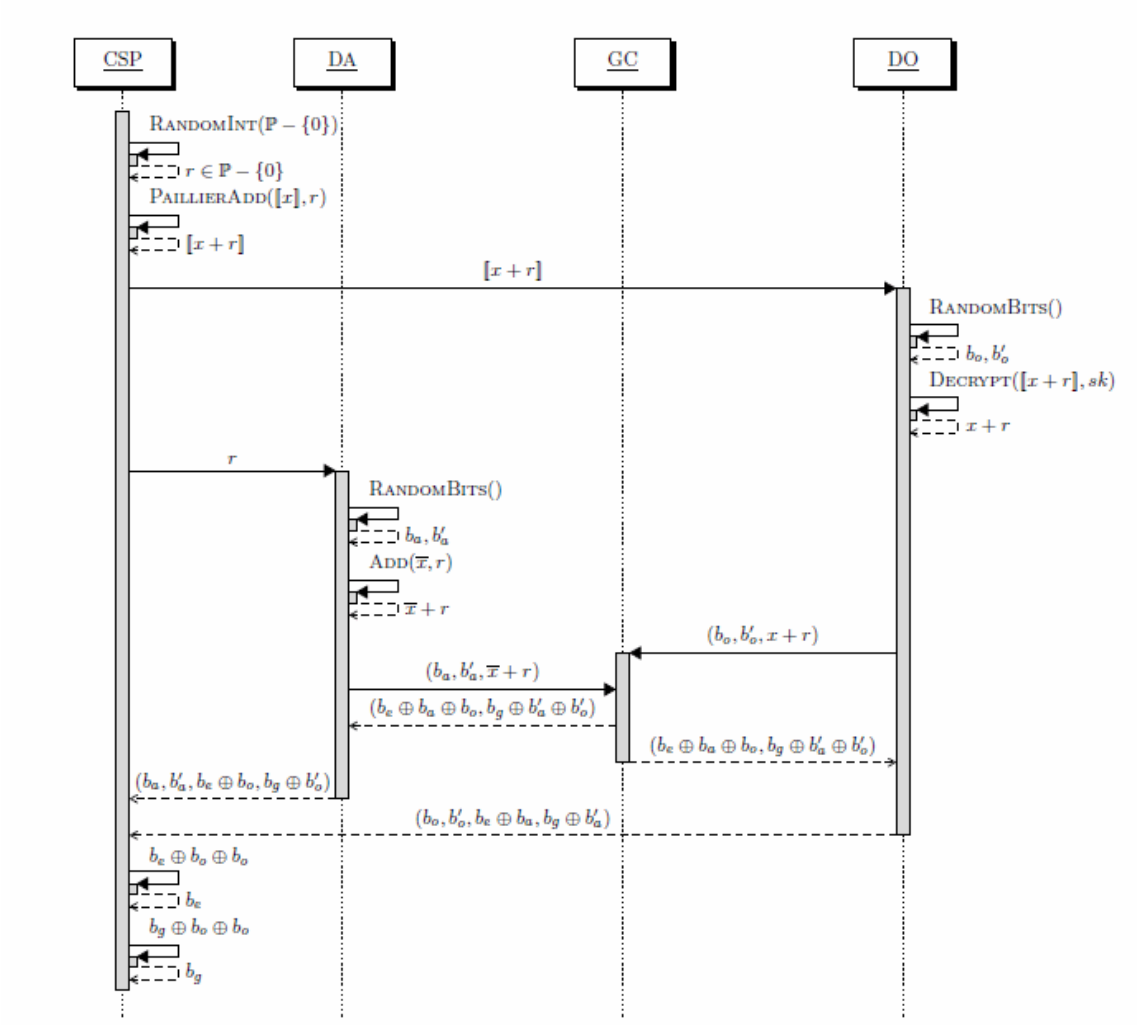


Figure 1.11: Oblivious Comparison Protocol

Encryption Algorithm

The encryption algorithm (Algorithm ENCRYPT) runs at the CSP as well and is called only if the tree traversal (Algorithm TRAVERSE) has to stop. Then the compared values are strictly ordered and depending on that the algorithm finds the closest element to the current node in the OPE-table. This element is either the predecessor if DA's input is smaller or the successor if DA's input is larger. Then if necessary, the CSP rebalances the tree before computing the ciphertext.

1.5.7 Protocol for Integer Comparison

Oblivious OPE protocol requires garbled circuit for comparison and equality test, hence we adapted the garbled circuits of [KS08, KSS09] to our needs. Firstly, instead of implementing one garbled circuit for comparison and another one for equality test, we combined both in the same circuit. This allows to use the advantage that almost the entire cost of garbled circuit protocols can be shifted into the setup phase. In Yao's protocol the setup phase contains all expensive operations (i.e., computationally expensive OT and creation of GC, as well as the transfer of GC that dominates the communication complexity) [KSS09]. Hence, by implementing both circuits in only one we reduce the two costly setup phases to one as well. Secondly, in our oblivious OPE protocol, integer comparison is an intermediate step, hence the output should not be revealed to the parties participating in the protocol, since this will leak information. Thus the input of the circuit contains a masking bit for each party that is used to mask the actual output. Only the party that receives the masked output and both masking bits can therefore recover the actual output. Let $GC_{=,>}$ denote this circuit.

Let P_1, P_2 be party one and two respectively and let $x = x_{l-1}\cdots x_0$, $\bar{x} = \bar{x}_{l-1}\cdots\bar{x}_0$ be their respective inputs in binary representation. Parties P_1 and P_2 choose masking bits b_x, b'_x, $b_{\bar{x}}$, $b'_{\bar{x}}$ and extend their input to $(b_x, b'_x, x_{l-1}\cdots x_0)$, $(b_{\bar{x}}, b'_{\bar{x}}, \bar{x}_{l-1}\cdots\bar{x}_0)$ respectively.

For equality test we use Equation[8] 1.4. The two first lines are from [KS08] and test from 0 to $l-1$ if the bits are pairwise different (i.e their exclusive-or is 1). If not we use the result of the previous bit test. Initially, this bit is set to 0.

$$\begin{cases} c_{e,0} & = 0 \\ c_{e,j+1} & = (\bar{x}_j \oplus x_j) \vee c_{e,j} \quad \text{if } j = 0\cdots l-1 \\ c_e & = c_{e,l} \oplus b_x \oplus b_{\bar{x}} \end{cases} \tag{1.4}$$

The actual output of the circuit $c_{e,l}$ is 1 if x and $\bar{x}$ are different and 0 otherwise (i.e. $c_{e,l} = [\bar{x} \neq x]?1 : 0$). Then we blind $c_{e,l}$ by applying exclusive-or operations with the masking bits b_x and $b_{\bar{x}}$.

The comparison functionality is defined as (if $\bar{x} > x$ then 1 else 0) (i.e $[\bar{x} > x]?1 : 0$). In [KSS09] the circuit is based on the fact that $[\bar{x} > x] \Leftrightarrow [\bar{x} - x - 1 \geq 0]$ and is summarized in Equation[9] 1.5, where again the two first are from [KSS09]. The second line represents the 1-bit comparator which depends on the previous bit comparison. This is initially 0.

$$\begin{cases} c_{g,0} & = 0 \\ c_{g,j+1} & = (\bar{x}_j \oplus c_{g,j}) \wedge (x_j \oplus c_{g,j}) \oplus \bar{x}_j, \quad j = 0\cdots l-1 \\ c_g & = c_{g,l} \oplus b'_x \oplus b'_{\bar{x}} \end{cases} \tag{1.5}$$

[8] In symbols $c_{e,j}$ and c_e, e stands for **equality test** and j is the bit index

[9] In symbols $c_{g,j}$ and c_g, g stands for **greater than** and j is as above

Again the actual output $c_{g,l}$ is blinded by applying exclusive-or operations with the masking bits b'_x and $b'_{\bar{x}}$.

1.5.8 Implementation and Evaluation

We have implemented our scheme using SCAPI (Secure Computation API)[EFLL12]. SCAPI is an open-source Java library for implementing secure two-party and multiparty computation protocols. It provides a reliable, efficient, and highly flexible cryptographic infrastructure. It also provides many optimizations of garbled circuits construction such as OT extensions, free-XOR, garbled row reduction [EFLL12]. Furthermore, there is a built-in communication layer that provides communication services for any interactive cryptographic protocol. This layer is comprised of two basic communication types: a two-party communication channel and a multiparty layer that arranges communication between multiple parties.

Parameters

The first parameter that should be defined for the experiment is the security parameter (i.e. bit length of the public key) of Paillier's scheme (e.g. 2048 or 4096). Paillier's scheme requires to choose two large prime numbers P and Q of equal length and to compute a modulus $N = PQ$ and the private key $\lambda = lcm(P-1, Q-1)$. Then select a random $g \in \mathbb{Z}^*_{N^2}$ such that if e is the smallest integer with $g^e = 1 \bmod N^2$, then N divides e. The public key is (g, N). To encrypt a plaintext m select a random $r \in \mathbb{Z}^*_N$ and compute Equation 1.6. To decrypt a ciphertext c compute Equation 1.7 with $L(u) = \frac{u-1}{N}$ and $\mu = (L(g^\lambda \bmod N^2))^{-1} \bmod N$.

$$c \leftarrow g^m r^N \bmod N^2 \tag{1.6}$$

$$m \leftarrow L(c^\lambda \bmod N^2) \cdot \mu \bmod N \tag{1.7}$$

The other parameters of the OOPE protocol are the length of the inputs (e.g. 32, 64, 128, 256 bits), the length of the order $\log_2 M$ - with M the maximal order - (e.g. 32, 64, 128 bits), and the size of the database (e.g. 10^3, 10^4, 10^5, 10^6 entries).

Optimization

To reduce the execution cost of our scheme we applied optimizations of Paillier's scheme as recommended in [Pai99]. We implemented our scheme with $g = 1 + N$. This transforms the modular exponentiation $g^m \bmod N^2$ to a multiplication, since $(1+N)^m \bmod N^2 = 1 + mN \bmod N^2$. Moreover, we precomputed μ in Equation 1.7, used Chinese remaindering for decryption and pre-generated randomness for encryption and homomorphic plaintext randomization (Figure 1.11). As a result, encryption, decryption and homomorphic addition take respectively $52\mu s$, $12ms$ and $67\mu s$ when the key length is 2048 bits.

Evaluation

To evaluate the performance of our scheme we answer the following questions:

- What time does the scheme take to encrypt an input of the DA?
- How does the network communication influence the protocol?

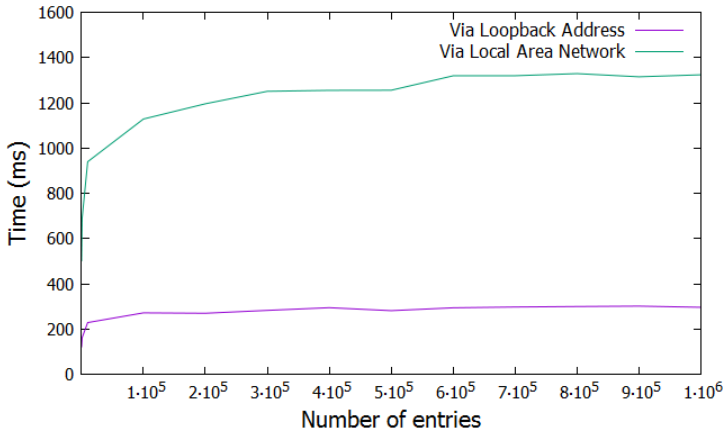
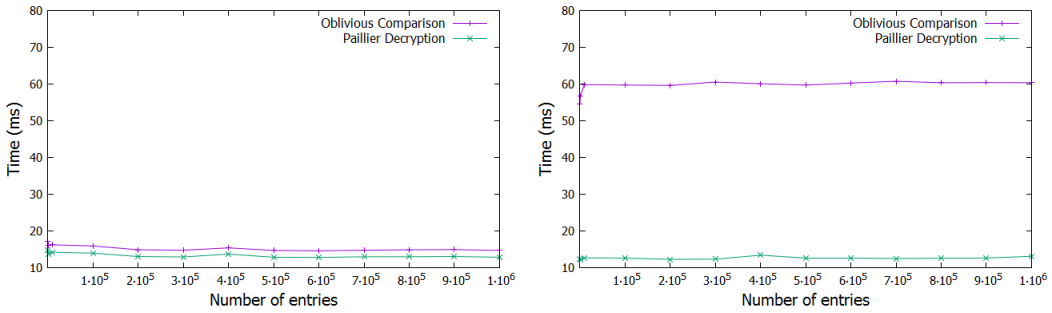


Figure 1.12: Encryption Cost of OOPE



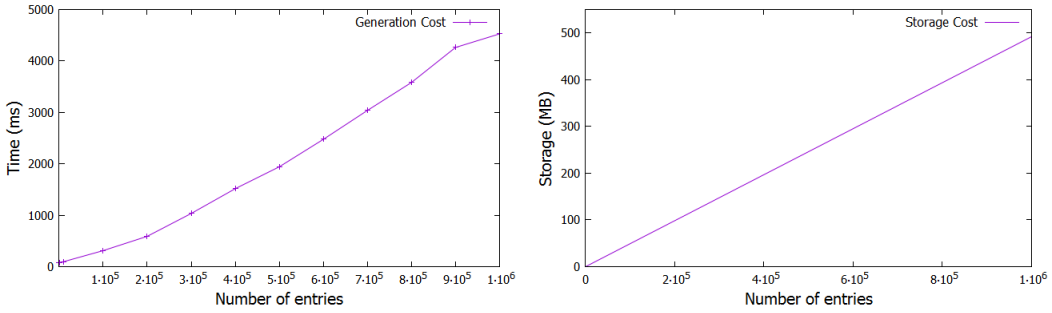
(a) Via Loopback (b) Via LAN

Figure 1.13: Cost of oblivious comparison

- What is the average generation time and the storage cost of the OPE-tree?

Experimental Setup. We chose 2048 bits as security parameter for Paillier's scheme and ran experiments via loopback address and via LAN using 3 machines with Intel(R) Xeon(R) CPU E7-4880 v2 at 2.50GHz. For the LAN experiment, the first machine with 4 CPUs and 8 GB RAM ran the CSP, the second machine with 4 CPUs and 4 GB RAM ran the DO and the last machine with 2 CPUs and 2 GB RAM ran the DA. For the loopback experiment we used the first machine. We generated the OPE-tree with random inputs, balanced it and encrypted the plaintexts with Paillier encryption. For the DA, we generated 100 random inputs. Then we executed the OOPE protocol 100 times and computed the average time spent in the overall protocol, in the oblivious comparison, in Yao's protocol, in Paillier's decryption.

Encryption Cost. Figure 1.12 shows the average cost (y-axis) needed to encrypt a value with the OOPE protocol for database sizes (x-axis) between 100 and 1,000,000. Overall, the cost for OOPE goes up as the size of the database increases. This is because the depth of the tree increases with its size. Hence, this implies a larger number of oblivious comparisons for larger trees. The average encryption time of OOPE for a database with one million entries is about 0.3 s via loopback (1.3 s via LAN). This cost corresponds to the cost of comparison multiplied by the number of comparisons (e.g. 20 comparisons for 1000000 entries).
The inherent sub-protocol for oblivious comparison does not depend on the database size but on the



(a) Average generation time

(b) Storage cost for $\log_2 N = 2048$

Figure 1.14: OPE-tree costs

input length and the security parameter $\log_2 N$. Figure 1.13 shows that this cost is almost constant for each database size. Via loopback (Figure 1.13a) the comparison costs about 14 ms which is dominated by the time (about 12 ms to the DO) to decrypt $[\![x+r]\!]$ in Protocol of Figure 1.11. The remaining 2 ms are due to the garbled circuit execution, since the overhead due to network communication is negligible. Figure 1.13b shows how the network communication affects the protocol. Via LAN (Figure 1.13b) the comparison costs about 60 ms where the computation is still dominated by the 12 ms for decryption. However, the network traffic causes an overhead of about 46 ms.

OPE-tree costs. The time to generate the OPE-tree also increases with the number of entries in the database and it is dominated by the time needed to encrypt the input data with Paillier's scheme. However, the above optimizations (i.e. choice of $g = 1+N$ and pregenerated randomness) enable a fast generation of the OPE-tree. Figure 1.14a illustrates the generation time on the y-axis for databases with size between 100 and 1,000,000 on the x-axis. For 1 million entries, the generation costs on average only about 4.5 seconds.
The storage cost of the tree depends on $\log_2 N$, the bit length of the order and the database size. Since Paillier ciphertexts are twice longer than $\log_2 N$, each OPE ciphertext $\langle [\![x]\!], y \rangle$ needs $2 \cdot \log_2 N + \log_2 M$ bits storage. This is illustrated in Figure 1.14b, with the x-axis representing the database size. The scheme needs 492.1 MB to store 1 million OPE ciphertexts, when the security parameter is 2048 and the order is 32-bit long.

1.6 Summary

Work in T3.1 focused on secure information sharing in the Cloud. For this, several solutions were formulated. First, a line of work was performed on Selective Encryption. In D3.1, a solution that supports updates to access privileges without requiring expensive re-encryption operations, thus limiting the overhead for the data owner was formulated as well as a solution based on RFIDs for the authentication of (authorized) partners within a supply chain. The shuffle index proposal (originally illustrated in D2.1) provides users knowing the encryption key access to the entire outsourced data collection. To address scenarios where users should be authorized for a different view over the data, D3.3 presented an extension of the shuffle index for fine grained access control enforcement. The proposed solution leverages selective encryption for access control enforcement on the data stored in a shuffle index. Another line of work builds on search over encrypted data (i.e., encrypted query processing). In D3.3 we formulate ENKI, the first system that efficiently supports

queries over data encrypted with different keys (i.e., between multiple users). We overcome the limitations of current approaches for multiple users offer either limited functionality or expose confidential information to the database server. It achieves modest overhead for the select queries of the TPC-C benchmark. Lastly, to increase privacy in encrypted query processing, this document introduces OOPE, a combination of property-preserving encryption and secure computation. Work on OOPE and search over encrypted data have been successfully transferred into the Use Case 2 demonstrator in WP1.

2. Secure multi-user interactions and sharing

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. Task 3.2 of ESCUDO-CLOUD is targeted at reducing this dependency, and we have developed protocols particularly to guarantee the integrity of the data stored in an untrusted Cloud.

2.1 ESCUDO-CLOUD innovation

The results of Task 3.2 consist of two main contributions.

- The first main contribution is 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, BCK17]. 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 type*. Moreover, its operations cover the hierarchical name space of a Cloud object store, supporting a real-world interface and not only a simplistic abstraction. VICOS has been covered in W3.1 as well as D3.3.
- The second main contribution is the development of generic authenticated data types (ADTs) and the provable-security analysis of a protocol for consistent multi-user access to a shared data type. In contrast to VICOS, in which the client access to the Cloud server follows the concept of a key-value store, generic ADTs allow for outsourcing arbitrary computation on the shared data to the server, in a way that the client can verify the correctness of the computation. Our generic instantiation of ADT can be seen as the stateful variant of the *hash&prove* scheme of Fiore et al. [FFG+16]. The protocol that builds on this generic ADT can be seen as a simplified variant of VICOS, but comes with a full provable-security analysis. This recent work has not yet been covered in an ESCUDO-CLOUD deliverable, but is accessible in the IACR eprint archive [CGPT17].

2.2 VICOS

VICOS is a system for verification of integrity and consistency of Cloud object storage, which has been developed in two ESCUDO-CLOUD publications [BCK15, BCK17]. It enables a group of mutually trusting clients to detect data-integrity and consistency violations when accessing

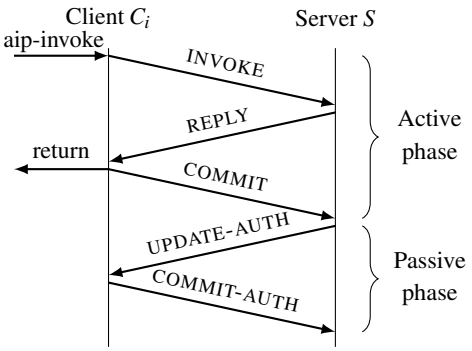


Figure 2.1: Protocol messages in AIP.

Cloud storage. In particular, it 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 has low computation and communication overhead. The VICOS system protects data stored on Cloud (object) storage services, such as OpenStack Swift or Amazon S3. As VICOS has been covered in ESCUDO-CLOUD documents W3.1 and D3.3, we only provide a short description here.

2.2.1 Concepts: The ADT integrity protocol (AIP)

VICOS builds on the *ADT integrity protocol (AIP)*, a generic protocol to verify the integrity and consistency for any authenticated data type (ADT) operated by a remote untrusted server. AIP extends and improves upon the *commutative-operation verification protocol (COP)* and its authenticated variant (ACOP) of [CO14] by reducing communication and improving parallelism. Section 2.2.2 briefly describes the architecture of the VICOS system.

Protocol AIP adopts the structure of previous protocols for verifying the consistency of an operation sequence executed by an untrusted server [MS02, CO14]. In the simplest form, each client would obtain the complete history of operations from the server S, verify everything, append its own operation, sign the history again, and send it to S. But because this is infeasible in practice, the history is represented compactly by a hash chain or through a vector clock. Furthermore, since S should not block one client C_i while another, potentially slow client C_j executes and authenticates its operation, S should respond to C_i right away. This means C_i can only verify the history speculatively since C_j has not yet verified and signed it. (The actual verification occurs asynchronously, and the protocol ensures that the security guarantees remain the same.) For a detailed illustration of such a simplified protocol, see the "bare-bones protocol" of [MS02] or the "lock-step protocol" of [CSS07].

Every client in VICOS builds a hash chain over the history of all operations in its view. It includes the hash chain output in signed messages, which makes it easy for other clients to detect violations of consistency by the server. The processing of one operation in AIP is structured into an *active* and a *passive* phase, as shown in Fig. 2.1. 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 sequence number (t) to o and responds with a REPLY message containing a list of *pending* operations, the response computed according to functionality F, 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, 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. If the pending-other operations are compatible with o, then C_i verifies (informally speaking) the pending-self operations including o with the help of the authenticator. The authenticator supposedly represents the current state held by S, it must be signed by the client whose operation produced the state and come with a hash chain output that matches the client's own hash chain. If these values are correct, C_i proceeds and outputs the response immediately. Along the way C_i verifies that all data received from S satisfies the 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. Note that C_i outputs the response of o: this must always be correct and consistent, i.e., respect fork-linearizability. The ordering, on the other hand, is speculative and assumes the pending operations will be executed and committed in this sequence.

