Secret-sharing and order-preserving encryption based private computation

D3.5

Project reference no. 653884

February 2017

European Commission

Horizon 2020
European Union funding
for Research & Innovation

Document information

Scheduled delivery 01.03.2017
Actual delivery 01.03.2017
Version 1.0
Responsible Partner Cybernetica

Dissemination level

Public

Revision history

Date	Editor	Status	Version	Changes
14.09.2016	B. Portela	Draft	0.1	ToC
15.12.2016	B. Portela, K. Tarbe, V. Sokk	Draft	0.2	Initial draft
20.12.2016	D. Bogdanov	Draft	0.3	Revised version
25.01.2017	B. Portela, F. Maia, J. Paulo, K. Tarbe, R. Pontes, T. Oliveira, V. Sokk	Draft	0.4	Revised version
10.02.2017	K. Tarbe	Draft	0.5	Final Draft
18.02.2017	H. Mercier	Draft	0.6	UniNe review – secure database server
20.02.2017	J. Paulo	Draft	0.7	INESC TEC review
27.02.2017	K. Tarbe	Draft	0.8	Incorporate changes from reviews
01.03.2017	J. Paulo, K. Tarbe	Final	0.9	Final review.
01.03.2017	J. Paulo	Final	1.0	Small fixes.

Contributors

Dan Bogdanov (CYBER)
Karl Tarbe (CYBER)
Ville Sokk (CYBER)
João Paulo (INESC TEC)
Francisco Maia (INESC TEC)
Tiago Oliveira (INESC TEC)
Rogério Pontes (INESC TEC)
Bernardo Portela (INESC TEC)

Internal reviewers

J. Paulo (INESC TEC)
H. Mercier (UniNe)

Acknowledgements

This project is partially funded by the European Commission Horizon 2020 work programme under grant agreement no. 653884.

More information

Additional information and public deliverables of SafeCloud can be found at http://www.safecloud-project.eu

Glossary of acronyms

Acronym	Definition
ABB	Arithmetic Black-Box
AES	Advanced Encryption Standard
API	Application Programming Interface
DDL	Data Definition Language
DE	Deterministic Encryption
DML	Data Manipulation Language
ECB	Electronic Codebook
GCM	Galois/Counter Mode
IND-CCA	Indistinguishability under chosen plaintext attack
JDBC	Java Database Connectivity
MPC	Secure Multi-Party Computation
OPE	Order-preserving Encryption
PPT	Probabilistic Polynomial Time
SE	Searchable Encryption
SHA	Secure Hash Algorithm
SQL	Structured Query Language
TTP	Trusted Third Party
WDOW	Window distance one-wayness
WOW	Window one-wayness
YCSB	Yahoo! Cloud Serving Benchmark

Table of contents

Document information 2
Dissemination level 2
Revision history 2
Contributors 2
Internal reviewers 2
Acknowledgements 2
More information 3
Glossary of acronyms 4
Table of contents 5
List of Figures 6
List of Tables 6
Executive summary 7
1 Introduction 8
2 Solution 1: Secure database server 9
2.1 Privacy-preserving techniques 9
2.1.1 Overview 10
2.1.2 Standard encryption 10
2.1.3 Deterministic encryption 11
2.1.4 Order-preserving encryption 12
2.1.5 Searchable encryption 13
2.2 Deployment 14
2.3 Security guarantees 16
2.3.1 Standard encryption 17
2.3.2 Deterministic encryption 18
2.3.3 Order-preserving encryption 19
2.3.4 Searchable encryption 21
2.3.5 Discussion 22
2.4 Performance analysis 22
3 Solution 2: Secure multi-cloud database server 23
3.1 Privacy-preserving techniques 23
3.1.1 Additively homomorphic secret sharing 24
3.1.2 Multi-party computation 25
3.1.3 Database API instantiation 26
3.2 Deployment 27
3.3 Security guarantees 28
3.3.1 Discussion 30
3.4 Performance analysis 30
4 Solution 3: Secure multi-cloud application server 32
4.1 Privacy-preserving techniques 32
4.1.1 Storage and query interface 32
4.1.2 Supported SQL primitives 32
4.1.3 Chapters explaining the algorithms used by categories 34
4.2 Deployment 38
4.3 Security guarantees 39
4.4 Performance analysis 40

5 Conclusion **41**

6 References **42**

List of Figures

Figure 1: Secure Queries Solutions. 8
Figure 2: Abstract Cryptobox 10
Figure 3: Put operation instantiation for standard encryption 11
Figure 4: Get operation instantiation for standard encryption 11
Figure 5: Scan operation instantiation for standard encryption 11
Figure 6: Put operation instantiation for deterministic encryption 12
Figure 7: Get operation instantiation for deterministic encryption 12
Figure 8: Scan operation instantiation for deterministic encryption 12
Figure 9: Put operation instantiation for order-preserving encryption 12
Figure 10: Get operation instantiation for order-preserving encryption 13
Figure 11: Scan operation instantiation for order-preserving encryption 13
Figure 12: Put operation instantiation for searchable encryption 13
Figure 13: Get operation instantiation for searchable encryption 14
Figure 14: Scan operation instantiation for searchable encryption 14
Figure 15: High level overview of Solution 1: Secure database server deployment scheme 15
Figure 16: Concrete SafeCloud deployment for Solution 1: Secure database server 16
Figure 17: Adversarial model for Solution 1: Secure database server 17
Figure 18: Game IND-CCA. 18
Figure 19: Game PRIV. 18
Figure 20: Game Wow. 20
Figure 21: Game Wdow. 20
Figure 22: Comparison of Solution 1: Secure database server with a default HBase deployment 23
Figure 23: Put operation instantiation for secret sharing and MPC 26
Figure 24: Get operation instantiation for secret sharing and MPC 26
Figure 25: Scan operation instantiation for secret sharing and MPC 26
Figure 26: Concrete SafeCloud deployment for Solution 2: Secure multi-cloud database server 27
Figure 27: Adversarial model for Solution 2: Secure multi-cloud database server 29
Figure 28: Solution 3: Secure multi-cloud application Server deployment 38

List of Tables

Table 1: Solution 2: Secure multi-cloud database server performance 30
Table 2: Solution 3: Secure multi-cloud application server performance 40

Executive summary

SafeCloud Secure Queries layer focuses on providing a set of software solutions that enable secure data processing in three different setups. Each setup considers a specific trust model and targets different application scenarios. In previous deliverables, we have described the design and architecture for each solution motivating our choices with the project's use cases.

In this report, we go a step further in detailing the SafeCloud Secure Queries solutions that were previously described in D3.1 and D3.2. Because each solution targets distinct workloads and infrastructural deployments, a different set of privacy-preserving techniques is suitable for each one of them. Moreover, these sets of techniques provide different tradeoffs in terms of performance, security and functionality guarantees. In order to better understand these tradeoffs and the applicability of each solution in the development privacy preserving applications, we discuss and detail the cryptographic privacy-preserving techniques that will be supporting each one of the Secure Queries layer solutions. The integration of each technique is highly dependent on the assumed stakeholders, deployment and security guarantees, which are discussed along the text.

Additionally, we provide performance results obtained from a preliminary performance evaluation of each solution prototype. The results show that the integration of the different techniques is possible and show preliminary performance numbers that will serve as reference for future implementations and optimizations.

1 Introduction

The SafeCloud secure queries architecture has three solutions as shown in Figure 1:

Secure queries		
SQ1 **Secure database server**	**SQ2** **Secure multi-cloud database server**	**SQ3** **Secure multi-cloud application server**

Figure 1: Secure Queries Solutions.

For each solution, a detailed architecture has been described in previous deliverables. Such architecture was driven by the goal of providing a comprehensive set of solutions for secure data processing. Accordingly, different trust models are considered and different guarantees emerge for each solution. However, in order to be able to integrate SafeCloud Secure Queries solutions into practical applications, it becomes essential to better understand each one of the solutions and to which scenarios are they actually suitable.

In this deliverable, we provide a thorough description of each solution focusing on the privacy preserving techniques used and their characteristics. We provide technical details whenever useful and also focus on detailing the specific characteristics that make each solution unique. We also specify the caveats of each privacy preserving technique with the goal of allowing to understand the different tradeoffs between privacy and performance that each solution offers.

In the following sections, we present, for each of the three solutions, the following technical details. First, we discuss the cryptographic privacy-preserving techniques for each solution. That is, what kind of cryptographic queries does the solution support and what kind of technology this is achieved with. Second, we present the stakeholders, components and their deployment to make it clear where the privacy-preserving operations are being performed. Third, we explain the security guarantees of the techniques and direct the reader to sources with more detailed security arguments. Finally, we present preliminary performance results for secure queries using that solution.

2 Solution 1: Secure database server

2.1 Privacy-preserving techniques

The discussion of implementation-agnostic techniques in the SafeCloud framework starts from an API that is generic enough to encompass the various techniques described in the literature review of Deliverable D3.2. As such, we will be presenting cryptographic components composed of a trusted side and an untrusted side (for the Solution 1: Secure database server scenario, those being the client trusted infrastructure side and the cloud/third-party untrusted server side), that collaboratively comply with the various cryptographic primitives to be instantiated for the SafeCloud use cases. More specifically, the general NoSQL operations of Put, Get and Scan will be translated into a sequence of cryptographic operations that will produce the expected result. This allows for the end-user to interact with a standard SQL/NoSQL API, while the underlying cryptographic mechanisms enforce security guarantees in a transparent way.

The set of cryptographic operations will be, on the trusted side, $putC$, $getC$, $getCDecode$, $scanC$ and $scanCDecode$, and on the untrusted side, $putS$, $getS$ and $scanS$. Intuitively, these operations correspond to a trusted-side encoding cryptographic operation, an untrusted-side processing of said operation, and a trusted-side decoding of the obtained result. To abstract the underlying mechanisms used for storage, we define the behaviour of untrusted-side operations to use an oracle db for performing standard storage operations. Details follow:

- putC(id,v) - this is the general operation for data encoding to be executed on a trusted environment. This will take an identifier for querying id and some value v, and produce a set of data d to be processed on the untrusted side.
- putS^{db}(d) - this is the operation for data storage to be performed on an untrusted environment. This has oracle access to the database, and will perform (potentially several) storage operations for storing data d, using the described oracle db.
- getC(id) - this is the trusted-side operation for retrieving some identifier id. This will produce a token t to be interpreted remotely, according to the crypto mechanism employed.
- getS^{db}(t) - this is the untrusted-side operation that will receive a token and, accordingly, retrieve the associated stored data (using oracle db) required to respond to the associated get request.
- getCDecode$(c_1, \dots, c_n)$ - this is the trusted-side operation that will receive the encodings from a getS^{db} call, and decode accordingly to produce a result consistent with the originally requested get operation.
- scanC(id_1, id_2) - this is the trusted-side operation that receives two identifiers id_1, id_2, and produces a token t to be interpreted remotely, according to the crypto mechanism employed. This t can be seen as an encoded query to be given to the underlying technique, which will produce all entries between id_1, inclusively, and id_2, exclusively.
- scanS^{db}(t) - this is the untrusted-side operation that will receive a scan query token and, accordingly, retrieve the associated stored data (using oracle db) required to respond to the associated scan request.
- scanCDecode$(c_1, \dots, c_n)$ - this is the trusted-side operation that will receive the encodings from a scanS^{db} call, and decode accordingly to produce a result consistent with the originally requested scan operation.

In particular, these operations allow for the execution of the standard NoSQL API for *Put*, *Get* and *Scan*, as can be seen on Figure 2. *Put* will straightforwardly require a trusted-side operation for encoding the data, and an untrusted-side operation for storing it. *Get* will encode the identifier on the client side, which will produce some token t to be then appropriately queried on the untrusted side. The resulting encoded value is then returned to the client, which will decode it to obtain the output for the original request v. Finally, *Scan* will take a pair of identifiers and compute a token for the operation on the trusted side, which will then be processed on the untrusted side to generate several encodings. These encodings will be given back to the trusted side for decoding, which will produce the set of values matching the result for the original *Scan* operation.

$\mathsf{Put}(\mathsf{id},\mathsf{v})$:	$\mathsf{Get}(\mathsf{id})$:	$\mathsf{Scan}(\mathsf{id}_1,\mathsf{id}_2)$:
$\mathsf{d}_1,\ldots,\mathsf{d}_n \twoheadleftarrow \mathsf{putC}(\mathsf{id},\mathsf{v})$	$\mathsf{t} \twoheadleftarrow \mathsf{getC}(\mathsf{id})$	$\mathsf{t} \twoheadleftarrow \mathsf{scanC}(\mathsf{id}_1,\mathsf{id}_2)$
$\mathsf{putS}^{\mathsf{db}}(\mathsf{d}_1,\ldots,\mathsf{d}_n)$	$\mathsf{c}_1,\ldots,\mathsf{c}_n \twoheadleftarrow \mathsf{getS}^{\mathsf{db}}(\mathsf{t})$	$\mathsf{c}_1,\ldots,\mathsf{c}_n \twoheadleftarrow \mathsf{scanS}^{\mathsf{db}}(\mathsf{t})$
	$\mathsf{v} \leftarrow \mathsf{getCDecode}(\mathsf{id},\mathsf{c}_1,\ldots,\mathsf{c}_n)$	$\mathsf{v}_1,\ldots,\mathsf{v}_n \leftarrow \mathsf{scanCDecode}(\mathsf{id}_1,\mathsf{id}_2,\mathsf{c}_1,\ldots,\mathsf{c}_n)$

Figure 2: Abstract Cryptobox

By following this abstraction, we achieve a cryptographic component API that is not only generic enough to enable usage of the techniques referred to in deliverable D3.2, but that is also compatible with the deployment of other protocols that were originally not considered in the project. For instance, the proposal of Intel's SGX trusted hardware [Intel14] is a recent novelty for security approaches, and the associated formalization consists of an even more recent set of theoretical security analysis [BPSW16, PST16].

