



Secure Key Value Store D3.7

Project reference No. 653884

December 2017



European
Commission

Horizon 2020
European Union funding
for Research & Innovation

Document information

Scheduled delivery	31.12.2017
Actual delivery	31.12.2017
Version	1.0
Responsible Partner	INESC TEC

Dissemination level

Public

Revision history

Date	Editor	Status	Version	Changes
10.11.2017	João Paulo	Draft	0.1	ToC
20.11.2017	João Paulo	Draft	0.2	First Draft
25.11.2017	Francisco Maia	Draft	0.3	Revision
26.11.2017	Rogério Pontes	Draft	0.4	Revision
12.12.2017	Dorian Burihabwa	Draft	0.5	Internal Revision
15.12.2017	Miguel Pardal	Draft	0.6	Internal Revision
19.12.2017	João Paulo	Final	1.0	Final Version

Contributors

João Paulo (INESC TEC)
Francisco Maia (INESC TEC)
Rogério Pontes (INESC TEC)

Internal reviewers

Miguel Pardal (INESC ID)
Dorian Burihabwa (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
IV	Initialization Vector
MPC	Multiparty Computation
OPE	Order-Preserving Encryption
DET	Deterministic Encryption
STD	Standard Encryption
SQ1	Secure Queries 1
SQ2	Secure Queries 2
YCSB	Yahoo Cloud Serving Benchmark
AES-GCM	Advanced Encryption Standard – Galois Counter Mode
MPFR	Multiple-Precision binary Floating-point library with correct Rounding

Table of contents


Document information	2
Dissemination level	2
Revision history	2
Contributors	2
Internal reviewers	2
Acknowledgements	2
More information	2
Glossary of acronyms	3
Table of contents	4
Executive summary	6
1 Introduction	7
2 Background	11
2.1 <i>Apache HBase</i>	11
2.1.1 HBase Architecture	12
2.1.2 HBase API.....	13
2.1.3 HBase Coprocessors	13
3 Secure Key Value Store Architecture	14
4 Solution 1: Secure processing in a single untrusted domain	17
4.1 <i>Overview</i>	17
4.2 <i>Architecture</i>	18
4.3 <i>Prototype</i>	20
4.3.1 Setup and Usage	21
5 Solution 2: Secure processing in multiple untrusted domains	23
5.1 <i>Overview</i>	23
5.2 <i>Architecture</i>	23
5.3 <i>Prototype</i>	25
5.3.1 Setup and Usage	25
6 Conclusion	27
7 References	28

List of Figures

Figure 1 - The SafeCloud framework.....6
Figure 2 - Overall architecture for secure queries.....8
Figure 3 - Secure Queries Solution 1 architecture.....9
Figure 4 - Secure Queries Solution 2 architecture..... 10
Figure 5 - HBase map structure..... 11
Figure 6 - HBase logical table view. 11
Figure 7 - HBase Architecture. 12
Figure 8 - Solutions 1 and 2 architectural components 14
Figure 9 - CryptoWorkers in the trusted site..... 16
Figure 10 - CryptoWorkers in the untrusted site. 17
Figure 11 - Solution 1 prototype architecture. 19
Figure 12 - HBase deployment configuration. 22
Figure 13 - YCSB deployment configuration..... 22
Figure 14 - Solution 2 prototype architecture. 24
Figure 15 - HBase deployment configuration 26
Figure 16 - YCSB deployment configuration..... 26

Executive summary

The framework proposed by SafeCloud consists of three layers: secure communication, secure storage, and secure queries. This deliverable is about the secure queries layer that provides cryptographic constructions from the database storage layer to the end-user processing requests. The overarching idea is to allow system developers to use the techniques provided by these three layers in order to achieve application-specific deployments. These deployments should surpass the state-of-the-art of existing tools with respect to functionality, performance and security. We recall Figure 1, from the general SafeCloud framework description.



SafeCloud architecture

Secure communication	State of the art: TLS secure channels	Solution:	Vulnerability-tolerant channels	Protected channels	Route-aware channels
		<i>Gives:</i>	Tolerance to vulnerabilities in components	Decreased risk of fake certificates; resistance to port scans and enumeration of network infrastructure	Improved confidentiality with warnings about route hijacking and making harder access to communication
<i>API:</i>	Extended secure socket API	Extended secure socket API	Extended secure socket API		
<i>Provided by:</i>	INESC-ID, TUM	INESC-ID, TUM	INESC-ID, TUM		
Secure storage	State of the art: Encrypted storage	Solution:	Secure block storage	Secure data archive	Secure file system
		<i>Gives:</i>	Block storage on individual data centers with fine control over data placement	Entangled immutable data storage for protection against tampering and censorship	Distributed secure file storage leveraging the secure block storage
<i>API:</i>	Key/value	REST (S3 or similar)	POSIX-like		
<i>Provided by:</i>	UniNE, INESC TEC	UniNE, INESC TEC	UniNE, INESC-ID		
Secure queries	State of the art: CryptDB	Solution:	Secure processing in a single untrusted domain	Secure processing in multiple untrusted domains	Secure processing in multiple untrusted domains with untrusted clients
		<i>Gives:</i>	Privacy of data against the server	Privacy of data against non-colluding servers	Privacy of data against non-colluding servers and clients
<i>API:</i>	SQL	SQL	SQL		
<i>Provided by:</i>	INESC TEC	INESC TEC, Cyber	Cyber		

Figure 1 - The SafeCloud framework.

In specific, this deliverable presents the final prototypes for the SafeCloud’s Secure Key Value Store components. These are fundamental components for the Secure Queries solutions 1 (Secure processing in a single untrusted domain) and 2 (Secure processing in multiple untrusted domains) proposed in WP3. In fact, the SQL engines of both solutions resort to the specific Key Value Store component to provide secure processing capabilities for the WP5 use-cases.

The proposed architectures and implementations are modular, which facilitates the integration of multiple cryptographic techniques in our solutions. In this deliverable, we present concrete examples of how different encryption techniques (i.e., standard encryption, deterministic encryption, order-preserving encryption) and multi-party computation plus secret sharing are employed in our prototypes. Finally, we also detail how the final prototypes can be configured and deployed for testing and production purposes.

1 Introduction

The SafeCloud project structure considers three main layers: secure communication, secure storage, and secure queries (or secure data processing). Secure communication provides solutions for the establishment of secure channels that ensure confidentiality and availability against adversaries that are more powerful than usually assumed. Secure storage provides techniques for reliable storage, such as long-term confidentiality, protection against file corruption, censorship or data deletion. Finally, secure queries provide cryptographic constructions from the database storage layer to the end-user data processing requests. The overarching idea is to allow system developers to combine implementations by employing these three layers to achieve application-specific deployments that choose different trade-offs between functionality, performance and security.

In this document, we will focus on the secure data processing and, in particular, in one of the key components that is used to implement the solutions of this layer: the Secure Key Value Store. Namely, to provide detailed and updated descriptions of the final NoSQL prototypes versions, we expand the descriptions of the architecture, privacy-preserving techniques and initial prototype presented in the following deliverables:

- **D3.1 - Privacy-preserving storage and computation architecture.** This report describes the architecture for the three solutions of the secure queries layer.
- **D3.2 - Privacy-preserving storage and computation techniques.** This deliverable presents the different privacy-preserving techniques (e.g., multi-party computation, standard encryption, deterministic encryption, order-preserving encryption) that are used in the secure queries layer solutions.
- **D3.3 - Non-elastic Secure Key Value Store.** This report presents the initial prototypes of the Key Value Store component for Secure Queries solutions 1 and 2.
- **D3.5 - Secret-sharing and order-preserving encryption based private computation.** This deliverable provides an extensive discussion about the privacy-preserving techniques used in the secure queries layer while pointing the different tradeoffs of each technique in terms of performance, security and functionality.
- **D3.6 - Elastic privacy-preserving storage and computation.** This deliverable discusses how elasticity is handled in each Secure Queries solution.