Processing of o continues with the passive phase. Its goal is to actually execute o on the remote state and to authenticate it within the ordering on which C_i speculated during the active phase. Hence, 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, verifies the execution of o except when o aborts, and checks that the operations which were pending for o have actually been executed and authenticated as claimed by S in REPLY. 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 a different order than assigned by the global 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.

Every client needs to know about all operations that the server has executed for checking consistency, and in particular for verifying UPDATE-AUTH messages. Therefore, when S responds to the invocation of an operation by C_i, it includes in the REPLY message a summary (δ) of all authenticated operations that C_i has missed since it last executed an operation. Prior to committing o, the client verifies these signatures and thereby *clears* the operations. The client also extends its hash chain with these operations.

2.2.2 System architecture and components

The VICOS system consists of three components:

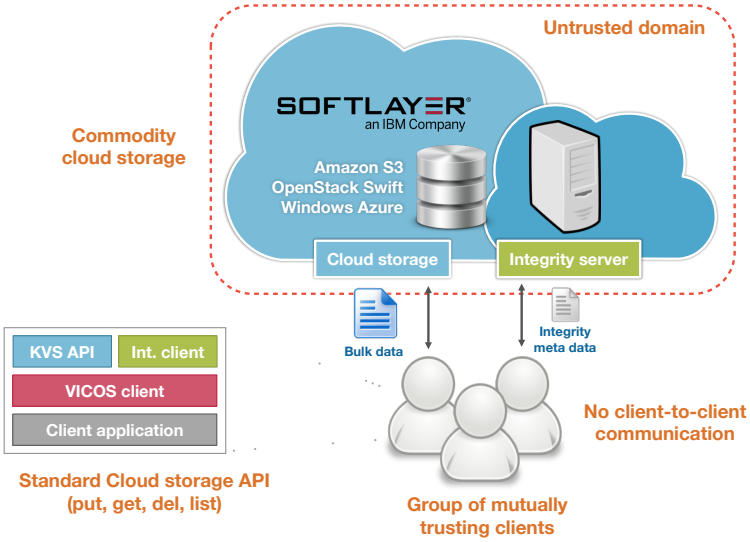


Figure 2.2: VICOS architecture

1. A *Cloud Object Store (COS)* service, as offered by commercial providers. It maintains the object data (bulk data) stored by the clients using VICOS.

2. The *VICOS server* that runs remotely as a Cloud services accessed by the VICOS client; it stores integrity-specific metadata of the object data being outsourced to the Cloud storage service. The metadata is protected through the AIP protocol for a simple key-value store.

3. The *VICOS client* enables clients to access the Cloud storage service and transparently protect the integrity and consistency of their object data. It exposes the COS interface to a client application. During each operation, the VICOS client consults the Cloud object store (using a COS API) for the object data itself and the VICOS server for integrity-specific metadata (through an AIP client). The integrity-specific metadata consists of an unique key of an object in the COS and its cryptographic hash.

The diagram in Figure 2.2 shows the architecture of VICOS. The COS and the VICOS server are *both in the untrusted domain*; they may, in fact, collude together against the clients. VICOS is written in Java, which is required to run the client and the server.

2.2.3 Conclusion

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 can 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. Indeed, the research described in Section 2.3 makes a step in this direction by generalizing the key-value store functionality to arbitrary data types.

2.3 Stateful multi-client verifiable computation

The scenario we are concerned with in this section involves, as in Section 2.2, multiple clients that mutually trust each other and collaborate through an untrusted server. In contrast to Section 2.2, which deals with a simple key-value store, the protocol in this section emulates multi-client access to any abstract data type F. On input an operation o, and current state s, the protocol computes $(s', r) \leftarrow F(s, o)$ to generate an updated state s' and an output r. The role of a client C_v is to invoke the operation o and obtain the response r; the purpose of the server is to store the state of F and to perform the computation. As an example, let F be defined for a set of elements where o can be adding or deleting an element to the set. The state of the functionality will consist of the entire set. We assume the availability of a public-key infrastructure, where each client registers its public key of a signature scheme. Clients communicate only with the server; no direct communication between the clients occurs. The goal of our protocol is to guarantee the integrity and freshness of responses, in the scenario where the server is untrusted and may be acting maliciously.

We describe the main ideas abstractly in the subsequent sections; the full details are available in the paper [CGPT17].

2.3.1 Overview

Our first contribution is a new and general security definition of a two-party ADT: The server manages the state of the computation, performs updates and queries; the client invokes operations and receives results. Our client only stores a short authenticator, and the server proves to the client that it performed the operation, i.e., compute the output and possibly a new authenticator, correctly. This significantly deviates from the standard three-party ADT definition (e.g. [Tam03, PTT11]) where there is a separation between data owner and client(s). The former needs to store the entire data in order to perform updates and publish the new authenticator in a *trusted* manner, while the latter one(s) may issue read-only queries to the untrusted server. Our definition allows the untrusted server to perform updates such that the resulting authenticator can be verified for its correctness, eliminating the need to have a trusted party store the entire data. The definition generalizes existing ones for two-party ADTs [Pap11, GGOT16] that only support deterministic schemes.

We then provide a *general-purpose* instantiation of an ADT, based on verifiable computation, by extending the work of Fiore et al. [FFG+16]. Our instantiation can capture *arbitrary* stateful deterministic computation, and the client stores only a short authenticator which consists of two elements in a bilinear group.

We also devise *computational* security definitions that model the distributed-systems concepts of *linearizability* and *fork linearizability* [MS02] in the code-based game-playing framework of Bellare and Rogaway [BR06]. This allows us to prove the security of our protocol in a computational

model by reducing from the security of digital signatures and ADTs—all previous work on fork linearizability used idealizations of the cryptographic schemes.

Finally, we describe a "lock-step" protocol to satisfy the computational fork linearizability notion, adapted from SUNDR [MS02] and Cachin et al. [CSS07]. The protocol guarantees consistent—in the sense of fork-linearizability—multi-client access to a data type. The protocol is based on our definition of ADTs; if instantiated by the general-purpose ADT construction we provide, it is a protocol for outsourcing any stateful (deterministic) computation with shared access in a multi-client setting.

2.3.2 Authenticated data types

Authenticated data types, which can be thought of as an abstraction and generalization of Merkle trees [Mer89], associate with a (potentially large) state of the data type a short *authenticator* (or *digest*) that is useful for verification of the integrity and authenticity of the data type. In more detail, an abstract data type is described by a state space S with a function $F : S \times O \to S \times A$ defined on it. F takes as input a state $s \in S$ of the data type and an operation $o \in O$ and returns a new state s' and the response $r \in A$. The data type also specifies the initial state $s_0 \in S$.

We present a definition for what is known in the literature as a "two-party" authenticated data type [Pap11]. The interaction is between a *client*, i.e., party that owns a data type which it wants to outsource, and an untrusted *server* that undertakes storing the state of this outsourced data type and responding to subsequent queries issued. The client, having access only to a succinct *authenticator* and the secret key of the scheme, wishes to be able to efficiently test that requested operations have been performed honestly by the server (see [Pap11] for a more detailed comparison of variants of ADT modes of operation).

An ADT has to satisfy two conditions, correctness and soundness. Correctness formalizes that if the ADT is used faithfully, then the outputs received by the client are according to the abstract data type F. Soundness formalizes that the server cannot cheat; i.e., any cheating attempt will be detected by the client. For the precise definition, we refer to the paper.

2.3.3 A general-purpose instantiation of ADT

We next describe one main technical contribution of the paper, namely a general-purpose instantiation of the definition of ADT described in Section 2.3.2. Our scheme builds on the work of Fiore et al. [FFG+16], which defined *hash&prove* schemes in which a server proves the correctness of a computation (relative to a state) to a client that only knows a hash value of the state. The main aspect missing from [FFG+16] is the capability for an untrusted server to *update* the state and provide the client with a new hash value that authenticates the new state.

Before we start describing our scheme, we describe some details of the *hash&prove* scheme of Fiore et al. [FFG+16]. Their construction builds on two schemes. An *verifiable computation* (VC) scheme and a *multi-exponentiation hash&prove* (MXP) scheme. In a nutshell, VC allows a (computationally powerful) server to perform a computation in a way that additionally results in a proof of the correctness of the computation. The verifier can then check the correctness of the proof and be convinced of the correctness of the result without performing the computation itself. Schemes for verifiable computation have been developed over the last couple of years, many of them (such as the one by Parno et al. [PHGR13]), use multi-exponentiations as an integral part of the proof. Fiore et al. [FFG+16] point out that multi-exponentiation can be seen as a type of hash

function, and provide a multi-exponentiation *hash&prove* scheme that, intuitively, allows to prove that two multi-exponentiations with different parameters have been computed on the same input.

The VC scheme used in [FFG+16] is actually a specific type of such scheme, namely an *offline-online verifiable computation scheme*. The offline-online property of VC states that the server can pre-compute a certain value c_z on the input, and proves to the client (in this case via MXP) that c_z contains the same value as the hash h_z known to the client. The client then concludes by verifying the proof via the online part of the verification with input c_z.

Our goal is to model stateful computations of the type $F(x,o) = (y,r)$, that is, in contrast to [FFG+16] we require that the server also prove the correctness of an updated output y. Our contribution is to extend the construction such that the offline phase of VC is applied to both x and y to obtain hashes h_x and h_y, and change the online phase to verify $F(x,o) = (y,r)$ using the correct values h_x and h_y. As in [FFG+16], we then use MXP to prove that the hashes h_x and h_y are computed correctly. The detailed description of the scheme and the proofs of correctness and security appear in the paper.

2.3.4 Computational fork-linearizable Byzantine emulation

The definitions of fork-linearizability in previous work [MS02, CSS07] state that a protocol achieves the property if *every* execution of the protocol leads to a transcripts of a certain type. This property can only be proven if cryptographic schemes are idealized; indeed, almost all practical cryptographic schemes are based on computational assumptions that can be broken by an attacker, albeit with very small probability. In order to prove fork-linearizability guarantees about protocols while using realistic models for the cryptographic components, we introduce computational versions of the properties required for fork-linearizability.

Our model follows the code-based game-playing framework of Bellare and Rogaway [BR06], and our models can be viewed in the same spirit as the definitions of authenticated key exchange of Bellare and Rogaway [BR93]. In more detail, we consider a (computationally bounded) attacker $\mathscr{A}$ that can access the individual protocol instances by means of oracle calls, and whose goal is to break the fork-linearizability properties. A protocol is then said to achieve the computational variant of these properties if no computationally bounded attacker can break them with non-negligible probability. The complete definition of the properties is provided in the paper.

2.3.5 A lock-step protocol for emulating shared data types

In the final section of the work, we describe a lock-step protocol that emulates a shared data type using the generic abstract data type instantiation described above. At a high level, the protocol can be seen as a simplified variant of the AIP protocol described in Section 2.2; the protocol has a similar message flow but does not exploit compatible operations. When instantiated with the generic abstract data type described in Section 2.3.3, the protocol consistently emulates a shared data type, although the computation is performed by the untrusted server. We then conclude the paper by formally proving that the lock-step protocol achieves computational fork-linearizability as described in Section 2.3.4.

2.3.6 Conclusion

The protocol presented in this section conceptually extends VICOS from a consistent key-value store to a Cloud platform that allows to outsource arbitrary stateful computation. The computational

effort on the client is small; indeed, clients are only required to verify signatures and to check the validity of the ADT operations. Both operations are efficient to implement, and computationally sufficiently efficient for today's mobile devices. The main computational burden is imposed on the server; indeed, although schemes for verifiable computation have improved significantly over the last years, we expect that further improvements in those schemes as well as efficiency improvements in our protocol will be necessary to make the approach practical.

2.4 Summary

When outsourcing their data to today's Cloud, clients have to trust the Cloud provider not to violate the privacy or integrity of their data. While the data of a single client can be protected using standard encryption and authentication mechanism, access by multiple interacting clients requires additional care because a misbehaving Cloud provider may attempt to provide inconsistent views on the data to multiple clients.

This chapter contains two solutions for the described scenario. First, VICOS is a protocol to secure multi-client access to a key-value store; a widely used functionality offered by Cloud providers and also referred to as *Cloud object storage (COS)*. VICOS acts as a transparent layer: it builds on top of widely-used COS interfaces such as OpenStack Swift or Amazon S3, and provides to client applications again a COS interface. Behind the scenes, however, VICOS ensures the consistency of the clients' views on the stored data. VICOS has been implemented, is available as open-source software, and exhibits performance that makes it usable in practical deployments [BCK17].

The second solution, multi-client stateful verifiable computation, extends the functionality of VICOS from a key-value store that only allows simple read and write operations to arbitrary stateful computation. This is achieved via constructing generic authenticated data types by extending the recent *hash&prove* construction of Fiore et al. [FFG+16] to untrusted updates, and then using this authenticated data type in a VICOS-like protocol. The solution is described and analyzed in a recent ESCUDO-CLOUD research paper [CGPT17]. As it builds on recent and not-yet-practical cryptographic methods, it has not yet been implemented, but we plan to further develop the protocol toward practicality.

3. Support for collaborative queries

Task 3.3 focuses on collaborative queries. Task 3.3 aims at: *i)* providing probabilistic techniques for verifying the integrity of the result of computations evaluated by collaborating parties (Section 3.2); and *ii)* enforcing selective access restrictions over data in collaborative scenarios (Sections 3.3 and 3.4). The innovative contribution by ESCUDO-CLOUD to the definition of techniques for probabilistic integrity verification is based on the combined adoption of two techniques, markers and twins, which have been refined to minimize integrity verification costs. Also, ESCUDO-CLOUD analyzed these solutions to precisely assess their effectiveness and to limit the overhead they cause. With respect to access control enforcement, the innovative contribution by ESCUDO-CLOUD is a solution, based on mixing encryption mode proposed in WP2, aimed at minimizing the data owner overhead for the enforcement of policy updates. This solution is based on the idea that it is sufficient to re-encrypt a small portion of a resource to update its access control policy. Also, ESCUDO-CLOUD proposed a novel approach for the specification and enforcement of authorizations that enables controlled data sharing for collaborative queries in the Cloud.

3.1 ESCUDO-CLOUD Innovation

Task 3.3 produced several advancements over the state-of-the-art.

- The first innovation is represented by an approach able to assess the integrity of approximate join queries (i.e., joins that combine tuples with similar, even if not equal, values for the join attributes). The proposed solution (D6.2) is based on the definition of a discretization technique that, applied to the join attribute, translates the approximate join into an equi-join. The result of the equi-join, which is delegated to an external CSP, is verified using twins and markers.

- Task 3.3 proposed refinements over existing integrity verification techniques that reduce their overhead with limited impact on integrity guarantees (D3.4). The proposed optimization is based on the idea that the size of control tuples (i.e., markers and twins) can be reduced by storing, for these tuples, their join attribute value only. Our analysis demonstrated that the adoption of slim control tuples considerably reduces the overhead, while not impacting integrity guarantees.

- Slim twins and markers enable the integrity verification of many-to-many join operations (D3.4). The adoption of slim markers and twins enables the support of many-to-many join operations without exposing the frequency distribution of join attribute values, and hence reveal twin pairs. In fact, slim twins and markers have a flat frequency distribution. They also support join operations among more than two relations, without the need of multiple interactions with the CSP (D3.4). This is due to the fact that the join between slim twins (and between markers) is always one-to-one.

- Task 3.3 provided a detailed analysis of the effectiveness of twins and markers for integrity verification aimed to support the data owner in choosing the number of twins and of markers to use for reaching the aimed integrity guarantee (Section 3.2.2).
- Task 3.3 designed a novel approach for the efficient enforcement of revoke operations. The proposed solution is based on the adoption of mixing encryption mode (proposed in WP2), which enables the data owner to change her access control policy by simply re-encrypting a small portion of the resource (Section 3.3).
- Task 3.3 developed a novel solution supporting collaborative queries in the cloud with access restrictions that regulate information flows among the parties (including external CSPs) involved in the query evaluation process (Section 3.4).

The results obtained by this task have been published in [BDF+16, DFJ+15, DFJ+16, DFP+16, DFJ+18].

3.2 Twins and Markers for Integrity Verification

The first issue addressed in this task is the definition and analysis of approaches aimed at providing integrity guarantees to the results of computations that involve multiple collaborating parties. The work within ESCUDO-CLOUD aimed first at extending two integrity verification techniques, twins and markers [DFJ+13], to limit their computation and communication overhead as discussed in D3.4 (Section 3.2.1). Then, the work focused on the analysis of the integrity guarantees provided by these techniques, to provide the user with support in the choice of the number of twins and markers to adopt for reaching a given integrity guarantee (Section 3.2.2).

3.2.1 Slim Twins and Markers

In D3.4, we presented an approach aimed at limiting the computation and communication overhead caused by the adoption of integrity verification approaches when a user delegates the evaluation of an equi-join operation to potentially untrusted CSPs. To achieve this, we proposed a variation of twins and markers.

The considered scenario is characterized by a user who is interested in delegating the evaluation of an equi-join operation to a CSP. The join query operates over the relations of two trusted parties, who collaborate for query evaluation. In the absence of security considerations, the owners of the two relations involved in the equi-join would send their data to the CSP, which would compute their join and return the result to the requesting user. However, since the CSP is potentially untrusted, it is necessary to protect both data confidentiality and integrity of the join result. To this end, the proposal in [DFJ+13] combines the following protection techniques.

- *Encryption on-the-fly*: the operand relations are encrypted by their owners, using a key known to the requesting user only, before being sent to the CSP. Encryption guarantees data confidentiality.
- *Markers*: fake tuples, not recognizable as such, are inserted into the operand relations by their owners. The set of join attribute values used for markers is the same in the two operand relations, and is disjoint from the actual domain of the join attribute. Absence of a marker signals the incompleteness of the join result.

- *Twins*: a subset of the tuples in the operand relations are duplicated by the relation owners. Since duplication is driven by a condition on the join attribute value, all the tuples satisfying the twinning condition must appear twice in the join result. A twin appearing solo signals the incompleteness of the join result.

- *Salts and buckets*: used to flatten the frequency distribution of the values of the join attribute (and hence also protect twin pairs) in one-to-many joins. Buckets, used on the *many* side of the join, of tuples with the same value for the join attribute create groups all of the same size. If a value is less frequent than the bucket size, the bucket is filled with dummy tuples. Salts consist in making different occurrences of the same join attribute value different by using (in encryption) a different salt for each occurrence. Each value in the relation on the *one* side of the join is replicated and encrypted with the different salts used by the relation on the side many of the operation. Salts and buckets can be used singularly or in combination.

Since the adoption of these integrity verification technique causes both a computation and a communication overhead, D3.4 proposed variations aimed at limiting integrity verification costs. The approaches proposed to limit integrity verification costs can be summarized as follows.

- *Semi-join*. The first solution presented in D3.4 for reducing the overhead due to markers, twins, salts and buckets consists of adopting the semi-join (in contrast to regular join) evaluation strategy. According to this strategy, the CSP operates only on the result of the projection on the join attribute of the two operand relations. Hence, the join is always a one-to-one join and the relations exchanged between the parties are smaller.

- *Limit salts and buckets to twins and markers*. Whenever the frequency distribution of the join attribute value is not considered sensitive, salts and buckets can be limited to twins and markers. Indeed, it is sufficient to flatten their distribution to prevent any observer from identifying markers and twin pairs based on the frequency of join attribute values.

- *Slim twins and slim markers*. Since integrity checks only consider the values of the join attribute of markers and twins, to reduce their costs we proposed to reduce the size of these control tuples by removing their content. In case of one-to-many joins, the adoption of slim twins reduces the number of twin tuples since the same slim tuple can act as twin for all the tuples having the same original join attribute value. In this case, the twin tuple will store the number of tuples for which it is a twin.

As discussed in detail in D3.4, all these solutions considerably reduce the cost of integrity checks, while not impacting integrity guarantees. We also note that the introduction of slim twins and markers also enables the evaluation of many-to-many joins, as well as of joins involving more than two relations.

3.2.2 Twins and Markers Analysis

The definition of probabilistic integrity verification technique naturally requires further studies to enable users to effectively use these techniques. Indeed, it is necessary to guide the user in the choice of the protection technique that provides higher integrity guarantees, while limiting costs. To this end, the work in Task 3.3 analyzed the effectiveness of twins and markers, with the aim of guiding end users in choosing the number of markers and twins she needs to adopt to obtain the desired integrity guarantees.

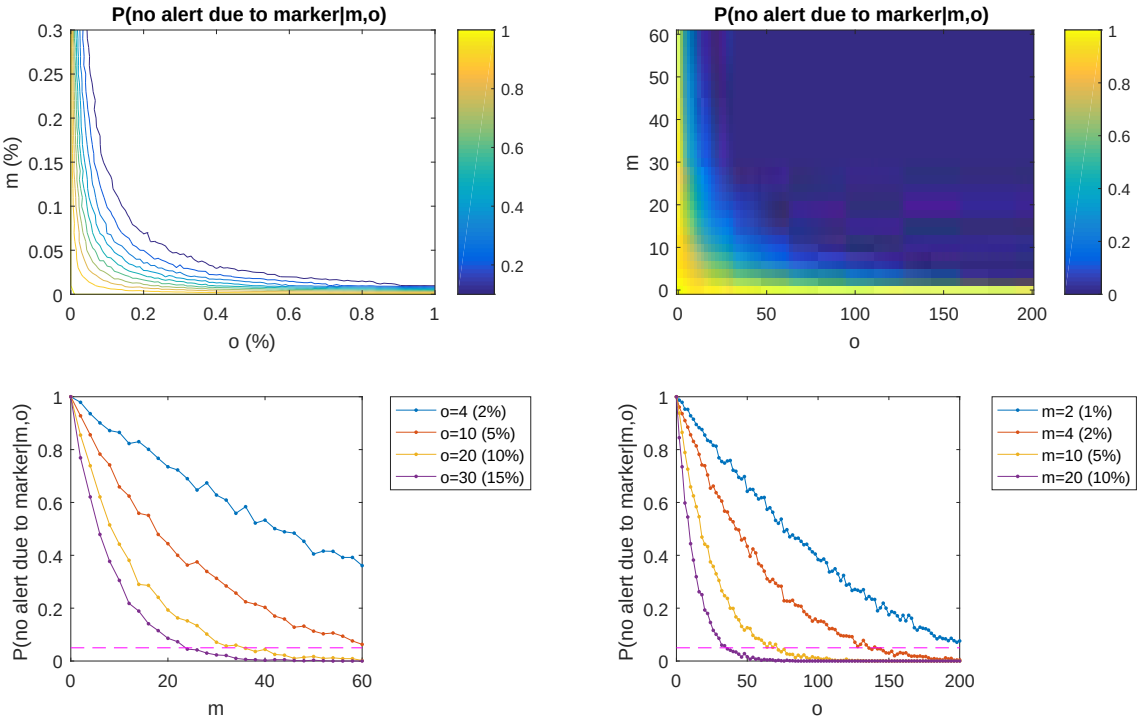


Figure 3.1: Probability that the misbehavior of a CSP goes undetected when using only markers

In our analysis, we considered a scenario characterized by a user who is interested in delegating a general *classification task* to a multitude of CSPs. Since CSPs might not be fully trusted, the user adopts probabilistic integrity verification technique to verify their behavior. Indeed, CSPs could randomly choose a class for their input data, instead of running the classification algorithm provided by the user, thereby saving computational resources. For simplicity, but without loss of generality, we assume that the result of the classification is complete (i.e., each input data is returned, together with its class, to the requesting user) and the integrity verification techniques are aimed at controlling the correctness of the classification result. We studied separately the integrity guarantees provided by the adoption of markers and twins.