This general description of components and behaviour becomes more familiar when each operation is instantiated according to the associated cryptographic technique. We will now show how this is done for each considered technique. Afterwards, this set of components will be shown in the context of the overall Secure Queries architecture, and the specific implementations are then detailed.

2.1.1 Overview

In order to prevent information leakage related to the length of the identifier id and value v, client methods may require access to the maximum possible values of id and v, defined as MAX_{id}, MAX_v, respectively. The maximum possible value of encrypted id, MAX_{cid}, can also be needed to produce a token when all elements are to be returned. *Pad* is a function that applies padding to a given argument and it is also assumed that decryption methods remove padding, if present. Initialization vectors are denoted by IV. Secret key is denoted by SK and public key by PK. If SK_i and PK_i share the same index it means that they are a key-pair. Throughout the following sections different instantiations will be presented.

2.1.2 Standard encryption

This section describes how the Standard Encryption technique can be instantiated accordingly to the previously defined API. Enc_{STD} and Dec_{STD} are instantiated with AES-GCM.

putC(id, v):	putS^{db}((c$_{id}$, c$_v$)):
$(IV_{id}, IV_v) \twoheadleftarrow (\{0,1\}^l, \{0,1\}^l)$ $c_{id} \twoheadleftarrow IV_{id} \| Enc_{STD}(SK, IV_{id}, Pad(MAX_{id}, id))$ $c_v \twoheadleftarrow IV_v \| Enc_{STD}(SK, IV_v, Pad(MAX_v, v))$ Return (c_{id}, c_v)	$Put^{db}(c_{id}, c_v)$

Figure 3: Put operation instantiation for standard encryption

In Figure 3, putC performs the encryption of id and v after padding is applied. It returns a tuple d_1 that contains c_{id} and c_v. Given c_{id} and c_v, putS^{DB} invokes a standard key-value storage operation, put^{DB}, to perform the insertion.

The instantiation of getC, getS^{DB} and getCDecode is the following:

getC(id):	getS^{db}((t$_1$, t$_2$)):	getCDecode(id, $(c_{id_1}, c_{v_1}), \ldots, (c_{id_n}, c_{v_n})$):

$t \leftarrow (MIN_{cid}, MAX_{cid} + 1)$
Return t
	Return $Scan^{db}(t_1, t_2)$	For $i \in [1 \ldots n]$:
		  If $id = Dec_{STD}(SK, c_{id_i})$:
		    Return $Dec_{STD}(SK, c_{v_i})$
		Return $\perp$

Figure 4: Get operation instantiation for standard encryption

In Figure 4 getC returns a tuple that contains the minimum and maximum, plus one, possible value of the ciphertext space for identifier id. getS^{DB} returns all elements that are within that interval. getCDecode decrypts and compares each one of the retrieved identifiers with id. If id and decrypted id are equal, the correspondent value is decrypted and returned. If id is not present in the retrieved data set, getCDecode returns an empty value.

The instantiation of scanC, scanS^{DB} and scanCDecode is the following:

scanC(id$_1$, id$_2$):	scanS^{db}((t$_1$, t$_2$)):	scanCDecode(id$_1$, id$_2$, $(c_{id_1}, c_{v_1}), \ldots, (c_{id_n}, c_{v_n})$):

$t \leftarrow (MIN_{cid}, MAX_{cid} + 1)$
Return t
	Return $Scan^{db}(t_1, t_2)$	$L \leftarrow []$
		For $i \in [1 \ldots n]$:
		  If $id_1 \leq (id_i \leftarrow Dec_{STD}(SK, c_{id_i})) < id_2$:
		    $L \leftarrow (id_i, Dec_{STD}(SK, c_{v_i})) : L$
		Return L

Figure 5: Scan operation instantiation for standard encryption

In Figure 5,scanC and scanS^{DB} are identical to getC and getS^{DB}, respectively. scanCDecode performs the decryption of each one of the encrypted identifiers and checks if it is greater or equal than id_1 and less than id_2. If it is, it decrypts the correspondent value and adds the identifier id_i and value v_i to list L. Otherwise, no operation is performed. L is returned at the end.

2.1.3 Deterministic encryption

This section describes how the Deterministic Encryption technique can be instantiated according to the previously defined API. Enc_{ST} and Dec_{STD} are instantiated with AES-GCM and H with SHA256 [RS07].

$\mathsf{putC}(\mathsf{id},\mathsf{v})$:	$\mathsf{putS}^{db}((\mathsf{c_{id}},\mathsf{c_v}))$:
$(\mathsf{IV_{id}},\mathsf{IV_v}) \twoheadleftarrow (\mathsf{H}(\mathsf{SK_1}\|\|\mathsf{id}),\{0,1\}^l)$ $\mathsf{c_{id}} \twoheadleftarrow \mathsf{IV_{id}}\|\|\mathsf{Enc_{STD}}(\mathsf{SK_2},\mathsf{IV_{id}},\mathsf{Pad}(\mathsf{MAX_{id}},\mathsf{id}))$ $\mathsf{c_v} \twoheadleftarrow \mathsf{IV_v}\|\|\mathsf{Enc_{STD}}(\mathsf{SK_2},\mathsf{IV_v},\mathsf{Pad}(\mathsf{MAX_v},\mathsf{v}))$ Return $(\mathsf{c_{id}},\mathsf{c_v})$	$\mathsf{Put}^{db}(\mathsf{c_{id}},\mathsf{c_v})$

Figure 6: Put operation instantiation for deterministic encryption

In Figure 6, putC performs the encryption of id with using a constant initialization vector IV_{id}. The encryption of v uses a random initialization vector. It then returns a tuple d_1 that contains $_{id}$ and c_v. Given c_{id} and c_v putS uses a standard key-value storage operation, put^{DB}, to perform the insertion operation.

The instantiation of getC, getS^{DB} and getCDecode is the following:

$\mathsf{getC}(\mathsf{id})$:	$\mathsf{getS}^{db}(\mathsf{t})$:	$\mathsf{getCDecode}(\mathsf{id},\mathsf{c_{v_1}})$:
$\mathsf{IV_{id}} \twoheadleftarrow \mathsf{H}(\mathsf{SK_1}\|\|\mathsf{id})$ $\mathsf{c_{id}} \leftarrow \mathsf{IV_{id}}\|\|\mathsf{Enc_{STD}}(\mathsf{SK_2},\mathsf{IV_{id}},\mathsf{Pad}(\mathsf{MAX_{id}},\mathsf{id}))$ Return $\mathsf{c_{id}}$	Return $\mathsf{Get}^{db}(\mathsf{t})$	Return $\mathsf{Dec_{STD}}(\mathsf{SK_2},\mathsf{c_{v_1}})$

Figure 7: Get operation instantiation for deterministic encryption

In Figure 7, getC returns a token that corresponds to the deterministic encryption of identifier id. getS^{DB} simply calls get^{DB} with token t, since equality is preserved. getCDecode will then decrypt and return the correspondent value.

The instantiation of scanC, scanS^{DB} and scanCDecode is the following:

$\mathsf{scanC}(\mathsf{id_1},\mathsf{id_2})$:	$\mathsf{scanS}^{db}((\mathsf{t_1},\mathsf{t_2}))$:	$\mathsf{scanCDecode}(\mathsf{id_1},\mathsf{id_2},(\mathsf{c_{id_1}},\mathsf{c_{v_1}}),\ldots,(\mathsf{c_{id_n}},\mathsf{c_{v_n}}))$:
$\mathsf{t} \leftarrow (\mathsf{MIN_{cid}},\mathsf{MAX_{cid}}+1)$ Return t	Return $\mathsf{Scan}^{db}(\mathsf{t_1},\mathsf{t_2})$	$L \leftarrow []$ For $i \in [1 \ldots n]$: If $\mathsf{id_1} \leq (\mathsf{id}_i \leftarrow \mathsf{Dec_{STD}}(\mathsf{SK},\mathsf{c_{id_i}})) < \mathsf{id_2}$: $L \leftarrow (\mathsf{id}_i,\mathsf{Dec_{STD}}(\mathsf{SK},\mathsf{c_{v_i}})) : L$ Return L

Figure 8: Scan operation instantiation for deterministic encryption

Since only equality is preserved when using a deterministic encryption scheme, in Figure 8, scanS^{DB} must return all key-value pairs in order to scanCDecode decrypt and compare them on the trusted side.

2.1.4 Order-preserving encryption

This section describes how the Order-preserving Encryption technique can be instantiated accordingly to the previously defined API. Enc_{OPE} is instantiated with the technique presented in [BCLO09].

$\mathsf{putC}(\mathsf{id},\mathsf{v})$:	$\mathsf{putS}^{db}((\mathsf{c_{id}},\mathsf{c_v}))$:
$\mathsf{IV} \twoheadleftarrow \{0,1\}^l$ $\mathsf{c_{id}} \leftarrow \mathsf{Enc_{OPE}}(\mathsf{SK_1},\mathsf{id})$ $\mathsf{c_v} \twoheadleftarrow \mathsf{IV}\|\|\mathsf{Enc_{STD}}(\mathsf{SK_2},\mathsf{IV},\mathsf{Pad}(\mathsf{MAX_{id}}+\mathsf{MAX_v},\mathsf{id}\|\|\mathsf{v}))$ Return $(\mathsf{c_{id}},\mathsf{c_v})$	$\mathsf{Put}^{db}(\mathsf{c_{id}},\mathsf{c_v})$

Figure 9: Put operation instantiation for order-preserving encryption

In Figure 9, putC performs the order-preserving encryption of id and also of id concatenated with v. The reason of why id is encrypted twice is because it is computationally inefficient to decrypt order-preserving encrypted values. putC returns a tuple d_1 that contains c_{id} and c_v. Given c_{id} and c_v putS uses a standard key-value storage operation, put^{DB}, to perform the insertion operation.

getC(id):	$\text{getS}^{\text{db}}(t)$:	$\text{getCDecode}(id, c_{v_1})$:
Return $\text{Enc}_{\text{OPE}}(SK_1, id)$	Return $\text{Get}^{\text{db}}(t)$	$(\cdot, v) \leftarrow \text{Dec}_{\text{STD}}(SK_2, c_{v_1})$ Return v

Figure 10: Get operation instantiation for order-preserving encryption

The instantiation of getC, getS^{DB} and getCDecode is described in Figure 10. GetC returns a token that corresponds to the order-preserving encryption of identifier id. getS^{DB} simply calls get^{DB} with token t, since equality is preserved. getCDecode will then decrypt and return the correspondent value.

$\text{scanC}(id_1, id_2)$:	$\text{scanS}^{\text{db}}((t_1, t_2))$:	$\text{scanCDecode}(id_1, id_2, (c_{id_1}, c_{v_1}), \ldots, (c_{id_n}, c_{v_n}))$:
$c_{id_1} \leftarrow \text{Enc}_{\text{OPE}}(SK_1, id_1)$ $c_{id_2} \leftarrow \text{Enc}_{\text{OPE}}(SK_1, id_2)$ Return (c_{id_1}, c_{id_2})	Return $\text{Scan}^{\text{db}}(t_1, t_2)$	$L \leftarrow []$ For $i \in [1 \ldots n]$: $L \leftarrow \text{Dec}_{\text{STD}}(SK_2, c_{v_i}) : L$ Return L

Figure 11: Scan operation instantiation for order-preserving encryption

The instantiation of scanC, scanS^{DB} and scanCDecode is depicted in Figure 11. ScanC returns an order-preserving encryption of id_1 and id_2, which allows scanS^{DB} to return only the necessary elements. scanCDecode decrypts the retrieved elements.

2.1.5 Searchable encryption

This section describes how the Searchable Encryption technique can be instantiated. Enc_{SE}, $Ge\ \ Token_{SE}$ and $Query_{SE}$ can be instantiated using the techniques described in [BW07].

putC(id, v):	$\text{putS}^{\text{db}}((c_{id}, c_v))$:
$IV \twoheadleftarrow \{0,1\}^l$ $c_{id} \twoheadleftarrow \text{Enc}_{\text{SE}}(PK, id)$ $c_v \twoheadleftarrow IV \| \text{Enc}_{\text{STD}}(SK_2, IV, \text{Pad}(MAX_{id} + MAX_v, id \| v))$ Return (c_{id}, c_v)	$\text{Put}^{\text{db}}(c_{id}, c_v)$

Figure 12: Put operation instantiation for searchable encryption

In Figure 12,putC performs the encryption of id and v using the correspondent methods. It returns a tuple d_1 that contains c_{id} and c_v. Given c_{id} and c_v putS uses a standard key-value storage operation, put^{DB}, to perform the insertion operation.