Recalling the Secure Queries layer architecture and description, three concrete solutions were proposed in the context of the project, all of them following the overall architecture depicted in Figure 2.

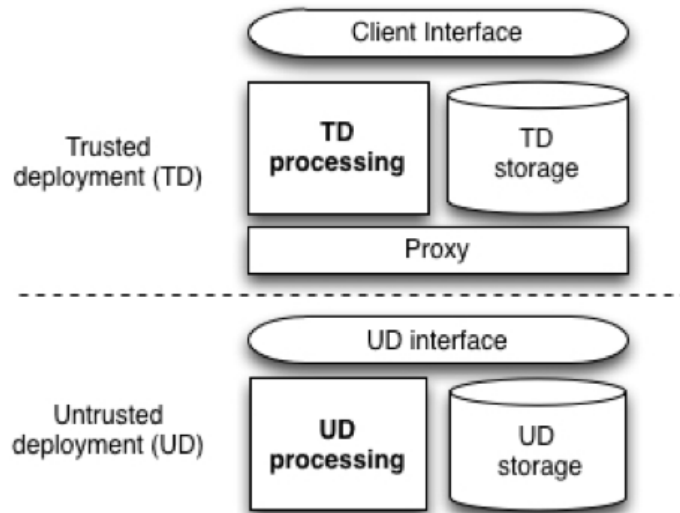


Figure 2 – Overall architecture for secure queries.

As observable in the figure, the architecture identifies two distinct sites where software components will be deployed: a *trusted* and an *untrusted* one. In both, storage and processing components can be deployed. Depending on the type of components chosen and the configuration of the sites, three different solutions emerge:

- **Secure Queries Solution 1 (SQ1):** Secure processing in a single untrusted domain
- **Secure Queries Solution 2 (SQ2):** Secure processing in multiple untrusted domains
- **Secure Queries Solution 3 (SQ3):** Secure processing in multiple untrusted domains with untrusted clouds

SQ1 and SQ2¹ aim at offering full SQL language support but require that most of the query processing is done on the trusted site. SQ3 focuses on different workloads and use cases, offering secure query processing mostly done on the untrusted sites but with limited SQL support. Moreover, Solution 1 and 2 distinctly separate the type of processing done in the trusted and untrusted sites. In fact, delving further in the concrete architecture for the different solutions, there is a common design feature in Solution 1 and 2. Both solutions rely on a key value store (NoSQL data storage system) component that is independent from the query processing components (the NoSQL data store components are delimited by the blue boxes in Figure 3 and Figure 4). Throughout this document, our focus will be the key value store components of Solution 1 and Solution 2, describing how this component is instantiated in a concrete implementation, and how it can be set up and used.

¹ In the document SQ1, SQ2 and SQ3 are also referred as Solutions 1, 2 and 3 respectively.

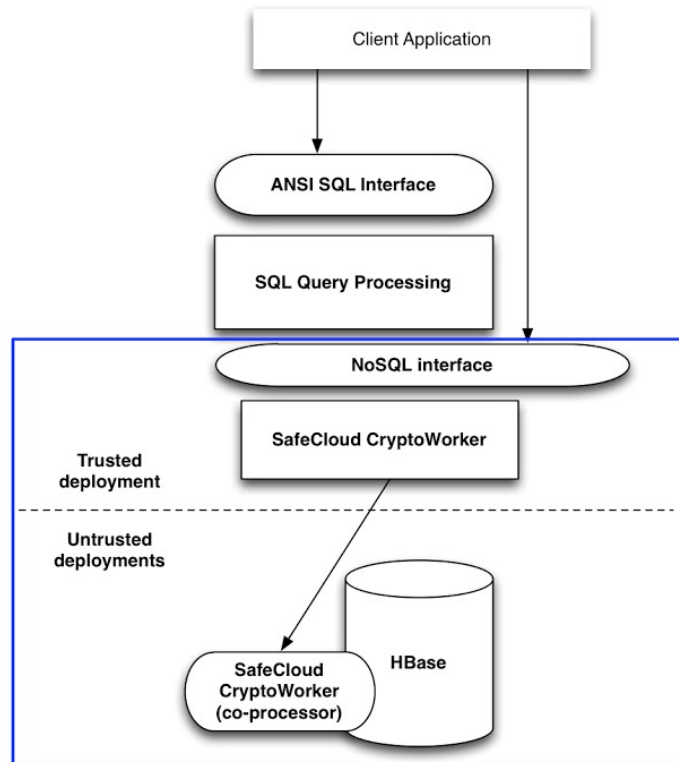


Figure 3 - Secure Queries Solution 1 architecture.

It is important to notice that, as explained in deliverables D3.1 and D3.3, our implementation of the secure key value components of Solutions 1 and 2 relies on Apache HBase [APACHE17a], which is considered a mature NoSQL databases available on the market. This decision was based both on the feasibility of solutions deployed over such a widely used technology, as well as on the high level of experience and expertise held by the SafeCloud’s partners with this system. Nevertheless, the proposed secure framework architecture is generic to allow a possible integration with other NoSQL databases e.g., Cassandra [APACHE17c], Redis [REDIS17].

The secure Key Value Store prototypes discussed in this document are designed while having in mind with the requirements of MaxData’s CLINIdATA® healthcare application for the following deployment scenarios:

- **SaaS deployment:** for small and medium-scale healthcare organizations that want to reduce costs on infrastructure, CLINIdATA® will be offered using the software- as-a-service (SaaS) model where all components are deployed on cloud providers contracted by Maxdata.
- **Hybrid deployment:** for large healthcare organizations that want to reduce costs on infrastructure but do not trust any cloud provider, CLINIdATA® computation/processing will be installed on customer’s premises making use of SafeCloud components to access data securely stored on untrusted cloud providers contracted by healthcare organizations.

Deliverable **D5.2 - Design and requirements, Maxdata SafeCloud-based healthcare platform** describes each of these cloud deployment scenarios in more detail, including the way CLINiDATA® will leverage the SafeCloud framework to achieve the desired goals.

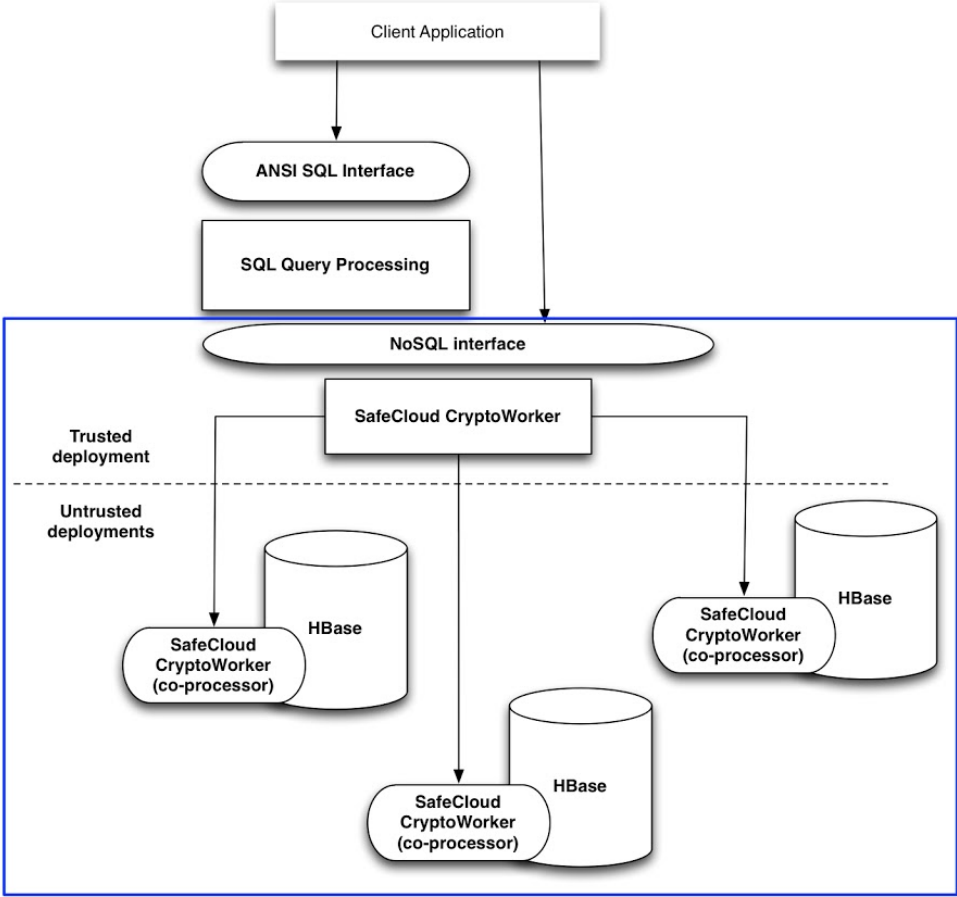


Figure 4 - Secure Queries Solution 2 architecture.