Markers. Markers are classification jobs that operate on artificial input data and that produce a known result. A result, for the evaluation of a marker, different from the expected one signals the incorrectness of the classification result (and the misbehavior of the CSPs). To be effective, markers should be non-recognizable as such by CSPs, who could otherwise selectively misbehave without being detected. Each marker has a cost for: *i)* its generation; *ii)* its evaluation by a CSP; *iii)* the verification of its result by the user; *iv)* the transmission of input data to the CSP and of the evaluation result to the user. Usually, the verification of marker results has limited costs, while their generation can be more expensive.

To assess the effectiveness of markers, we analyzed the probability that a CSP that does not classify a subset of the data items (but randomly selects a class for each of them) goes undetected. With this aim, we modeled the behavior of the CSP and of the user and performed a set of Monte Carlo simulations. Figure 3.1 reports the results of 1000 simulations, considering a dataset including 200 input data items, a classification with 4 classes, and a Zipf distribution of the 200 data items in the 4 classes. The figure illustrates the probability that a mis-behavior by the CSP does not trigger any alert (i.e., all markers produce the expected results), varying the number m of markers injected into the input dataset and the omissions by the CSP (i.e., the number o or percentage $o\%$ of data

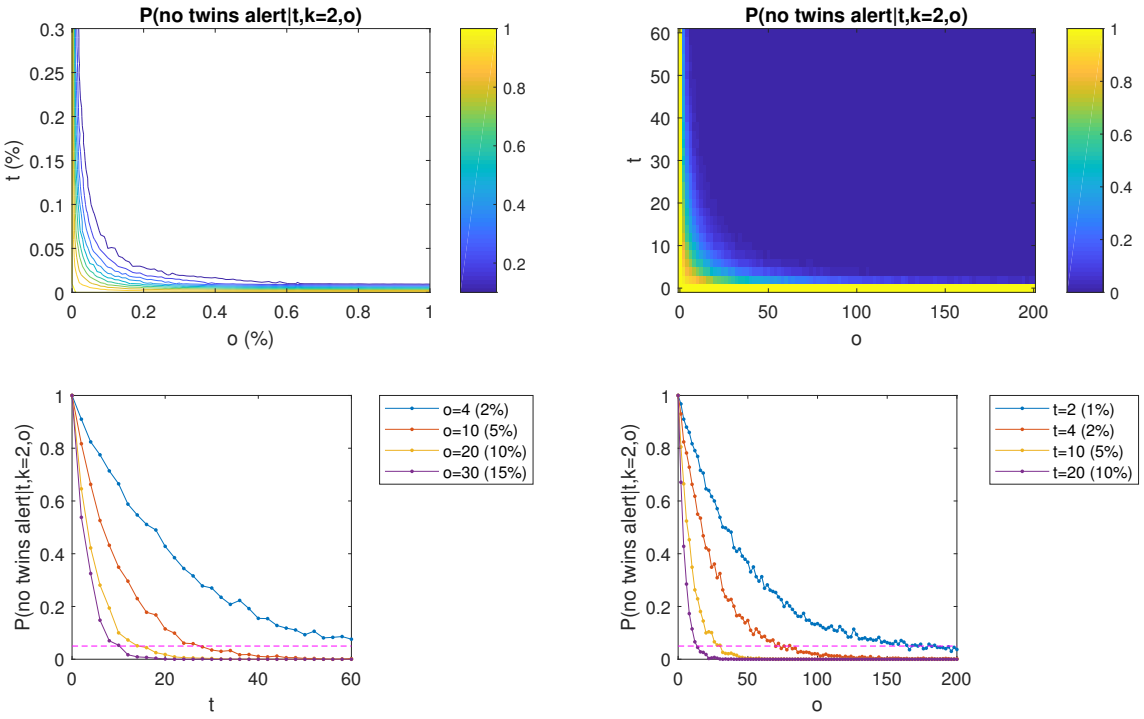


Figure 3.2: Probability that CSP misbehavior goes undetected when using only twins

items on which the CSP does not run the classification algorithm). As visible from the figure, the probability that an omission goes undetected decreases as the number of markers grows. Similarly, it decreases as the number of omissions grows.

Twins. Twins are replicated jobs, which should produce exactly the same result and can then be used to verify the correctness of the classification result. If the result of the evaluation of twin jobs is not the same, the user can easily determine that the classification result is not correct. To be effective, twin jobs should not be recognizable by CSPs: if a CSP is able to identify twin jobs, it can evaluate them in the same (possibly wrong) way, thus eluding the verification. Similarly to markers, twins have a cost due to: *i)* replication of jobs; *ii)* evaluation of the same job multiple times; *iii)* comparison of the results of twin jobs; *iv)* multiple transmissions of the same input data and of the results of twins.

To assess the effectiveness of twins, we studied the probability that at least one of the CSPs to which twin jobs are assigned does not perform all the classification jobs assigned to it, but goes undetected. As with markers, we modeled the CSPs and user behavior and performed a set of Monte Carlo simulations. Figure 3.2 reports the results of 1000 simulations, considering the same configuration used for the simulations in Figure 3.1 with 10 CSPs and assuming that each twin is replicated once. The figure illustrates the probability that CSPs misbehavior do not trigger any alert (i.e., all the twin pairs produced the same result), varying the number t of twinned jobs and the omissions by the CSP (i.e., the number o or percentage $o\%$ of data items on which the CSP does not run the classification algorithm). As can be seen from the figure, the probability that an omission goes undetected decreases as the number of twins grows. Similarly, it decreases as the number of omissions grows.

3.3 Mix&Slice For Efficient Access Revocation

One of the complex aspects in using encryption to enforce access control policy concerns access revocation. If granting an authorization is easy (it is sufficient to give the newly authorized user access to the key), revoking an authorization is a completely different problem. There are essentially two approaches to enforce revocation: *i)* re-encrypt the resource with a new key or *ii)* revoke access to the key itself. Re-encryption of the resource entails, for the data owner, downloading the resource, decrypting it and re-encrypting it with a new key, re-uploading the resource, and re-distributing the key to the users who still hold authorizations. If decryption, re-encryption, and even key management (for this specific context) can be considered a trivial issue, the remaining challenge is represented by the need to download and re-upload the resource, with a considerable overhead for the data owner. This overhead will continue to grow as usage of Cloud resources grows, in particular in the context of emerging big data applications. The alternative approach of enforcing revocation on the resource by preventing access to the key with which the resource is encrypted cannot be considered a solution. As a matter of fact, it protects the key, not the resource itself, and it is inevitably fragile against a user who - while having been revoked from an access - has maintained a local copy of the key.

In this deliverable, we present a novel approach to enforce access revocation that provides efficiency, as it does not require expensive upload/re-upload of (large) resources, and robustness, as it is resilient against the threat of users who might have maintained copies of the keys protecting resources on which they have been revoked access.

3.3.1 Mix&Slice

The basic idea of our approach is to use *mixing* encryption mode, presented in D2.6, which guarantees complete interdependence (*mixing*) among the bits of the encrypted content. In this way, unavailability of even a small portion of the encrypted version of a resource completely prevents the reconstruction of the resource or even of portions of it. Brute-force attacks guessing possible values of the missing bits are possible, but even for small missing portions of the encrypted resource, the required effort would be prohibitive. The *all-or-nothing transform* (AONT) [Riv97] considers similar requirements, but the techniques proposed for it are not suited to our scenario, because they are based on the assumption that keys are not known to users, whereas in our scenario revoked users can know the encryption key and may plan ahead to locally store critical pieces of information.

Our approach trades off between the requirement to connect all bits of a resource (to provide the desired interdependency of the content), and the requirement to maintain fine grained access of the resource itself. This is a particular challenge due to the potentially huge size of the resources. To achieve this, we apply the idea of mixing content within portions of the resource, enforcing then revocation by overwriting encrypted bits in every such portion. Before mixing, our approach partitions the resource in different, equally sized, chunks, called *macro-blocks*. Then, as the name hints, it is based on the following concepts.

- *Mix:* the content of each macro-block is processed by an iterative application of different encryption rounds together with a carefully designed bit mixing, that ensures, at the end of the process, that every individual bit in the input has had impact on each of the bits in the encrypted output.
- *Slice:* the mixed macro-blocks are sliced into fragments so that fragments provide complete coverage of the resource content and each fragment represents a minimal (in terms of number

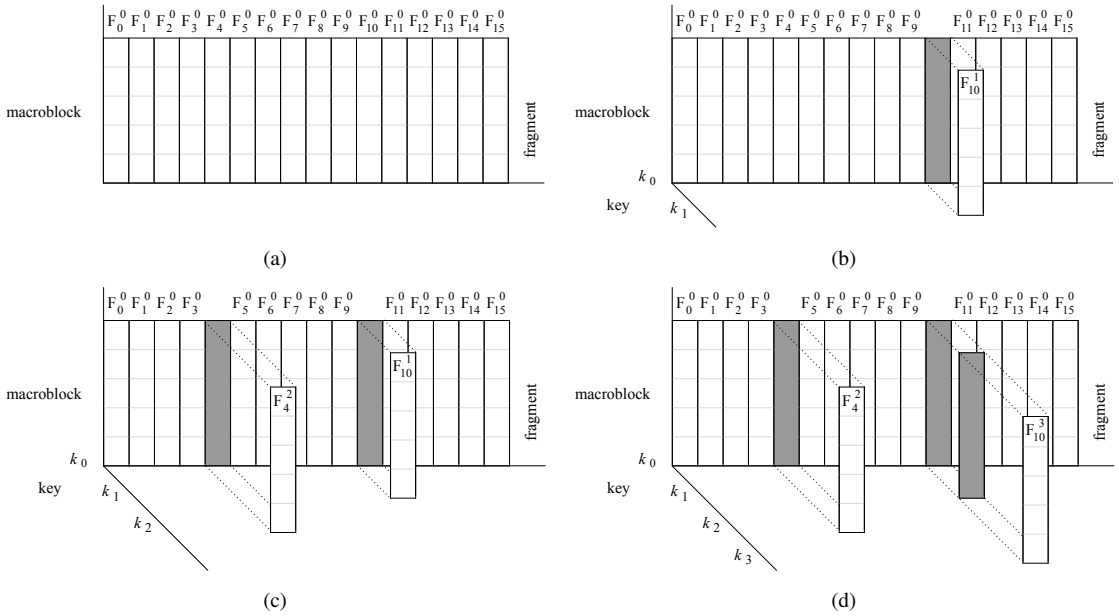


Figure 3.3: An example of fragments evolution

of bits of protection, which we call *mini-block*) unit of revocation: lack of any single fragment of the resource completely prevents reconstruction of the resource or of portions of it.

To revoke access from a user, it is sufficient to re-encrypt one (any one) of the resource fragments with a new key not known to the user. The advantage is clear: re-encrypting a tiny chunk of the resource guarantees protection of the whole resource itself. Also, the CSP simply needs to provide storage functionality and is not required to play an active role for enforcing access control or providing user authentication. Our *Mix&Slice* proposal is complemented with a convenient approach for key management that, based on key regression, avoids any storage overhead for key distribution.

3.3.2 Access management

Accessing a resource (or a macro-block in the resource, resp.) requires availability of all its fragments (its mini-blocks in all the fragments, resp.), and of the key used for encryption. Policy changes corresponding to granting access to new users can be simply enforced, as usual, by giving them the encryption key. In principle, policy changes corresponding to revocation of access would instead normally entail downloading the resource, re-encrypting it with a new key, re-uploading the resource, and distributing the new encryption key to all the users who still hold authorizations. Our approach enables the enforcement of access revocation to a resource by simply making any of its fragments unavailable to the users from whom the access is revoked. Since lack of a fragment implies lack of a mini-block for each macro-block of a resource, and lack of a mini-block prevents reconstruction of the whole macro-block, lack of a fragment equates to complete inability, for the revoked users, to reconstruct the plaintext resource or any portion of it. In other words, it equates to revocation.

Access revocations are then enforced by the data owner by randomly picking a fragment, which is then downloaded, re-encrypted with a new key (which will be made known only to users still authorized for the access), and re-uploaded at the server overwriting its previous version. While

Revoke

1: randomly select a fragment F_i of R — /* fragment to be rewritten */
2: download F_i^c from the server — /* version of the fragment stored */
3: **if** $c > 0$ **then** — /* F_i^0 has been overwritten in a revocation */
4: derive key k_c — /* derive k_c using key regression */
5: $F_i^0 := D(k_c, F_i^c)$ — /* retrieve the original version of the fragment */
6: determine the last key k_{l-1} used — /* it is stored in R's descriptor */
7: generate new key k_l
8: $F_i^l := E(k_l, F_i^0)$
9: upload F_i^l overwriting F_i^c — /* overwrite previous version */
10: encrypt s_l with the key of acl(R) — /* limits it to authorized users */
11: update R's descriptor — /* including the new s_l */

Figure 3.4: Revoke on resource R

still requesting some download/re-upload, operating on a fragment clearly brings large advantages (in terms of throughput) with respect to operating on the whole resource (see Section 3.3.4). Revocation can be enforced on any randomly picked fragment (even if already re-written in a previous revocation) and a fresh new key is employed at every revoke operation. Figure 3.3 illustrates an example of fragments evolution due to the enforcement of a sequence of revoke operations. Figure 3.3(a) is the starting situation with the original fragments computed as illustrated in Section 3.3.1. Figure 3.3(b-d) is the sequence of rewriting to enforce revocations, which involve, respectively, fragment F_{10}, re-encrypted with key k_1, fragment F_4, re-encrypted with key k_2, and fragment F_{10} again, now re-encrypted with key k_3. In the following, we use notation F_i^j to denote a version of fragment F_i encrypted with key k_j, being F_i^0 the version of the fragment obtained through the mixing process. In the figure, the resource is represented in a three-dimensional space, with axes corresponding to fragments, macro-blocks, and keys. The re-writing of a fragment is represented by placing it in correspondence to the new key used for its encryption. The shadowing in correspondence to the previous versions of the fragments denote the fact that they are not available anymore as they are overwritten by the new versions.

Each revoke operation requires the use of a fresh new key and, due to policy changes, fragments of a resource might be encrypted with different keys. Such a situation does not cause any complication for key management, which can be conveniently and efficiently handled with a *key regression* technique [FKK06]. Key regression is an RSA-based cryptographically strong technique (the generated keys appear as pseudorandom) allowing a data owner to generate, starting from a seed s_0, an unlimited sequence of symmetric keys $k_0, \ldots, k_u$, so that simple knowledge of a key k_i (or the compact secret seed s_i of constant size related to it) permits to efficiently derive all keys k_j with $j \leq i$. Only the data owner (who knows the private key used for generation) can perform forward derivation, that is, from k_i, derive keys following it in the sequence (i.e., k_z with $z \geq i$). Note instead that, not knowing the private key, users cannot perform forward derivation. The cost that users must pay for key derivation is small. On a single core, the computer we used for the experiments is able to process several hundred thousand key derivations per second.

With key regression, every user authorized to access a resource just needs to know the seed corresponding to the most recent key used for it (s_0 if the policy has not changed, s_3 in the example of Figure 3.3(d)). To this end, there is no need for key distribution, rather, such a seed can be stored in the resource descriptor and protected (encrypted) with a key corresponding to the resource's *acl* (i.e., known or derivable by all authorized users) [AFB05, DFJ+07]. Enforcing revocation entails then, besides re-encrypting a randomly picked fragment with a fresh new key k_i, rewriting its

Access

download R's descriptor and all its fragments
retrieve seed s_l used for the last encryption
compute keys $k_0, \ldots, k_l$
for each downloaded fragment F_i^x **do**
  if $x > 0$ **then**
    $F_i^0 := D(k_x, F_i^x)$ /* retrieve the original version of fragments */
for $j = 0, \ldots, M-1$ **do** /* reconstruct and decrypt macro-blocks */
  M_j := concatenation of mini-blocks $F_i^0[j]$, $i = 0, \ldots, (m \cdot b) - 1$
  decrypt M_j

Figure 3.5: Access to resource R

corresponding seed s_i, encrypted with a key associated with the new *acl* of the resource. Figure 3.4 illustrates the revocation process.

To access a resource, a user then first downloads the resource descriptor, to retrieve the most recent seed s_l, and all the fragments. With the seed, she computes the keys necessary to decrypt fragments that have been overwritten, to retrieve their version encrypted with k_0. Then, she combines the mini-blocks in fragments to reconstruct the macro-blocks in the resource. She then applies mixing in decrypt mode to macro-blocks to retrieve the plaintext resource. Figure 3.5 illustrates the process to access a resource.

Note that the size of macro-blocks influences the performance of both revoke and access operations. Larger macro-blocks naturally provide greater policy update performance as they decrease policy update cost linearly, with limited impact on the efficiency of decryption, since its cost increases logarithmically (Section 3.3.4).

3.3.3 Effectiveness of the approach

In this section, we elaborate on the effectiveness of our approach for enforcing revocation. For the discussion, we denote with *msize* the size of individual mini-blocks, m the number of mini-blocks in a block, b the number of blocks in a macro-block, M the number of macro-blocks, and f the number of fragments (i.e., $f = m \cdot b$).

We consider the threat coming from a user whose access to the resource has been revoked, and who downloads the resource from the server. With access policy enforced by encryption, not being authorized for an access should not prevent downloading the resource but rather it should prevent reconstruction of its plaintext representation. We then evaluate the protection against the user's attempts to reconstruct the plaintext resource. In doing so, we consider the worst case scenario, with respect to key management, where the user has maintained memory of the last key (or the corresponding seed) used for the resource, up to the point in which she was authorized for the access. In other words, we assume the user to be able to decrypt the fragments that have been overwritten before she has been revoked access, and hence, to know the original version encrypted with k_0 of the fragments that have not been overwritten since she has been revoked access. Since seeds are compact, such a threat is indeed realistic. To reconstruct the resource when missing a fragment, the user would have to perform a brute force attack attempting all possible combinations of values of the missing bits, that is, 2^{msize} attempts for each of the M macro-blocks. If more fragments, let's say f_{miss}, are missing, the user would have to perform $2^{msize \cdot f_{miss}}$ attempts for each of the M macro-blocks.

The inability of the user to reconstruct a resource if some fragments have been overwritten is

because, without such fragments, the user cannot retrieve the corresponding original version (the one encrypted with k_0) needed to correctly reconstruct the resource plaintext. A potential threat can then come if the user maintains a local storage with the original version of part of the resource. We distinguish two cases, depending on whether the user stores complete fragments or portions of them across the whole resource.

Local storage of fragments. Suppose a user locally stores (when authorized) some fragments of the resource. Even if such fragments are later overwritten for revoking access to the user, and then their most recent version stored at the server is unintelligible to her, she has them available for reconstructing the resource. However, the fragment to be overwritten in a policy revocation is chosen randomly by the owner. Therefore, the user can still reconstruct the resource after one fragment has been overwritten if the fragment that the owner has overwritten is the same fragment that the user has also stored locally, which has probability $1/f$ to occur. Generalizing the reasoning to the consideration of the user locally storing more than one fragment and the policy naturally changing even after the specific user revocation, we determine the probability P_A of the user's ability to access the resource assuming local storage of f_{loc} fragments to be $P_A = (f_{loc}/f)^{f_{miss}}$. The probability clearly increases with the number of fragments stored locally, but quickly reaches extremely low values after a few updates of the policy, approximating zero even for high percentage of fragments locally stored. The low probability (and the high storage effort requested to the user) essentially makes such attack not suitable: if the user has to pay a storage cost that approaches the maintenance of the whole resource, then the user would have stored the plaintext resource when authorized in the first place. We also note that a possible extension of our approach could consider overwriting, instead of pre-defined fragments, a randomly chosen set of mini-blocks (ensuring coverage of all macro-blocks), to enforce a revocation. In this case, the probability of the user storing m_{loc} mini-blocks per macro-block (also randomly chosen) to be able to access the resource immediately after her revocation would be $(m_{loc}/(m \cdot b))^M$, which would become $(m_{loc}/(m \cdot b))^{M \cdot m_{miss}}$, (i.e., negligible), if she misses m_{miss} mini-blocks per macro-block. We note, however, that overwriting randomly picked mini-blocks across the resource would considerably increase the complexity in the management of fragments, and it would make it harder to provide an efficient physical structure for fragments (Section 3.3.4). Given the observations above about the high storage cost that would be required to the user and the low probability of her success as policy changes, we argue that the regular structure for the fragments is preferable.

Keeping portions of all mini-blocks. Instead of locally storing some selected fragments, a user can opt for using storage to maintain portions of all the mini-blocks in each fragment. In this case, whatever the fragment overwritten in the revocation, the user will have to perform some effort to realize a brute-force attack to retrieve the missing bits (she does not have the complete fragment), but such an effort will be lower, given the availability of the locally stored bits. For instance, assuming the user to keep 50% (i.e., half of the bits) of each mini-block, the effort for reconstructing the resource given a missing fragment would now be $2^{(msize/2)}$ attempts for each of the M macro-blocks (in contrast to the 2^{msize} required if all the bits in the fragment were unknown). However, again, if more fragments are missing, the required effort would quickly escalate, being equal to $2^{(msize/2) \cdot f_{miss}}$ when f_{miss} fragments are missing. For each attempt, the verification that a guess is correct would require to apply all the decryption rounds until the plaintext is reconstructed, with a great cost. We note that the user can cut down on such cost if she locally maintains, in addition to the portions of the original mini-blocks, also some bits of the partial results of the computation (which would allow her to test correctness of a guess without performing all the

encryption rounds). Availability of such partial results can help testing the guesses for a mini-block if the other mini-blocks in the same block are available (i.e., when the user misses only one fragment per block). However, from the birthday paradox, we note that the probability of two revocations hitting the same block (but a different fragment) quickly increases with the number of revocations. Then, after a few updates the advantage of the user keeping partial results of the computation will become ineffective. In addition to this, we note that, in this case as well, the storage and computational efforts required to the user do not seem to make this attack much preferable for her with respect to the choice of locally storing the whole plaintext resource itself in the first place.