The instantiation of getC, getS^{DB} and getCDecode is the following:

$\mathsf{getC}(\mathsf{id})$:	$\mathsf{getS}^{\mathsf{db}}(\mathsf{t})$:	$\mathsf{getCDecode}(\mathsf{id}, \mathsf{c}_{\mathsf{v}_1})$:

$\mathsf{t}_{\mathsf{id}} \twoheadleftarrow \mathsf{GenToken}_{\mathsf{SE}}(\mathsf{SK}, \mathsf{id})$
Return t_{id}
	For $(\mathsf{c}_{\mathsf{id}_i}, \mathsf{c}_{\mathsf{v}_i}) \in \mathsf{db}$:
	    If $\mathsf{Query}_{\mathsf{SE}}(\mathsf{t}, \mathsf{c}_{\mathsf{id}_i})$:
	        Return $(\mathsf{c}_{\mathsf{id}_i}, \mathsf{c}_{\mathsf{v}_i})$
	Return $\perp$
		Return $\mathsf{Dec}_{\mathsf{STD}}(\mathsf{SK}_2, \mathsf{c}_{\mathsf{v}_1})$

Figure 13: Get operation instantiation for searchable encryption

In Figure 13, getC uses $GenToken_{SE}$ to create a special search token t_{id} that contains a predicate that will be used by $Query_{SE}$ to check if a given ciphertext corresponds to the requested identifier id. getCDecode simply decrypts the returned value.

$\mathsf{scanC}(\mathsf{id}_1, \mathsf{id}_2)$:	$\mathsf{scanS}^{\mathsf{db}}(\mathsf{t})$:	$\mathsf{scanCDecode}(\mathsf{id}_1, \mathsf{id}_2, (\mathsf{c}_{\mathsf{id}_1}, \mathsf{c}_{\mathsf{v}_1}), \ldots, (\mathsf{c}_{\mathsf{id}_n}, \mathsf{c}_{\mathsf{v}_n}))$:

$\mathsf{t}_{\mathsf{id}} \twoheadleftarrow \mathsf{GenToken}_{\mathsf{SE}}(\mathsf{SK}, (\mathsf{id}_1, \mathsf{id}_2))$
Return t_{id}
	$L \leftarrow []$
	For $(\mathsf{c}_{\mathsf{id}_i}, \mathsf{c}_{\mathsf{v}_i}) \in \mathsf{db}$:
	    If $\mathsf{Query}_{\mathsf{SE}}(\mathsf{t}, \mathsf{c}_{\mathsf{id}_i})$:
	        $L \leftarrow (\mathsf{c}_{\mathsf{id}_i}, \mathsf{c}_{\mathsf{v}_i}) : L$
	Return L
		$L \leftarrow []$
		For $i \in [1 \ldots n]$:
		    $L \leftarrow \mathsf{Dec}_{\mathsf{STD}}(\mathsf{SK}_2, \mathsf{c}_{\mathsf{v}_i}) : L$
		Return L

Figure 14: Scan operation instantiation for searchable encryption

In Figure 14, scanC returns a token that basically encodes the predicates $>= id_1$ *and* $< id_2$, which allows $\mathrm{scanS}^{\mathrm{DB}}$ to return only the corresponding elements. scanCDecode decrypts the retrieved elements.

2.2 Deployment

Any application that wants to integrate SafeCloud secure queries solutions has two distinct APIs available. It can use either a SQL interface or a NoSQL one. For Solution 1: Secure database server in particular, SafeCloud will provide almost full SQL compatibility and a full HBase-like NoSQL interface. The extent of the restrictions on SQL compliance are out of the scope of the present deliverable. They are partially addressed in D3.4 and will also be extensively addressed in future deliverables.

To offer a SQL and NoSQL integration for the client application, SafeCloud solutions are deployed across two main sites (one trusted site and one untrusted site). Figure 15 depicts an high level overview of such deployment scheme.

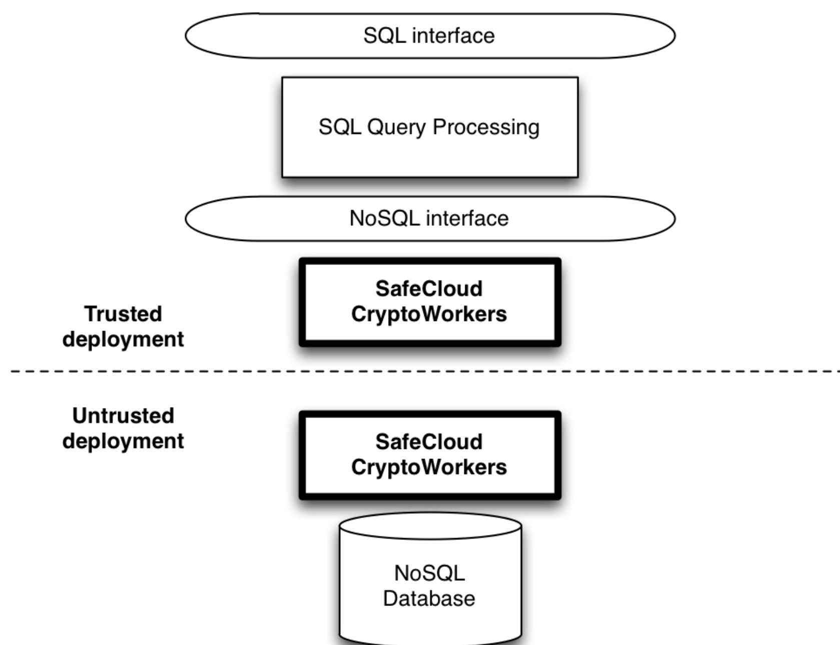

Figure 15: High level overview of Solution 1: Secure database server deployment scheme

Concretely, the client application has access to the trusted deployment site where it can issue requests to the desired API - SQL or NoSQL. When the application is using the SQL interface, a query processing component is used, which translates the application queries into NoSQL ones. If not the application can directly issue NoSQL request to the SafeCloud system.

Upon the reception of a NoSQL request (*put, get* or *scan*) a SafeCloud CryptoWorker will be called to handle it. These CryptoWorker components are installed in the same site as the interface handles and query processing components, i.e., all in the trusted site. The main job of the CryptoWorker is, according to its configuration, translate the received NoSQL operations into SafeCloud operations as described in Section 2.1. These operations are *putC*, *getC*, *getCDecode*, *scanC*, *scanCDecode*, *putS*, *getS* and *scanS.* Depending on the configured technique, each CryptoWorker will issue these operations following a specific workflow that allows secure data processing at the untrusted site. These workflows can be revisited in the Section 2.1.

Having translated NoSQL operations into SafeCloud workflow operations, the untrusted site CryptoWorkers will perform the converse task: they handle SafeCloud operations, translating them to operations the NoSQL database can process. To achieve this two components must be deployed on the untrusted site: a SafeCloud CryptoWorker and a NoSQL database. In detail, considering the SafeCloud prototypes, this means having an HBase deployment. Moreover, in order to improve the performance of the system and leveraging specific characteristics of this database, SafeCloud CryptoWorkers are deployed as HBase co-processors thus running collocated with HBase Region Servers.

In Figure 16 we depict the concrete deployment of the SafeCloud Solution 1: Secure database server system.

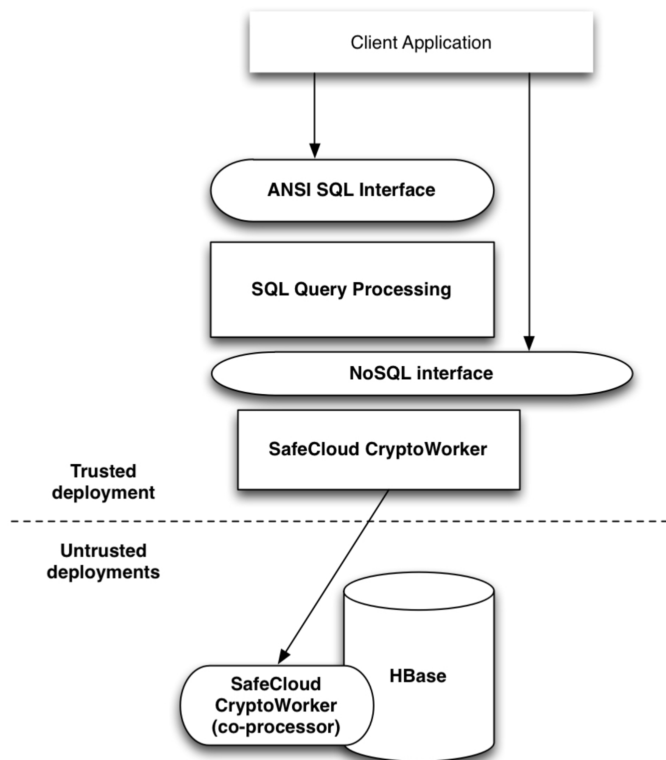

Figure 16: Concrete SafeCloud deployment for Solution 1: Secure database server

More details regarding this architecture refinement, including cryptographic components for NoSQL translation and how these are instantiated in the current prototype, can be read in D3.3.

2.3 Security guarantees

Deliverable D3.2 presented a state-of-the-art analysis regarding privacy-preserving techniques that could be deployed for the SafeCloud framework. The security analysis and evaluation of such techniques and the feasibility of their implementations is inherently associated with the power of the system's adversary. To rigorously define such behaviour, the adversary is assumed to be a monolithic entity that may be corrupting multiple participants and/or eavesdropping communication channels, which is a strictly stronger (and simpler) model than considering multiple adversaries that might act in a coordinated way. Furthermore, the adversarial power considered can also be specified over multiple axis, as described in Deliverable D3.2, which are now recalled:

- The adversary can be *active*, which implies an arbitrary behaviour regardless of the expected protocol, *covert*, also acting arbitrarily but adverse to a meaningful possibility of getting caught, and *semi-honest*, following the specified protocol while attempting to obtain additional information from the message transcript [AL07].
- The adversary can perform corruptions *statically*, i.e., the set of corrupt participants is established prior to the protocol, and remains unchanged throughout, or *adaptively*, i.e., corruptions of participants can occur during the protocol, based on data gathered by the adversary during its execution [DN14].
- The type of corruptions can also vary depending on the considered system. *Snapshot* corruptions imply that an adversary receives a snapshot of the entire data held by the corrupt party, while *persistent* corruptions mean that an adversary is

additionally capable of taking control of the corrupt party operations from the moment of corruption onwards [PBP16].

For Solution 1: Secure database server specifically, we consider an adversary that corrupts the untrusted deployment side. The corruption will always be static in this sense, as there are no additional untrusted participants, but adversarial power and type of corruptions can still depend on the underlying technique. Given the API presented in Section 3.1, we can now specify what a SafeCloud adversary can see in the context of our cryptographic implementations. This is defined exactly as the data sent to (and from) untrusted operations *putS*, *getS* and *scanS*, as well as the database in the untrusted environment, as depicted in Figure 17. Our security analysis for SafeCloud techniques will take these adversary definitions into consideration, and will be performed at a technique-by-technique basis. This will allow for a more detailed evaluation of adversarial assumptions, leakage, and security requirements assessment. Afterwards, we reason over the applicability of the different proposed protocols for meeting security criteria required from real-world solutions.

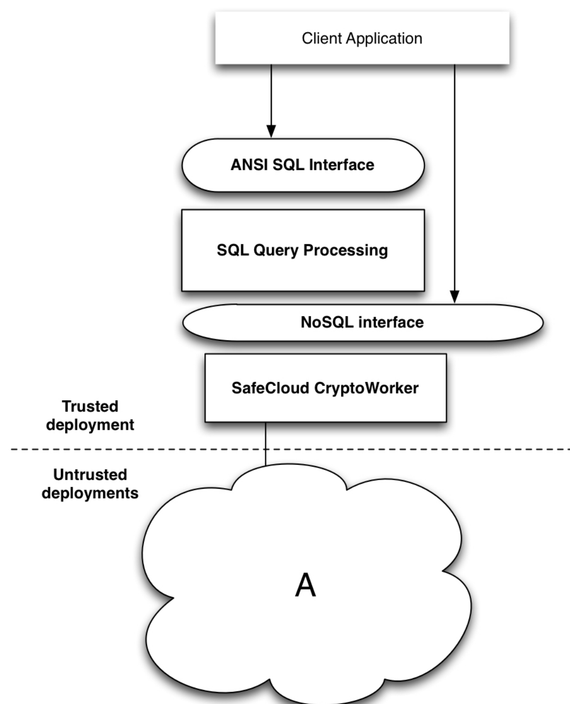

Figure 17: Adversarial model for Solution 1: Secure database server

2.3.1 Standard encryption

By employing a standard encryption scheme, we are allowing for the adversary to *persistently* corrupt our untrusted deployment in an *active* way. More specifically, we are implementing scheme AES-GCM, which satisfies the standard security guarantees required for symmetric encryption schemes. More formally, our guarantees obtained from our implementation can be inferred from the fact that our scheme $\boldsymbol{S}$ provides security under chosen-ciphertext attacks (*IND-CCA*) [GM82] against some Probabilistic Polynomial Time (PPT) adversary $\boldsymbol{A}$, as detailed in Figure 18.

