SafeCloud

Non-elastic Secure Key Value Store D3.3

Project reference No. 653884

February 2016

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	INESC TEC

Dissemination level

Public

Revision history

Date	Editor	Status	Version	Changes
10.12.2016	João Paulo	Draft	0.1	ToC
12.01.2017	Francisco Maia	Draft	0.2	First Draft
20.01.2017	Bernardo Portela, Rogério Pontes	Revision	0.3	Revision
30.01.2017	João Paulo	Draft	0.4	Final Draft
12.02.2017	Miguel Correia	Review	0.5	Review from INESC ID
20.02.2017	Karl Tarbe	Review	0.6	Review from CYBERNETICA
27.02.2017	João Paulo	Final	1.0	Final Version

Contributors

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

Internal reviewers

Miguel Correia (INESC ID)
Karl Tarbe (CYBERNETICA)
Ville Sokk (CYBERNETICA)

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

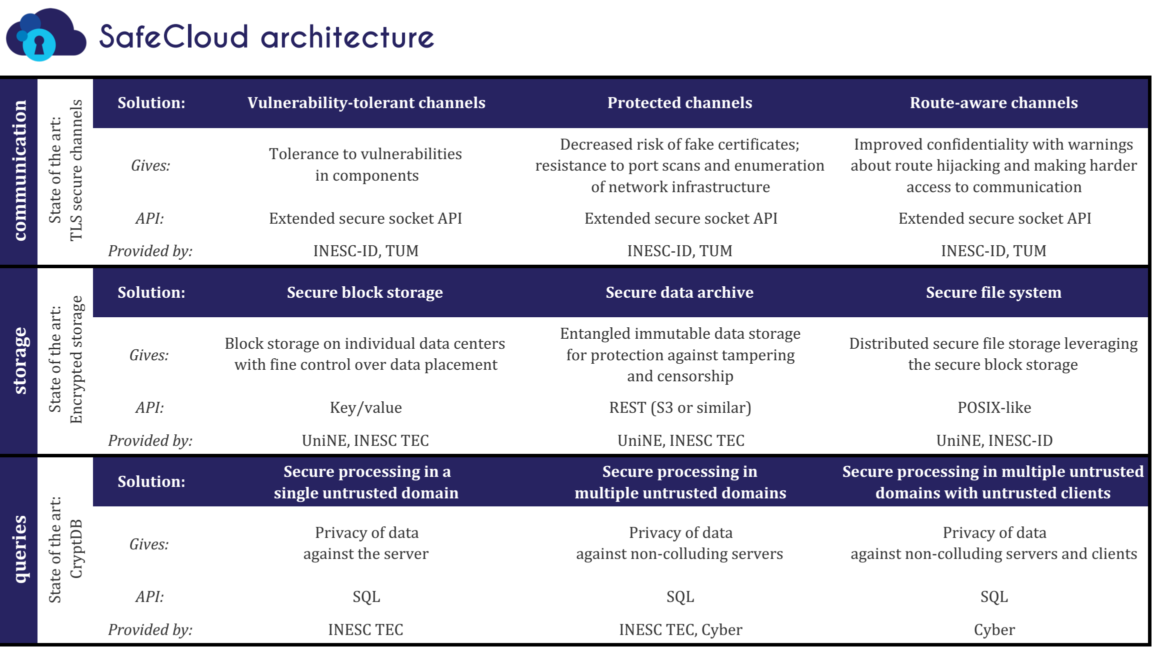
Acronym	Definition
IV	Initialization Vector
MPC	Multiparty Computation
OPE	Order-preserving Encryption

Table of contents

List of Figures

Executive summary

The framework proposed by SafeCloud consists of three layers: secure communication, secure storage, and secure queries. Secure communication provides schemes for the establishment of channels amongst protocol participants employing technologies for tamper-resistant channels, ensuring confidentiality and availability. Secure storage provides techniques for reliable storage, such as long-term confidentiality, protection against file corruption or data deletion. Finally, secure queries provide 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**
		Gives:	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
		API:	Extended secure socket API	Extended secure socket API	Extended secure socket API
		Provided by:	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**
		Gives:	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
		API:	Key/value	REST (S3 or similar)	POSIX-like
		Provided by:	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**
		Gives:	Privacy of data against the server	Privacy of data against non-colluding servers	Privacy of data against non-colluding servers and clients
		API:	SQL	SQL	SQL
		Provided by:	INESC TEC	INESC TEC, Cyber	Cyber

Figure 1 - The SafeCloud framework.