A note on collusion. Collusion can happen when two users join effort to gain access to a resource that neither of them can access (we do not consider collusions with the server, which is assumed trustworthy to enforce the re-writing requested by the owner). In fact, if one of the users is authorized for the resource, she has no incentive and therefore there is no collusion. Also, the case of users working together to grant each other access to resources on which they individually have authorization cannot be considered collusion, since merging their knowledge they collectively do not go beyond their privileges. Collusion is then represented by users who join effort in locally storing portions of the resource (e.g., fragments or parts of mini-blocks as discussed above). For instance, each of the users could keep half of the fragments and they can merge their knowledge to patch for missing fragments. Such a situation does not add any complication with respect to the previous discussion, as it simply reduces to consider the group of colluding users as an individual attacker. We then note again that the collective effort, in terms of storage and/or computation, required to gain access would easily approximate the effort of locally storing the original plaintext resource itself. In other words, the attack strategy does not offer any advantages to users attempting to access the resources for which they are no more authorized.

3.3.4 Implementation

In this section, we discuss the realization of our approach for its practical deployment. The components that have to be considered are the *client*, who decrypts resources to access them (Section 3.3.4), and the *protocol* used for the interaction between client and server. The protocol has a significant impact on the profile of the *server* responsible for hosting the resources and for authenticating the data owner who is the only party authorized to modify the data. In particular, we will consider two options for the realization of the interaction protocol: *i) Overlay* (Section 3.3.4), which operates on top of a common Cloud object service (the server is unaware of the adoption of our approach and is a standard object server); and *ii) Ad-hoc* (Section 3.3.4), which directly supports the primitives to update a fragment and to get the current state of the resource (the server is aware of the features of our approach and attention will have to be paid to its internal structure).

We found from this analysis that the client is able to make use of our approach without restrictions, with a performance in the application of the encryption technique within a common personal computer that is compatible with any network bandwidth; the network then proves to be the bottleneck in the access to external resources. For the protocol, when the technique is applied in a transparent way on top of existing object storage solutions (Overlay), we observe several orders of magnitude in performance improvement for some configurations. The realization of the technique using an ad-hoc protocol further improves the benefits with its greater flexibility, but it also requires to consider the mapping of the logical structure to its physical representation - for which we have identified an adequate solution. All these results prove the applicability of the technique in the current technological landscape and the benefits that it can provide for many application domains.

It is important to observe that the primary parameters influencing the performance are the size *msize* of mini-block and the number f of fragments. While the size of mini-blocks represents our security parameter and must be chosen by the data owner based on her security requirements, the number of fragments is chosen considering performance only. In the following, we will then focus on the tuning of the number of fragments, considering resources of variable sizes. Note that the choice of the number of fragments implies also the definition of the number of macro-blocks, as the product of the number of macro-blocks by the number of fragments is equal to the number of mini-blocks of the resource.

The evaluation of the best value for the number of fragments will have to consider a number of aspects that characterize the application domain. The major ones are: frequency of policy updates; frequency and average size of `get` requests; network bandwidth, for the upload and download direction. All these aspects have a direct impact on the overall throughput offered by our solution, which confirms its advantage in the prompt enforcement of revoke operations, measured by the average transfer rate for `get` requests.

The experimental results illustrated in this section have been obtained using, for the client, a machine with Linux Ubuntu 16.04 LTS, Intel i7-4770K, 3.50 GHz, 4 cores. For the server, we used an Amazon EC2 m4.large instance, with 4 CPUs and 8 GB of RAM. The client was connected to the Internet by a symmetric 100 Mbps connection.

Client

Our approach requires the client to execute a more complex decryption compared to the use of AES with a traditional encryption mode (e.g., CTR or CBC). The cost of decryption (which is comparable to the cost of encryption by the data owner) is nearly $\log_m(m \cdot b)$ times the cost of applying a single AES decryption, while the impact of reorganizing the data structure at each round is limited. Due to the high performance of modern processors in the execution of block ciphers, this logarithmic cost factor is not critical. Also, decryption can be parallelized on multi-core CPUs, making the client processing even more efficient.

An aspect that has to be considered in the implementation of the client is the possible need to keep large amounts of data in memory. This may occur when fragments are downloaded one after the other and decryption can start only after the last fragment has been downloaded, which, for example, happens with the Overlay solution. If the resource size exceeds the available memory at the client, this leads to an extremely significant performance hit. The configuration of the system can (and should) avoid this possibility by splitting the resource into sub-resources (Section 3.3.4).

Experiments on the client All code has been written in Python, because for all the functions the computational performance is not a constraint. The only component written in C was the invocation of the mixing for encryption and decryption functions. Since most current Intel x86 CPUs offer the support for a hardware implementation of AES, named AES-NI, we considered its adoption in our experiments. Figure 3.6 shows that the cost of decryption is compatible with all reasonable scenarios for the application of our technique. In particular, the figure illustrates the throughput obtained, varying the number of threads, by the application of our approach in different configurations characterized by macro-blocks of size (*Msize*) 4 KiB, mini-blocks of size (*msize*) 32 and 64 bits (which imply 5 and 9 encryption rounds, resp.), when using AES-NI and when not using it (AES). Mixing was applied on data that were already available in memory. We notice that even the single-threaded 9-round non-hardware-supported implementation (line ‘AES, *msize*=64,

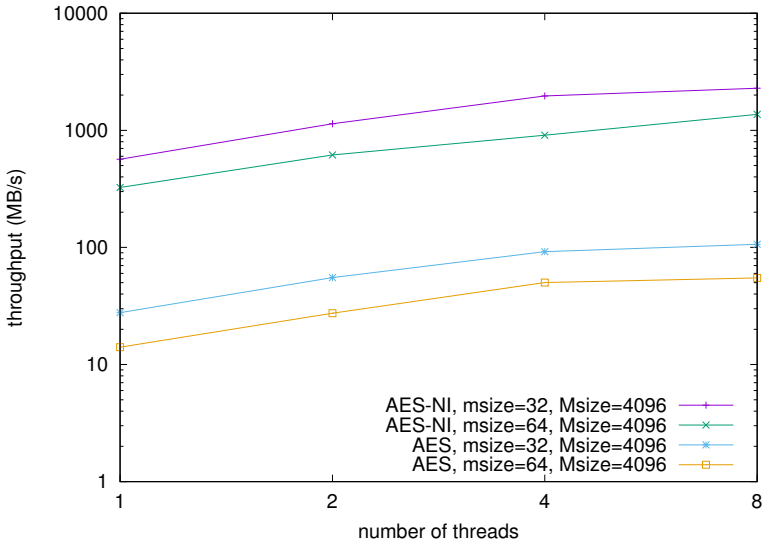


Figure 3.6: Throughput varying the number of threads

Msize=4096') offers a throughput that is greater than 100 Mbps. For the AES-NI multi-threaded 5-round implementation we reach a 2.5 GB/s throughput (line 'AES-NI, *msize*=32, *Msize*=4096'). The figure also shows that, increasing the number of threads, we reach a performance level that is 4 times the one obtained by the single-threaded implementation. This is consistent with the presence of 4 physical cores in the CPU we used, each with a dedicated AES-NI circuitry.

The performance, even for a large number of fragments, shows to be orders of magnitude better than the bandwidth of current network connections. Even without the hardware support (lines 'AES, *msize*=32, *Msize*=4096' and 'AES, *msize*= 64, *Msize*=4096'), the application of the cryptographic transformation shows greater throughput than the data transfer rate of most Internet connections. An experiment on 1 GiB size macro-blocks and 32 bit mini-blocks showed the expected slow down in throughput, managing the decryption in less than 5 seconds (still above the bandwidth of long-distance connections).

Overlay solution

The Overlay solution is analyzed using as a reference the Swift service. Swift has been selected due to its popularity, availability as open source, and technical features that are good representatives of what is offered by a modern object storage service for the Cloud (resources are called *objects* in this discussion, to align with the Swift terminology). The Swift server instance has been installed on the Amazon EC2 platform. We consider two main alternatives for the realization of our approach on Swift[1] without any changes to the server.[2] The first option assumes to manage each fragment as a separate object. The second option makes use of the ability to access portions of objects and specifically considers the use of *Dynamic Large Objects* (DLOs). Our experiments show that this latter option provides significant benefits in performance with respect to managing fragments as separate objects. DLOs deserve then to be used when available.

[1] Swift organizes objects within *containers*. The current structure of Swift supports access control only at the level of containers. The analysis we present can be immediately adapted to the management of the access policy at the container granularity rather than the object granularity. We keep the analysis at the level of object to be consistent with the discussion in the deliverable.

[2] We had to change a parameter in the server, to support a large number of fragments in the DLO mode.

Fragments as atomic separate objects. This approach is the most adaptable one, as it can be used with any object storage service. Also, the support for a policy update will be immediate, as it will be mapped to a single update to the object containing the corresponding fragment. However, these advantages come together with some potential restrictions. The client would be responsible for managing mixing and slicing. The approach requires the introduction of some metadata associated with each of the fragments or stored in a dedicated supporting object. The client has to be able to concurrently access all the fragments of the object to exhibit good performance when accessing large resources. If there are many fragments, this requires creating and keeping open a large number of connections with the server.

Use of DLOs. The *Dynamic Large Object*[3] (DLO) service of Swift has been introduced to support the management of large objects, going beyond the size limits of storage devices and providing finer granularity with the access. When using DLOs, an object is separated into a number of sub-objects that can be downloaded with a single request. The fragments of our approach can then be stored into separate DLO fragments. The Swift server is responsible for the management of the mapping from an object to its fragments, splitting a request for downloading an object into a number of independent requests to the server nodes that are responsible to store the data (the Swift architecture has a server node directly offering an interface to the clients and uses a number of independent storage nodes; this architecture provides redundancy and availability). In this way, the client only generates a single `get` request for the object, independently from the number of fragments. The descriptor of the object can be extended with the representation of the version of each fragment. A similar approach can be realized when the object service offers the flexibility to operate with `get` and `put` only on a portion of the object.

The major constraint of this approach is the need to wait for the download of all the fragments before the decryption of the first macro-block can start. As anticipated in Section 3.3.4, this causes delays and requires the client to keep available in RAM the complete encrypted representation of the object before it can be processed. To mitigate this problem, fragments can also be split into sub-fragments. In this way, the download will be organized with a serial download of all the sub-fragments representing the same set of macro-blocks. This is consistent with approaches used in Cloud storage, where there is a common guideline to split resources larger than a few GiB (Swift forces a split at 5 GiB in its standard configuration). Experiments confirm that beyond 1 GiB, the throughput remains stable even for configurations with a large number of fragments.

Experiments on the Overlay solution We built a Swift client application in Python that implements the `get` and `put_fragment` methods that characterize our technique. We followed two implementation strategies, one using fragments as atomic separate objects, and the other adopting the DLO support offered by Swift.

Figure 3.7 compares, for different numbers of fragments, the time required for the execution of `get` requests assuming to map each fragment to a separate object. The lines correspond to distinct values for the number f of fragments (i.e., 1, 4, 16, 64, 256, and 1024). The parameters that drive the performance are the network bandwidth and the overhead imposed by the management of each request. For `get` requests, the overhead introduced by the management of one request for each fragment dominates when the resource is small, whereas the increase in object size makes the network bandwidth the bottleneck. The profile of `put` requests uploading the complete resource proved to be identical to the profile of `get` requests using a single fragment. The execution of

[3] `https://docs.openstack.org/swift/latest/overview_large_objects.html`

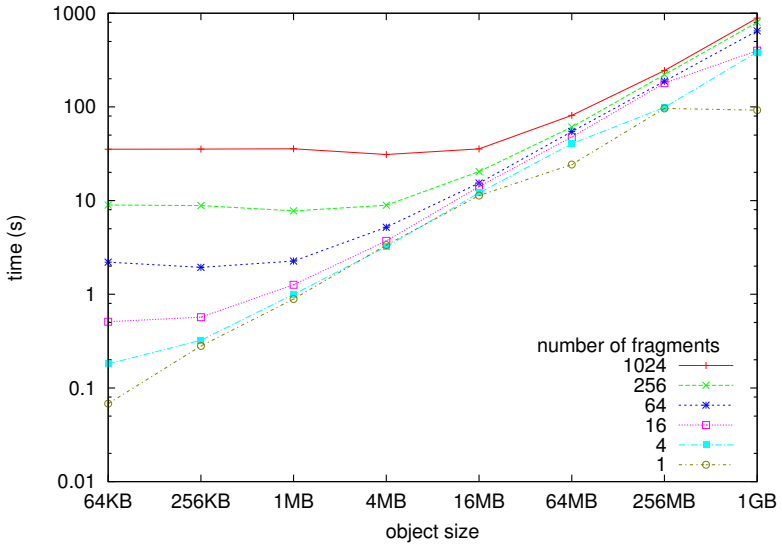


Figure 3.7: Time for the execution of `get` requests on Swift

`put_fragment` requests grows linearly with the size of the fragment.

The identification of the best number of fragments requires to consider the profile of the scenario. We evaluated the behavior of a system on a collection of 1000 objects where, after each `put_fragment` request, a sequence of 50 `get` requests were executed on objects in the collection, all of the same size. Figure 3.8 reports the results of these experiments. As objects become larger, the benefits of fragmentation in the application of policy updates compensate for the overhead imposed on the retrieval of the objects. It is important to note that the performance of the solution that does not use our technique corresponds to the line with one fragment. The throughput of the configurations using fragments is orders of magnitude higher already for medium-size objects. The graph also shows that the best number of fragments depends on the resource size. The identification of the value to use requires to consider the configuration of the system and the expected workload.

A second set of experiments followed the same approach, but considering the use of DLOs in Swift. The number of fragments still has a significant impact on the performance of the `get` request, because the server has to generate internally the mapping for the single request originating from the client and the multiple requests addressed to the storage nodes. The application of the same workload considered for the experiments in Figure 3.7, which interleaves `get` and `put_fragment` requests, produces the results presented in Figure 3.9. Comparing the cost with and without DLO we notice a significant benefit deriving from the use of DLOs.

Ad-hoc solution

The use of an ad-hoc protocol is able to provide the full range of benefits of our approach. The protocol will have to support the basic primitives to upload (`put`) and download (`get`) a resource. The `put` primitive, when used to upload the initial state of the resource, will have to provide a resource descriptor that defines: the identifier of the key k_0 used by the owner to encrypt the resource; the size of mini-blocks and the number of fragments (which determine the size of the macro-block); an array with an element for every fragment describing its version. In addition to the `put` primitive, the server will recognize the `put_fragment` primitive, which will allow the owner to update a fragment. Parameters of this primitive, in addition to the resource identifier and

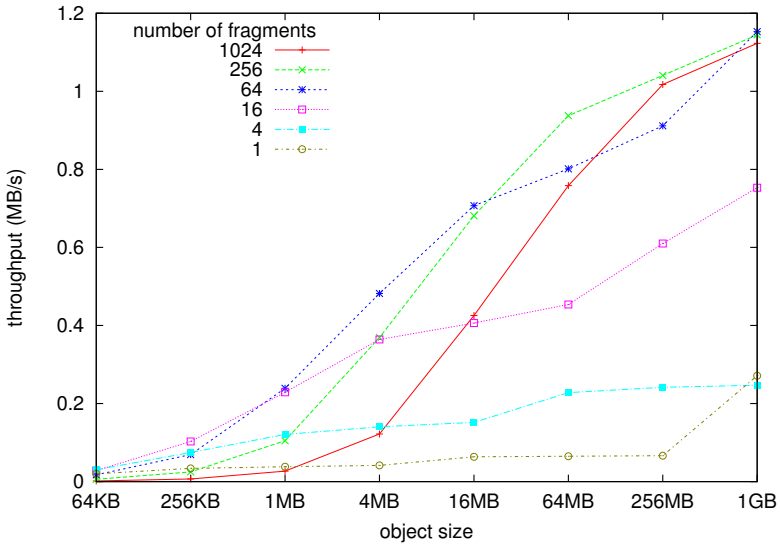


Figure 3.8: Throughput for a workload combining `get` and `put_fragment` requests on Swift

fragment content, will be the identifier of the fragment and its version number. The `put_fragment` primitive requires the authentication of the user issuing the request, in the same way as the `put` primitive.

The `get` primitive can return to the user the resource, one macro-block after the other. The client will be able to immediately start the decryption of macro-blocks, after a preliminary decryption with key k_i of the mini-blocks belonging to the fragments at version $i > 0$. In this way, the client does not have to wait for the completion of the download of all the fragments. The answer to the `get` request always provides first the resource descriptor, with the representation of the version of each of the fragments. Among the parameters of the `get` primitive we have the option to retrieve only a specific portion of the resource.

For this solution, we have to dedicate attention to the mapping of the logical structure to the physical representation of data. At the logical level, the resource is divided into fragments, and the content is represented by a sequence of macro-blocks. At the physical level, the resource can be stored as a collection of separate fragments or as a sequence of macro-blocks. In addition to these two options, there is a range of intermediate alternatives, with the interleaved representation of multiple fragments.

Experiments on the Ad-hoc solution The advantage of a dedicated server is the ability to use an efficient protocol. The use of an ad-hoc server makes the management of fragments more flexible and avoids the overheads associated with the generation of a number of independent `get` requests equal to the number of fragments that are produced by the Overlay solution. Still, the use of a potentially large number of fragments can introduce non-negligible costs. In the extreme case where a large resource is managed with a single macro-block (i.e., the number of fragments corresponds to the number of mini-blocks of the whole resource), the client will have to wait for the download to complete to start decryption, and decryption will involve a high number of rounds. Also, when only a portion of the resource is needed, our approach requires the client to download the macro-blocks that contain the portion of interest; if macro-blocks are large, this may lead to a significant overhead. As already discussed, the identification of the optimal number of fragments has to consider several features of the application domain. In the current technological scenario,

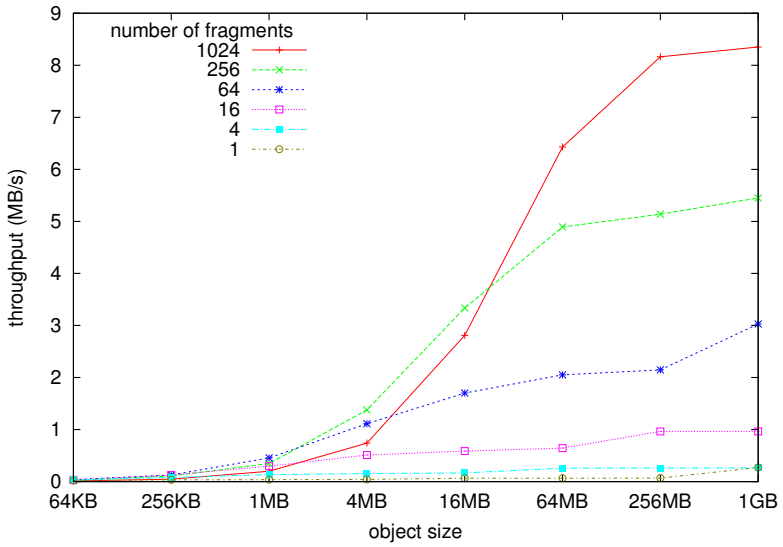


Figure 3.9: Throughput for a workload combining `get` and `put_fragment` requests with Swift DLOs

we notice that the use of an ad-hoc server can support a number of fragments larger than what is adequate for the Overlay solution, but extreme values cause inefficiencies.

As mentioned above, an important aspect that the implementation of the ad-hoc server has to consider is the mapping from the logical structure to its physical representation. In this analysis, we will consider a traditional scenario where the server uses the functions of the operating system to access the storage ability of mass memory devices. In the experiments we used the Amazon EC2 instance and its access to the Elastic Block Storage. The operating system offers an interface that allows to read and write physical blocks, typically a few KiB in size. The mapping of the bi-dimensional logical structure with macro-blocks and fragments to the concrete physical structure realized by a sequence of physical blocks can follow several strategies. To compare these alternatives, we assume a scenario where we have 1024 fragments and map the structure to 4KiB physical blocks. A first strategy consists in storing the resource one macro-block after the other. The dual strategy consists in storing the resource one fragment after the other. Between these two extremes, we have strategies that split each macro-block into a number of parts and store contiguously into a physical disk block all the macro-block portions that correspond to the mini-blocks in the same position. The rationale is that the organization along macro-blocks will be the most efficient to support `get` requests, but it will require to access all the physical disk blocks when a `put_fragment` request is received. The representation based on fragments will instead be the most efficient to support `put_fragment` requests, but it will introduce a significant overhead when managing `get` requests. For small resources these aspects do not have a large impact, whereas for large resources the performance benefit can be significant. Figure 3.10 illustrates the results obtained on a container with 1000 files, each of 1 GiB in size. The horizontal axis denotes the number of shares of each macro-block (1 represents the strategy with the macro-blocks stored in sequence, and 1024 represents the strategy with fragments stored in sequence). For a workload that interleaves a `get` request for every `put_fragment` request, the total cost is minimized when we use a solution with 256 fragments. Interestingly, the two extremes with this workload do not represent the best option. In these experiments, we measured the time required to access the data from storage. In most systems we expect the network to be the bottleneck that limits the performance and the

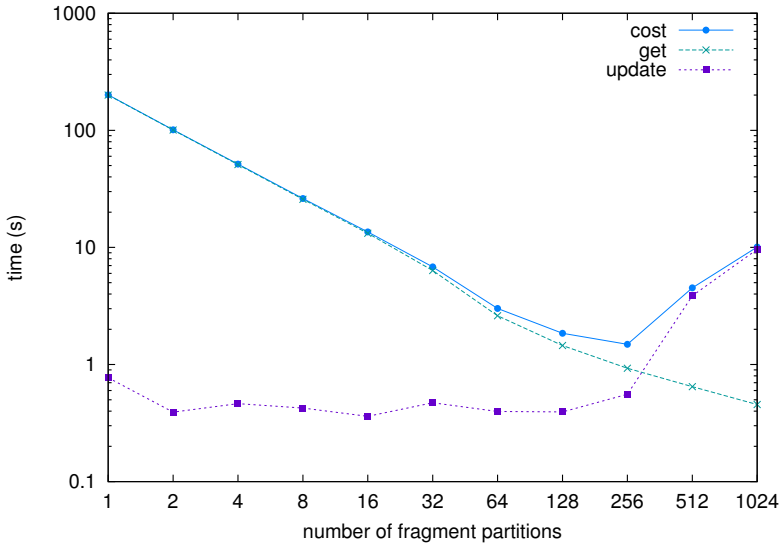


Figure 3.10: Configurations for physical blocks

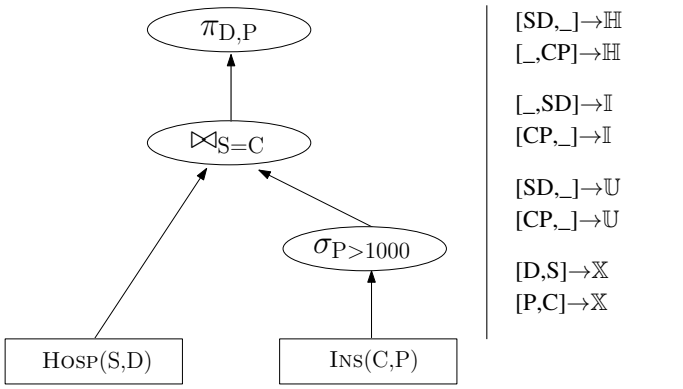
choice of physical representation will rarely be observed by the clients, but the performance benefit that is shown by the experiment can lead to a more efficient implementation of the server.