The experiment behaves as follows. First, a key is generated, and a coin is flipped. Afterwards, the adversary is given the possibility of encrypting plaintexts and decrypting ciphertexts arbitrarily, towards producing two challenge messages m_0 and m_1. A message is encrypted according to the secret coin flip, and the resulting ciphertext is delivered to the adversary. The adversary wins if he can guess the plaintext associated with the received ciphertext (the coin flip result), which implies that it was able to extract some information about the received ciphertext without ever having access to the secret key. Note that, despite being able to query decryptions, the adversary is not supposed to decrypt the received ciphertext ($c' \,!= c$).

game IND-CCA$^{\mathsf{S},\mathcal{A}}$:	Enc(m):
$\mathsf{k} \twoheadleftarrow \mathsf{S.Gen}(1^l)$	$\mathsf{c} \twoheadleftarrow \mathsf{S.Encrypt}(\mathsf{k},\mathsf{m})$
$b \twoheadleftarrow \{0,1\}$	Return c
$(\mathsf{m}_0,\mathsf{m}_1) \twoheadleftarrow \mathcal{A}_1^{\mathsf{Enc},\mathsf{Dec}}(1^l)$	
$\mathsf{c} \twoheadleftarrow \mathsf{S.Encrypt}(\mathsf{k},\mathsf{m}_b)$	Dec(c′):
$b' \twoheadleftarrow \mathcal{A}_2^{\mathsf{Enc},\mathsf{Dec}}(\mathsf{c})$	If $\mathsf{c}' \neq \mathsf{c}$:
Return $b = b'$	Return $\mathsf{S.Decrypt}(\mathsf{k},\mathsf{c}')$
	Return $\perp$

Figure 18: Game IND-CCA.

Given that the data provided to *PutS*, *GetS* and *ScanS* consists of messages produced by Encrypt, and given that the key remains on the trusted side at all times, we can ensure that our adversary will not be able to infer any useful information from the ciphertexts received, under the classical security definitions. More specifically, we have that the advantage of adversary $\boldsymbol{A}$ against our SafeCloud deployment $STDEnc$ to infer information from received inputs is upper bounded by the existence some adversary $\boldsymbol{A}$ breaking *IND-CCA* on our scheme.

2.3.2 Deterministic encryption

The usage of deterministic encryption allows the adversary to *persistently* corrupt our untrusted deployment in an *active* way. We infer the security guarantees obtained from the specification of our scheme, detailed in Section 3.1. More formally, we present the security properties of our deterministic scheme $\boldsymbol{D}$ under ciphertext indistinguishability for deterministic encryption $PRIV$ against some PPT adversary $\boldsymbol{A}$, as detailed in Figure 19.

game PRIV$^{\mathsf{D},\mathcal{A}}$:	Enc(m):
$L \leftarrow [\,]$	$L \leftarrow \mathsf{m} : L$
$\mathsf{k} \twoheadleftarrow \mathsf{D.Gen}(1^l)$	$\mathsf{c} \twoheadleftarrow \mathsf{D.Encrypt}(\mathsf{k},\mathsf{m})$
$b \twoheadleftarrow \{0,1\}$	Return c
$(\mathsf{m}_0,\mathsf{m}_1) \twoheadleftarrow \mathcal{A}_1^{\mathsf{Enc},\mathsf{Dec}}(1^l)$	
$\mathsf{c} \twoheadleftarrow \mathsf{D.Encrypt}(\mathsf{k},\mathsf{m}_b)$	Dec(c′):
$b' \twoheadleftarrow \mathcal{A}_2^{\mathsf{Enc},\mathsf{Dec}}(\mathsf{c})$	If $\mathsf{c}' \neq \mathsf{c}$:
If $(\mathsf{m}_0 \in L \vee \mathsf{m}_1 \in L)$:	Return $\mathsf{D.Decrypt}(\mathsf{k},\mathsf{c}')$
Return F	Return $\perp$
Return $b = b'$	

Figure 19: Game PRIV.

The experiment behaves as follows. First, a key is generated and a coin is flipped. Afterwards, the adversary is given the possibility of encrypting plaintexts and

decrypting ciphertexts arbitrarily, towards producing two challenge messages m_0 and m_1. The adversary wins if it guesses correctly the result of the coin flip, given the associated received ciphertext. Since the scheme implies a deterministic encryption scheme, the adversary is prevented from querying encryptions of any of the challenge messages m_0 or m_1.

Observe that, contrary to the previous argument, we can no longer claim that the adversary is unable to infer any useful information from the ciphertexts. In fact, the adversary will be able to know exactly which duplicate ciphertexts exist in the system, which is a bad event in experiment $PRIV$, bound by excluding $(m_0 \in List \vee m_1 \in List)$ [BBO07]. As such, we have that the advantage of $\boldsymbol{A}$ against our SafeCloud deployment $DETEnc$ to infer information from non-duplicate inputs is upper-bounded by the existence some $\boldsymbol{A}$ breaking the security property of our scheme.

2.3.3 Order-preserving encryption

The usage of order-preserving encryption allows the adversary to *persistently* corrupt our untrusted deployment in a *semi-honest* way. Order-preserving encryption schemes fundamentally leak information regarding distance among plaintexts. As such, security of our order-preserving encryption scheme follows the analysis presented in [BCO11], ensuring two general security properties in this context: *window one-wayness*, and *window distance one-wayness*. We now present the security definitions referring to our scheme, and discuss their implications.

Window one-wayness (WOW) evaluates the one-wayness of an order-preserving scheme, i.e., if an adversary that is given a set of ciphertexts is capable of accurately guessing an interval within one of the underlying plaintext lies. More specifically, let r define an interval size, and let z define the number of uniformly generated encrypted messages received by the adversary $\boldsymbol{A}$. We say that our order-preserving encryption scheme $\boldsymbol{O}$ satisfies r, z-WOW if the probability for an adversary $\boldsymbol{A}$ to produce a successful result is negligible for some value ε,

$$Pr[Wow_{O,A}^{r,z} = 1] < \varepsilon$$

where the security experiment is detailed in Figure 20, and behaves as follows. First, a key is generated and a set of z messages is sampled uniformly from the message space, which will then be encrypted and given to the adversary. The adversary wins if it is able to guess an interval over which some original plaintext message resides $(\exists m \in \{m_1, \ldots, m_z\} : m \in [m_l, m_r])$, which implies the adversary was able to extract some information from the set of received ciphertexts.

```
game Wow^{O,A}(r, z):
k ↞ O.Gen(1^l)
m_1, ..., m_z ↞ UGen_M(z)
For i ∈ [1 ... z]:
      c_i ↞ O.Enc(k, m_i)
(m_l, m_r) ↞ A_1(c_1, ..., c_z)
If ((m_r − m_l) mod M + 1 ≤ r) ∧ (∃m ∈ {m_1, ..., m_z} : m ∈ [m_l, ..., m_r])
      Return T
Return F
```

Figure 20: Game Wow.

Window distance one-wayness (WDOW) evaluates the extent to which an order-preserving encryption scheme is expected to leak the distance between underlying plaintexts. More specifically, let r define an interval size, and let z define the number of uniformly generated encrypted messages received by the adversary $\boldsymbol{A}$. We say that our order-preserving encryption scheme $\boldsymbol{O}$ satisfies r, z-WDOW if the probability for an adversary $\boldsymbol{A}$ to produce a successful result is negligible for some value ε.

$$Pr[Wdow_{O,A}^{r,z} = 1] < \varepsilon$$

The security experiment is detailed in Figure 21, where M is the plaintext message space, behaving as follows. First, the key is generated, and a set of z messages is uniformly sampled from the message space, which will then be encrypted and given to the adversary. The adversary wins if it is able to guess an interval separating two of the originally given plaintext messages $(\exists m_i, m_j \in \{m_1, \dots, m_z\} : \ |m_i - m_j| \in [d_1, d_2])$, which implies the adversary was ample to extract some information regarding plaintext distance from the set of received ciphertexts.

```
game Wdow^{O,A}(r, z):
k ↞ O.Gen(1^l)
m_1, ..., m_z ↞ UGen_M(z)
For i ∈ [1 ... z]:
      c_i ↞ O.Enc(k, m_i)
(d_1, d_2) ↞ A_1(c_1, ..., c_z)
If ((d_1 − d_2) + 1 ≤ r) ∧ (∃m_i, m_j ∈ {m_1, ..., m_z} : |m_i − m_j| ∈ [d_1, ..., d_2])
      Return T
Return F
```

Figure 21: Game Wdow.

Contrary to the previous definitions, the leakage of order-preserving schemes is harder to quantify, since one cannot easily employ classical definitions of security. Informally, one should expect these schemes to leak some non-negligible amount of data for real-world adversaries on databases sufficiently large. This is a direct consequence of the lower-bounds established in [BCO11], which are shown to be fundamentally the best possible for a scheme that enables order-preserving encryption. To illustrate these

limitations, for an adversary guessing the exact value of the plaintext, we have an adversarial advantage defined as

$$Adv_{O,A}^{1,z-wow} < \frac{9z}{\sqrt{M-z+1}}$$

which essentially entails that the challenge set size z should be small in comparison to the plaintext message space size M. Furthermore, and to prevent attacks where adversaries have additional information regarding the context of stored data (e.g. they are aware that some hospital dataset contains sensitive information regarding patient age), one should also ensure that the set of data encrypted with order-preserving encryption is close to an uniform distribution, so as to avoid potential inference attacks [NKW15], which are outside the scope of these definitions (the adversary receives uniformly generated plaintexts).

2.3.4 Searchable encryption

The usage of searchable encryption provides a different level of security depending if the *semi-honest* adversary performs corruptions *persistently*, or in a *snapshot* way. For *snapshot* corruptions, schemes for searchable encryption provide security similar to the usage of standard encryption (following the IND-CCA model), since the adversary will only compute over an encrypted database, disregarding execution queries. For *persistent* attackers, leakage will be dependent on operations performed, and is thus parametrized by the operations made available [HK14]: *putS* and *scanS*, in this case.

As such, we define the overall leakage considering a set of leakage functions L_{get}, L_{put}andL_{scan}, such that

$$L_{put}(id, v) = (len(\underline{id}), SRCH_t(\underline{id}))$$
$$L_{get}(id) = ACCP_t(id, id)$$
$$L_{scan}(id_1, id_2) = ACCP_t(id_1, id_2)$$

where $\underline{id}$ is the set of unique searchable keywords in id, $ACCP_t(id_1, id_2)$ is the access pattern at time t defined as the set of$\{ID(i_i): i_i \in [id_1, id_2]\}$, and $SRCH_t(\underline{id})$is the set of entries for all searched terms until time t that also appear in $\underline{id}$.

The quantification of leakage towards a persistent attacker can be calculated directly by adding the leakage of Kinsertions, M gets and N scans:

$$L_{system} = \sum_{k=1}^{N} \sum_{m=1}^{M} \sum_{n=1}^{M} L_{put}(id_k, v_k) + L_{get}(id_m) + L_{scan}(id_{n_1}, id_{n_2})$$

Intuitively this implies that as the number of queries observed by the adversary increases, the security of a searchable encryption will become more and more similar to the one provided by an order-preserving encryption scheme, as the adversary will eventually have access patterns $ACCP_t$ for all searchable keywords on the database. As such, feasibility of searchable encryption schemes is dependent on two main criteria. If the corruptions can be modelled as *snapshots*, then the security is similar to standard encryption, while providing strictly better functional properties. Otherwise, our

searchable encryption scheme presents a security middle-ground between standard encryption and order-preserving encryption.

2.3.5 Discussion

The refinements of these security trade-offs will allow SafeCloud solutions to remain secure according to the use case security requirements while maximizing efficiency. For instance, if only equality computations are required, then employing Deterministic Encryption allows us to have a system with stronger security guarantees than Order-Preserving encryption. Furthermore, if the system considers only snapshot attackers, then employing Searchable Encryption schemes allows for ciphertext indistinguishability, since leakage is only associated with access patterns from *putS*, *getS* and *scanS*, which are not provided to this specific kind of adversary. This level of fine-grained optimization will typically translate into an extension of the database description to be stored and managed to specify the security guarantees/operations required of each data type.

By having these levels of security defined by the different underlying cryptographic components, we show that the guarantees provided by the SafeCloud framework have versatility in implementation. This is a crucial characteristic to ensure, since the project aims to provide solutions that go beyond the specified use cases, which may require different combinations of data security and performance requirements.

2.4 Performance analysis

This section presents preliminary evaluation results that compare our current prototype, supporting deterministic encryption, with an original HBase deployment without any privacy-aware guarantees. The goal of this section is not to provide a complete analysis of the system performance but to show that our design and prototype are able to perform secure HBase queries over protected data, and to show a hint of what can be expected once the development of the solutions is more mature. The evaluation was performed with the YCSB benchmark [YCSB16]. This widely used benchmark provides a standard method to compare the performance of different NoSQL databases. In other words, *Yahoo!* developed this benchmark because both the paradigm and the access pattern of such data stores are quite different from the ones used by traditional benchmark systems, mostly designed for relational databases.

The YCSB benchmark starts by creating a defined number of concurrent clients that try to perform a set of operations according to a predefined workload. The type of operations available include Read, Scan, Delete and Update operations. The evaluation discussed in this deliverable is made through a user specified workload where the number of records of the NoSQL database, the number of requests and the frequency of different of requests is parameterized. The requests follow a uniform distribution to choose what database rows are accessed during the duration of the benchmark.