The software components described in this deliverable are available for demonstration upon explicit request. These components are not open source software to be inline with the project’s exploitation plan. To get more information and place requests please go to <http://www.safecloud-project.eu>.

The remainder of the document is structured as follows. We begin by providing a background on Apache HBase in Section 2. In Section 3 we describe the high-level architecture and design decisions for our Secure Key-Value Store solutions. We then instantiate this generic design as Solution 1 in Section 4 and as Solution 2 in Section 5. We conclude the document providing some remarks regarding these final prototypes.

2 Background

2.1 Apache HBase

Apache HBase is a distributed, scalable and open-source non-relational database [APACHE17a]. Inspired by Google's BigTable [CDG+08], it can be thought of as a multi-dimensional sorted map or table. The map is indexed by a tuple composed by row key, column family, column qualifier and timestamp, which is used as a key for a given value, as illustrated in Figure 5.

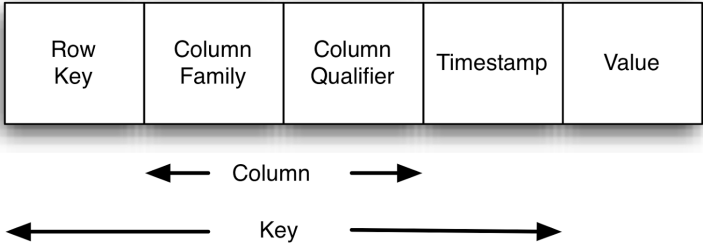


Figure 5 - HBase map structure.

Row keys might be associated with an unbounded and dynamic number of qualifiers (columns) grouped into column families (groups of columns). As an example, in an HBase table storing information of a given company's employees, "employee" may be a column family and the employee's "name", "age", "salary" can be distinct column qualifiers grouped by that column family. Each qualifier is then identified by concatenating its column family's name and qualifier, i.e, family:qualifier. A number of rows form a table, and each row may specify a distinct number of column families. Both the row key and the associated values are arbitrary, non-interpreted arrays of bytes. Data is maintained in a lexicographic order: first by row key, second by column's family name followed by qualifier and, in descendent order, by timestamp.

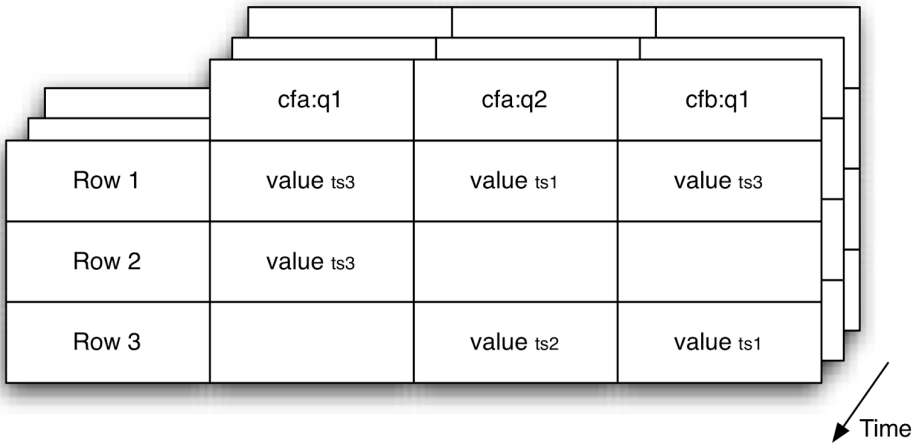


Figure 6 - HBase logical table view.

A logical view of an HBase table is presented in Figure 6. In this view, each cell (value) is the intersection of a row key, a column family (cf) and a column qualifier (q).

Additionally, timestamps (ts) may be used to have a multi-dimensional table, since it means that several versions may exist simultaneously. From this point onwards, we will use the term “row” to denote a single row according to this logical view. Typically, column families (cf) are well-defined and must be created before data can be stored. In contrast, qualifiers are created at runtime by inserting new key-value pairs.

2.1.1 HBase Architecture

Figure 7 depicts the HBase architecture and its main components.

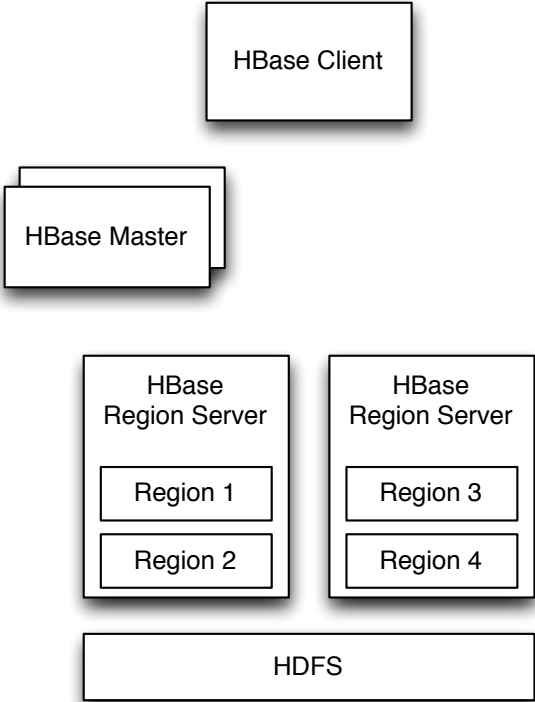


Figure 7 - HBase Architecture.

An **HBase client** component is provided so that applications are able to perform queries, following the HBASE API, to the HBase backend (combination of HBase Masters and Region Servers). Succinctly explained, the HBase client contacts the Master component to know what Region Servers are responsible for storing the rows for a specific request. After knowing such information, the HBase client issues the request to the appropriate Region Servers that reply back with the query results.

The **HBase Master**, is responsible for redirecting HBase client requests to the appropriate Region Servers, where the keys being stored/retrieved are kept. The HBase Master may be deployed in a primary/secondary replication mode to ensure a fault-tolerant design via redundancy.

Rows in a table are partitioned horizontally² and each partition is called a Region. After partition, resulting regions can be distributed across several nodes named

² Horizontal partitioning divides a table into multiple ones. Each table then contains the same number of columns, but fewer rows. In HBase, the rows of a table can be stored (partitioned) in different Region Servers.

RegionServers that are responsible for serving one or more regions. All columns and values of a given key (row) are available in the same Region Server. Regions Servers store and retrieve Regions' data using the Hadoop Distributed File System (HDFS) [APACHE17b]. Ideally, each Region Server is collocated in the same machine with the HDFS data node serving the data for the Regions belonging to that Region Server. This option promotes data locality and allows Regions Servers to have a more efficient access to their data. In addition, this capability of partitioning data horizontally across different Region Servers provides HBase with a scalable design. In other words, by launching additional Region Servers and partitioning data across these, it becomes possible to serve requests from an increasing number of clients while maintaining the NoSQL database's throughput and latency.