3.4 Collaborative Queries in the Cloud with Access Restrictions

With the Cloud market evolving every day more into a rich and diversified marketplace, users can choose among a wide spectrum of solutions made available by Cloud providers to execute (computation-intensive) data computations. The evolution of technology makes available a variety of storage and computation providers, with different costs, performance, and security guarantees. In such a scenario, ensuring appropriate protection models and techniques to data that can be sensitive, proprietary, or simply subject to access restrictions becomes a key requirement for allowing users to leverage (low-costs) Cloud providers for storing and elaborating data. In the third year of the project, we have address this problem and proposed a novel approach enabling collaborative and distributed query execution with the controlled involvement of providers that might be not fully trusted to access the data content. Our goal is twofold: first, to allow data authorities to make their data available for possible collaborative processing, while maintaining control over them; second, to allow users accessing such data to leverage the rich and diverse offer of the Cloud market, by relying on providers for performing queries over such data. In this section, we illustrate our approach for the specification and enforcement of authorizations that enables controlled data sharing for collaborative queries in the Cloud.

3.4.1 Rationale and Running Example

The core of our proposal is a simple, yet flexible, authorization model that enjoys the great advantage of simplicity of specification and management. Each data authority can establish authorizations regulating the release to other subjects (i.e., users, providers, and other data authorities) of data under its control. Authorizations are specified by each authority independently (no cross-domain authorization or collaborative administration is required) and selectively grant visibility on the data to other subjects. Visibility can be granted either plaintext or encrypted. Subjects authorized for



(a) Query plan (b) Authorizations

Figure 3.11: An example of a query plan (a) and of authorizations on relations HOSP and INS (b)

encrypted visibility over some data can perform computations (e.g., evaluate conditions or perform joins) over the data without accessing the actual data values. Leveraging the availability of solutions that support operations on encrypted data (e.g., CryptDB [PRZB11a] and the SEEED framework over the SAP Hana DBMS [G+14]), this feature increases the spectrum of potential providers to which operations within a query can be assigned. Query execution can then selectively involve, in the different steps of the computation, different data authorities and Cloud providers as deemed desirable for economic or performance reasons. Authorizations imposed by data authorities are enforced by applying encryption/decryption on-the-fly as needed to disable/enable data visibility as demanded by authorizations and operation requirements. Authorization enforcement will entail controlling not only direct data access, or release, but also accounting for information implicitly conveyed as a result of a computation.

For concreteness, but without loss of generality, we frame our work in the context of relational database systems. We consider queries of the general form "SELECT FROM WHERE GROUP BY HAVING" that can include joins among distinct relations under control of different data authorities. Execution of queries is performed according to a query plan established by the query optimizer. The query plan is represented as a tree `T(N)` whose leaves are base relations and whose non-leaf nodes are operations to be executed to perform the query. We assume the query plan to be produced with classical optimization criteria, and in particular we assume that projections are pushed down to avoid retrieving data that are not of interest for the query. Graphically, we represent a leaf node as a square box that contains (the projection of) a source relation. We refer to leaf nodes as base relations. In this section, we consider as a running example two data authorities: a hospital $\mathbb{H}$, storing relation HOSP(S,D), reporting the SSN (S) and Disease (D) of hospitalized patients, and an insurance company $\mathbb{I}$ storing relation INS(C,P), reporting for each Customer (C) the insurance Premium (P). We consider a user $\mathbb{U}$ as well as a third Cloud provider $\mathbb{X}$ offering computational power. Our running example considers the execution, on behalf of user $\mathbb{U}$, of query "SELECT D, P FROM HOSP JOIN INS ON S=C WHERE P>1000" retrieving the diseases and premiums of hospitalized patients whose premium is greater than 1000. Figure 3.11(a) reports the query plan for our query.

3.4.2 Authorization Model

We assume a simple, yet expressive, authorization model in which each data authority specifies authorizations regulating the release of its data. Authorizations are defined at the fine-grained level of attribute specifying, for every attribute, whether a *subject* (i.e., a user, a data authority, or a provider) can have:

- *plaintext visibility*: the subject has complete visibility on the values of the attribute;
- *encrypted visibility*: the subject cannot view the plaintext values of the attribute, but can view an encrypted version of them;
- *no visibility*: the subject cannot view the values of the attribute at all (neither plaintext nor encrypted).

While plaintext and no visibility do not require explanation, since they correspond to traditional ways of regulating access, the encrypted visibility, which represents a characteristic and strength of our proposal, deserves some clarification. The reason behind the consideration of the encrypted visibility is to provide a subject with the ability to operate on an attribute for performing joins with other relations or for evaluating conditions on encrypted values (supported by the kind of encryption used), while not releasing to the subject the actual values of the attribute. In the authorization model we do not distinguish among different encryption schemes, so to leave the model simple and the approach flexible. In fact, expressing the encryption scheme in the authorizations would introduce considerable complexity in the specifications, without providing an actual advantage in the end. The distinction among encryption schemes will be made by the query optimizer in the generation of the query plan, depending also on the operations that are to be executed on the encrypted data.

Consistently with standard security practice, we assume a "closed" policy for the specification of authorizations, meaning that only accesses explicitly authorized are allowed (i.e., 'no visibility' does not need to be specified, as it applies whenever the other two do not). Authorizations are then defined as follows.

Definition 3.4.1 (Authorization) *Let R be a relation and $\mathcal{S}$ be a set of subjects. An* authorization *is a rule of the form $[P,E]\rightarrow S$, where $P\subseteq R$ and $E\subseteq R$ are subsets of attributes in R such that $P\cap E=\emptyset$, and $S\in\mathcal{S}\cup\{any\}$.*

Authorization $[P,E]\rightarrow S$ states that subject S can view attributes P in plaintext and attributes E encrypted. Sets P and E are required to be disjoint. However, we note that an authorization that permits a subject S to access an attribute a in plaintext also allows S to access the encrypted version of the attribute. We assume that, for each relation, a subject can hold at most one authorization (the consideration of multiple authorizations would not increase expressivity). Since the set of subjects who might be involved in a query, and for whom release of data may be requested, may not be completely known a priori, a default authorization can be specified, which applies to all subjects for which no explicit authorization already exists for the interested relation. This is accommodated by the consideration of value 'any' as subject of the authorization.

We expect users to have authorizations that include plaintext attributes only, since users need to be able to access the queries' responses and manage keys for attributes encrypted in the computation. The data authority storing a relation can be expected to hold an authorization for accessing its content in plaintext (i.e., S storing $R(a_1,\ldots,a_n)$ is authorized for $[\{a_1,\ldots,a_n\},\_]\rightarrow S$). Providers and other data authorities may instead have authorizations that also include encrypted attributes,

allowing them to operate on relations that include some attributes without disclosing the attributes' values to them. Figure 3.11(b) illustrates an example of authorizations for our running example. For simplicity, in the figure and in the remainder of this section, we denote a set of attributes simply with the sequence of the attributes composing it, omitting the curly brackets and commas (e.g., SD stands for $\{S, B\}$).

3.4.3 Relation Content Model

To determine whether the release of a relation to a subject should be accepted according to authorizations, it is necessary to capture the informative content of a relation, which might not be completely expressed by the attributes appearing in its schema. This may happen, for instance, due to the evaluation of a selection condition, or of a grouping operation, on attributes that are then removed from the relation schema through a projection. As a simple example, the relation resulting from "SELECT A FROM *R* WHERE B='10'", while containing only A in its schema, indirectly leaks information on the values of attribute B as well, and should therefore not be visible to subjects not authorized to see both A and B. Capturing the informative content of a relation *R* (resulting from a computation) requires then to take into account such indirect information leakage and relationships among attributes, which we characterize through the concepts of *implicit* and *equivalent* attributes.

- *Implicit attributes.* Implicit attributes are attributes not necessarily appearing in a relation schema but that have been taken into account in the computation of the relation. Basically, implicit attributes for a relation *R* are all those attributes that appear in a selection condition or grouping operation in the (sub-)query producing *R*. The information indirectly conveyed differs depending on the selection condition considered. For instance, a selection condition 'B=10' leaks the fact that all the tuples in the result have value of B equal to 10, disclosing B precisely even if it is not explicitly visible in the relation. A selection condition 'B>10' leaks instead the fact that the tuples appearing in the relation have a value for B greater than 10, but without leaking B's actual values. The evaluation of a GROUP BY clause over B is similar to the evaluation of equality condition 'B=*value*', where *value* may be unknown. Consistently with the fact that we operate at the schema level, we do not distinguish among the degrees of leakage and assume an attribute to be *implicitly visible* in a relation (i.e., indirectly exposed) if the attribute was taken into account – in some way – in the computation of the relation. The concept of implicit visibility applies to both plaintext and encrypted attributes.

- *Equivalent attributes.* Equivalence among attributes captures the fact that some attributes have been connected in a computation (i.e., some conditions among them have been applied) and therefore visibility of one attribute indirectly leaks the other(s). Like for implicit attributes, the degree of such a leakage can depend on the condition enforced. For instance, condition 'A=B' implies precise leakage of the values of B from the visibility of A, while condition 'A>B' entails a partial leakage, as a subject viewing A can only infer the fact that B has a value lower than the one visible for A. Again, we do not consider different degrees of leakage (which would introduce considerable complexity and fuzziness in the approach, with limited advantages in the enforcement of authorizations), but simply capture such a connection between the attributes, considering them as equivalent from the point of view of authorization enforcement (as visibility of one entails visibility of the other). Given a relation *R*, we say that two attributes are *equivalent* if the (sub-)query producing *R* involves a condition comparing them. The equivalence relationship is symmetric and transitive. Different sets of

equivalent attributes can exist for a given relation. The equivalence relationship can apply to both explicit as well as implicit attributes, and to plaintext as well as encrypted attributes.

In the following, we refer to attributes explicitly visible in a relation as *visible* attributes, and to those implicitly leaked as *implicit*. In addition, attributes can be *plaintext* or *encrypted*.

3.4.4 Authorization Enforcement

We now illustrate how query execution can be regulated ensuring obedience to authorizations.

Authorized Visibility

The consideration of implicit and equivalent attributes allows us to capture the entire informative content carried by a relation, regulating query execution ensuring that no subject can explicitly or implicitly access data for which it is not authorized. More precisely, a subject S is authorized to access a relation R iff the following three conditions hold:

1. S is authorized to access in plaintext all the (visible or implicit) attributes represented in plaintext in R;
2. S is authorized to access in plaintext or in encrypted form all the (visible or implicit) attributes represented in encrypted form in R;
3. S is authorized to access in the same form (either plaintext or encrypted) all the equivalent attributes (uniform visibility).

Conditions 1 and 2 correspond to a simple enforcement of authorizations, taking into account both the visible and implicit attributes. Also, condition 2 considers the fact that subjects authorized for plaintext visibility over an attribute can also have encrypted visibility over the same (since the encrypted representation conveys less information than the plaintext one). Condition 3 enforces control on indirect information leakage caused by equivalence relationships established in query computation, to prevent unauthorized exposure of information. It requires the subject to have the authorizations for the attributes in equivalence sets, since the relation implicitly carries information about them. In other words, since they leave a trace in the computation result, all attributes in equivalence sets are always treated as implicit attributes. It also imposes that, within each equivalence set, the authorizations be the same (either plaintext or encrypted) for all attributes in the set. In fact, equivalence relationships in a profile express the fact that some attributes have been related in a computation (e.g., an equi-join operation) and therefore visibility of one attribute in an equivalence set leaks information on the other attributes in the same set. Imposing uniform visibility allows us to account for such inference channels, blocking them when not consistent with the authorizations. Note that uniform visibility must be satisfied for all attributes in an equivalence set, regardless of whether they actually belong to the relational schema (i.e., they are visible).

The above discussion illustrates the conditions that make a subject authorized for a relation, based on authorizations and the informative content of the relation. The enforcement of authorizations in our context concerns however regulating the assignment to authorized subjects of operation execution within a query plan. An operation of the query plan, corresponding to a non-leaf node in the query plan `T(N)`, operates on one or two operand relations, and produces a relation as output. A subject can be considered authorized for the execution of an operation if and only if it is authorized

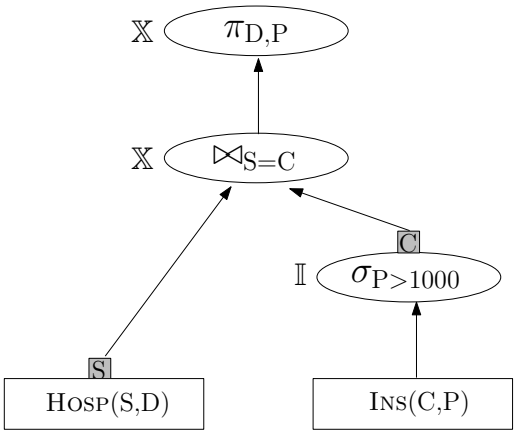


Figure 3.12: An extended query plan

for all the relations involved: the operand(s) as well as the result. The authorized visibility for the operand(s) is needed since otherwise the subject could not access them. The authorized visibility for the result enforces the control over the information entailed by the execution of the operation itself. For instance, with reference to the tree in Figure 3.11, only the user $\mathbb{U}$ is authorized for the execution of the join operation: $\mathbb{H}$ cannot access the right operand, $\mathbb{I}$ cannot access the left operand, $\mathbb{X}$ cannot access any of them. Also, $\mathbb{H}$ and $\mathbb{I}$ cannot access the result of the join execution as neither of them has uniform visibility over the join attributes S and C.

Given a query plan, our goal is to produce an authorized assignment of operations to subjects. While our discussion so far accounted for the possible presence of encrypted attributes, the original query plan, including only operations requested by the query computation, does not include any encryption/decryption operation (see for example the plan in Figure 3.11). Encryption and decryption operations are inserted on-the-fly by our approach to adjust visibility of attributes as required by operation requirements or authorizations. Encryption protects attributes so to permit the assignment of operations to subjects that could not otherwise be considered. Decryption permits accessing plaintext values of encrypted attributes when needed in the computation.

In the remainder of this section, we refer to a query plan $\mathtt{T}'$ that is obtained by inserting encryption and decryption operations into another query plan T as an *extended query plan* for T. As said, encrypting attributes enables the consideration, for the assignment of an operation, of subjects not otherwise authorized for the execution of the operation. Figure 3.12 illustrates the query plan in Figure 3.11 extended with two encryption operations, reporting on the left-hand side of each node a subject that could now be considered for the execution the node's operation. It is easy to see that, by encrypting attributes S and C, the join operation could now also be executed by subject $\mathbb{X}$, which is authorized to access both of them in encrypted form.

Clearly, the encryption needed to make assignments authorized eventually depends on the actual subjects to which operations are assigned. There are basically two opposite approaches that can be followed in the insertion of encryption/decryption operations in the query plan, corresponding to maximizing or minimizing visibility of attributes. Maximizing visibility corresponds to always leave visibility of data in the clear, applying encryption only when strictly needed for protecting attributes visibility from the subject executing a specific operation. Minimizing visibility corresponds to always apply encryption by default, decrypting attributes only as needed for operation execution. Each of the two extremes has some pros and cons and, to avoid predetermining one of the possible scenarios above, we adopt a more flexible approach by first determining the candidate subjects for

operations, and then injecting encryption and decryption operations only as needed, depending on the decided assignment of operations to subjects. The query optimizer can then decide assignments of operations based on cost and performance aspects.

Injecting on-the-fly Encryption

Assignment of operations to subjects must be bounded by the authorizations and the operation requirements, which can limit the application of encryption (as some operations need to access some attributes in plaintext for execution). To define the potential subjects that can be assigned an operation (i.e., the operation candidates), our approach then first needs to characterize the operation requirements that may limit the application of encryption. We capture this by defining the *minimal visibility* needed over an operand to allow the evaluation of an operator. Intuitively, the minimum required view over an operand for the execution of an operation is the operand relation where all the (visible) attributes, but those that need to be in plaintext for operation execution, are encrypted. Minimum required views allow us to take into account the visibility requirements for operation execution: only subjects authorized for the minimum required views can be *candidates for the assignment* (since for them the operand could be always protected with encryption without affecting operation execution).

Given a query plan and a possible assignment of operations taken from the potential candidates, there are different ways in which encryption and decryption could be inserted to make the assignment authorized. Since it is desirable to avoid the use of encryption if not needed for protection, a good strategy produces a plan that encrypts only those attributes that need to be encrypted for obeying authorizations (and later decrypts them if needed for the execution of an operation). We then aim at computing a *minimally extended authorized query plan*, that is, an extended plan where encryption and decryption operations are minimized. Query operation assignments can then be performed by the query optimizer, as follows:

1. perform a post-order visit of the query plan identifying candidates for each operation;
2. establish an assignment for each operation using classical approaches [Kos00];
3. perform a post-order visit of the query plan extending the plan with encryption and decryption operations.

The sub-queries assigned to each subject can then be dispatched, together with the required encryption keys (if the subject is required to perform encryption or decryption operations). It is easy to see that the plan in Figure 3.12 implies an authorized assignment of operations to subjects, and only encrypts attributes to obey authorizations (i.e., it encrypts attributes S and C to allow $\mathbb{X}$ to execute the join enforcing the authorizations).

3.5 Summary

The work in Task 3.3 focused on the support of collaborative queries, providing solutions for verifying the integrity of computations among collaborative parties as well as techniques for selective sharing among collaborative parties.
Task 3.3 proposed the combined adoption of twins and markers in the evaluation of join operations among relations hold by different parties, to provide a probabilisitic guarantees of the correctness and completeness of the results. The base techniques have then been refined to reduce the overhead they cause for integrity verification. Task 3.3 also analyzed the integrity guarantees provided by

markers and twins.
Task 3.3 developed a novel approach for supporting efficient revocation of authorizations. This approach has the advantage that it is sufficient to re-encrypt a small portion of the resource to make it unintelligible to non authorized users.
Task 3.3 finally proposed an approach that allows data owners to make their data selectively available for access and collaborative query execution, and enables users to execute queries over such data with selective and controlled involvement of external CSPs.

4. Security testing

The work under this task focused on providing security mechanisms that help guarantee the protection of customer data. The deployment of security and privacy mechanisms is essential in scenarios and applications involving the management of personal data. The changing context of threats and growing vulnerabilities forces to design techniques to verify the expected security and privacy levels. Hence, security testing becomes a necessity in the service developer community and also in the Cloud service provisioning community.

D3.2 compiles the advocated security testing techniques applicable to the ESCUDO-CLOUD environment. We have grouped and compared the testing techniques considering varied parameters that allows us to determine in what aspect of the service they should be applied. We have evaluated these techniques to analyze their applicability to the Use Cases (UC) identified in the WP1 of ESCUDO-CLOUD. As additional T3.4 activities, the open source testing frameworks of PAIN[1] and GRINDER[2] are developed. We have also extended the T3.4 testing approach to apply to the full Cloud security lifecyle via a novel schema that monitors SLA (Service Level Agreements) compliance levels. The ESCUDO-CLOUD UC relevant validation is shown on Amazon Web Service and OpenStack.

4.1 ESCUDO-CLOUD Innovation

This task produced several advancements over the state-of-the-art.

- We established the viability & also applicability of specific security testing approaches to the ESCUDO UC's. This was reported in D3.2.
- We developed a novel test acceleration technique (PAIN: Parallel Injections) that provides tunability of test parallelization over the tradeoff criteria of test acceleration and accuracy. The developed technique is generically applicable to multiple test generation/execution schemes taking test injections, resource availability and test interference/timeouts criteria as the basic inputs. PAIN is complemented by a generic test harness (GRINDER) that can provide a customizable testing interface over the changes of the target systems. The initial approach of PAIN and GRINDER was reported in D3.2.
- All classical testing approaches require a reference value to ascertain deviations over testing. This does not apply to emergent multi-threaded or concurrent software that increasingly figure in Cloud systems. Consequently, we developed the innovative concept of Invariant Propagation Analysis to efficiently handle the testing multiple execution traces. This work was published in [CWS+17]. The corresponding techniques for formal verification of concurrent software were published in [MSBS17, MSB+16].

[1] www.deeds.informatik.tu-darmstadt.de/deeds/research/tools/pain
[2] www.deeds.informatik.tu-darmstadt.de/deeds/research/tools/grinder

- While PAIN/GRINDER provide the UC owners with tools for supporting the direct security testing of code fragments, protocols, components or inter-component interfaces, the testing of Cloud system needs to be supported over the entire security lifecycle. Additionally, both security relevant functional and non-functional attributes in Cloud systems are typically specified by SLAs/SLOs[3] (as developed in T4.1). Our innovative approach has been to address the issue of security lifecycle testing with the use of compliance monitoring of the SLAs/SLOs along with the aspects of performance overhead of conformance testing/monitoring. The compliance approach to "system-level" testing via direct monitoring is detailed in Section 4.3.2 below and reported in [AWT+17]; the approach of indirect monitoring is reported in [ZLTS17].