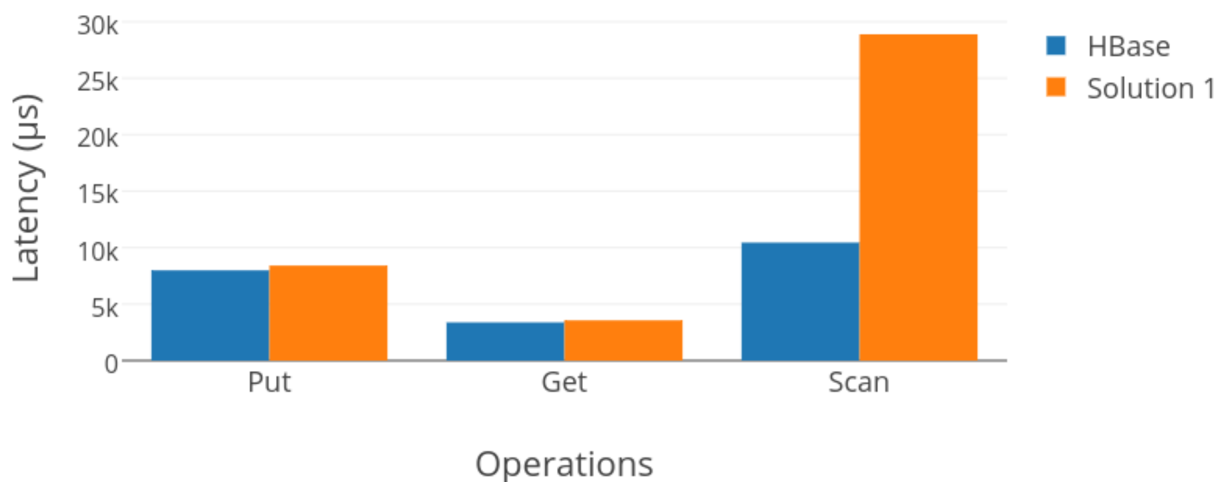

Figure 22: Comparison of Solution 1: Secure database server with a default HBase deployment

The experiments used a single HBase database with 100 records. The machines used for the evaluation had an Intel i3-2100 CPU with a clock rate of 3.10GHz and 8GB of main memory, and a 500GB 7200 RPM hard drive. 40% of the requests made to the database were PUT requests, with the remaining request being 30% Gets and 30% evenly distributed Scans. Multiple requests were sent to the database and, for each request, the latency of the operation was measured. Figure 22 shows the results for three HBase operations: PUT, GET and SCAN. As observed, determinist encryption has a small performance overhead on the PUT and GET requests, since the only extra step required is ciphering and deciphering plaintext information. However, the SCAN operation has a significant impact. As explained in the previous sections, this overhead is due to the need to bring every record to the trusted deployment and search for the requested rows. These results are only preliminary and attempt to give an idea of the current status of the solutions, thus further performance optimizations are still possible in order to reduce the observed overhead. For instance, caching and batching techniques for queries done to the untrusted HBase backend will allow us to significantly improve the performance of our current prototype.

3 Solution 2: Secure multi-cloud database server

3.1 Privacy-preserving techniques

SafeCloud's Secure multi-cloud database server (solution 2) targets a scenario where the untrusted deployment consists of three distinct entities and one of them can behave adversarially. Such scenario contrasts with the single untrusted deployment of solution 1 and represents a different trust model where data is required not to be in the control of a single entity. A multiple untrusted deployment scenario is justified when, for instance, cloud infrastructures are known to be vulnerable to attacks and have software backdoors that make traditional data encryption insufficient. Advanced cryptographic techniques, such as multi-party computation algorithms, are suitable in this scenario. Additionally, these techniques place solution 2 within a considerably different

deployment scenario and with vastly different performance and security implications, when compared to solution 1. This allows for the SafeCloud Secure Queries layer to provide a wider range of solutions, so as to cater to a broader scope of real-world applications.

The architectures of the Secure database server (solution 1) and the Secure multi-cloud database server (solution 2) share a lot of similarities with respect to trusted/untrusted components. As such, and for consistency of presentation, the techniques employed in this section will also follow the API proposed in Section 3.1. More specifically, this section will detail the basic mechanisms used for multi-party computation, describe how these can be deployed within the SafeCloud framework, specify the security guarantees obtained from applying these techniques, and provide preliminary results with respect to the implementations.

3.1.1 Additively homomorphic secret sharing

As described previously, in solution 2, we consider a trust model where data cannot be stored in a single entity. This change takes into account scenarios where encrypting data and storing it in a single third-party untrusted deployment is not considered secure enough. This lack of trust in the single entity scenario comes from considering a stronger adversary that, for instance, is able to decrypt data stored in such entity. This is addressed through secret sharing techniques. Originally, these have been proposed to protect sensitive data such as an encryption key while not diminishing their availability [Shamir79]. If an encryption key was stored in a single location and then lost, the encrypted data with that key would also be lost. However, if the key is stored in multiple places, then risk of exposing it also increases. To address these two concerns, secret sharing schemes contain two types of entities, a dealer and players. A dealer is capable of securely dividing a secret s into p shares. Each share is then stored on a single player. In order to reconstruct a secret, the dealer must retrieve a subset of the shares from the players. Depending on the secret sharing technique used to generate the secrets, some subsets of shares can not be used to reconstruct the secret. The secret is secure as long as players players do not collude to exchange shares.

In our Secure multi-cloud database server, secrets are protected by an additively homomorphic secret sharing scheme described by *Bogdanov et al.* [Bog13]. This scheme has two important properties. First, the only subset of shares that can be used to reconstruct a secret is the complete set of shares. This first property ensures that a malicious entity can only have access to the private data, if it gains access to every share. The second property is the creation of a Galois Field. Such a field is crucial to enable players to compute a given function $\boldsymbol{F}$ over the shares without disclosing any information.

The secret sharing scheme exports two functions in its API, a $\boldsymbol{encrypt}(MAX_{id}, id)$ and a $\boldsymbol{decrypt}(MAX_{id}, shares)$ function. Similar to the standard encryption described in section 3.1, the encrypt and decrypt function require the maximum possible value of an identifier id. The other argument of the $\boldsymbol{encrypt}$ function is the secret id to be divided into shares. The output of the $\boldsymbol{encrypt}$ functions is a token holding all the shares. The other argument to $\boldsymbol{decrypt}$ function is the token containing every share and it outputs the original secret. The number of shares in to which the secret must be divided is

currently bounded to three. This number is chosen specifically because the multi-party protocols used were designed for three parties.

3.1.2 Multi-party computation

While secret sharing provides the foundation for data privacy by storing secrets in distinct entities, the capacity to perform the computation required by databases is given by Multi-party computation protocols. These protocols empower the parties (players) with the capacity to compute a function over the shares without disclosing any information. In order to do so, the parties must collaborate and share pieces ofinformation, however such information never reveals anything about the original data. In fact, the protocols described and used in this solution have been proved secure [Bog13]. The only entity that can calculate the output of the protocols is the dealer, and it does so only when it receives the output of the protocol from each party.

Although different computation primitives are possible in a multi-party scenario, only two are relevant for the implementation of our solution 2. These are a protocol that computes the equality of two secrets and a protocol the calculates if a secret is greater or equal than another. Both of these protocols are actually built on top of other protocols, however they provide the highest level of abstraction essential to the queries of a NoSQL database. Both protocolos provide a similar API with just two arguments:

- equal($share1_i$, $share2_i$)
- greaterOrEqual($share1_i$, $share2_i$)

This API is exported by each computing party. Note that, each party stores multiple shares, each belonging to a different secret. When one of these computations is needed for a certain secret, separate requests are issued to each of the parties. Accordingly, the subscript i on the shares represents the computing party and ranges from 1 to 3. Each party will perform the necessary operations on their respective share of the secret and the output will be a new share. The combination of these generated shares is used by the dealer to figure out the result of the computation.

Despite the multi-party protocols providing only two basic functions, these can be used to do the remaining order comparisons. For instance, a greaterThan function can be created by executing the greaterOrEqual and equal protocol. While both protocols must be executed independently, their outputs enables the dealer to understand the order of the secrets. As such, the API can be extended for every order comparison.

3.1.3 Database API instantiation

This section focuses on instantiating the cryptographic operations on secret sharing and multi-party API's presented in the previous sections. From a high level perspective, the secret sharing API's are used on the trusted side, while the multi-party API on the untrusted party.

$\mathsf{putC}(\mathsf{id}, \mathsf{v})$:	$\mathsf{putS}^{\mathsf{db}}((\mathsf{shares}, \mathsf{c_v}))$:
$\mathsf{shares} \twoheadleftarrow \mathsf{Encrypt}(\mathsf{MAX_{id}}, \mathsf{id})$ $\mathsf{c_v} \twoheadleftarrow \mathsf{Enc_{STD}}(\mathsf{SK}, \mathsf{Pad}(\mathsf{MAX_v}, \mathsf{v}))$ Return $(\mathsf{shares}, \mathsf{c_v})$	$\mathsf{Put}_1^{\mathsf{db}}(\mathsf{shares}[1], \mathsf{c_v})$ $\mathsf{Put}_2^{\mathsf{db}}(\mathsf{shares}[2], \mathsf{c_v})$ $\mathsf{Put}_3^{\mathsf{db}}(\mathsf{shares}[3], \mathsf{c_v})$

Figure 23: Put operation instantiation for secret sharing and MPC

In Figure 23 ,*putC* applies the secret sharing Encrypt function to the identifier *id* and the standard encryption function to the value *v* as in the section 2.1. The output of the function *putC* is a tuple containing an array of shares and an encrypted value. Once these two values are obtained, the $putS^{DB}$ function can be issued to store a secret on each party. The execution of $putS^{DB}$ performs three put operations, one on each backend server.

$\mathsf{getC}(\mathsf{id})$:	$\mathsf{getS}^{\mathsf{db}}(\mathsf{shares}, (\mathsf{c_{id_1}}, \mathsf{v_{id_1}}), .., (\mathsf{c_{id_n}}, \mathsf{v_{id_n}}))$:	$\mathsf{getCDecode}(\mathsf{s}_{11}, .., \mathsf{s}_{jn}, \mathsf{c_{v_1}}), .., (\mathsf{c_{id_n}}, \mathsf{c_{v_n}}))$:
Return $\mathsf{Encrypt}(\mathsf{MAX_{id}}, \mathsf{id})$	For $i \in [1..n]$: $\quad \mathsf{res_share}_i \leftarrow \mathsf{equal}(\mathsf{share}_i, \mathsf{c_{id_i}})$ Return $(C_i, V_i, \mathsf{res_share}_i)$	For $i \in [1..n]$: $\quad$ If $\mathsf{Decrypt}(\mathsf{MAX_{id}}, \mathsf{s}_{i1}, \mathsf{s}_{i2}, \mathsf{s}_{i3})$ $\quad\quad$ Return $\mathsf{Unpad}(\mathsf{Dec_{STD}}(\mathsf{SK}, \mathsf{c_{v_i}}))$ Return $\perp$

Figure 24: Get operation instantiation for secret sharing and MPC

For the get operation, and as depicted in Figure 24, a call for *getC* encrypts an identifier *id* into three shares. Each share is sent to the respective party which will call $getS^{DB}$. This function receives a single share, $share_i$, and executes the equal protocol on every record of the database. The output of this function, composed by the set of records that satisfy the query, is sent to the trusted deployment. Having collected the output of the computation of each computing party, the trusted domain processes such output in the *getCDecode* function. This function, goes through every record it received and decrypts the output of the equal protocol. If such output is the value *0* that means it is a valid record and the corresponding decrypted value is returned.

$\mathsf{scanC}(\mathsf{id}_1, \mathsf{id}_2)$:	$\mathsf{scanS}^{\mathsf{db}}(\mathsf{share}_1, \mathsf{share}_2, (\mathsf{c_{id_1}}, \mathsf{v_{id_1}}), .., (\mathsf{c_{id_n}}, \mathsf{v_{id_n}}))$:	$\mathsf{scanCDecode}(\mathsf{s}_{11}, .., \mathsf{s}_{jn}, \mathsf{c_{v_1}}), .., (\mathsf{c_{id_n}}, \mathsf{c_{v_n}}))$:
$\mathsf{shares_id}_1 \twoheadleftarrow \mathsf{Encrypt}(\mathsf{MAX_{id_1}}, \mathsf{id})$ $\mathsf{shares_id}_2 \twoheadleftarrow \mathsf{Encrypt}(\mathsf{MAX_{id_2}}, \mathsf{id})$ Return $(\mathsf{id}_1, \mathsf{id}_2)$	For $i \in [1..n]$: $\quad \mathsf{res_share}_i \leftarrow \mathsf{greaterOrEqual}(\mathsf{share}_{1_i}, \mathsf{c_{id_i}})$ $\quad\quad \&\& \ \mathsf{lessThan}(\mathsf{share}_{2_i}, \mathsf{c_{id_i}})$ Return $(C_i, V_i, \mathsf{res_share}_i)$	$L \leftarrow []$ For $i \in [1..n]$: $\quad$ If $\mathsf{Decrypt}(\mathsf{MAX_{id}}, \mathsf{s}_{i1}, \mathsf{s}_{i2}, \mathsf{s}_{i3})$ $\quad\quad$ Return $\mathsf{Unpad}(\mathsf{Dec_{STD}}(\mathsf{SK}, \mathsf{c_{v_i}}))$ Return L

Figure 25: Scan operation instantiation for secret sharing and MPC

In Figure 25, *scanC* function is similar to the *getC* function, but accepts two input values and generates shares for each one of them. A single share from each secret is sent to each server on the untrusted parties. Every server will execute the $scanS^{DB}$ where the protocols required to calculate the order of the shares are executed. While lessThan has not been presented in the previous section it can be constructed from the existing API. The resulting shares, the corresponding identifiers and values are sent to the trusted

domain. The trusted domain, completes the scan algorithm, by going through every record and decrypting the resulting shares. If a resulting share outputs a valid result, the row identifier and values are decrypted and returned back to the client.