2.1.2 HBase API

HBase exposes a set of operations for data access that is quite similar to the one used in other key-value data stores. This interface is provided by the HBase client component and encompasses the following operations:

- GET - Get key-value pairs of a given row, identified by row key;
- PUT - Insert or update a key-value pair for an existing or a new row;
- SCAN - Get all key-value pairs for a specific range of rows;
- DELETE - Remove one or more key-value pairs belonging to one or more rows;

Note that, for GET and PUT requests, it is only possible to retrieve or update specific column qualifiers if the requests specify the row key, column family and column qualifier being targeted. Additionally, HBase provides filter operations for both GET and SCAN requests. Namely, it is possible to request several key-value pairs with a scan request and then to filter only the key-values where a specific column has a certain value. As an example, if a company stores in HBase information about its employees, it is possible to query all employees with identification numbers (row key) between number 100 and 1000, and then to filter the request to only retrieve the entries for employees that were born in 1986 (assuming that age is a column qualifier).

2.1.3 HBase Coprocessors

The computation done at the HBase Backend can be extended with novel functionalities, without modifying the core implementation of HBase, by using the HBase coprocessor mechanism. Coprocessors can be seen as plugins that are implemented and added to HBase backend extending its capabilities [APACHE12].

Two types of coprocessors are available: *observers* and *endpoints*. Observer coprocessors bind a piece of code to system events. For instance, they may be used to add access control when a client requests a GET operation. Furthermore, endpoint coprocessors can also extend the client-server protocol communication with arbitrary remote code execution through Remote Procedure Calls (RPC). This kind of coprocessors is similar to stored procedures of traditional relational databases.

For some of the SafeCloud solutions, observer and endpoint coprocessors are essential because additional computation must be done at the HBase backend. For instance,

multi-party computation protocols (MPC) require performing computation over secrets stored at the HBase backend [PMP+16]. As further explained in Deliverables 3.1, 3.2 and 3.5, this computation is essential for supporting GET and SCAN requests when row identifiers are protected (private). Moreover, MPC requires computing over stored secrets and exchanging the computation results with other parties (HBase clusters), which also needs to resort to coprocessors. To sum up, when a GET request is issued by the client, coprocessors will be key for performing the additional computation and exchange results across parties, before replying back to the client. Finally, coprocessors are also required for SafeCloud SQ1 solution when the cryptographic techniques being used require maintaining some sort of index at the HBase backend in order to do computation over private data. The access to the index and computation to process a request also requires coprocessors [PRZ+11]. Without this mechanism, we would have to change the core implementation of HBase backend components, which would significantly increase the implementation and maintenance costs of our solution.

3 Secure Key Value Store Architecture

In order to provide usable and manageable solutions we designed our solution in such a way that completely avoids changing the Apache HBase core implementation. This allows our system to be compatible with evolving versions of HBase and allows its own development to be independent of HBase releases or roadmap. Additionally, it also renders the process of changing the underlying system from HBase to another NoSQL store easier. To achieve this design, we base our approach in software components that can be placed in the middle of the normal HBase workflow and that actively modify such workload transparently from the perspective of the client application and HBase itself. These components are deployed both in the untrusted and the trusted sites, and are named *CryptoWorkers*.

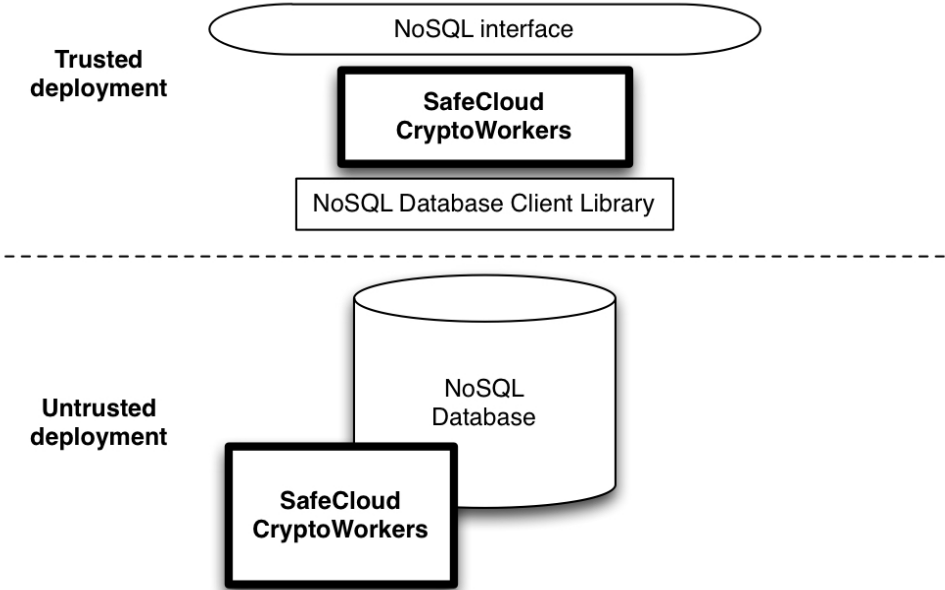


Figure 8 - Solutions 1 and 2 architectural components.

In Figure 8 we depict the organization of the different components that are used for the secure key value stores prototypes supporting SQ1 and SQ2 solutions. It is important to

note that, for Solution 2 (secure multi-cloud database server), the components of the untrusted deployment are instantiated multiple times to account for the multiple untrusted domains that are considered in such solution. These details will be addressed in subsequent sections. For now, we briefly describe the design of the *CryptoWorkers*.

From a high-level perspective, *CryptoWorkers* are responsible for two tasks:

- **Trusted site:** *CryptoWorkers* transform plain NoSQL operations in secure NoSQL operations according to the requested privacy technique;
- **Untrusted site:** *CryptoWorkers* add extra behavior to allow data processing over encrypted data. Depending on the privacy technique in use, this may require the addition of extra communication steps.

It is important to notice that, in our prototype, HBase is the NoSQL database and *CryptoWorkers* currently take advantage of its specific characteristics such as co-processors. However, the software itself does not depend on HBase, making our design extensible and potentially compatible with other NoSQL databases. This HBase-agnostic approach allows general cryptographic mechanisms to overlay the data storage and management in a close to black-box way, which can afterwards be instantiated and optimized according to the specific circumstances in which they are deployed, e.g. in our case employing co-processors for performing general remote computations.

In Section 2 we have detailed the HBase API that is composed of three main operations: PUT, GET and SCAN. As illustrated in Figure 9, the basic functionality of the *CryptoWorker* is to translate those plain data requests into *secure requests*. Secure requests are encoded according to some associated cryptographic technique (e.g., standard encryption, deterministic encryption), so that when they are transmitted to some untrusted (potentially adversarial) site, the required security guarantees will hold. The encryption techniques used for protecting a specific request are defined in a configuration file. For instance, for a certain GET operation issued in the trusted deployment, the *CryptoWorker* will issue one or more concrete HBase operations (via the HBase Client library) to the HBase backend running on the untrusted deployment. That set of operations will ensure that such GET operation is made with specific data privacy guarantees according to the *CryptoWorker* configuration, which specifies the protocol encoding the data.