This deliverable presents the first prototypes of the SafeCloud's Secure Key Value Store component. This is a fundamental component for the secure queries layer solution proposed in WP3. In fact, the SQL engine for the first two processing solutions, to be leveraged in WP3, will resort to these Key Value Store solutions to provide secure processing capabilities for applications using SQL databases.

The document discusses the implementations of two NoSQL prototypes developed to allow computation on top of single and multiple untrusted third-party infrastructures. The proposed architectures and implementations are modular, which facilitates the integration of multiple cryptographic techniques in our solutions. In this deliverable, we present a concrete example for how deterministic encryption techniques and multi-party computation via secret sharing can be employed in our prototypes.

Finally, we also detail how the prototype can be deployed and we show preliminary evaluation results.

1 Introduction

The SafeCloud project structure considers three main layers: secure communication, secure storage, and secure queries (or secure data processing). Secure communication provides schemes for the establishment of channels amongst protocol participants employing technologies for tamper-resistant channels, ensuring confidentiality and availability. 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 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.

In this document, we will focus on the secure data processing layer and, in particular, on one of the key components that are used to implement this layer solutions, the Secure Key Value Store. Recalling the secure data processing 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.

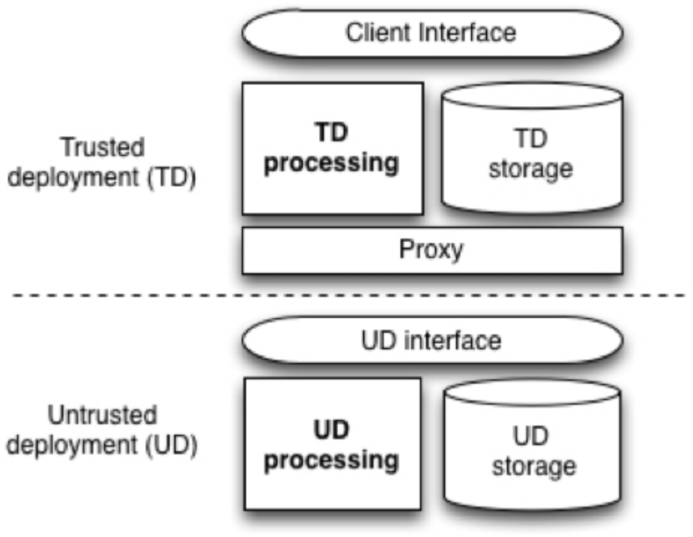


Figure 2 - Secure processing overall architecture.

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

- Solution 1: Secure database server
- Solution 2: Secure multi-cloud database server
- Solution 3: Secure multi-cloud application server

Each one of the solutions follows slightly different architecture designs. Solution 1 and Solution 2 aim at offering full SQL language support but require that most query processing is made on the trusted environment. Solution 3 focuses on different workloads and use cases offering secure query processing mostly done on the untrusted environments but with limited SQL support. Moreover, solutions 1 and 2 more 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 separate from the query processing components (Figures 3 and 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.