3.2 Deployment

The deployment scheme for solution 2 is very similar of that previously presented for solution 1. The significant difference between them is the fact that, for this solution, there are three HBase installations in three different untrusted sites. The other components remain the same with the exception of the untrusted site cryptoworkers that are now also three. The concrete deployement scheme for this solution is depicted in Figure 26.

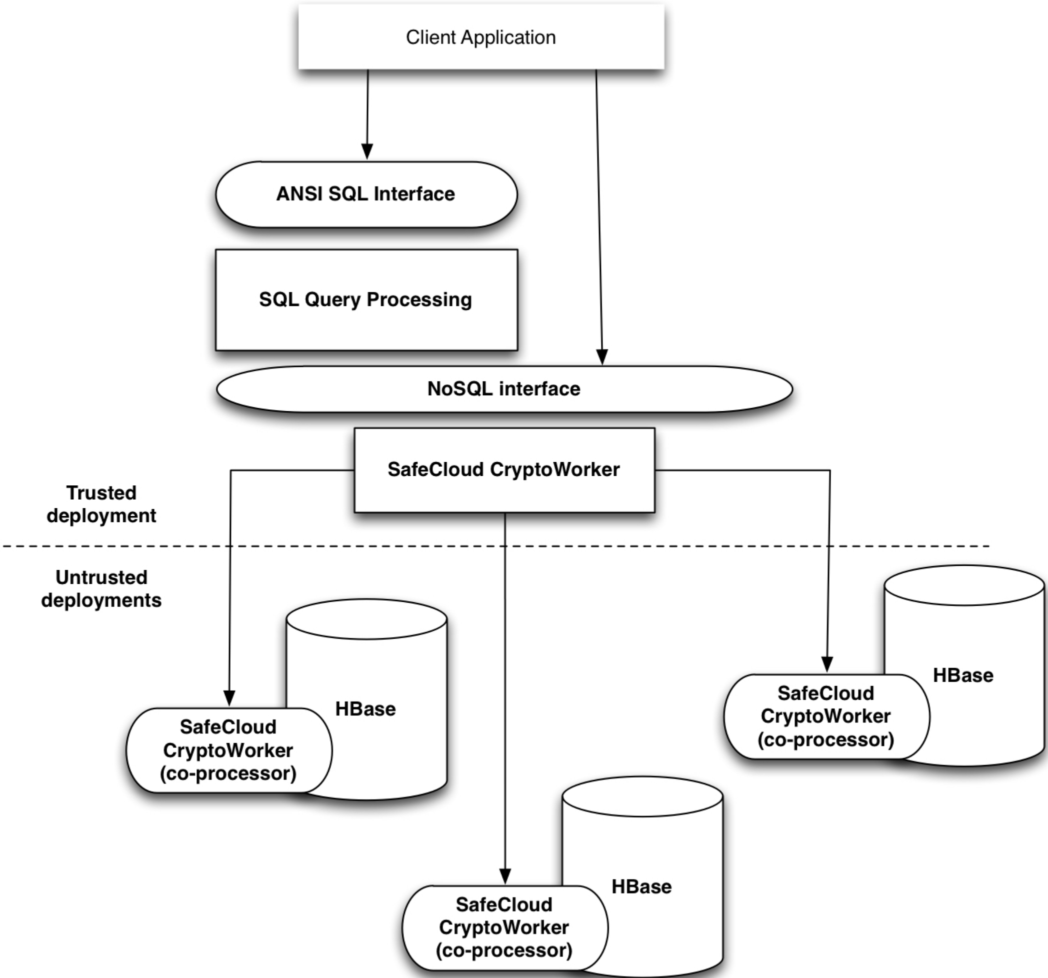

Figure 26: Concrete SafeCloud deployment for Solution 2: Secure multi-cloud database server

3.3 Security guarantees

The Secure multi-cloud database server considers the problem of securely executing code over multiple untrusted participants, which is within the domain of multi-party computation (MPC). Standard definitions of MPC involve a number of players $P_1, \dots, P_n$ in possession of inputs $x_1, \dots, x_n$ agree to compute a function $f(x_1, \dots, x_n) = (y_1, \dots, y_n)$, so that each P_i knows y_i and learns nothing additional that could not be deduced from its personal input and output.

Ideally, this would be achievable by using a trusted third party (TTP), where each P_i would send its x_i to the TTP, and expect to receive the appropriate y_i which, by the definition of a TTP, would meet all security criteria. Finding such a trusted participant is unrealistic for real-world scenarios, however the notion of having a TTP is useful for modelling the security of MPC protocols in general. This approach is commonly used for formalizing composable security proofs such as universal composability [Can01], reactive simulatability [PW00], abstract cryptography [MR11] or inexhaustible Turing machines [KT13].

The real-world describes the behaviour expected from an actual protocol execution for some specified number of participants. The ideal world is a formal specification of the same protocol under the existence of a TTP: some ideal functionality executes the protocol in an incorruptible way, while a simulator fully aware of corruptions simulates towards the adversary a protocol view that should be indistinguishable from the one observed on a real-world scenario. The reasoning is that, if such a simulator exists, then a real-world adversary cannot possibly gain more information than what would be obtainable in the idealized version of the protocol.

Security analysis in this sense is also dependent on basic definitions for adversarial power presented in Deliverable D3.2 and recalled in Section 3.3 of this document. Intuitively, it is easier to present a MPC protocol as secure (indistinguishable from an idealized TTP) the more limited we consider our adversary to be. Feasibility of implementations is inherently associated with these assumptions, as security against very powerful adversaries implies extremely costly and complex protocols, while potentially not being representative of the realistic malicious entities acting upon the system.

By enabling the usage of multi-party computation, solution 2 leverages an execution model with multiple participants and distributed trust, towards expanding the trade-offs offered to deployed solutions. More specifically, considers a scenario where untrusted computation is distributed over three participants, where at most one is adversarially controlled (see Figure 27). This translates into a trust model for an adversary of threshold 1. Furthermore, for the deployed algorithms referred to in Section 4.1, we will also be considering a *semi-honest* adversary. As previously contextualized, these Solutions should be seen as alternative deployment scenarios considering different security/performance requirements, e.g. if *semi-honest* adversaries are an inadequate assumption, then the Secure database server (solution 1) proposes alternative mechanisms under stronger adversarial scenarios.

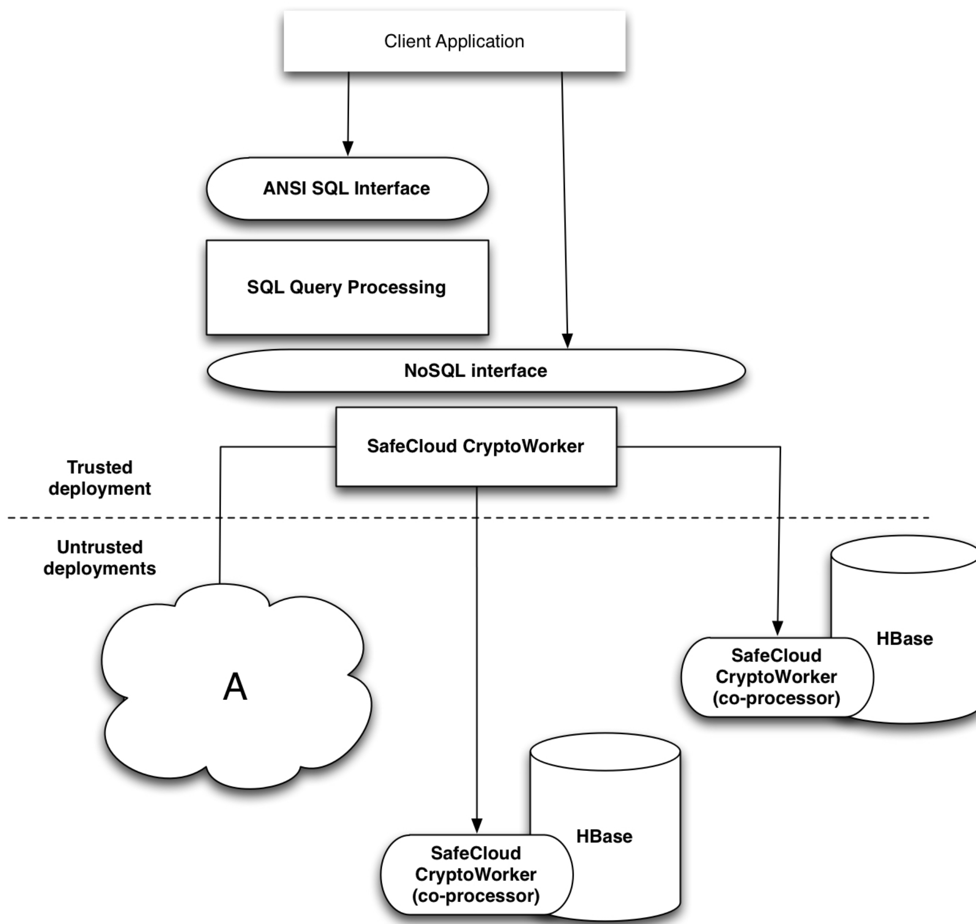

Figure 27: Adversarial model for Solution 2: Secure multi-cloud database server

Under these specifications, the behaviour displayed by applications deployed with the Secure multi-cloud database server (solution 2) are indistinguishable from ideally executing these operations in a trusted party. This is achieved from the security of the underlying Sharemind algorithms presented in Section 4.1, where the view of the system is simulatable given only the view of the passive, adaptively corrupting participant [Bog14]. More specifically, security of Sharemind is presented as an arithmetic black-box (ABB) [DN14], where several operations are shown to be secure under a weaker notion of *input privacy* security, followed by a stronger operation shown to be *secure*, which will confer indistinguishability and composability to the whole set of operations.

- *Secure* refers to the classical, stronger scenario, where some adversary A selects inputs to machine H and observes computation outputs written onto H, while comparing the view that is shown regarding the protocol. This will be either the real execution, having the messages exchanged between participants and the corrupt party internal state, or the ideal execution having the same data being presented by a simulator that is only given the input/output of the corrupt party.
- Input privacy refers to operations that provide a weaker notion of security, where an adversary A selects inputs and observes the system (including the state of the corrupt participant), but is not able to verify computation outputs. This is modelled by splitting the participants on both real and ideal world into two machines $H = H' \cup H_{\perp}$. A will then set the computation inputs onto H' and the outputs will be eventually written on $H_{\perp}$ which in this setting is disabled from interacting with A.

Similar to the previous notion of security, *A* will also observe either a real-world execution of the protocol, or a similar view as presented by a simulator that is only given the input of the corrupt party.

The central idea of this model is that intermediate operations, i.e. all of them except the final one returning the result, do not have to provide the strongest level of security, provided that these intermediate results are not presented to the adversary. This builds upon composability results that show that a set of *input private* protocols, followed by a *secure* protocol constitutes a *secure* protocol in itself, according to the specified underlying security model.

3.3.1 Discussion

The security model of the Secure multi-cloud database server (solution 2) and the protocols employed over it follow a different approach than the ones available in the Secure database server (solution 1) in two fundamental ways. On one hand, performing computation over data in solution 1 inherently requires some relaxation of security guarantees, as can be observed in the less strict security models presented in Section 2.3 for techniques such as Deterministic encryption, Order-preserving encryption, or Searchable encryption. All of these consider different levels of security that partially reveal properties of the original plaintexts. Meanwhile, multi-party computation present in solution 2 enables the execution of arbitrary functions over the same strong security assurances. On the other hand, all solutions deployed over MPC algorithms are expected to have meaningful computation and communication costs, which is expected to reduce throughput of operations. Alternatively, the flexibility of solution 1 allows for specifying data protection techniques following performance considerations, which inherently increases flexibility in fulfilling system performance requirements.

3.4 Performance analysis

This section presents a performance analysis of Solution 2: Secure multi-cloud database server in comparison to a baseline HBase deployment without privacy guarantees. Again, YCSB benchmark was used with similar settings to the experiments described in Section 2.4. Results are shown in Table 1 and show the latency of HBase operations in milliseconds.

Operation	HBase	Secure multi-cloud database server
PUT	7.9	35.6
GET	3.4	1493.4
SCAN	10.4	20198.2

Table 1: Solution 2: Secure multi-cloud database server performance

The PUT operation is approximately 4 times slower than the one observed for the baseline HBase deployment. The decrease of performance is associated with the generation of three secrets and with issuing three parallel operations to the different untrusted Hbase clusters. Moreover, this decrease in performance is even more noticeable in the GET and SCAN operations, as both operations require the HBase clusters to transverse every record of the database, to perform multi-party protocols that involve computation and the exchange of messages across clusters, and finally to send the results to the trusted deployment. Despite the execution of the operations

being similar, the SCAN operation is slower, as the GrearOrEqual protocol used in this type of operations requires a higher number of computation rounds between the computing parties. In both cases, the overhead of the system appears from the communication rounds between the computing parties. It is expected for the performance to decrease with a linear growth as the number of records stored in the database increase. Again, by resorting to caching and batching optimizations we expect the performance to increase significantly and relief the current communication bottleneck.

4 Solution 3: Secure multi-cloud application server

4.1 Privacy-preserving techniques

4.1.1 Storage and query interface

This section is to remind the reader of the primitives that are used in Solution 3: Secure multi-cloud application server. The basic building blocks of Solution 3 are additive secret sharing and multi-party computation over those shares. We are standing on the shoulders of giants, i.e., we leverage existing work and build on top of that.