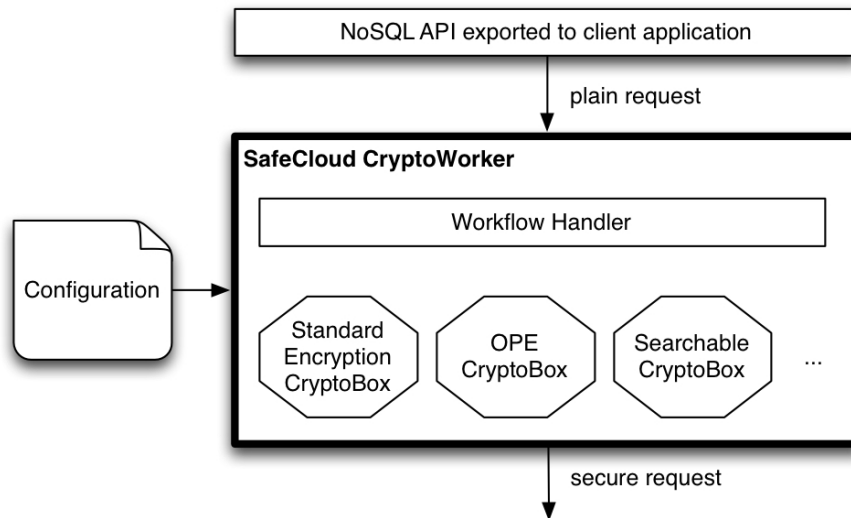


Figure 9 - CryptoWorkers in the trusted site.

Each CryptoWorker on the trusted deployment is responsible for a specific workflow that allows the translation of plain requests to secure requests and their correspondent replies with respect to a specific cryptographic technique or set of techniques. In order to manage the workflow, each cryptographic technique employs a set of operations. These operations will take care of requests in a secure fashion, involving three general sequential operations: a client-side encode, a server-side operation, and a client-side decode. Each *cryptobox* offers a set of cryptographic implementations that are useful in certain privacy workflows. In other words, a *cryptobox* can be seen as a black box that offers a specific encryption scheme. For instance, in order to provide standard encryption over HBase it is necessary that a plain request content is encrypted using AES-GCM. Consequently, a SafeCloud standard encryption *cryptobox* is made accessible by CryptoWorkers.

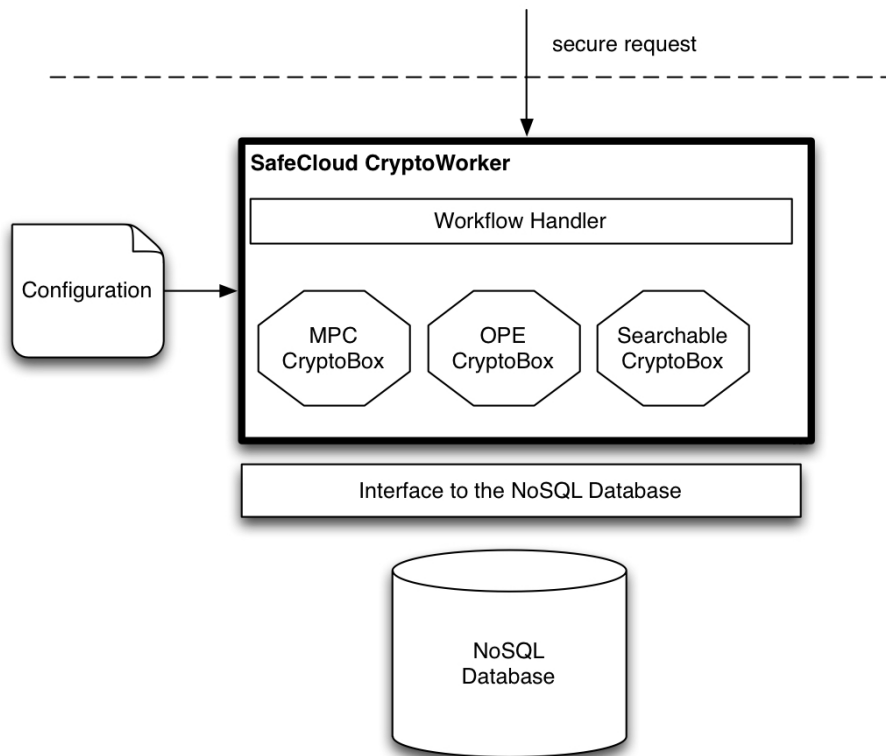


Figure 10 - CryptoWorkers in the untrusted site.

Similarly, in the untrusted domain, *CryptoWorker* co-processors provide additional behavior that allows the system to reply according to the data privacy level required (Figure 10). For example, CryptoWorkers that use techniques such as multi-party computation provide additional computation over encoded data and communication steps that must be added to the untrusted NoSQL backend in order to comply with the protocols. The specific operations used and the different configurations that are considered in the current design of the system are described in detail in deliverable D3.5 (Sections 2 to 4).

In the context of the present document, it suffices to state that HBase itself remains unmodified in our design and that our system simply adds functionality to the HBase system by translating typical NoSQL operations to operations that are data privacy-aware. A more in-depth discussion of the security techniques, their deployment within SafeCloud, and the associated security guarantees is provided in Deliverable D3.5. In the following sections, we focus on the actual system prototype and how it can be installed, run and tested.

4 Solution 1: Secure processing in a single untrusted domain

4.1 Overview

Solution 1 of the secure processing layer of SafeCloud focuses on providing secure data processing when a single untrusted domain is used. This corresponds to the typical use of a single cloud provider for data storage and processing. As noted previously, this solution relies on a secure key value store component that is composed of both trusted and untrusted deployment sub-components.

In this Section, we describe the prototype for the HBase secure key value store. This prototype was designed to allow for the configuration of the different privacy preserving techniques described in D3.5. These techniques include order preserving encryption, deterministic and standard encryption.

4.2 Architecture

The architecture of the final prototype for Solution 1 follows the originally planned architecture. In terms of implementation, it supports all the techniques mentioned in the previous WP3 deliverables (i.e, standard encryption, deterministic encryption, and order-preserving encryption). The architecture was designed with strong modularity concerns, which allows the prototype to be ready for quick integration with the SQL Engine components developed along in the project. In Figure 11 we present the concrete architecture for the prototype.

The prototype follows the architecture presented in the previous sections and, as described previously, uses Apache HBase. The SafeCloud *CryptoWorker* code works as an extension of the HBase client library and, on the server side, runs in co-processors at each HBase Region Server. Looking at the specific case of solution 1, we consider only one remote site with a single HBase deployment, which in turn can hold multiple Region Servers. Region Server management is left to HBase itself and SafeCloud co-processors only extend their behavior.

Figure 11 depicts the basic workflow for the deterministic encryption case that serves as an illustrative example of how the solution 1 prototype works. Note however that the configuration file, present at the client premises, can specify different combinations of the supported cryptographic techniques for protecting specific database values. As an example, suppose the database is holding sensitive information regarding the appointments of patients for a given Hospital. In order to keep the *Date* of the appointments private while still being able to query all the appointments for a specific period of time, one can resort to order-preserving encryption as the protection technique for this *Date* field. On the other hand, for searching the appointments of a specific patient by her *Name*, the *Name* database field can be protected with deterministic encryption. The combinations of cryptographic techniques and the database fields to which these are applied to, are specified in a configuration file.