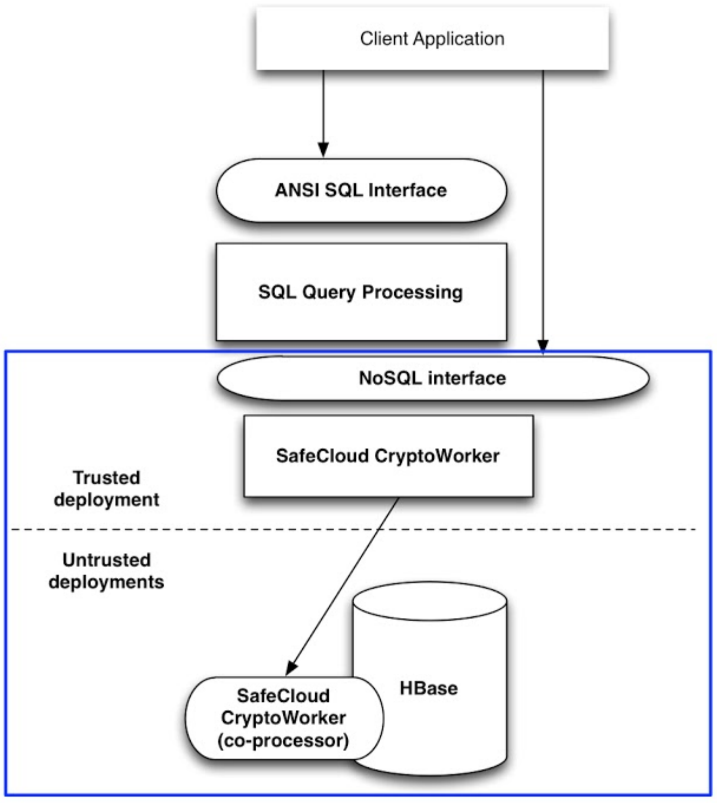


Figure 3 - Secure processing solution 1 architecture.

It is important to notice that, as explained in deliverable D3.1, our implementation of the secure key value components of Solutions 1 and 2 relies on Apache HBase [APACHE16a], which is considered one of the most mature NoSQL databases available in the market. This decision was based both on the feasibility of solutions deployed over such a widespread technology, as well as on the high level of experience and expertise held by the SafeCloud's partners with this system.

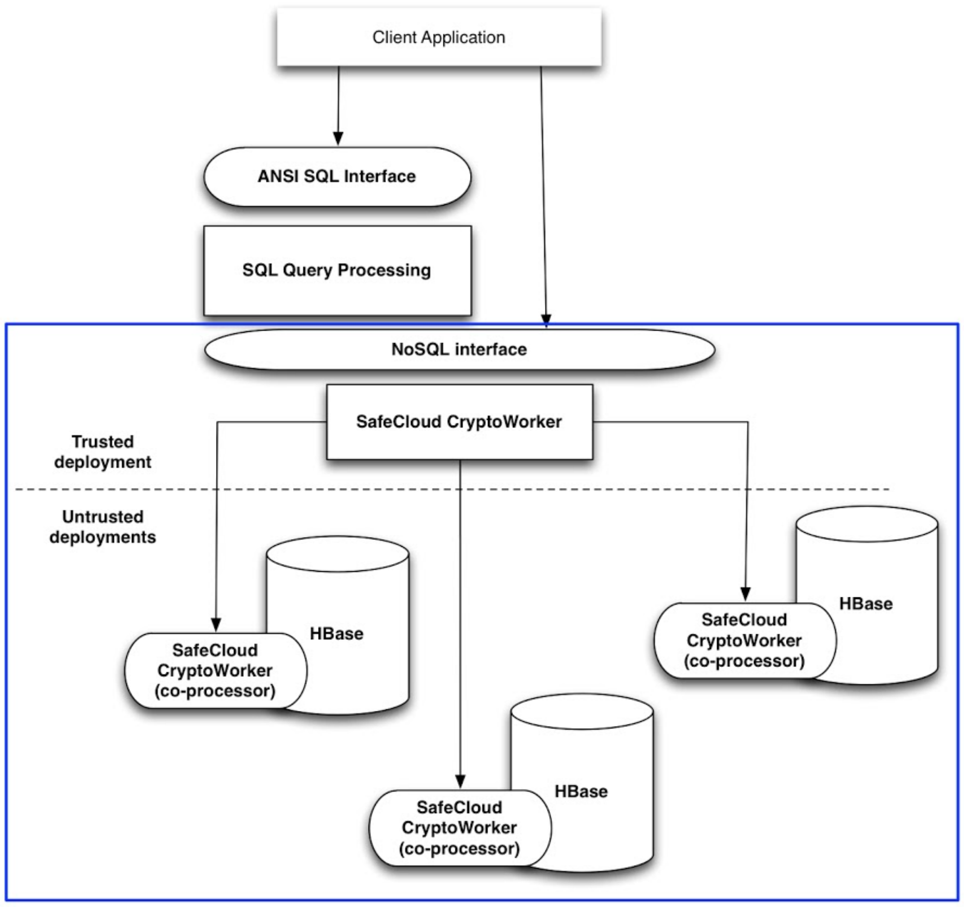


Figure 4 - Secure processing Solution 2 architecture.