Solution 3 differs from the other solutions by enabling the possibility of having multiple local domains, where not all of them are trusted. The general idea for this scenario is to allow multiple client entities to upload their sensitive data to a shared database that can be later queried over by potentially untrusted end-users, i.e., we have many data owners and we protect the privacy of individual data owners by only allowing aggregation queries over the data.

We use the Sharemind MPC framework [Bog13], which provides us with cryptographic protocols and tools for computing with secret-shared values. We utilize three party secret sharing, where each secret value is shared between three non-colluding compute parties. Primitive operation between secret-shared values are provided by the framework, but larger operations and applications are implemented using a domain specific language called SecreC [BLR14]. In SecreC, private and public data is separated on type system level and operations with private data are automatically converted to invocations of cryptographic protocols.

The Sharemind framework also has storage capabilities. To be more specific, there is a database module that stores tables. Every computing party has access to only its shares of the secrets and therefore can store only its shares. The database module can work as both row-oriented and column-oriented database, but in this solution we use only column-oriented mode.

We have an extensive suite of common high level operations for dealing with the data tables. The suite of high level operations is implemented in SecreC and was first designed and implemented as the backend of the statistical analysis tool Rmind [BKLS16]. Later the suite of operations was extracted from Rmind, and was renamed to Sharemind Analytics Engine. The Sharemind Analytics Engine contains operations for sorting, aggregating, joining and so on.

To recap, we have all the tools needed for doing secure data analytics in the Sharemind framework. SafeCloud secure queries layer Solution 3 builds a SQL interface on top of the operations provided by the Sharemind Analytics Engine.

4.1.2 Supported SQL primitives

It should be noted that we use PostgreSQL dialect of SQL which is SQL as supported by PostgreSQL. We also support some PostgreSQL specific extensions to SQL. For example in PostgreSQL there are extension to DML called RETURNING which allow to return queries over the affected rows from INSERT, UPDATE and DELETE statements. Also in DELETE and UPDATE you can use joins with USING list or FROM list to specify which rows to operate on.

4.1.2.1 Data Types

We support some exact numeric types, they are various bit lengths (8, 16, 32 and 64) of INTEGER. Unlike PostgreSQL, we also support the unsigned variants of INTEGER with the name being uintX, where X is the number of bytes (1,2,4,8). Unsigned types are supported by the Sharemind Analytics Engine and therefore it made sense to add them. We also support approximate numeric types float4 and float8.
From string types we only support bounded length strings, or in SQL terminology: varchar(n).
At the moment support for datetime types is not done, but could be emulated by using integers and suitable conversions.

4.1.2.2 Scalar Expressions

We support basic binary operations: (+, -, *, /, and, or, =, !=, <, <=, >, >=). We also support the unary negation (-). We also support aggregate expressions: (all, any, avg, count, max, min, sum). The list is not very large, and we are definitely missing string operations, but they are easy to add in future versions of Solution 3.

4.1.2.3 DDL - Data Definition Language

We support a very basic part of the DDL: CREATE TABLE and DROP TABLE. For new tables we support only one kind of column constraint, that is NON NULL. All other column and table constraints are not supported.
This means that there are many things missing from a standard relational database. For example there are no constraints (other than non-null), views, foreign keys or triggers. Support for them is not planned.

4.1.2.4 DML - Data Manipulation Language

We support INSERT with the RETURNING clause. We do not support default values and conflict resolution which is not a problem, since conflicts are impossible because we do not support constraints other than NON NULL.
DELETE statements are supported with the USING list and the RETURNING clause.
The UPDATE statement is supported, including both forms of SET syntax, the RETURNING clause and FROM list.

4.1.2.4.1 Queries

We support only regular select queries. We do not support WITH queries (Common Table Expressions).
We support combining queries with the UNION ALL and EXCEPT operations. INTERSECT and distinct UNION are not yet implemented.
We do not support removing duplicate rows with the DISTINCT keyword of SELECT.

4.1.2.4.1.1 Joins

We support full outer, left outer, right outer, inner and cross joins. However, there is a limitation, we only support equi-joins. Other types of joins can be emulated by the user. For an example, we do not support “SELECT * from t1 join t2 on t1.a < t2.b;”, but we do support “SELECT * from t1, t2 WHERE t1.a < t2.b;”. The result sets are identical, but the performance of such a query will be suboptimal, because of the cross join, that makes the intermediate table (before filtering) very large.
We support joins with the ON expression, USING list and natural join.

4.1.2.4.1.2 Filtering

We support the same operations as in Scalar Expressions. This means that we do not currently support BETWEEN operator. As mentioned before, we do not support string operations and functions, that means that the WHERE clause can not contain the LIKE operator and other pattern matching operators.

4.1.2.4.1.3 Group by

We support GROUP BY and HAVING clauses. The aggregates that can be used were already mentioned in the Scalar expressions section, but for easier reference we will list them again here: (all, any, avg, count, max, min, sum). For the HAVING clause we have the same limitations as for the WHERE clause.
We do not support window functions.

4.1.2.4.1.4 Order by

We support sorting based on multiple columns. We also can sort in both ascending and descending order and the ordering of NULLs can be specified with NULLS FIRST and NULLS LAST directives.

4.1.2.4.1.5 Limit and Offset

We currently do not support limiting and offsetting the result table rows. These will be added in the future.

4.1.2.5 DCL - Data Control Language

Not supported at the moment. We might support it, when we have figured out how to handle access policies.

4.1.2.6 TCL - Transaction Control Language

Not supported at the moment. Instead of giving an explicit error, we ignore some of the commands. Ignoring is needed because some JDBC drivers always use transactions even for SQL statements that have nothing to do with transactions or even modifying the database. For an example a simple query "SELECT * FROM t1;" is peppered with "BEGIN" and "COMMIT" by the JDBC driver, thus becoming "BEGIN; SELECT * FORM t1; COMMIT;".

4.1.3 Chapters explaining the algorithms used by categories

The following subsections give an overview of the algorithms used for supporting the database operations.

4.1.3.1 Join

Database table join is an operation which concatenates rows of two tables to create a merged table. There are different types of join operations according to the criteria used to decide which rows are included in the output table. We support inner join, left and right outer joins, full outer join and cross join. The rows are joined based on values in key columns of the two input tables. Key values can only be matched by equality but not other relations like greater than. This type of join is known as equi-join. A key can consist of multiple columns.

- Inner join computes the Cartesian product (cross join) of the input tables and drops rows where the key values are not equal.
- Left outer join creates a table with all the rows from the first input table. Rows that have a matching key value in the second table are concatenated with the row from

the second table. Rows that do not have a match in the second table will be given missing values for the fields of the second table.

- Right outer join is like a left outer join but now all the rows from the right table are included in the result.
- Full outer join includes all rows from both tables in the output. Rows with matching keys are concatenated and the rest of the rows are given missing values for the fields of the other table.

The cross join algorithm is straightforward and computes a cartesian product of the secret-shared values using local computations only. Each row of the first table is added to the result in combination with each row from the second table.

The other types of joins require matching values of the key columns of the joined tables. The algorithm is described in [LTW13].

- The rows of both input tables are shuffled [LWZ11] using a random permutation p_1. None of the computing parties knows the permutation.
- A random secret-shared Advanced Encryption Standard (AES) encryption key s is generated. None of the computing parties knows the key.
- An implementation of AES operating on secret-shared values [LTW13] is used to encrypt the keys of both tables using the encryption key s. Using electronic codebook mode (ECB), the same values of the keys will have the same ciphertext.
- The encrypted key column values will be declassified. The number of matching rows is leaked but since the shuffle permutation p_1 and AES encryption key s are not know, it cannot be determined which input key values correspond to which ciphertext.
- The public ciphertexts can be used to match rows of the two tables and the resulting table can be constructed from the shares using local computations.
- If a key value occurs multiple times in a table, the output can be shuffled using a second random permutation p_2. The number of occurrences of a key can leak some information. For example, if an individual in a database has a particularly large number of entries, the number of occurrences of a key can be used to determine which ciphertext corresponds to the individual.

This algorithm works for a single key column. If we need a key consisting of n fields then the n fields can be hashed into a single value and we can use the same algorithm. We use the Carter-Wegman keyed hash function [CW77] with a random secret-shared key because it is efficient in a secure multi-party computation setting. The hash function with key k and input x is:

$$h(k, x) = x_s k_s + \ldots + x_2 k_2 + x_1 k_1$$

4.1.3.2 Group by

Group by allows grouping rows by a key and aggregating groups (rows with the same key) using a function such as mean. The result of a group by operation is a table with a row per key value where the fields are group aggregates. We support multiple key columns and the sum, mean, all, any, count, maximum and minimum functions. The group by algorithm is described in [BKK+16] and is similar to the join algorithm.

- The rows of the input table are shuffled using a random permutation.

- The key column is encrypted using AES with a random secret-shared key.
- The encrypted key column values are declassified.
- Each party now knows which rows have equal key values (i.e. belong to the same group) and we can apply standard privacy preserving algorithms on the groups to compute the aggregates.
- If a key value occurs multiple times in a table, the output can be shuffled using a second random permutation as in the case of the JOIN operation.

The group by algorithm leaks the sizes of the groups but does not leak any value in the input table or the aggregated table since no party knows the permutations or the AES key and the aggregates are computed using composable privacy preserving algorithms.

If a key consists of multiple columns we can again use the Carter-Wegman hash function.

4.1.3.3 Sorting

We support sorting database tables, i.e. reordering the rows based on the ordering of some columns in the table. This is implemented using oblivious sorting algorithms which are described in [BLT14].

Any sorting algorithm which is based on comparisons can be used obliviously if the input is shuffled with a random permutation and the results of the private comparisons are declassified [HKI+13]. The declassified values can be used for reordering the shares locally. This will leak the number of equal values but this can be avoided by adding a position index to the values which also makes the algorithm stable and in the case of quicksort, avoids the worst case quadratic complexity since no elements are equal.

Multiple oblivious sorting algorithms are described in [BLT14] such as oblivious radix sort and sorting using sorting networks which are graphs of compare and swap gates where the control flow is fixed, i.e. the comparisons depend on the size of the input but not the values. The algorithms have been implemented on Sharemind and are evaluated in the article. We will describe oblivious quicksort since it was deemed the most efficient.

To recap, the standard quicksort algorithm consists of recursive calls to the partition function which reorders the array into elements less than a pivot element and elements greater than the pivot. The partition function is then recursively called on the subarrays until the subarrays have size one or zero. On a high level the oblivious algorithm works as follows:

- Shuffle the input vector using a random permutation.
- Follow the standard quicksort algorithm.
    a. Perform comparisons using the less than or equal protocol.
    b. Declassify the results of the comparisons.
    c. Locally reorder shares of array elements when partitioning.

In Sharemind, we can do arithmetic with vectors by applying the operator pointwise. A pointwise multiplication of n element vectors is more efficient than n scalar multiplications because it requires only one invocation of the multiplication protocol instead of n invocations. To optimise quicksort we can collect the operands of the

comparisons on the same level of the call tree into vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ and do a single vectorised comparison.

In the optimised oblivious quicksort algorithm, we maintain a queue of subarrays that need to be partitioned. Operands of comparisons in the partition function for each subarray in the queue are collected into vectors $\boldsymbol{a}$ and $\boldsymbol{b}$. The result of $\boldsymbol{a} \leq \boldsymbol{b}$ is declassified and used to locally swap shares of elements in the subarrays. If the subarrays resulting from the partitioning need to be partitioned further, they are added to the queue. This means that all the comparisons needed in partitioning are collected into rounds of comparisons and comparisons of a round are executed using a single protocol invocation. The total number of invocations of the less than or equal protocol is reduced significantly compared to the straightforward implementation.

4.1.3.4 Deleting rows

The application supports deleting table rows based on a condition. We first compute the filtered table leaving rows that match the condition. To do this, we obliviously shuffle the rows including values of the condition, declassify the shuffled condition and remove rows that do not match the condition. This will leak the number of rows that match the deleting condition. This filtered table is needed for the RETURNING clause that is supported by many SQL implementations.

Next we compute the set difference between the original table and the table of rows that match the condition. The set difference algorithm is again similar to the table join algorithm:

- Shuffle the input tables using random permutations.
- Compute Carter-Wegman hashes of each row of each table using the same random key.
- Encrypt the hashes with the same random AES key.
- Declassify the ciphertexts.
- Add a public boolean column $\boldsymbol{left}$ to each table which indicates if the row is from the first table. This column is constantly true for the first table and constantly false for the second table.
- Concatenate the two tables and sort by the public ciphertext.
- Now, by traversing the public cryptograms, the same cryptograms will be located consecutively. If a group contains false values in the $\boldsymbol{left}$ column, then these rows will not be included in the output because one of them is from the second table. If every row in a group has the value true in the $\boldsymbol{left}$ column then this row only occurred in the first table and is included in the output.
- Drop the $\boldsymbol{left}$ column from the resulting table and shuffle the rows using a random permutation.

Finally, the original table is replaced by the output of the set difference operation.

4.1.3.5 Updating values

SQL allows updating columns of database tables. In the simplest case, every value of a column is set to a new value using a vector-valued expression. Supporting this is straightforward because we already support arithmetic. SQL also allows to update

values only in rows that match a certain condition. The algorithm for this case is as follows:

- Compute the filter condition.
- Compute new column values as if there was no condition. New values are also computed in positions where the condition is false.
- Use an oblivious choice to update the column value to the new value in positions where the condition is true.