4.2 Techniques and Approaches for Security Testing

The results of this task were reported in D3.2 at M21. The activites subsequent to M21 resulted in the additional publications of [AWT+17, ZLTS17, MSBS17, MSB+16, CWS+17].

4.2.1 Security Testing Techniques Overview

In D3.2, we discussed different security testing techniques as *White Box* and *Black Box* approaches. Next, we discussed the applicability of test parallelization to security testing based on experiences with test parallelization in dependability assessment. Finally, we presented a brief overview of widely used security testing guidelines and methodologies.

4.2.2 Testing Support Tools

The initial work was reported in D3.2. We have extensively enhanced both PAIN and GRINDER for functionality and also usability. Specifically, PAIN can now handle the aspects of (a) parallel experiment instances interference on resources, (b) ascertain result deviations due to timeout-based detectors, (c) assess predictive power of collected taskstats data for result validity, (d) provide prediction of achievable throughput (analytical model) and validation/tuning using experimental data, and (e) provide timeout and throughput tuning at runtime.

Both GRINDER's and PAIN's source codes are publicly available on github under the AGPL v3 license[4][5].

4.2.3 Testing Multi-Threaded Software and Systems

Bug injection is an established security testing technique to measure the robustness of a program to errors by introducing bugs/anomalies/errors into the program under test. Following an injection experiment, Error Propagation Analysis (EPA) is deployed to understand how errors affect a program's execution. EPA typically compares the traces of a fault-free (golden) run with those from a faulty run of the program. While this suffices for deterministic programs, EPA approaches are unsound for multi-threaded programs that have a non-deterministic golden run. Over ESCUDO-CLOUD we have proposed the use of automatically inferred likely invariants in lieu of golden traces for conducting EPA in multi-threaded programs. We present this approach as Invariant Propagation

[3] Service Level Objectives
[4] https://github.com/DEEDS-TUD/GRINDER
[5] https://github.com/DEEDS-TUD/PAIN

Analysis (IPA). We evaluate the stability and fault coverage of invariants derived by IPA across six different fault types across six representative programs through injection experiments. We find that stable invariants can be inferred in all six programs, although their coverage of faults depends on the application and the bug type. This work has been reported in [CWS+17] and the developed techniques for formal verification in [MSBS17, MSB+16].

4.2.4 Testing the Security Lifecycle: Compliance Monitoring

The usage of computing resources as a service makes Cloud computing an attractive solution for enterprises with fluctuating needs for information processing. However, the lack of security specifications and guarantees in most Cloud service offerings leaves customers uncertain about whether or not the provided service satisfies their security requirements. As an effort towards managing service security between Cloud Service Providers (CSPs) and customers, Security Service Level Agreements (secSLAs) were proposed as an extension to existing Cloud SLAs that usually comprise performance properties of services [LTTS15]. SecSLAs, in contrast, are used to specify the security properties of the provided service. The specification of a security property defines security mechanisms to achieve this property and their implementations.

Moreover, a secSLA includes SLOs, which are target values for different service levels of the specified security properties. Consequently, secSLAs can serve two purposes. First, according to the security level determined by specified values of the SLOs, customers can decide whether or not the required security level for the provided service is satisfied. Second, binding the properties with the SLOs in an agreement obligates the CSP to provide the service with the contracted SLO values.

Effectively, secSLAs are supposed to act as a contract between customers and CSPs. However, to date there is no[6] mechanism to enable customers to verify the CSPs' compliance to such contracts. The inability to validate if a contract is honored by both parties renders the utility of such contracts questionable. To develop assurance in the security of the provided service, customers needs to be able to validate the compliance of the CSP to the secSLA. While a number of approaches for service providers exist to assess the compliance of their services to the corresponding SLAs, there is virtually no support for customers to detect if the services they use comply to the specified security levels. To close this gap, we propose "C'MON: Monitoring the Compliance of Cloud Services to Contracted Properties", an approach to continuously monitor the compliance of Cloud services to SLAs as detailed in the next section. C'MON was tested on both Amazon EC2 and OpenStack systems relating to the ESCUDO-CLOUD UC's. Our evaluation of C'MON shows its ability to identify violations of contracted security properties in an IaaS setting with very low performance overheads.

4.3 C'MON: Monitoring the Compliance of Cloud Services to Contracted Properties

In this section, we propose a framework for secSLA compliance validation which enables Cloud customers to validate the compliance of the provided Cloud service to the contracted security level in a secSLA. The approach is realized by first defining means to evaluate the SLOs associated with each security property defined in a secSLA. By using these means to monitor the values of the SLOs, the compliance of the service to secSLA is validated against the secSLA throughout the life

[6]The sole exception being the framework initiated by TUD in EC FP7 SPECS www.specs-project.eu

cycle of the service and any detected violations are reported. This section provides the following contributions:

1. A survey of SLOs proposed by different security control frameworks and standards to be included in the secSLA. SLOs are surveyed to decide upon the possibility of measuring their value from the customer side.

2. Definition of measurement procedures for SLOs.

3. Design and implementation of a monitoring tool which monitors the values of the SLOs throughout the service life cycle, to ensure their compliance to the secSLA.

4. Evaluation of the functionality and performance of the approach on two examples of IaaS services using a self-hosted OpenStack lab setup and Amazon Elastic Compute Cloud (Amazon EC2) instances.

4.3.1 Terminology

SecSLAs have been proposed to enhance customers' assurance in the security of Cloud services [LTTS15]. The secSLA of a service enables customers to understand which security properties are implemented in the service and obliges the CSP to deliver the properties with the contracted values of the SLOs. However, a secSLA by itself does not guarantee that the contracted security level is actually met. The CSP might fail to satisfy the contracted security level at any time during the service life cycle. SecSLAs also specify the compensations to be paid in such cases. The aim of the proposed approach is to provide customers with means to detect these violations of secSLAs. This is achieved by measuring the SLOs contained in the secSLA and ensuring the compliance of the measured values to the contracted ones. Before discussing the details of the approach, we introduce the terminology and assumptions that our approach and its discussion are based on in the following.

- **Security Service Level Agreement (secSLA)**

  A secSLA is a documented agreement between the CSP and the customer, which describes the security properties of the service to be delivered [LTTS15]. Security control frameworks (e.g. National Institute of Standards and Technology (NIST) Special Publication (SP) 800-53 [Nat14] and Cloud Security Alliance (CSA)'s Cloud Control Matrix (CCM) [Clo16a]) describe security properties in secSLA as a hierarchy of security controls refined into SLOs. An example of a security control defined in CSA's CCM is *incident management*. This control is refined into *mean time between incidents* and *percentage of timely incident resolutions* SLOs, for which a CSP specifies target values and commits to meeting these values throughout the service life cycle.

- **Service Level Objective (SLO)**

  SLOs are the targets for service levels that the CSP specifies in the secSLA and commits to achieve. They represent the measurable elements of the SLA that allow assessing the security properties of the service. In the secSLA, the SLO is specified as an upper bound, a lower bound, or both. The value of the SLO should not exceed, fall below or lay outside the specified bounds, respectively. An example of an SLO is the *mean time between incidents*, which refers to the average time between the discovery of two consecutive incidents [EC 13]. The specified value for this SLO can be a lower bound which would imply that the reported average time between incidents should not fall below the defined lower bound at any time of the service life cycle.

- **Measurement**

  A measurement is a set of operations used to determine the value of an SLO. The measurement may be a formula, a process, a test or anything needed to assign a value to the SLO. In reference to the same example of the SLO *mean time between incidents*, a measurement for this SLO would involve means for incident detection and logging, in combination with means to extract incident detection time from the log and calculate the average time between two consecutive incidents.

4.3.2 The C'MON Cloud Monitoring Approach

In order to enable customer-side SLO measurements, we first analyze the secSLA to (a) understand which SLOs it comprises and (b) identify the quantifiable and measurable SLOs among that set of SLOs. For the evaluation C'MON of discussed in Section 4.3.6, we performed an extensive literature survey to identify SLOs that are commonly contained in secSLAs and investigated each SLO to decide whether or not it can be measured by the customer. The results of this survey are detailed in Section 4.3.3.

The next step, detailed in Section 4.3.4, derives measurement mechanisms for the identified SLOs that allow assessments from the customer side. Based on the identified measurement mechanisms, a framework for continuously monitoring the values of the SLOs is developed in Section 4.3.5.

The proposed approach focuses on the validation of the security of the IaaS Cloud service model. IaaS includes the infrastructure resource stack hardware platforms (server, raw storage, and networking resources) as well as connectivity to these resources [RRS11]. IaaS is the foundation on which "higher" service models, such as PaaS or SaaS, are built. Hence, building secure SaaS and PaaS services also mandates securing the infrastructure on which they are built. Accordingly, the SLOs contained in the proposed validation model are defined for IaaS services and also form the basis for ensuring the security of the other models.

4.3.3 SLO Classification

The aim of our approach is to develop a customer side compliance validation approach. Since the ability of the customer to validate the compliance of a CSP to a secSLA is restricted by his ability to measure the values of the SLOs contained in the secSLA, the first step in our approach is to identify the SLOs which can be measured by a customer. An analysis of SLOs proposed by security controls frameworks and security research projects is performed to investigate the SLOs and decide upon the feasibility of measuring their values. SLOs from five different security catalogs proposed by NIST [Nat08], CIS [NIS10], CUMULUS [EC 13], A4Cloud [NFG14] and SPECS [LB14] are surveyed. The surveyed SLOs are classified according to their measurability from the customer side into three categories:

- **Measurable:** The values of the SLOs belonging to this category can be measured from the customer side. The measurement can yield exact values which can be directly compared to the values provided by the CSP.

- **Can be approximated:** This category refers to the SLOs which cannot be exactly measured but an approximation of their values can be interpreted. An example for an SLO which belongs to this category is *mean time to incident recovery* proposed in [NIS10]. This SLO measures the

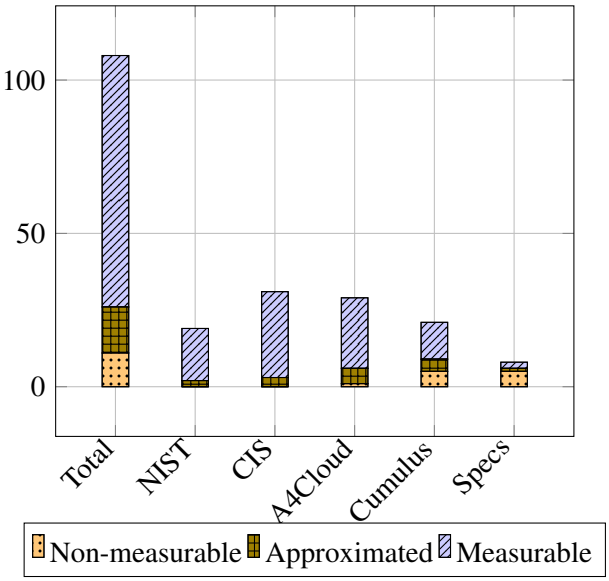


Figure 4.1: Quantitative SLO Classification

mean time taken to recover from a security incident. Incident detection solutions can be used at the customer side to detect incidents occurring at the customer's resources and track them until they are recovered. However, the value specified by the CSP is calculated across the full organization for all resources. Thus, the calculated value at the customer side is a approximation of the actual value given the assumption that the occurrence and recover time of incidents are similarly distributed across all resources.

- **Non-Measurable:** Non-measurable SLOs are those which cannot be measured or approximated from the customer side. In particular, these SLOs depend on information that is exclusive to the CSP. *Security Budget Measure* [Nat08] is an example of an SLO that belongs to this category. The SLO measures the percentage of information system budget devoted to information security of incidents. A customer has no means to measure a CSP's budgeting or costs for security incidents.

A total of 142 SLOs were examined for this work. During the examination of the SLOs, two types of SLOs have been identified, SLOs with quantitative values (i.e., values with numeric types and semantics of a quantity), and SLOs with qualitative values (i.e., nominal or ordinal values) [Int16]. With the aim of providing an automated compliance validation approach, only quantitative SLOs are considered. Out of the total number of SLOs, 82.5% were identified as quantitative. After extracting the quantitative SLOs and eliminating the underspecified SLOs (i.e. SLOs whose definitions are not clear enough to decide upon their measurability), 108 were classified as shown in Figure 4.1. The figure reports the number of SLOs in each category for each considered security catalog. Out of all quantitative SLOs, only 11 SLOs turned out to be measurable. Table 4.1 lists the measurable SLOs along with their description.

4.3.4 SLO Measurement Definition

In this step, we define measurements for the 11 SLOs classified as measurable in the previous step. A measurement definition describes the process used to determine the value of each SLOs from

Table 4.1: Measurable SLOs

Standard	**SLO**	**Description**
CUMULUS: 01	Percentage of uptime	The percentage of time the resource was considered available, in comparison with the total elapsed time.
CUMULUS: 02	Percentage of processed requests	The percentage of successful resource requests processed by the provider over the total number of submitted requests.
CUMULUS: 03	Percentage of timely recoveries	The ability to recover from an unavailability event within a maximum predefined delay.
CUMULUS: 04	Mean time between failures	The mean time between two consecutive failures to process a request.
CUMULUS: 20	Country-level anchoring	Indicates that the CSP guarantees that all storage, processing and remote access only takes place within a set of predefined countries.
SPECS: 02	Forward secrecy	Enables the use of forward secrecy on a cryptographic channel.
SPECS: 03	HSTS (HTTP Strict Transport Security)	Usage of HSTS protocol.
SPECS: 05	Secure cookies	Enables the use of secure cookies.
SPECS: 06	Client certificates	Enables the use of client certificates for SSL/TLS-based authentication.
SPECS: 07	OCSP (Online Certificate Status Protocol) stapling	Enables the use of OCSP for requesting the status of a digital certificate.
A4Cloud: 20	Readability (Flesch Reading Ease Test)	Level of readability of text produced by the CSP.

the customer side. For each SLO, the description provided in the catalog is studied to define an appropriate measurement to determine its value. The measurements are discussed below.

Percentage of Uptime

The SLO specifies a lower bound for the percentage of time over a predefined measurement period in which the service should be available. Based on the description of the SLO, the availability of a service is determined by the status of a request sent to the service at the time of measurement. If a service request response returns an error code or the CSP fails to deliver the service within a predefined time frame, the service is considered unavailable. The description of the SLO specifies additional attributes *timeslotFailThreshold* and *slotSize* to calculate the percentage of uptime. The measurement period is divided into timeslots of fixed length *slotSize*. A slot is considered available if the percentage of failed requests within a timeslot is less than defined percentage *timeslotFailThreshold*. Accordingly, the percentage of uptime is calculated as the percentage of time slots in which the service was considered available.

Measurement Test service requests are generated and sent to check if the service is available or not. To monitor the availability of the service throughout the full life cycle, requests are sent periodically to the service with a predefined frequency and the response of the CSP is verified. The status of the requests are used to calculate the percentage of uptime using Equation 4.1.

$$\frac{\sum Available\ slots}{\sum Slots} \tag{4.1}$$

Percentage of Timely Recoveries

The SLO specifies a lower bound for the percentage of unavailability events which last less than the predefined delay *maxTime* [EC 13]. The SLO has the same attributes *slotSize* and *timeslotFailThreshold* as the previous SLO. In addition, it follows the same definition of service availability defined over time slots of size *slotSize* to determine unavailability events and their duration.

Measurement The SLO is measured using the same method defined for measuring *percentage of uptime*. Service requests are sent at a predefined frequency to test the availability of the service throughout the full measurement period. Upon the detection of an unavailable slot, the start of an unavailability event is marked. The number of slots until the next available slot is recorded as the unavailability event duration. The percentage of the unavailability events with duration less than *maxTime* is then calculated using Equation 4.2.

$$\frac{\sum Unavailability\ events(T < maxTime)}{\sum Unavailability\ events} \tag{4.2}$$

Percentage of Processed Requests

A lower bound for the percentage of successfully processed requests by the CSP over a given measurement period is specified by this SLO. A request is considered successful, if the service was delivered without an error and within a predefined time frame. The SLO is reported over the *measurementPeriod* of the service.

Measurement To measure this SLO, service requests are generated at a predefined frequency and the percentage of successful requests is calculated over the measurement period using Equation 4.3.

$$\frac{\sum Successful\ requests}{\sum Requests} \tag{4.3}$$

Mean time between failure

This SLO is used to specify a lower bound for the mean time between two consecutive failures of processing a request [EC 13]. It is important to note that the failure here is defined as the failure of a single request. The mean time is calculated over the duration of the measurement period.

Measurement Test requests are continuously sent with a predefined frequency to the service. For each reported failure of processing a request, the time of the failure is logged. Accordingly, the time between every two consecutive failures is calculated. The mean time of all the reported times between consecutive failures throughout the measurement period is calculated using Equation 4.4 and reported as the value for this SLO.

$$\frac{\sum |Start\ of\ downtime - Start\ of\ next\ uptime|}{\sum Failed\ requests} \tag{4.4}$$

HSTS

This SLO refers to the usage HTTP Strict Transport Security (HSTS) [LB14]. Since this SLO is concerned with web session security, it can be defined for SaaS services only. This applies also for the next four SLOs in Table 4.1.

Measurement An HTTP host is declared as an HSTS host by issuing an HSTS policy, which is represented by and conveyed via the Strict-Transport-Security HTTP response header field [HJB12]. Therefore, to check for HSTS usage, an HTTP GET request is sent to the service, and the Strict-Transport-Security HTTP header field in the response is investigated. To validate the continuous usage of HSTS during the service life cycle, HTTP GET requests are repeatedly sent to the service, checking the header field on every request.

Secure Cookies Forced

The secure cookies SLO [LB14] reports whether the service enforces the usage of secure cookies or not. This SLO is used to serve sensitive data protection by protects the confidentiality of cookies in transmission.

Measurement By sending an HTTP GET request to the service and examining the Set-Cookie header in the responses, one can check if the secure attribute is set to true to validate the usage of secure cookies. The value of the secure attribute can be monitored continuously by sending the HTTP GET request regularly. The value of the flags of all the received responses are checked to validate whether secure cookies are enforced or not.

Forward Secrecy

The SLO reports whether the Cloud service supports forward secrecy in its cryptographic channels.

Measurement Connecting to a server which supports forward secrecy implies that the session key used during Secure Sockets Layer/Transport Layer Security (SSL/TLS) session establishment is independent from the server's private key [HC98]. Accordingly, achieving forward secrecy is reliant on the cipher suite chosen when initiating the SSL/TLS session since the cipher suite determines the key exchange algorithm used. The use of Diffie-Hellman key exchange supports this property. Hence, to check for the usage of forward secrecy, an SSL session is initiated with the server, including only the cipher suites which uses Diffie-Hellman key exchange in the client cipher suites. If the handshake was successful then the server supports the forward secrecy. Otherwise, if a handshake failure alert is raised and the session is terminated, then forward secrecy is not supported. To monitor the value of this SLO throughout the life cycle of the service, this process is carried out repeatedly throughout the service's life cycle.

Client Certificates

This SLO refers to whether the service enables the use of client certificate in SSL/TLS-based authentication. The SLO can take one of 3 values "required", "optional" or "not required". If the client certificate is "required" then the client can only be authenticated if it sends an valid certificate during the handshake. If the value is "optional" the client may or may not send its certificate during the handshake. If the server does not want to accept the client certificate even if it was presented, the value is set to "not required".

Measurement An SSL session is initiated with the server to check whether the server uses the client certificate or not. To test all three possibilities, a simple algorithm is proposed. A session is first initiated without a client certificate. If a handshake failure alert is raised [DR08], then the server needs to verify the client certificate for authentication (i.e., a certificate is "required"). Else, if the handshake did not fail then the client certificate may be "optional" or "not required". Next we initiate another session, this time with an invalid client certificate. If the session is successfully initiated, it can be deduced that the server does not check the client certificate even if presented (i.e., the certificate is "not required"). Otherwise, if a handshake failure alert is raised, then client certificate authentication is "optional" presenting the certificate is not essential for authentication but if a certificate is presented it is checked. As a result, the handshake failure alert is raised when trying to authenticate with an invalid certificate. This process is repeated periodically to continuously monitor the value of the SLO.

OCSP Stapling

Online Certificate Status Protocol (OCSP) stapling [GSM+99] reports if the use of OCSP for requesting the status of a digital certificate is enabled or not. A server that uses OCSP stapling sends the OCSP request on the behalf of the client staples the OCSP response to the handshake.

Measurement To check whether OCSP is supported or not, an SSL session is initiated with the server while including the Certificate Status Request extension in the "client hello" SSL handshake

message. If the Certificate Status response extension is attached with the SSL handshake, then the server supports OCSP stapling.

Country Level Anchoring

The country level anchoring SLO [EC 13] defines a set of countries within which all the storage, processing and remote access takes place.

Measurement The countries can be inferred from the IP addresses of the storage and access entities using IP to geolocation databases. Given the IP address of the provided Cloud service, a route trace is initiated to the IP address of the machine to check the IP addresses of the host and the gateway and infer the corresponding countries [7]. Moreover, the countries from which the service is accessed can be inferred from the source IP addresses.

Readability (Flesch Reading Ease Test)

This SLO specifies a lower bound for the level of readability of text produced by the CSP. The readability of a given text is quantitatively calculated based on the number of sentences, words and syllables contained in the text.

Measurement Equation 4.5 in [NFG14] is used to calculate readability, where S is the total number of sentences, W is the number of words and Y is the total number of syllables in the text. A minimum value of 45 is defined to regard a text as reasonably readable. The equation is used to calculate the readability of the secSLA written by the CSP to describe the service provided.

$$206.835 - 1.015\frac{W}{S} - 84.6\frac{Y}{W} \tag{4.5}$$

4.3.5 C'MON Monitoring Framework

We propose C'MON as a solution for customers to monitor the compliance of the service to the secSLA throughout its life cycle. This is achieved by tracking the values of the SLOs using the measurements defined in the previous subsection. By comparing the measured values to the contracted ones, the framework enables the detection of violations to the secSLA. The architecture of the framework is shown in Figure 4.2. The components of the framework and their interactions are discussed in the following.

Monitoring manager: The monitoring manager regulates the validation process. It takes the validation request from the customer as input, starts the monitoring and outputs the validation status.

SLOs tests repository: The measurements defined in 4.3.4 are formulated as tests to be executed during the monitoring to determine the real-time values of the SLOs. The tests are stored in this repository. The tests are loaded according to the SLOs present in the secSLA of the resource to be monitored.