The remainder of the document is structured as follows. We begin by providing some background on Apache HBase in Section 2. In Section 3 we describe the high-level approach towards providing a secure key value store relying on HBase. We then instantiate that approach to Solution 1 in Section 4 and to Solution 2 in Section 5. We conclude the document providing some remarks regarding the work developed thus far, and the future development required to successfully achieve SafeCloud's proposed objectives.

2 Background

2.1 Apache HBase

Apache HBase is a distributed, scalable and open-source non-relational database [APACHE16a]. Inspired by Google's BigTable, 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 name and timestamp, which is used as a key for a given value, as illustrated in Figure 5.

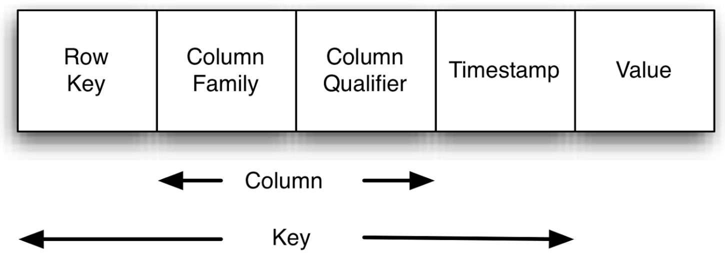


Figure 5 - HBase map structure.

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 a 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 byte array, *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 not-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.

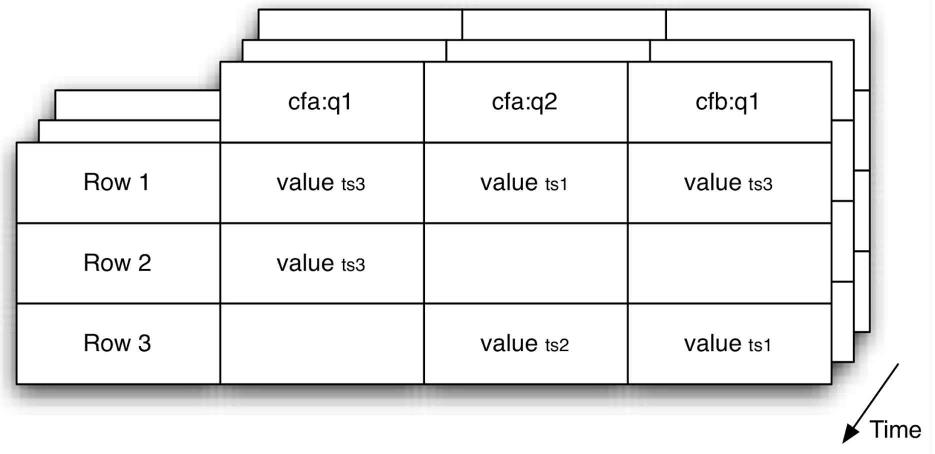


Figure 6 - HBase logical table view.

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 and a column qualifier (cq). 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 in runtime by inserting new key-value pairs.

2.1.1 HBase Architecture

Figure 7 depicts the HBase architecture and main components.

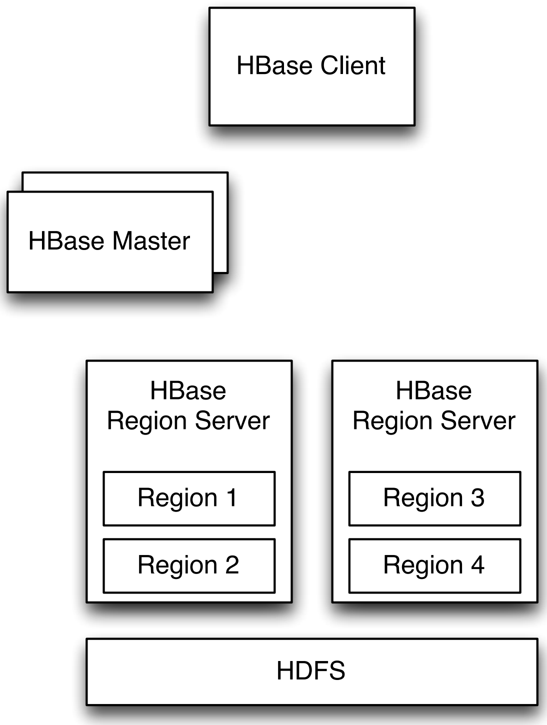


Figure 7 - HBase Architecture.