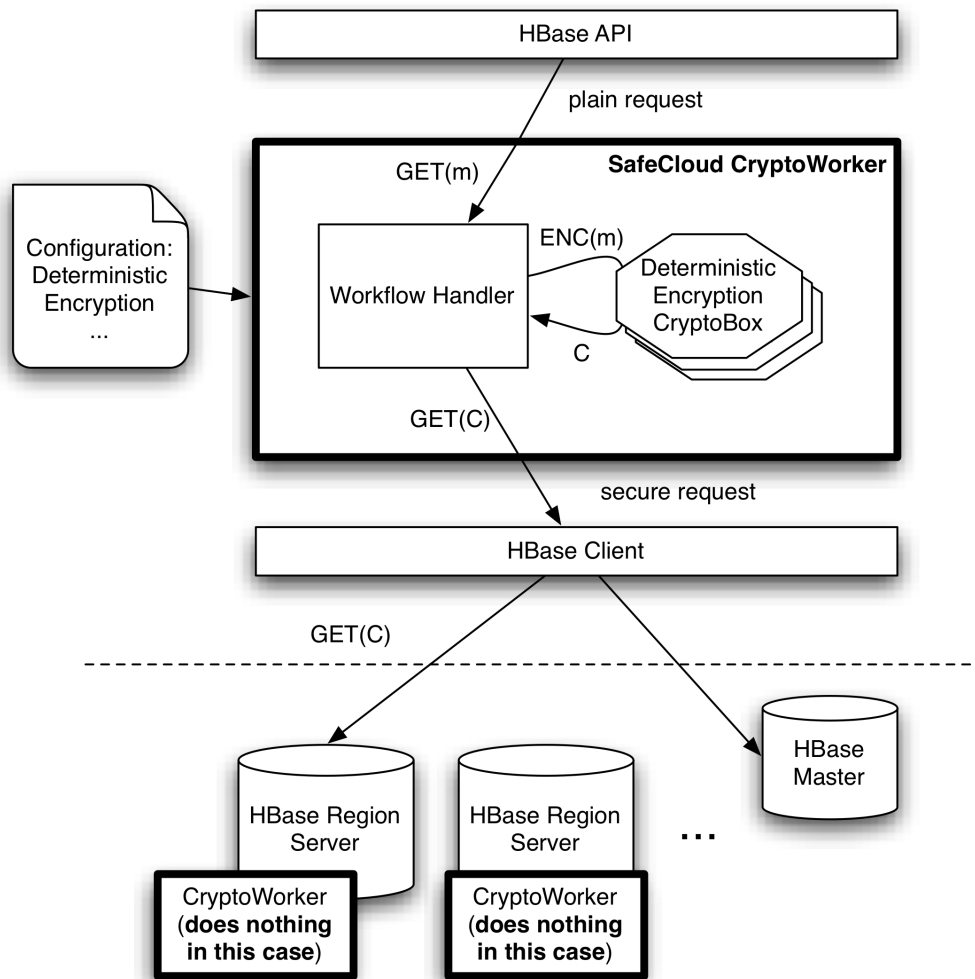


Figure 11 - Solution 1 prototype architecture.

Going back to the example depicted in Figure 11, upon a request, the *CryptoWorker*, configured to use deterministic encryption, knows it has to instantiate a deterministic encryption *cryptobox* to deal with cryptographic operations. It also knows that requests contain plain (unencrypted) values that do not make any sense to the HBase server, which only contains encrypted data. Accordingly, the *CryptoWorker* uses the *cryptobox* to cipher the request values in order to be able to issue secure operations to the remote HBase server. Because we are using deterministic encryption, equality operations are still valid over the encrypted data. As a consequence, the *GET* operation can still be performed normally and no extra server-side behavior is needed. However, the *scan* operation requires relative order to be maintained in the encrypted data, which is not the case for deterministic encryption. In fact, this operation requires all data that could possibly answer the scan operation to be retrieved and processed in the trusted site. Observe that this specific instantiation of deterministic encryption still follows the more abstract API proposed in Deliverable D3.5, since there is an initial step of computation on the client-side encrypting data, a step of server-side operations for storing/retrieving data, and a final step for decoding upon obtaining encrypted values. There is no extended behavior deployed on the server side (at the coprocessors) that can prevent this overhead without compromising deterministic encryption privacy levels. This will

significantly impact the performance of the prototype and such impact is different when using different techniques, as we will see further on. This substantiates our initial claim that there is no solution that fits all cases. Compromises between privacy levels and performance must be taken into account when choosing a specific privacy technique or set of techniques.

In terms of scalability, as further discussed in deliverable Deliverable D3.6, SQ1 leverages HBase horizontal partitioning to scale for deployments with an increasing number of clients. The encryption techniques supported by our prototype do not have an impact in the original partitioning design, which allows the solution to inherit the scalability of NoSQL solutions.

The workflow presented for Solution 1 is somewhat straightforward, which will get gradually more complex as *CryptoWorkers* have to deal with more convoluted workflows, as detailed in deliverable D3.5. An example of this is the *CryptoWorker* deployed in the prototype for Solution 2.

4.3 Prototype

The final NoSQL prototype of SQ1 follows the architecture described in the previous section and, as one of the main concerns, it follows a modular and flexible approach where several cryptographic techniques can be easily integrated in the future.

In more detail, the final prototype contemplates three distinct *CryptoBoxes*. The standard encryption *CryptoBox* relies on OpenSSL [OPENSSL17] cryptographic library. The deterministic encryption is implemented in accordance to the construction described in [RS07]. Finally, the Order-Preserving Encryption *CryptoBox* is implemented following the design of [BCL+09] and it relies on OpenSSL and MPFR [FHL+07], a multiple-precision floating-point library. The implementation of these encryption techniques allows supporting the vanilla HBase operations *i.e.*, PUT, GET, SCAN, DELETE and Filters.

During the experimental evaluation of the prototype, we noticed that decoding information encrypted with order-preserving encryption has a significant penalty in the latency and throughput of HBase operations. As such, we propose an optimization that trades additional storage space for a considerable performance improvement. In our system, every column qualifier encrypted with order-preserving encryption will be accompanied by the same value protected with standard encryption. Then, when a value protected with order-preserving encryption must be retrieved by the client, instead of decoding the order-preserving encryption, the *CryptoWorker* module decodes the value protected with standard encryption instead, which is considerably faster. For instance, decoding a 14 byte length ciphertext with order-preserving encryption takes 567.434 μ s and with standard encryption takes 5.884 μ s. Moreover, for a 256 byte ciphertext, order-preserving encryption takes 2.861s to decode while standard encryption takes 8.028 μ s. As this optimization still contemplates the storage of data encrypted with order-preserving encryption, filtering operations such as equality or range queries are also supported.

Note that the main advantage of our architecture and implementation is that it can be easily integrated with other techniques such as searchable encryption. Implementation-

wise, the supported techniques do not require any additional computation at the HBase backend (co-processors) since the *CryptoWorker* deployed at the trusted infrastructure is responsible for ciphering and deciphering the data as well as doing the additional computation required to support the full HBase API. However, for supporting cryptographic techniques that require state/computation at the HBase backend, e.g. maintaining some obfuscated function for comparison, we can employ co-processors in a similar fashion to the approach described in Solution 2, where this mechanism is necessary.

4.3.1 Setup and Usage

SQ1 NoSQL prototype is currently installable through docker containers [DOCKER16]. Docker containers are isolated components that possess a configuration file where it is detailed what are the software packages that must be installed and how to deploy these components. This allows having a stable “recipe” that establishes a docker setup for automatically setting up our SafeCloud solution.

In our docker containers we have the required configuration to install HBase components, our HBase client supporting the *CryptoWorker*, and all the necessary dependencies. Currently we have two docker images relevant to this solution in the docker public registry. The first image is an HBase backend on a standalone configuration. This configuration deploys all the necessary HBase components on a single server³. The second image contains the necessary dependencies to use our HBase Client, plus our model for the *CryptoWorker*.

Finally, to perform an evaluation of our proposal, we integrated our final prototype (M28) with the Yahoo Cloud Serving Benchmark (YCSB) [YCSB17], a widely used framework to evaluate the performance of different NoSQL databases. For these experiments, we chose workloads that were inspired on the Health WP5 use-case that will leverage SQ1. Briefly, the workload mimics a typical Hospital Database that must store query information about patients and their medical appointments. Most of this information contains sensitive information about patients (e.g., name, address, date of the appointments). All this information is protected with the appropriate techniques (i.e., standard encryption, deterministic encryption and order-preserving encryption) in a way that it is possible to query such data without disclosing any sensitive information from it, even if the infrastructure where the NoSQL database is deployed becomes compromised. The experimental results show that by combining different cryptographic techniques, it is possible to have a practical solution that balances the desired functionality, performance and security for different applications. In average, when compared with a baseline HBase deployment without any data privacy guarantees, our prototype introduces less than 15% of performance overhead across the different realistic workloads tested. Further details on these experiments are discussed in [MPP+17].