Tester: The tester performs the tests loaded from the repository to measure the values of the SLOs of the delivered service. The testing of the service is performed remotely using the provided information about the service with the validation request.

[7] This measurement is limited by the ability of attaining the route to the service. Any intermediate firewalls between the machine performing the measurement and the target machine might obstruct the packets sent to determine the route.

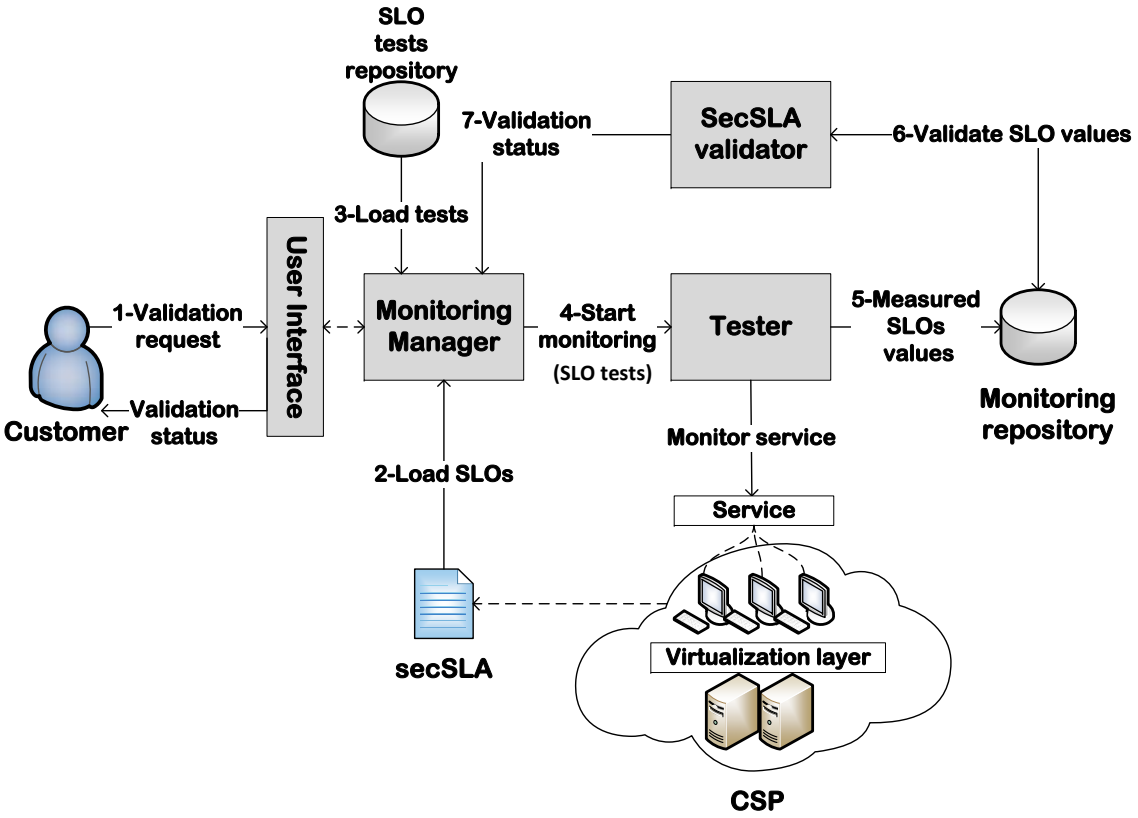


Figure 4.2: Monitoring Framework Architecture

Monitoring repository: The monitoring repository maintains the monitoring results collected by the tester throughout the measurement period.

SecSLA validator: The secSLA validator is responsible for reporting the compliance status of the service. The validator compares the measured value of each SLO to its corresponding contracted value to check for violations. If the measured values of all SLOs are equal to the contracted values or is within the contracted threshold then the service is compliant to the secSLA . Otherwise, the service is said to violate the secSLA.

The compliance validation with C'MON comprises the following steps.

1. The customer initiates a validation session by sending a validation request to the monitoring framework. The request includes some information (e.g., IP address or host-name) about the service to validate along with the secSLA.

2. The monitoring manager loads the contracted values of the SLOs from the secSLA.

3. The tests corresponding to the loaded SLOs are selected from the SLO tests repository to be used during the monitoring session.

4. The tester executes the tests with a predefined monitoring frequency to measure the values of the SLOs.

5. The tester stores the collected results of the tests in the monitoring repository.

6. After the validation session is completed, the measured values of the service's SLOs collected over the validation session are compared against the SLO values from the secSLA.

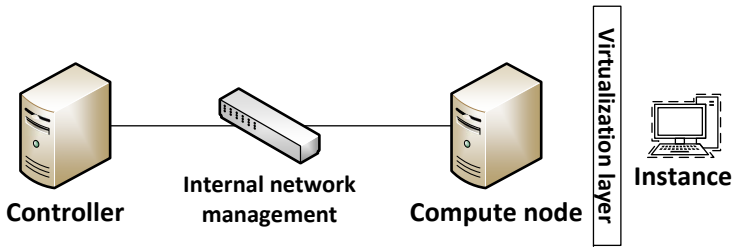


Figure 4.3: OpenStack Cloud Platform Architecture

7. The validation status is returned to the customer along with any detected violation.

The service compliance validation status can be reported periodically or upon customer's demand. The proposed monitoring framework enables the customer to adjust the monitoring configuration (e.g. monitoring frequency, monitoring duration, validation period) as he wishes to achieve the required monitoring coverage.

4.3.6 Evaluation

Three experiments have been conducted to evaluate the approach. The first two experiments are concerned with evaluating the functionality of the approach on two examples of IaaS services: OpenStack and Amazon EC2. The third experiment evaluates the overhead imposed by the proposed framework on both the monitoring server and the monitored Cloud service.

Experiment 1: OpenStack Compute Instance

In the first experiment, the effectiveness of C'MON for managing the compliance validation process has been evaluated by assessing C'MON's ability to measure the SLOs and detect violations of predefined values.The experiment was performed in a controlled environment using an IaaS service provided by a self-hosted OpenStack Cloud platform. Figure 4.3 shows the setup of the experiment. The IaaS Cloud platform is built using one controller node, which hosts all OpenStack services to manage compute resources, and a compute node that hosts the created instances. The platform is used to create a compute instance, which is used as a target for monitoring the SLOs *percentage of uptime, percentage of timely recoveries, meantime between failures* and *percentage of processed requests* by monitoring the availability of the instance using ping messages. In addition, OpenStack's dashboard *Horizon* provides a web based interface to manage the instance. The SLOs *Secure cookie forced, HSTS, client certificate, forward secrecy* and *OCSP* are monitored for the dashboard to evaluate the session security properties of the management interface. C'MON was run on a server outside the Cloud platform. The IP address of the instance and the host name of the dashboard were used to access the corresponding resources.

The experiment was performed by setting up a running instance with the management interface and enabling all the session security properties by editing the configurations of the dashboard. Violations of SLO values were deliberately caused at different times during the experiment to test C'MON's ability to detect them. Two types of violations were used: an instance availability outage, in which the instance was forced into an unavailable state, and a session security violation, in which support for the session security properties was detained (i.e., the usage of secure cookies is disabled after it was originally enabled).

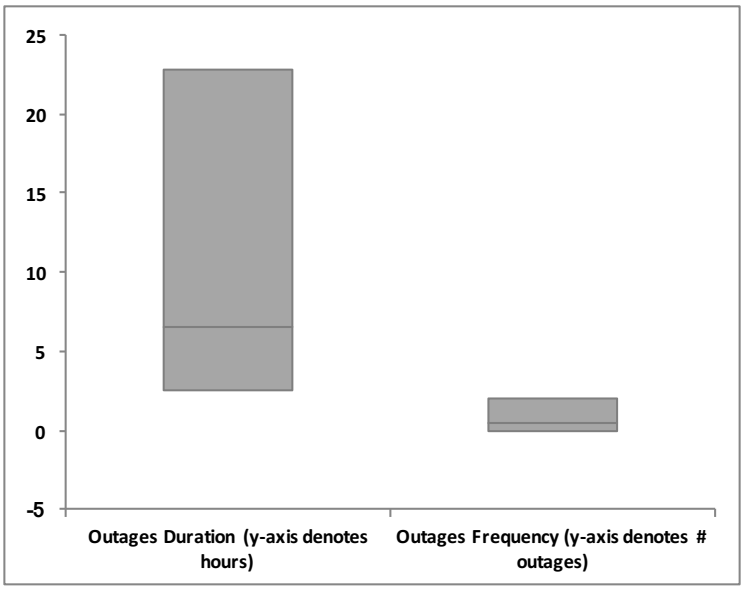


Figure 4.4: Outages Distribution per Month

We took great care to simulate *realistic* outages in our experiments and conducted a study to derive realistic outage distributions, covering both the frequency and the duration of the outages. We surveyed a number of data sets ([Ama17b, Clo16b]), but only found the data provided by the International Working Group on Cloud Computing Resiliency (IWGCR) [IWG17] to be sufficiently concise.

The box plot in Figure 4.4 shows the distribution of outage durations (in hours) from the collected data and outage frequencies per hour on the same scale, respectively. As the usable data set only comprised a total of 47 outage reports, a triangular distribution for the duration of outages has been fitted to the data, using the 25% quartile as the lower bound, the 75% quartile as the upper bound, and the median value as the mode. According to our analysis of the data, the mean outage frequency is less than one outage per month. However, relying on this low frequency would result in too few data points for the experiment. Hence, we used a uniformly distributed random variable to determine the occurrence of outages, while avoiding outage overlaps by setting its lower bound to a value greater than the 75% quartile of the outage duration. The upper bound of the random variable's range was set to 72 hours to ensure a sufficient number of samples for the measurement period. The obtained probability distributions of outage durations and frequencies were used to schedule the outages over the time of the experiment.

As our analysis of real service outages shows that outages of less than 2.5 hours are a relatively rare occurrence (less than 25% of the reported outages), we use three tests per hour as a conservative configuration for the monitoring frequency. We ran the experiments for one month. Over the entire experiment duration C'MON accurately detected the artificially injected security property violations without any false negatives or false positives.

Experiment 2: Amazon EC2 Instances

In the second experiment, we evaluated our approach on a commercial Cloud service setup using AWS EC2 instances. Three EC2 instances each in a different region, along with their web based management interfaces (Amazon management console) were monitored. The configurations for the instances are shown in table 4.2. C'MON was deployed on a server outside the Cloud network. The

SLOs monitored for the instances and the management interfaces were the same as the previous experiment. In addition, the *country-level anchoring* SLO was monitored for the instances.

Table 4.2: Instances Configurations

	US_EC2	**FRA_EC2**	**TYK_EC2**
Region	US East (N. Virginia)	EU (Frankfurt)	Asia Pacific (Tokyo)
Instance type	t2.nano		
Amazon Machine Image	Amazon Linux AMI 2016.09.0 (HVM) [Ama17a]		
Network	Default		
Availability zone	us-east-1d	eu-central-1b	ap-northeast-1a
Tenancy	Shared		

C'MON Configurations With the aim to detect the shortest possible change and thereby achieve maximal coverage of changes in the monitored SLO values, a fine-grained monitoring was used. For availability, the shortest possible change was assumed to be the reboot time of an instance during which the instance is unavailable. The reboot time of an instance has been experimentally assessed to be 5 seconds. Hence, the monitoring frequency is set to one test every five seconds. For the session security properties of the management subsystem, a lower monitoring frequency was used, since changes in the configuration of the web server hosting the console are expected to occur less frequently. Moreover, automated access to the console at high rates might be considered malicious and thus the requests might get blocked by the CSP. Accordingly, the monitoring frequency for the consoles was configured as six times per hour.

We used the following values for the additional attributes defined for the SLOs to calculate the values of the SLOs in this experiment. The *slotSize* was set to 15 minutes with a *timeslotFailThreshold* equal to 10%. The *maxTime* used to calculate the *percentage of timely recoveries* was one slot, i.e., 15 minutes.

The experiment was also run for a month. Table 4.3 shows the measured values of the SLOs for all three instances. According to the results, the instance deployed in the US East (New Virginia) offered the least *percentage of uptime* with a percentage still greater than 99%. The lowest *percentage of processed requests* is reported for the instance located in Asia Pacific (Tokyo). By comparing the results of the *percentage of uptime* and *percentage of processed requests* for TKY_EC2 and US_EC2, it can be noticed a lower *percentage of processed requests* does not necessarily imply a lower *percentage of uptime*. This is due to the fact that the *percentage of uptime* is calculated over the availability of the slots rather than individual requests. The instance deployed in the EU (Frankfurt) had the highest *mean time to failure* with a value equal to 17 hours. All instances achieved 100% timely recoveries from outages for a *maxTime* of one slot duration, i.e., no outage lasted for more than 15 minutes for any instance.

The location of the instances did not change during the experiment period. The locations reported by the monitoring framework for each instance are displayed in the table. We analyzed the collected data investigate the frequencies and durations of the outages during the measurement period. According to the recorded data, the shortest observed outage is the failure of a single request, i.e., an outage duration of less than 10 seconds. The longest outage duration extracted from the results of all instances is 255 seconds experienced in the US_EC2 instance.

The reported results for the management interfaces are almost identical for all three consoles. All session security SLOs were enabled and the client certificate was "Not required" for all consoles. The FRA_Console apparently did not support OCSP stapling. The assessment of these SLOs did not change over the measurement period. Therefore, no conclusions can be drawn from our experiment regarding the change frequency or duration for session security SLOs in real world settings.

Table 4.3: Amazon EC2 Instances Results

	US_EC2	**FRA_EC2**	**TKY_EC2**
Percentage of Uptime	99.0915%	100%	99.9303%
Percentage of Processed Requests	99.8088%	99.9930%	99.7571%
Percentage of Timely Recoveries	100%	100%	100%
Mean Time to Failure	36.41 minutes	1023.63 minutes	35.91 minutes
Country-Level Anchoring	United States (Gateway: United States, Seattle Host: United States, Ashburn)	Germany (Gateway: Germany, Frankfurt Host: Germany, Frankfurt)	Japan (Gateway: Japan,Tokyo Host: Japan,Tokyo)

Experiment 3: Monitoring overhead

In the third experiment, the performance of the proposed approach has been evaluated with the goal to estimate the communication and computational overhead C'MON imposes. This experiment was conducted on the same setup of the first experiment (Figure 4.3), as it provides higher controllability of the factors that potentially influence our performance measurements. The imposed overhead was evaluated for the monitoring server, the OpenStack instance, and the server hosting the dashboard. To estimate the highest overhead imposed by the monitoring framework, the highest monitoring frequency from our prior experiments was used, i.e., one test every five seconds.

The monitoring overhead was assessed by measuring communication and computation performance metrics on the monitored and monitoring machines with and without C'MON being active. The comparison between performance with and without C'MON is conducted using a two sample Kolmogorov-Smirnov (K-S) test [MF51]. If the null hypothesis (that the two sample sets of performance measurements belong to the same population) cannot be rejected, then there was no significant performance overhead imposed by C'MON. Otherwise, an overhead was observed and we report the average difference between the values as the imposed overhead. The conducted performance tests are detailed below.

Communication Overhead. The metric considered for measuring the communication overhead is *network throughput*. Network throughput was measured using the netperf [Jon17] benchmark. The server under measurement executed a netperf client and the *TCP_STREAM* benchmark profile was used to transfer data and report the throughput of the uplink during the transmission. Three tests were performed, one for the monitoring server and one for each target instance. Each test was performed once while monitoring was active and once while it was inactive. For every test for both cases (monitoring in/active) the test has been performed 30 times yielding 30 throughput samples per test.

The K-S tests are performed on the collected samples from each test using a statistical significance level of $\alpha = 0.05$. The results are shown in Table 4.4. For all three cases the p-value is greater than α. Thus, the null hypothesis cannot be rejected, which shows that there is no evidence of statistically significant communication overhead using C'MON.

Table 4.4: Network Throughput K-S Tests Results

	Monitoring Server	**Instance**	**Dashboard**
p-value	0.236	0.954	0.132
D	0.267	0.333	0.300

Computational Overhead To measure the performance of computational workloads, two metrics were chosen: *CPU total time* and *file Input/Output (I/O) throughput*. We used Sysbench [Kop17] to measure the chosen metrics. The CPU total time was measured for the monitoring server, the instance, and the machine running the dashboard, whereas file I/O throughput was only measured for the monitoring server and the dashboard, as the ping-based SLO measurements do not impose file I/O overheads on the IaaS instances. Each test has been performed 30 times for each machine and monitoring being active or inactive. K-S tests are used to compare the samples collected for each case using $\alpha = 0.05$ as for the communication overhead assessment before.

The results of the K-S tests are listed in Table 4.5. According to the p-values of the CPU total time, tests for both the monitoring server and the server hosting the dashboard, the null hypothesis is rejected. This implies an imposed overhead on the server caused by monitoring. Figures 4.5 and 4.6 show the cumulative frequency distributions of the CPU time samples collected for the monitoring server and the dashboard server respectively. We estimate the imposed overhead as the average difference between the measurements with and without monitoring, which is 1.28 seconds, i.e., a 0.0076% increase for the monitoring server. For the dashboard server, the monitoring overhead is 0.072 seconds, i.e., a 0.0003% increase. For the CPU time test performed on the instance, the null hypothesis of the K-S tests could not be rejected. Hence, there is no evidence of any statistically significant overhead.

Table 4.5: Computational Performance K-S Tests Results

	CPU Total Time			**I/O Throughput**	
	Monitoring Server	**Instance**	**Dashboard**	**Monitoring Server**	**Dashboard**
p-value	< 0.0001	0.007	0.393	0.219	< 0.0001
D	0.633	0.433	0.233	0.267	0.567

The results of the file I/O throughput test performed for the monitoring server also yields no evidence of significant overhead. However, an overhead is observed on the dashboard for the same test. Figure 4.7 shows the cumulative frequency distributions of the throughput for both cases (monitoring active/inactive). The average difference between the distributions is 0.045 Mb/sec, i.e., a 0.03% decrease resulting from monitoring.

Threats to Validity

The first threat to validity is the assumption made about the minimum outage duration. The way an instance is managed (launched, scheduled or booted) differs from one CSP to another, hence,

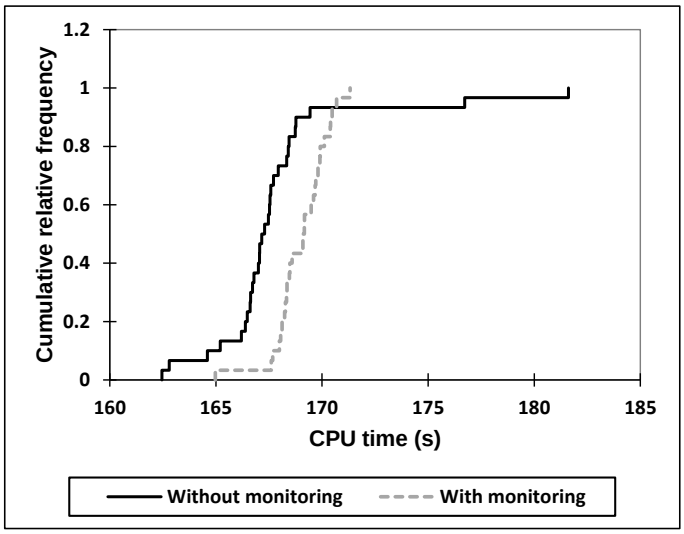


Figure 4.5: Monitoring Server CPU Total Time

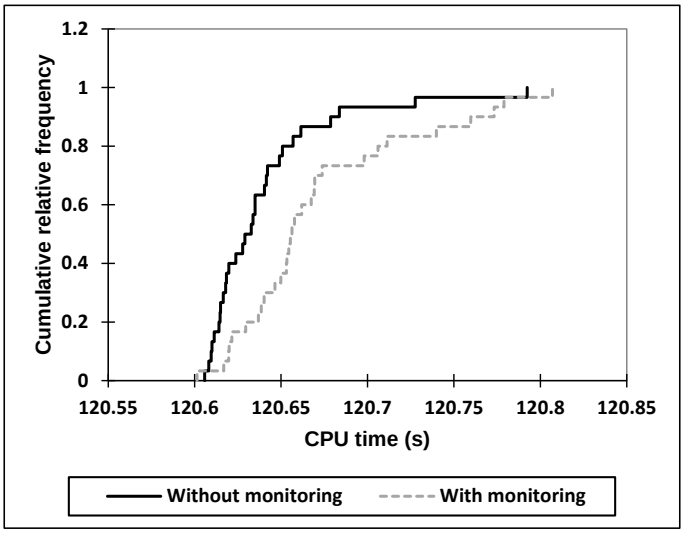


Figure 4.6: Dashboard CPU Total Time

the expected outage duration of the services differ. The absence of false negatives observed in our experiments depends on whether the violation duration is longer than the time interval between two consecutive monitoring events. Hence, the coverage of violations is dependent on the monitoring frequency. High monitoring frequencies yield high coverage, but also entail high overheads. In our evaluation we conservatively chose high monitoring frequencies and still obtained modest performance overheads for C'MON. We are therefore confident that C'MON provides reasonable performance in any realistic use case.

The second threat to validity is caused by assuming that any request failure is caused by an availability of the service at the CSP side. Although, the Cumulus catalog [EC 13] defines the availability of the service based on the status of requests sent to it, there is a chance that the failure of the request is caused due to the failure of the path taken to reach the service, not the failure of service itself. On the one hand, this means that CSPs are not necessarily responsible for secSLA violations observed by C'MON, as these may also be caused by perturbations in the network infrastructure. On the other hand, this still accurately reflects the service level as perceived

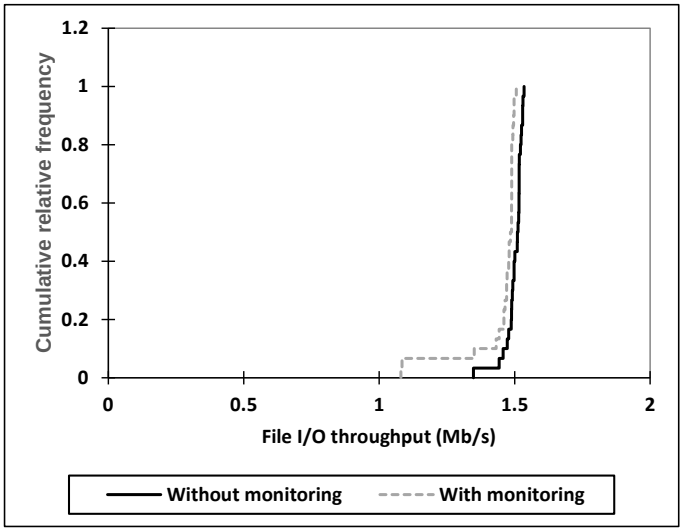


Figure 4.7: Dashboard File I/O Throughput

by the user. Therefore, the threat to validity only affects conclusions regarding CSPs' provided service levels. The service level observed by the user is accurately reflected by C'MON's monitoring results.