An **HBase client** component is provided, so that applications using HBase on the client-side 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, and then the client issues the request to the appropriate Region Servers, which, in turn, reply to the client 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 horizontally and each partition is called a Region. After partition, resulting regions can be distributed across several nodes named **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 from the Hadoop Distributed File System (HDFS) [APACHE16b]. Ideally, each Region Server is hosted 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.

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 both 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 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 filter the request to only retrieve the entries for employees that were born in 1986 (considering 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 HBase core implementation, by using the HBase coprocessors mechanism. These can be seen as plugins that are implemented and added to HBase backend components that allow extending the computation done when NoSQL queries are performed [APACHE12].

Two types of coprocessors are available: observers and endpoints. Observer coprocessors bind a piece of code to system events. For instance, it 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 and thus, arbitrary code execution is allowed 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 ids 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 the key components for performing the additional computation and exchange results across parties before replying to the client. Finally, coprocessors are also required for other SafeCloud solutions that require maintaining some sort of indexes at the HBase backend in order to do computation over private data. The access to the index and the computation done when a HBase request is done also requires coprocessors [PRZ+11]. Without this mechanism we would have to change the core implementation of HBase backend components, which would increase significantly 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 deployment 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 from HBase itself. These components are deployed both in the untrusted and the trusted sites, and are named *CryptoWorkers*.

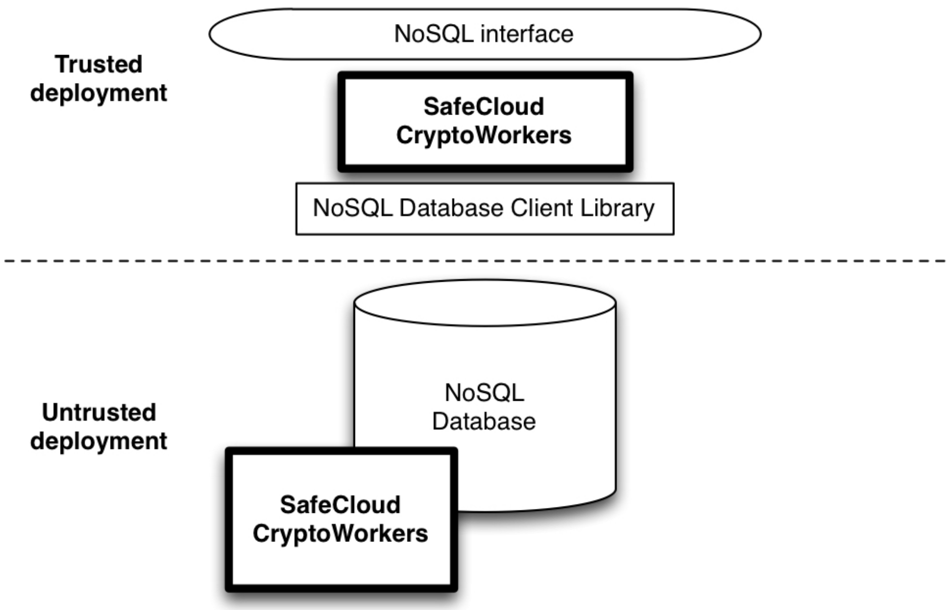


Figure 8 - Solutions 1 and 2 architectural components.

In Figure 8 we depict the organization of the different components that are used for the secure key value stores used in Solution 1 and Solution 2 of the secure data processing layer of the SafeCloud project. It is important to notice that, for Solution 2, 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 how the CryptoWorkers work.

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

- **Trusted site**: CryptoWorkers transform plain NoSQL operations into 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 can 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 coprocessors. However, the software itself does not depend on HBase, making our design extensible and compatible with other NoSQL databases. This HBase-agnostic approach is what allows for 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 system interface that is composed of three main operations: PUT, GET and SCAN. The basic functionality of the CryptoWorker is to translate those requests into *secure requests*. Secure requests are encoded according to some associated cryptographic technique, so that when they are transmitted to some untrusted (potentially adversarial) environment, the required security guarantees will hold. 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 (Figure 9). That set of operations will ensure that such GET is made with specific data privacy guarantees according to the *CryptoWorker* configuration, which specifies the protocol encoding the data.