SafeCloud docker containers can be deployed either on a distributed cluster or on a single machine. As there are a variety of docker orchestration tools that can be used to deploy containers in a distributed setting, this Section only presents the steps required

³ This setup is used for demonstration purposes as it allows the evaluation of our solutions in a more contained and easy to run setup. Production-ready versions are available upon explicit request

to do a local deployment. A local deployment only requires an operating system with a docker engine and the deployment steps required are the same for every platform. Nonetheless, the process of installing a docker engine also changes depending on the operating system. As this process is beyond the scope of this document, we refer the installation process to the docker official website documentation [DOCKER17]. An additional tool, docker-compose [DOCKERC17], is also required to simplify the deployment process. Such tool allows defining in what machines the SafeCloud components will be deployed and the necessary network configuration so that these different components can interact with each other.

Besides the referred essential tools, two configuration files are required. Figure 12 and Figure 13 display the exact content that each file must have. These files incorporate the required configuration to deploy the two docker images on a local server.

```
baseline:
  image: safecloud/hbase:standalone
  net: ncwork
  hostname: baseline
  container_name: baseline
```

Figure 12 - HBase deployment configuration.

```
yccb:
  image: safecloud/yccb:sq1
  net: ncwork
  hostname: yccb
  container_name: yccb
```

Figure 13 - YCSB deployment configuration.

To deploy and test our current demonstrator the following steps are necessary:

- Write the contents of Figure 12 to a new file *hbase.yml*
- Write the contents of Figure 13 to a new file *yccb.yml*
- Create a new docker network named *ncwork* with the following command:
docker network create ncwork
- Deploy the hbase backend with the following command:
docker-compose -f hbase.yml up
- Deploy the yccb image with the following command:
docker-compose -f yccb.yml up

After executing these commands the YCSB benchmark starts issuing requests to the database and, after completing the benchmark, an output with the achievable latency and throughput for requests is displayed.

5 Solution 2: Secure processing in multiple untrusted domains

5.1 Overview

Similarly to SQ1, this solution is based on a secure Key Value Store backend. However, due to the different security model of multi-party protocols that defines SQ2, the system architecture requires multiple untrusted domains and an additional discovery service. Nevertheless, the high-level approach taken is similar to the one previously described for SQ1, i.e., we do not modify HBase itself but rather externally add the necessary behavior via the appropriate embedding mechanisms.

For the particular case of SQ2, we consider three independent untrusted clusters, each with a separate HBase deployment. The added behavior enables each HBase instance to provide secure operations by following multi-party computation algorithms. Moreover, this entails the requirement for communication capabilities between the different entities since this is a fundamental necessity for general multi-party protocols.

In the following Sections, we present the prototype for the multi-party computation HBase solution.

5.2 Architecture

The concrete architecture for this second final prototype is presented in Figure 14. As expected, this prototype considers three different remote deployments, where three different HBase clusters are deployed. Supporting distributed HBase clusters with multiple Region Servers that play the role of a party in multi-party protocols requires an additional *discovery service*. This service supports dynamic and scalable clusters with different configurations as it enables region servers of a cluster to learn with which region servers from the other clusters they should communicate (i.e., the ones that have the corresponding data secrets) and correctly evaluate an MPC protocol. This discovery service can be deployed in any one of the clusters and is transparent to the client. More details on how this discovery service is key for enabling the scalability of the NoSQL prototype are available in Deliverable D3.6.

A client-side *CryptoWorker*, at the trusted site, is instantiated and configured with the appropriate multi-party computation parameters and workflow. In contrast with the previous prototype, the backend-side *CryptoWorkers* that take part in the MPC are now responsible for request processing and engage in a specific workflow to be able to address the data requests from the client. In particular, they follow multi-party computation protocols that require communication between the three *CryptoWorkers* deployed in the three different untrusted sites.

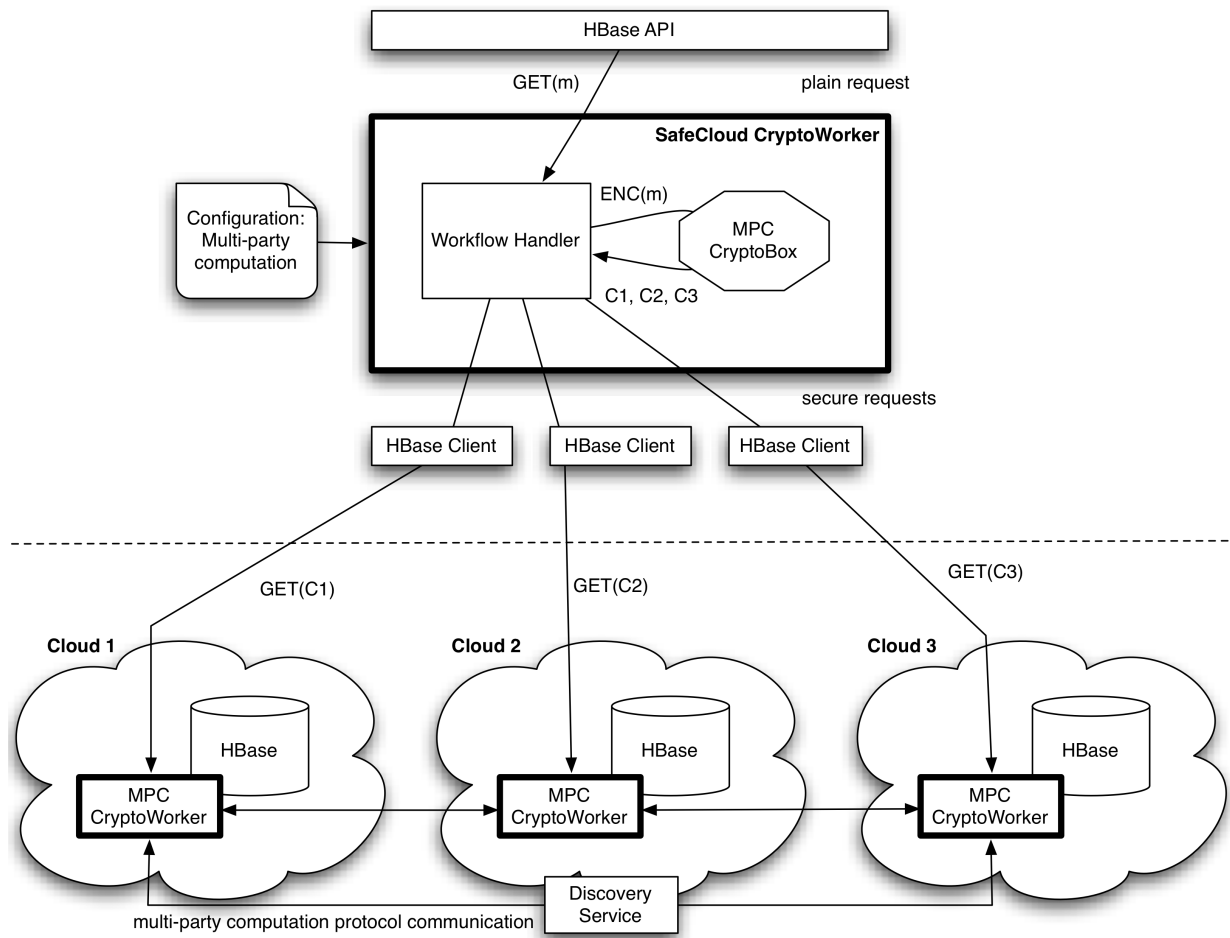


Figure 14 - Solution 2 prototype architecture.