A third threat to validity concerns the measurement defined to check for the enforcement of forward secrecy, the measurement considers variants of Diffe-Hellman key exchange as the only algorithms which support forward secrecy and the enforcement is decided according. The existence of other algorithms might threaten the validity of the "not enforced" result. Finally, a fourth threat to validity is the choice of CSPs. The approach was only evaluated on an OpenStack and on Amazon EC2. The validation of services, which are differently set up and managed, might yield different results.

4.4 Summary

The work of D3.2 was reported at M21 and resulted in the publications [WSN+15, WPS+15, LNW+14]. The work subsequent to M21 resulted in the additional publications of [AWT+17, ZLTS17, MSBS17, MSB+16, CWS+17]. Overall we were able to achieve the following:

- Address the problem of the applicability of security testing techniques according to the availability of Cloud resources.
- Evaluate and map the range of available security testing techniques to the UC's requirements to advocate the most appropriate testing technique to use in each UC.
- Propose a customer side monitoring framework for monitoring the compliance of Cloud services to the contracted properties in secSLAs.
- Evaluate the proposed approach on both a self-hosted OpenStack platform and Amazon EC2. The results have proven that our approach is suitable for measuring the values of the SLOs and identifying violations of contracted SLO values.

5. Conclusion

In this document, we presented the activities and results of the work carried out in ESCUDO-CLOUD Work Package 3 between M4 and M34. WP3 addressed the problem of enabling the data owner with the ability to selectively share her protected data with other users in the Cloud. ESCUDO-CLOUD partners in WP3 approached this from four directions: empowering a data owner for selective sharing (T3.1) of protected data with other users, ensuring secure multi-user interactions and sharing (T3.2), guaranteeing integrity and correctness of the data and responses in presence of write operations by different users, and supporting collaborative queries (T3.3) involving access to data of different owners. Furthermore, security testing mechanisms and guidelines (T3.4) for the developed solutions were presented. These directions yielded the following results within ESCUDO-CLOUD.

Selective sharing. Task 3.1 had the goal to empower Cloud users with security means for data sharing. To achieve this, partners in T3.1 worked on selective encryption, the shuffle index, oblivious order preserving encryption, and search over encrypted data. The shuffle index in combination with selective encryption enables data owners to ensure confidentiality and integrity of data. Furthermore, a multi-user encryption schema for search over encrypted data was formulated for scenarios where encrypted data is required to be processed at a honest-but-curious CSP. Both means enable to enforce access restrictions at the provider-side, without requiring trust in the CSP. In addition, Oblivious Order Preserving Encryption represents a novel approach for increasing self-protection (i.e., privacy) in multi-party evaluation of decision trees over encrypted data.

Secure multi-user interactions and sharing. The goal of Task 3.2 was the development of solutions for multi-client interactions in the Cloud setting, with a focus on guaranteed integrity. We developed two solutions, with varying scopes. The first solution, VICOS, guarantees multi-client consistent access to a Cloud object store, which is accessed as a key-value store. VICOS has been implemented, can be used as an additional layer on commodity Cloud object store systems, and its performance is practical for real-world deployment. The second solution guarantees multi-user consistent access to an arbitrary data type, i.e., arbitrary stateful computation can be outsourced to an untrusted Cloud provider. While this solution allows for more general applications than VICOS, it uses computationally more expensive cryptographic tools and has not yet been implemented. We expect, however, that improvements in the active research areas related to the cryptographic components used in the protocol, together with efficiency improvements in our protocol, can make the protocol practical in the near future.

Support for collaborative queries. The task first studied twins and markers as integrity verification techniques to be used in collaborative computation scenarios, proposing refinements

for reducing verification costs. The task also analyzed the effectiveness of twins and markers to help the user in choosing how many markers and twins to use to obtain the desired integrity guarantees. The second problem addressed in this task is related to access control enforcement in collaborative scenarios. The task developed a novel solution, based on mixing encryption mode studied in WP 2, that enables the efficient enforcement of a revoke operation through the re-encryption of a small portion of the revoked resource, as well as a novel approach for specifying and enforcing authorizations to enable controlled data sharing for collaborative query execution in the Cloud.

Security testing. The task established the utility of varied security testing schema to the ESCUDO-CLOUD UCs. In addition, it developed two publicly available tools as (a) a generic testing harness, GRINDER, customizable for the test target, and (b) a testing framework, PAIN, to tune the tradeoffs across test acceleration, test correctness and test timings/allocation on distributed resources. A novel approach to handle testing for multi-threaded software was developed. Additionally, an extension from component testing to security lifecycle testing was developed as a secSLA compliance monitoring schema along with the validation on Amazon AWS and Openstack.

Bibliography

[ABL+04] Mikhail J. Atallah, Marina Bykova, Jiangtao Li, Keith B. Frikken, and Mercan Topkara. Private collaborative forecasting and benchmarking. In *Proceedings of the 2004 ACM Workshop on Privacy in the Electronic Society, WPES 2004, Washington, DC, USA, October 28, 2004*, pages 103–114, 2004.

[AEDS03] Mikhail J. Atallah, Hicham G. Elmongui, Vinayak Deshpande, and Leroy B. Schwarz. Secure supply-chain protocols. In *2003 IEEE International Conference on Electronic Commerce (CEC 2003), 24-27 June 2003, Newport Beach, CA, USA*, pages 293–302, 2003.

[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.

[AKSX04] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. Order preserving encryption for numeric data. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 563–574, New York, NY, USA, 2004. ACM.

[Ama17a] Amazon. Amazon web services. `http://aws.amazon.com/ec2/`, 2017.

[Ama17b] Amazon Web Services, Inc. Summary of the Amazon DynamoDB Service Disruption and Related Impacts in the US-East Region. `https://aws.amazon.com/message/5467D2/`, 2017.

[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.

[AWT+17] S. Alboghdady, S. Winter, A. Taha, H. Zhang, and N. Suri. C'mon: Monitoring the compliance of cloud services to contracted properties. In *International Conference on Availability, Reliability and Security*, page 36, 2017.

[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.

[BCK17] M. Brandenburger, C. Cachin, and N. Knežević. Don't trust the cloud, verify: Integrity and consistency for cloud object stores. *ACM Transactions on Privacy and Security (TOPS)*, 20(3), 2017.

[BCLO09] Alexandra Boldyreva, Nathan Chenette, Younho Lee, and Adam O'Neill. Order-preserving symmetric encryption. In *Proceedings of the 28th Annual International Conference on Advances in Cryptology: The Theory and Applications of Cryptographic Techniques*, EUROCRYPT '09, pages 224–241, Berlin, Heidelberg, 2009. Springer-Verlag.

[BCO11] Alexandra Boldyreva, Nathan Chenette, and Adam O'Neill. Order-preserving encryption revisited: Improved security analysis and alternative solutions. In *Proceedings of the 31st Annual Conference on Advances in Cryptology*, CRYPTO'11, pages 578–595, Berlin, Heidelberg, 2011. Springer-Verlag.

[BDF+16] E. Bacis, S. De Capitani di Vimercati, S. Foresti, S. Paraboschi, M. Rosa, and P. Samarati. Mix&Slice: Efficient access revocation in the cloud. In *Proc. of the 23rd ACM Conference on Computer and Communication Security (CCS 2016)*, Vienna, Austria, October 2016.

[BR93] Mihir Bellare and Phillip Rogaway. Entity authentication and key distribution. In Douglas R. Stinson, editor, *CRYPTO*, volume 773 of *LNCS*, pages 232–249. Springer, 1993.

[BR06] Mihir Bellare and Phillip Rogaway. Code-based game-playing proofs and the security of triple encryption. In Serge Vaudenay, editor, *EUROCRYPT*, volume 4004 of *LNCS*. Springer, 2006.

[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.

[CDN15] Ronald Cramer, Ivan Damgård, and Jesper Buus Nielsen. *Secure Multiparty Computation and Secret Sharing*. Cambridge University Press, New York, NY, USA, 2015.

[CGPT17] Christian Cachin, Esha Ghosh, Dimitrios Papadopoulos, and Björn Tackmann. Stateful multi-client verifiable computation. Cryptology ePrint Archive, Report 2017/901, September 2017.

[CK08] Octavian Catrina and Florian Kerschbaum. Fostering the uptake of secure multiparty computation in e-commerce. In *Proceedings of the The Third International Conference on Availability, Reliability and Security, ARES 2008, March 4-7, 2008, Technical University of Catalonia, Barcelona , Spain*, pages 693–700, 2008.

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

[Clo16a] Cloud Security Alliance. Cloud Controls Matrix v3 . `https://cloudsecurityalliance.org/research/ccm/`, 2016.

[Clo16b] CloudHarmony. CloudSquare Service Status. `https://cloudharmony.com/status`, 2016.

[CO14] Christian Cachin and Olga Ohrimenko. Verifying the consistency of remote untrusted services with commutative operations. In *OPODIS*, volume 8878 of *LNCS*. Springer, 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.

[CWS+17] A. Chan, S. Winter, H. Saissi, K. Pattabiraman, and N. Suri. IPA: error propagation analysis of multi-threaded programs using likely invariants. In *Proc. of International Conference on Software Testing, Verification and Validation*, pages 184–195, 2017.

[DA01] Wenliang Du and Mikhail J. Atallah. Privacy-preserving cooperative scientific computations. In *Proceedings of the 14th IEEE Workshop on Computer Security Foundations*, CSFW '01, pages 273–, Washington, DC, USA, 2001. IEEE Computer Society.

[DDC16] Betül Durak, Thomas DuBuisson, and David Cash. What else is revealed by order-revealing encryption? Technical Report 786, IACR Cryptology ePrint Archive, 2016.

[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+13] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati. Integrity for join queries in the cloud. *IEEE Transactions on Cloud Computing (TCC)*, 1(2):187–200, July-December 2013.

[DFJ+15] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati. Integrity for approximate joins on untrusted computational servers. In *Proc. of the 30th International Information Security and Privacy Conference (SEC 2015)*, Hamburg, Germany, May 2015.

[DFJ+16] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati. Efficient integrity checks for join queries in the cloud. *Journal of Computer Security (JCS)*, 24(3):347–378, 2016.

[DFJ+18] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, G. Livraga, S. Paraboschi, and P. Samarati. An authorization model for multi-provider queries. *Proc. of the VLDB Endowment (PVLDB)*, 2018. to appear.

[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 Transactions on Storage (TOS)*, 11(4):19:1–19:54, November 2015.

[DFP+16] S. De Capitani di Vimercati, S. Foresti, S. Paraboschi, G. Pelosi, and P. Samarati. Access control for the shuffle index. In *Proc. of the 30th Annual IFIP WG 11.3 Working Conference on Data and Applications Security and Privacy (DBSec 2016)*, Trento, Italy, July 2016.

[DFP+17] S. De Capitani di Vimercati, S. Foresti, S. Paraboschi, G. Pelosi, and P. Samarati. Enforcing authorizations while protecting access confidentiality. *Journal of Computer Security (JCS)*, 2017. to appear.

[DJ01] Ivan Damgård and Mats Jurik. A generalisation, a simplification and some applications of paillier's probabilistic public-key system. In *Proceedings of the 4th International Workshop on Practice and Theory in Public Key Cryptography: Public Key Cryptography*, PKC '01, pages 119–136, London, UK, UK, 2001. Springer-Verlag.

[DR08] T. Dierks and E. Rescorla. The transport layer security (tls) protocol version 1.2. https://www.ietf.org/rfc/rfc5246.txt, 2008.

[DT08] Ivan Damgård and Rune Thorbek. Efficient conversion of secret-shared values between different fields. *IACR Cryptology ePrint Archive*, 2008:221, 2008.

[EC 13] EC FP7 CUMULUS Consortium. D2.1 Security-Aware SLA Specification Language and Cloud Security Dependency model. `http://www.cumulus-project.eu/index.php/public-deliverables`, 2013.

[EFLL12] Yael Ejgenberg, Moriya Farbstein, Meital Levy, and Yehuda Lindell. SCAPI: the secure computation application programming interface. *IACR Cryptology ePrint Archive*, 2012:629, 2012.

[FFG+16] Dario Fiore, Cédric Fournet, Esha Ghosh, Markulf Kohlweiss, Olga Ohrimenko, and Bryan Parno. Hash first, argue later: Adaptive verifiable computations on outsourced data. In *ACM CCS*, pages 1304–1316. ACM, 2016.

[FKK06] K. Fu, S. Kamara, and Y. Kohno. Key regression: Enabling efficient key distribution for secure distributed storage. In *Proc. of NDSS*, San Diego, CA, USA, February 2006.

[Fri10] Keith B. Frikken. Secure multiparty computation. In *Algorithms and Theory of Computation Handbook*, pages 14–14. Chapman & Hall/CRC, 2010.

[G+14] P. Grofig et al. Experiences and observations on the industrial implementation of a system to search over outsourced encrypted data. In *Proc. of Sicherheit 2014*, Vienna, Austria, March 2014.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing*, STOC '09, pages 169–178, New York, NY, USA, 2009. ACM.

[GGOT16] Esha Ghosh, Michael T. Goodrich, Olga Ohrimenko, and Roberto Tamassia. Verifiable zero-knowledge order queries and updates for fully dynamic lists and trees. In *SCN*, volume 9841 of *LNCS*, pages 216–236, 2016.

[Gol04] Oded Goldreich. *Foundations of Cryptography: Volume 2, Basic Applications*. Cambridge University Press, New York, NY, USA, 2004.

[GSB+16] Paul Grubbs, Kevin Sekniqi, Vincent Bindschaedler, Muhammad Naveed, and Thomas Ristenpart. Leakage-abuse attacks against order-revealing encryption. Technical Report 895, IACR Cryptology ePrint Archive, 2016.

[GSM+99] S. Galperin, S. Santesson, M. Myers, A. Malpani, and C. Adams. X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP . https://www.ietf.org/rfc/rfc2560.txt, 1999.

[HC98] D. Harkins and D. Carrel. The Internet Key Exchange (IKE). `https://tools.ietf.org/html/rfc2409\#section-3.3`, 1998.

[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.

[HJB12] J. Hodges, C. Jackson, and A. Barth. Http strict transport security (hsts). `https://tools.ietf.org/html/rfc6797`, 2012.

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

[Int16] International Organization for Standardization ISO. Information Technology - Cloud Computing - Service Level Agreement (SLA) Framework and Terminology. `https://www.iso.org/standard/67545.html`, 2016.

[IWG17] IWGCR. International working group on cloud computing resiliency. `http://www.iwgcr.org/`, 2017.

[Jon17] R. Jones. Netperf Homepage. `http://www.netperf.org/netperf/`, 2017.

[Ker12] Florian Kerschbaum. Privacy-preserving computation - (position paper). In *Privacy Technologies and Policy - First Annual Privacy Forum, APF 2012, Limassol, Cyprus, October 10-11, 2012, Revised Selected Papers*, pages 41–54, 2012.

[Ker15] Florian Kerschbaum. Frequency-hiding order-preserving encryption. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 656–667, New York, NY, USA, 2015. ACM.

[KKBF17] A. Kemgne Tueno, F. Kerschbaum, D. Bernau, and S. Foresti. Selective access for supply chain management in the cloud. In *Proc. of the 2017 IEEE Workshop on Security and Privacy in the Cloud (SPC)*, October 2017.

[Kop17] A. Kopytov. Sysbench. `https://github.com/akopytov/sysbench`, 2017.

[Kos00] D. Kossmann. The state of the art in distributed query processing. *ACM CSUR*, 32(4):422–469, 2000.

[KS08] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free XOR gates and applications. In *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part II - Track B: Logic, Semantics, and Theory of Programming & Track C: Security and Cryptography Foundations*, pages 486–498, 2008.

[KS14] Florian Kerschbaum and Axel Schröpfer. Optimal average-complexity ideal-security order-preserving encryption. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 275–286, 2014.

[KSS09] Vladimir Kolesnikov, Ahmad-Reza Sadeghi, and Thomas Schneider. Improved garbled circuit building blocks and applications to auctions and computing minima. In *Cryptology and Network Security, 8th International Conference, CANS 2009, Kanazawa, Japan, December 12-14, 2009. Proceedings*, pages 1–20, 2009.

[LB14] J. Luna and M. Bregu. EC FP7 SPECS Project - Report on requirements for Cloud SLA negotiation. `http://www.specs-project.eu/publications/public-deliverables/d2-1-2/`, 2014.

[LNW+14] A. Lanzaro, R. Natella, S. Winter, D. Cotroneo, and N. Suri. An Empirical Study of Injected versus Actual Interface Errors. In *Proc. of the International Symposium on Software Testing and Analysis*, pages 397–408, 2014.

[LP09a] Yehuda Lindell and Benny Pinkas. A proof of security of yao's protocol for two-party computation. *J. Cryptol.*, 22(2):161–188, April 2009.

[LP09b] Yehuda Lindell and Benny Pinkas. Secure multiparty computation for privacy-preserving data mining. *The Journal of Privacy and Confidentiality*, 2009(1):59–98, 2009.

[LTTS15] J. Luna, A. Taha, R. Trapero, and N. Suri. Quantitative reasoning about cloud security using service level agreements. *IEEE Transactions on Cloud Computing*, 5(3):457–471, 2015.

[MCO+15] Charalampos Mavroforakis, Nathan Chenette, Adam O'Neill, George Kollios, and Ran Canetti. Modular order-preserving encryption, revisited. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 763–777, 2015.

[Mer89] Ralph C. Merkle. A certified digital signature. In *CRYPTO*, pages 218–238, 1989.

[MF51] J. Massey and J. Frank. The kolmogorov-smirnov test for goodness of fit. *Journal of the American Statistical Association*, 46(253):68–78, 1951.

[MS02] David Mazières and Dennis Shasha. Building secure file systems out of byzantine storage. In *ACM PODC*, pages 108–117. ACM, 2002.

[MSB+16] P. Metzler, H. Saissi, P. Bokor, R. Hesse, and N. Suri. Efficient Verification of Program Fragments: Eager POR. In *International Symposium on Automated Technology for Verification and Analysis*, pages 375–391, 2016.

[MSBS17] P. Metzler, H. Saissi, P. Bokor, and N. Suri. Quick verification of concurrent programs by iteratively relaxed scheduling. In *ACM Automated Software Enginering*, 2017.

[Nat08] National Institute of Standards and Technology NIST, Cloud Computing Reference Architecture and Taxonomy Working Group. Performance and Measurements Guide for Information Technology. NIST 800-55 Revision 1. `http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-55r1.pdf`, 2008.

[Nat14] National Institute of Standards and Technology NIST, Joint Taskforce Transformation Initiative. Security and Privacy Controls for Federal Information Systems and Organizations. *NIST 800-53v4*, 2014.

[NFG14] D. Nuñez and C. Fernandez-Gago. EC FP7 A4Cloud Project - Validation of the Accountability Metrics. `http://www.a4cloud.eu/sites/default/files/D35.2%20Validation%20of%20the%20accountability%20metrics.pdf`, 2014.

[NIS10] NIST. The cis security metrics v1.1.0. `www.nist.gov`, 2010.

[NKW15] Muhammad Naveed, Seny Kamara, and Charles V. Wright. Inference attacks on property-preserving encrypted databases. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 644–655, New York, NY, USA, 2015. ACM.

[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.

[Pai99] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Proceedings of the 17th International Conference on Theory and Application of Cryptographic Techniques*, EUROCRYPT'99, pages 223–238, Berlin, Heidelberg, 1999. Springer-Verlag.

[Pap11] Charalampos Papamanthou. *Cryptography for Efficiency: New Directions in Authenticated Data Structures*. PhD thesis, Brown University, 2011.

[PHGR13] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *2013 IEEE Symposium on Security and Privacy (SP)*, 2013.

[PLZ13] Raluca Ada Popa, Frank H. Li, and Nickolai Zeldovich. An ideal-security protocol for order-preserving encoding. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 463–477, Washington, DC, USA, 2013. IEEE Computer Society.

[PRZB11a] R.A. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. In *Proc. of SOSP 2011*, Cascais, Portugal, October 2011.

[PRZB11b] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. Cryptdb: Protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 85–100, New York, NY, USA, 2011. ACM.

[PSSW09] Benny Pinkas, Thomas Schneider, Nigel P. Smart, and Stephen C. Williams. Secure two-party computation is practical. *IACR Cryptology ePrint Archive*, 2009:314, 2009.

[PTT11] Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos. Optimal verification of operations on dynamic sets. In *CRYPTO*, pages 91–110, 2011.

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

[Riv97] R.L. Rivest. All-or-nothing encryption and the package transform. In *Proc. of FSE*, Haifa, Israel, January 1997.

[RRS11] A. Reed, C. Rezek, and P. Simmonds. Security guidance for critical areas of focus in cloud computing v3. 0. `https://downloads.cloudsecurityalliance.org/assets/research/security-guidance/csaguide.v3.0.pdf`, 2011.

[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.

[Tam03] Roberto Tamassia. Authenticated data structures. In *Algorithms - ESA 2003*, pages 2–5, 2003.

[TYM14] Isamu Teranishi, Moti Yung, and Tal Malkin. Order-preserving encryption secure beyond one-wayness. In *Proceedings of the 20th International Conference on Advances in Cryptology*, ASIACRYPT, 2014.

[WPS+15] S. Winter, T. Piper, O. Schwahn, R. Natella, N. Suri, and D. Cotroneo. Grinder: On reusability of fault injection tools. In *Proc. of the International Workshop on Automation of Software Testing*, pages 75–79, 2015.

[WSN+15] S. Winter, O. Schwahn, R. Natella, N. Suri, and D. Cotroneo. No pain, no gain?: The utility of parallel fault injections. In *Proc. of the International Conference on Software Engineering*, pages 494–505, 2015.

[Yao82] Andrew C. Yao. Protocols for secure computations. In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, SFCS '82, pages 160–164, Washington, DC, USA, 1982. IEEE Computer Society.

[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.

[ZLTS17] H. Zhang, J. Luna, R. Trapero, and N. Suri. deqam: A dependency based indirect monitoring approach for cloud services. In *IEEE Services Computing (SCC)*, pages 27–34, 2017.