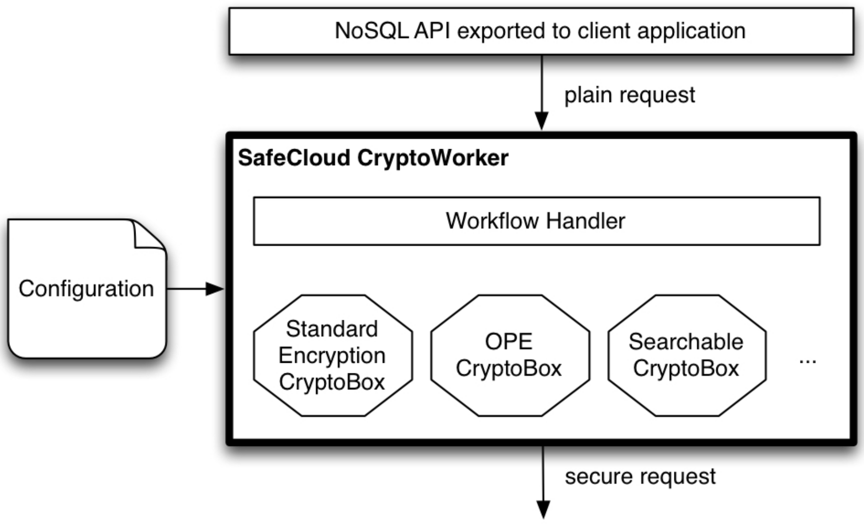


Figure 9 - CryptoWorkers in the trusted site.

Each CryptoWorker on the trusted deployment is responsible for a specific workflow that allows the translation of plain requests to secure requests and their corresponding 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 overlaying requests in a secure fashion, involving three general sequential operations: a client-side encode, a server-side operation, and a client-side decode. For example, to perform a GET, our prototype executes getC on the client, getS on the server, and then getCDecode, to retrieve the original request result. Each *cryptobox* offers a set of cryptographic implementations that are useful in certain privacy workflows. For example, in order to provide standard encryption over HBase it is necessary that the content of plain requests is encrypted using, for instance, AES-GCM.

Consequently, a SafeCloud Standard Encryption *cryptobox* is made accessible by CryptoWorkers. Different *cryptoboxes* provided are depicted in Figures 9 and 10.

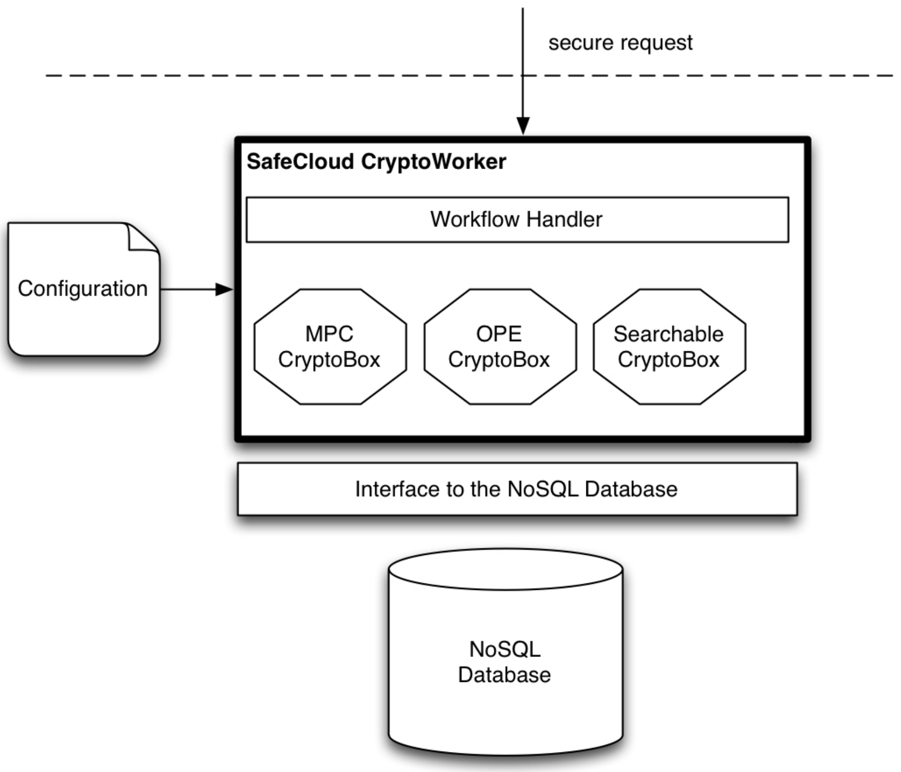


Figure 10 - CryptoWorkers in the untrusted site.