Briefly, a GET operation is encoded into three different operations to be issued to the three independent backends. When a backend-side *CryptoWorker* identifies such request as a multi-party computation request, it starts a specific set of procedures to process it. This typically includes querying the local HBase instance for data, communicating with the discovery service to identify which region server is handling the same request, then connect to the other backend-side *CryptoWorkers* and, finally, replying to the trusted site. The client-side *CryptoWorker* processes the answers and translates them into a plain data reply to the client application.

The computation and communication steps necessary for this workflow are thorough and subtle, since they depend on the underlying complex cryptographic mechanisms that allow for general secure computation of functionalities over several participants. For a more in-depth overview of MPC protocol details, please refer to deliverable D3.5.

5.3 Prototype

The final demonstrator of Solution 2 follows the architecture described in the previous section. In more detail, the prototype supports multi-party computation over secret shared data by implementing CryptoWorkers both at the trusted domain (HBase client) and untrusted domain (HBase coprocessors). The MPC CryptoBox used by these CryptoWorkers is written in Java and is a tailored implementation of Sharemind protocols, thus leveraging previous knowledge of the CYBERNETICA project partner. The communication done across CryptoWorkers in different HBase clusters is done by resorting to a middleware component implemented using Java NIO.

The implementation of this technique allows supporting the vanilla HBase operations *i.e.*, PUT, GET, SCAN and DELETE and filters. Furthermore, unlike the initial implementation, this version already supports concurrent client requests and batch operations. Note that the architecture used in this prototype is based on the one used for SQ1 so it is still highly modular and, any other security techniques that resort to multiple untrusted domains could be integrated with Solution 2.

5.3.1 Setup and Usage

The deployment of the NoSQL component of SQ2 follows a very similar approach to SQ1. It uses the same tools and technologies. However, the docker images and configurations are updated to reflect the multiple backend architecture of this solution. Three instances of the HBase backend components have to be instantiated with different configurations. As such, Figure 14 contains the configuration file that must be used to deploy three HBase clusters, each with an integrated *CryptoWorker* module, which are named cluster1, cluster2 and cluster3. To deploy the YCSB benchmark and HBase client plus *CryptoWorker* bundle, only a minor detail must be changed. Instead of using the tag *aes*, the tag *mpc* must be used instead, as shown in Figure 15.

To deploy this solution⁴, it is assumed that a docker engine and the docker-compose tool are installed on a single machine as in the deployment of solution 1. Again, this section only presents the steps required to do a local deploy. The necessary commands are:

- Write the contents of Figure 11 to a new file *hbase.yml*
- Write the contents of Figure 12 to a new file *yccb.yml*
- Create a new docker network named *ncwork* with the following command:
`docker network create ncwork`
- Deploy the hbase backend with the following command:
`docker-compose -f hbase.yml up`
- Wait around 40 seconds for the MPC cryptoworkers to establish a connection and for the instances to be operational.
- Deploy the yccb image with the following command:
`docker-compose -f yccb.yml up`

After executing these commands, the YCSB benchmark starts issuing requests to the prototype and after completing them, the benchmark should produce a final output with

⁴ This setup is used for demonstration purposes as it allows the evaluation of our solutions in a more contained and easy to run setup. Production-ready versions are available upon explicit request.

the throughput and latency metrics for the NoSQL operations issued. Preliminary results for this prototype are further detailed in deliverable D3.5 and in [PMP+16].

```
cluster1:
  image: safecloud/untrusted:latest
  ports:
    - "60010:60010" # Master info web portal port
    - "60000:60000" # Master port for client to connect
    - "16262:16262" # Zookeeper port for the client to connect
  net: ncwork
  hostname: cluster1
  container_name: cluster1
  command: "-s 0 6262 cluster2 6262 cluster3 6262 60000 16262"

cluster2:
  image: safecloud/untrusted:latest
  ports:
    - "60020:60010"
    - "61000:61000"
    - "17262:17262"
  net: ncwork
  hostname: cluster2
  container_name: cluster2
  command: "-s 1 6262 cluster3 6262 cluster1 6262 61000 17262"

cluster3:
  image: safecloud/untrusted:latest
  ports:
    - "60030:60010"
    - "62000:62000"
    - "18262:18262"
  net: ncwork
  hostname: cluster3
  container_name: cluster3
  command: "-s 2 6262 cluster1 6262 cluster2 6262 62000 18262"
```

Figure 15 - HBase deployment configuration

```
yccb:
  image: safecloud/yccb:sq2
  net: ncwork
  hostname: yccb
  container_name: yccb
```

Figure 16 - YCSB deployment configuration

6 Conclusion

Along this document we have described the final prototype versions for the NoSQL components of Secure Queries Solutions 1 and 2. Although two different prototypes are described, both are based on a common secure framework architecture, which not only enables code reuse, but also allows easier extensions to each prototype. For example, prototype extensions with novel cryptographic techniques or even secure frameworks (e.g., Intel SGX [INTEL17]).

Both prototypes are part of a bigger picture and were already integrated with other components, such as query engines and transactional management systems, in order to provide full functionality. This is key for achieving final products that will run the full-fledged use-cases detailed in WP5.

Currently, both prototypes are being further tested and optimized to improve both performance and scalability. Such optimizations will allow reaching the non-functional requirements of the use cases.

7 References

- [APACHE17a] Apache HBase Team (2017). “Apache HBase™ Reference Guide”.
(<https://hbase.apache.org/book.html>)
- [APACHE17b] Apache Hadoop documentation (2017).
(<https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsUserGuide.html>)
- [APACHE17c] Apache Cassandra documentation (2017).
(<http://cassandra.apache.org>)
- [APACHE12] Apache HBase Team (2012). “Coprocessor Introduction”.
(https://blogs.apache.org/hbase/entry/coprocessor_introduction)
- [BCL+09] Boldyreva A, Chenette N, Lee Y, O’Neill A. “Order-preserving symmetric encryption,” in Int. Conference on Advances in Cryptology: The Theory and Applications of Cryptographic Techniques, 2009.
- [CDG+08] Chang F, Dean J, Ghemawat S, Hsieh W, Wallach D, Burrows M, Chandra T, Fikes A, and Gruber R. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.* 26, 2, Article 4, 26 pages, 2008.
- [DOCKER17] Docker containers web page (2017).
(<https://www.docker.com>)
- [DOCKERC17] Docker-Compose web page (2017).
(<https://docs.docker.com/compose>)
- [FHL+07] Fousse L, Hanrot G, Lefèvre B, Péliissier P, Zimmermann P. “Mpfr: A multiple-precision binary floating-point library with correct rounding,” *ACM Transactions on Mathematical Software*, vol. 33, no. 2, p. 13, 2007.
- [INTEL17] Intel SGX web page (2017).
(<https://software.intel.com/en-us/sgx>)
- [MPP+17] Macedo R, Paulo J, Pontes R, Portela B, Oliveira T, Matos M, Oliveira R. “A Practical Framework for Privacy-Preserving NoSQL Databases”. 36th IEEE International Symposium on Reliable Distributed Systems (SRDS), 2017.
- [OPENSSL17] Openssl web page (2017).
(<http://www.openssl.org/>)
- [PMP+16] Pontes R, Maia F, Paulo J, Vilaça R. “SafeRegions: Performance Evaluation of Multi-party Protocols on HBase”. 2016 IEEE 35th Symposium on Reliable Distributed Systems Workshops (SRDSW), 2016.
- [PRZ+11] Popa R, Redfield C, Zeldovich N and Balakrishnan H. “Cryptdb: protecting confidentiality with encrypted query processing”. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 85-100. ACM, 2011.
- [REDIS17] REDIS database web page (2017).
(<https://redis.io>)
- [RS07] Rogaway P, and Shrimpton T. “Deterministic authenticated encryption,” in *Advances in Cryptology–EUROCRYPT*, vol. 6, 2007.
- [YCSB17] YCSB benchmark repository (2017).
(<https://github.com/brianfrankcooper/YCSB>)