Oblivious choice is a method for computing the expression *if c then a else b* without leaking $\boldsymbol{c}$, $\boldsymbol{a}$ or $\boldsymbol{b}$. If the underlying secure multi-party computation technology supports multiplication and addition, this can be computed with the expression:

$$\boldsymbol{ca} + (1 - \boldsymbol{c})\boldsymbol{b}$$

where $\boldsymbol{c}$ has been converted to a numeric vector of zeroes and ones.

4.2 Deployment

In principle, the deployment has not changed from what was proposed in the D3.2 and it is depicted on Figure 28.

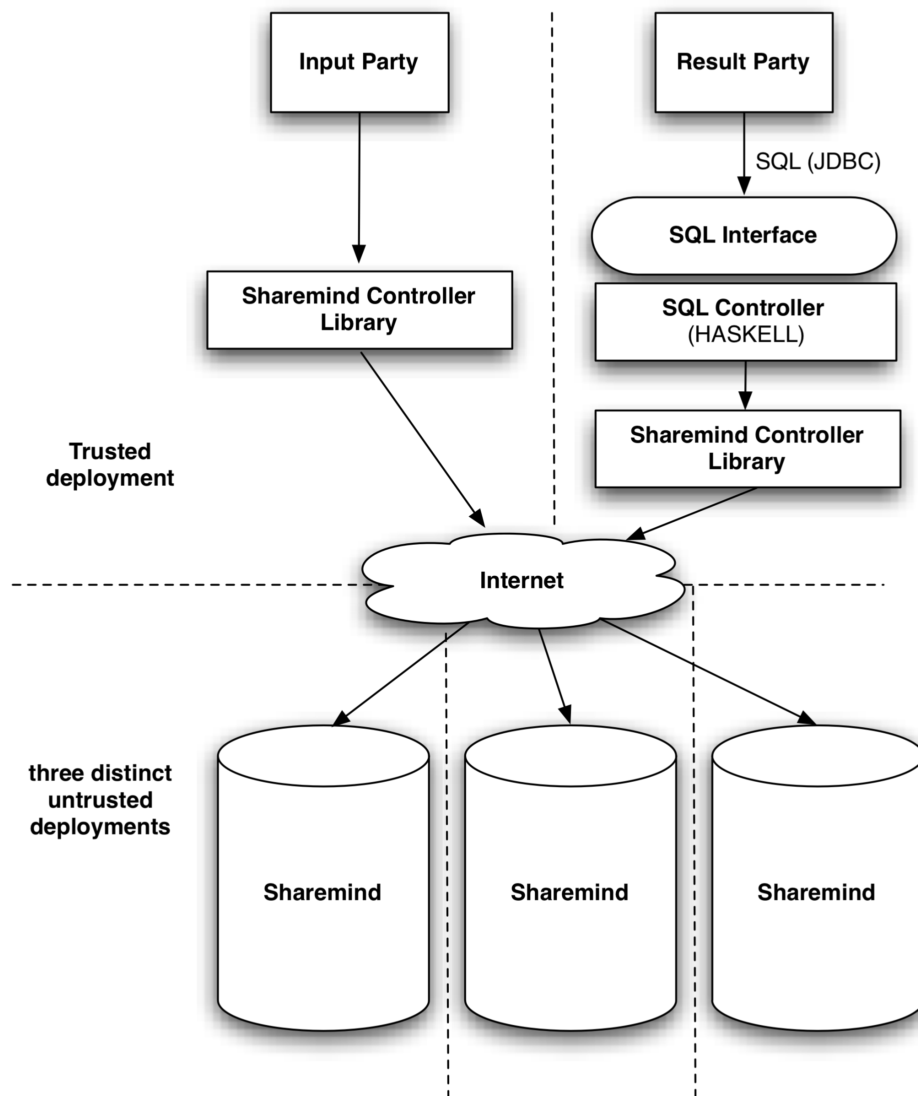

Figure 28: Solution 3: Secure multi-cloud application Server deployment

In detail, there are two binaries that are used for the client proxy. The first is called *sql-controller* and it provides a text-based user interface and the other is called *server-emu* and it emulates a PostgreSQL server, so one can use existing PostgreSQL drivers to talk to the Solution 3: Secure multi-cloud application server.

4.3 Security guarantees

We built our solution on top of the Sharemind Application server which uses homomorphic secret sharing. The standard Sharemind security guarantees apply. Section 3.3 already addresses the security properties of the underlying Sharemind protocols, therefore we do not duplicate them here. Instead, we talk about the higher level concepts.

Some information is leaked in operations for more efficient protocols. For example in JOIN, the number of matching rows is leaked, but it cannot be determined which of the input keys correspond to which count. In GROUP BY (aggregation) we leak the sizes of groups, but no values. In most cases these leaks are not important and we chose the extra performance against sealing those information leaks.

There are still some problems left to address. For example, if you calculate the average height of a person in different age groups and the oldest person is alone in the 110+ group, her height would leak. We counter this by not returning results when a group has less than k rows. However, if you can construct queries on two different groups that have a single individual in their intersection then the difference of the query results will leak that individual's value. Protection against this is not enforced, but provided with auditing. We are looking into better ways to provide more fine grained access policies.

4.4 Performance analysis

The system was benchmarked on a cluster with three machines with dedicated 10 Gb/s links. The machines have Intel Xeon E5-2640 v3 processors and 128 GB of memory. The running times of different SQL statements (averaged over 10 runs) are given in Table 2.

Operation	Mean time (s)	Standard deviation	Test setup
Inner join	44.66	2.01	Two tables. 100000 rows, 10 columns each.
Delete	26.34	0.19	100000 rows, 10 columns. 50000 rows for which the WHERE condition (id > 50000) holds will be deleted.
Order by	40.07	0.73	100000 rows, 10 columns. Sorted by two columns.
Group by	3.42	0.1	100000 rows, 2 columns. One is used for grouping and the other is averaged per group. There are 10000 groups.
Update	3.29	0.15	100000 rows, 10 columns. 5 columns are updated with the identity function in positions where a WHERE filter with a single comparison operation is true (id > 50000).
Insert	66.58	0.19	100 inserts into an empty table with 5 columns. The inserts are executed one by one instead of a single insert with 100 rows.
Select (with filter)	0.82	0.04	100000 rows, 10 columns. The WHERE filter has a single comparison (id > 50000).

Table 2: Solution 3: Secure multi-cloud application server performance

All of the test tables had 64-bit integer columns. The INSERT test had nullable columns, the others did not. The client application was on the same local area network as the test cluster.

5 Conclusion

This report focused on the integration of different privacy techniques in SafeCloud Secure Queries solutions. As stated in the text, each solution may leverage specific privacy-aware techniques to explore distinct tradeoffs between security, performance and database functionalities. The set of techniques that can be integrated with each solution depends on the specific stakeholders, deployment and security models expected for that solution.

Our preliminary results show that the current prototypes can indeed support the different privacy-preserving techniques, discussed along this deliverable, and serve as a first baseline for further implementations and optimizations of each solution.

6 References

[AL07] Aumann, Yonatan, and Yehuda Lindell. "Security against covert adversaries: Efficient protocols for realistic adversaries." *Theory of Cryptography Conference*. Springer Berlin Heidelberg, 2007.

[BBO07] Bellare, Mihir, Alexandra Boldyreva, and Adam O'Neill. "Deterministic and efficiently searchable encryption." *Annual International Cryptology Conference*. Springer Berlin Heidelberg, 2007.

[BCLO09] Alexandra Boldyreva, Nathan Chenette, Younho Lee, and Adam O'neill. "Order-preserving symmetric encryption." In Annual International Conference on the Theory and Applications of Cryptographic Techniques, pp. 224-241. Springer Berlin Heidelberg, 2009.

[BCO11] Boldyreva, Alexandra, Nathan Chenette, and Adam O'Neill. "Order-preserving encryption revisited: Improved security analysis and alternative solutions." *Annual Cryptology Conference*. Springer Berlin Heidelberg, 2011.

[BPSW16] M. Barbosa, B. Portela, G. Scerri, and B. Warinschi. Foundations of hardware-based attested computation and application to SGX. In EuroS&P, pages 245–260. IEEE, 2016.

[BKK+16] Dan Bogdanov, Liina Kamm, Baldur Kubo, Reimo Rebane, Ville Sokk, and Riivo Talviste. Students and Taxes: a Privacy-Preserving Study Using Secure Computation. PoPETs, 2016(3):117135, 2016.

[BKLS16] Dan Bogdanov, Liina Kamm, Sven Laur, and Ville Sokk. Rmind: a tool for cryptographically secure statistical analysis. IEEE Transactions on Dependable and Secure Computing, PP(99):1–1, 2016.

[BLR14] Dan Bogdanov, Peeter Laud, and Jaak Randmets. Domain-polymorphic programming of privacy-preserving applications. In Proceedings of the Ninth Workshop on Programming Languages and Analysis for Security, PLAS'14, pages 53–65. ACM, 2014.

[BLT14] Dan Bogdanov, Sven Laur, and Riivo Talviste. A Practical Analysis of Oblivious Sorting Algorithms for Secure Multi-party Computation. In Proceedings of the 19th Nordic Conference on Secure IT Systems, NordSec 2014, volume 8788 of LNCS, pages 59–74. Springer, 2014.

[Bog13] Dan Bogdanov. Sharemind: programmable secure computations with practical applications. PhD thesis, University of Tartu, 2013.

[Bog14] Bogdanov, Dan, et al. "From input private to universally composable secure multi-party computation primitives." *Computer Security Foundations Symposium (CSF), 2014 IEEE 27th*. IEEE, 2014.

[BW07] Dan Boneh, and Brent Waters. "Conjunctive, subset, and range queries on encrypted data." Theory of Cryptography Conference. Springer Berlin Heidelberg, 2007.

[Can01] Canetti, Ran. "Universally composable security: A new paradigm for cryptographic protocols." *Foundations of Computer Science, 2001. Proceedings. 42nd IEEE Symposium on*. IEEE, 2001.

[CW79] Larry Carter and Mark N. Wegman. Universal Classes of Hash Functions. J. Comput. Syst. Sci., 18(2):143–154, 1979.

[DN14] Damgård, Ivan, and Jesper Buus Nielsen. "Universally composable efficient multiparty computation from threshold homomorphic encryption." *Annual International Cryptology Conference*. Springer Berlin Heidelberg, 2003.

[DN14] Damgård, Ivan, and Jesper Buus Nielsen. "Adaptive versus static security in the UC model." *International Conference on Provable Security*. Springer International Publishing, 2014.

[GM82] Goldwasser, Shafi, and Silvio Micali. "Probabilistic encryption & how to play mental poker keeping secret all partial information." *Proceedings of the fourteenth annual ACM symposium on Theory of computing*. ACM, 1982.

[HK14] Hahn, Florian, and Florian Kerschbaum. "Searchable encryption with secure and efficient updates." *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014.

[HKI+13] Koki Hamada, Ryo Kikuchi, Dai Ikarashi, Koji Chida, and Katsumi Takahashi. Practically Efficient Multi-party Sorting Protocols from Comparison Sort Algorithms, pages 202–216. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[Intel14] Intel. Software Guard Extensions Programming Reference, 2014. https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf.

[KT13] Küsters, Ralf, and Max Tuengerthal. "The IITM Model: a Simple and Expressive Model for Universal Composability." *IACR Cryptology ePrint Archive* 2013 (2013): 25.

[LTW13] Sven Laur, Riivo Talviste, and Jan Willemson. From Oblivious AES to Efficient and Secure Database Join in the Multiparty Setting. In Applied Cryptography and Network Security, volume 7954 of LNCS, pages 84–101. Springer, 2013.

[LWZ11] Sven Laur, Jan Willemson, and Bingsheng Zhang. Round-Efficient Oblivious Database Manipulation. In Proceedings of the 14th International Conference on Information Security. ISC'11, pages 262–277, 2011.

[MR11] Maurer, Ueli, and Renato Renner. "Abstract cryptography." *In Innovations in Computer Science*. 2011.

[NKW15] Naveed, Muhammad, Seny Kamara, and Charles V. Wright. "Inference attacks on property-preserving encrypted databases." *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015.

[PW00] Pfitzmann, Birgit, and Michael Waidner. "Composition and integrity preservation of secure reactive systems." *Proceedings of the 7th ACM conference on Computer and communications security*. ACM, 2000.

[RS07] Rogaway, P., and T. Shrimpton. "Deterministic Authenticated-Encryption." Advances in Cryptology–EUROCRYPT. Vol. 6. 2007.

[Shamir79] Adi Shamir. 1979. How to share a secret. *Commun. ACM* 22, 11 (November 1979), 612-613. DOI=http://dx.doi.org/10.1145/359168.359176

[PBP16] Poddar, Rishabh, Tobias Boelter, and Raluca Ada Popa. Arx: A Strongly Encrypted Database System. Cryptology ePrint Archive, Report 2016/591, 2016. http://eprint.iacr.org/2016/591.

[PST16] R. Pass, E. Shi, and F. Tramer. Formal abstractions for attested execution secure processors. Cryptology ePrint Archive, Report 2016/1027, 2016. http://eprint.iacr.org/2016/1027.

[YCSB16] YCSB benchmark repository (2016). (https://github.com/brianfrankcooper/YCSB)