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 6). For example, techniques such as multi-party computation CryptoWorkers 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. Note that these operations will be performed in possibly corrupt servers. The specific operations used and the different configurations that are considered in the current design of the system are described in detail in D3.5.

In the context of the present document it suffices to understand 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 D3.5. In the following sections we focus on the actual system prototype and how it can be installed, run and tested. When necessary, we will provide context for the specific set of CryptoWorker configurations used in this first implementation of the prototype.

4 Solution 1: Secure processing in a single untrusted domain

4.1 Overview

Solution 1 of the secure processing layer of SafeCloud is focused 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-based secure key value store. This prototype was designed to allow the configuration of the different privacy preserving techniques described in D3.5. These techniques include order preserving encryption, deterministic and standard encryption and searchable encryption. The current version (M18) of the prototype has already been tested with deterministic encryption. The preliminary results are presented in D3.5.

4.2 Architecture

The design of the current prototype for Solution 1 follows the architecture planned for the final software package. In terms of implementation it lacks support for some of the techniques mentioned in the previous WP3 deliverables and still constitutes a non-optimized implementation. However, the architecture was designed with strong modularity concerns, which allows the prototype to be ready for quick integration with the components that will be developed along the rest of the project. More specifically, this prototype only includes a CryptoWorker and correspondent cryptobox for deterministic encryption. Other components will be added in the following months.
In Figure 11 we present the concrete architecture for the prototype instantiated with the deterministic encryption CryptoWorker.

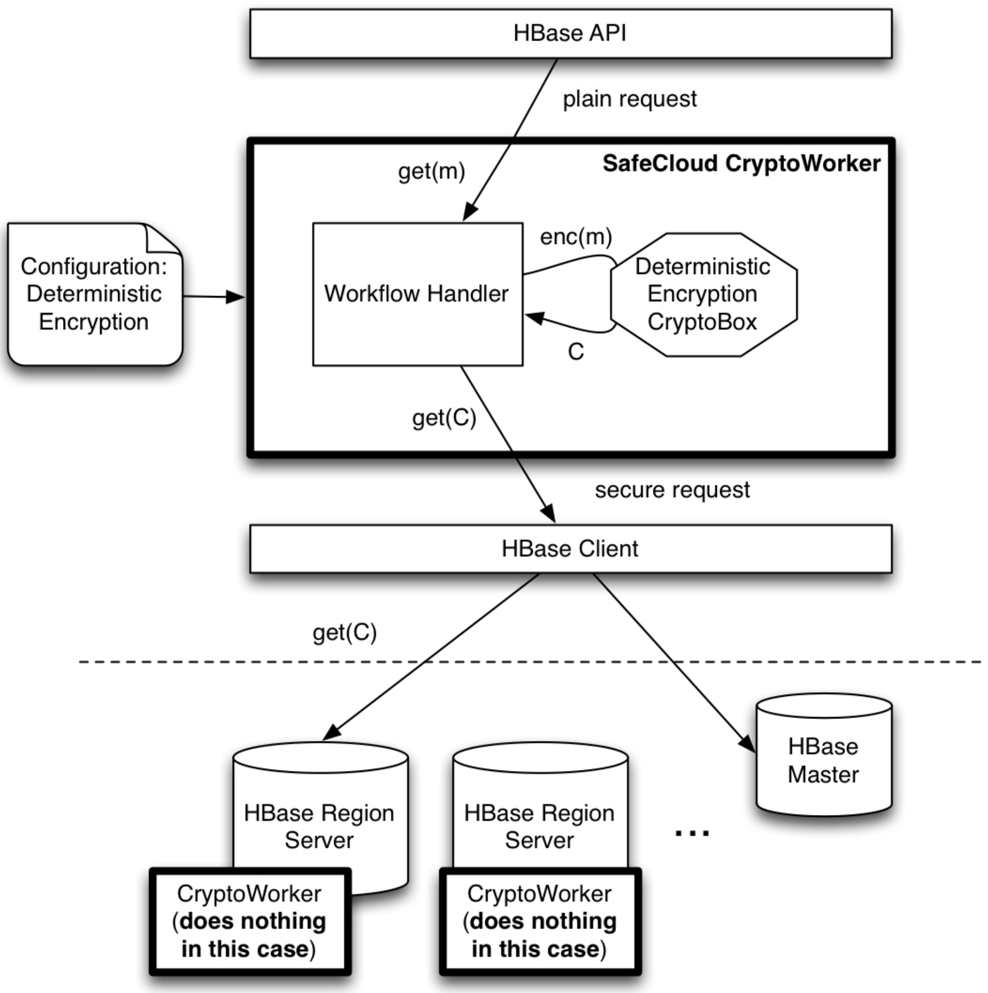


Figure 11 - Solution 1 prototype architecture.

The prototype follows the architecture presented in the previous sections and, as described previously, uses Apache HBase. SafeCloud CryptoWorker code works as an extension of the HBase client library and, on the server side, runs in coprocessors 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 coprocessors only extend their behavior.

In Figure 11 we depict the basic workflow for the deterministic encryption case that serves as an illustrative example about how Solution 1 prototype works. Upon a request the CryptoWorker, configured to use deterministic encryption, knows it should 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 environment. Observe that this specific instantiation of deterministic encryption still follows the more abstract API proposed in 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 avoid this overhead without compromising deterministic encryption privacy levels. This will impact significantly 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 and compromises between privacy levels and performance must be taken into account when choosing a specific privacy technique or set of techniques.
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 initial demonstrator of Solution 1 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 near future.

In more detail, the current prototype supports AES encryption with a fixed initialization vector, which makes encryption deterministic. The library used is the one provided in Java crypto [JAVAC16] and it serves as an example of the type of techniques that could be supported in this solution. This technique is added by implementing the deterministic encryption cryptobox. This implementation assumes that keys and values are protected with AES-GCM.

The implementation of this technique allows supporting the vanilla HBase operations *i.e.,* PUT, GET, SCAN, DELETE and filters.

Note that the main advantage of our architecture and implementation is that it can be easily integrated with other techniques, such as order preserving encryption and standard encryption. Implementation-wise most of these techniques do not require any additional computation at the HBase backend (coprocessors) since the CryptoWorker deployed at the trusted infrastructure is responsible for ciphering and deciphering the data and to do the additional computation to support the full HBase API. For other techniques that also do not require additional computation at the HBase backend, the design pattern is similar. In fact, for techniques such as order preserving encryption, the work done at the CryptoWorker for all remote operations is reduced, since the data stored at the untrusted HBase backend can be compared even when encrypted. This means that the CryptoWorker is able to support SCAN queries without having to transfer rows to the trusted infrastructure in order to validate their values with the requested query.

Finally, 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 way similar to the approach described in Solution 2, where this mechanism is necessary.

4.3.1 Setup and Usage

Solution 1 prototype is currently installable through Docker containers [DOCKER16]. Docker containers are isolated components that are initialized with a configuration file where it is detailed what are the software packages (Java, HBase, etc) that must be installed and how to deploy these components. With this configuration file it becomes easy to automatically deploy our solution in any machine supporting the Docker environment.

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 in a single server. The second image contains the necessary dependencies to use our HBase Client, plus our model for the CryptoWorker. Finally, to perform an initial evaluation of our proposal, we integrated our solution with the YCSB benchmark [YCSB16], a widely used framework to evaluate the performance of different NoSQL databases.

Upon SafeCloud distribution, a configuration of these images 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 images in a distributed setting, this section only presents the steps required to do a local deploy. A local deploy 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 is beyond the scope of this document, we refer the installation process to the Docker official website documentation [DOCKER16]. An additional tool is also required to simplify the deployment process, Docker-compose[1].

Besides the referred essential tools, two configuration files are required. Figure 12 and 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: rogerp/baseline
        net: ncwork
        hostname: baseline
        container_name: baseline
```

Figure 12 - HBase deployment configuration.

[1] https://docs.docker.com/compose/install/

```
ycsb:
        image: safecloud/ycsb:aes
           net: ncwork
           hostname: ycsb
           container_name: ycsb
```

Figure 13 - YSCB 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 *ycsb.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 ycsb image with the following command:

  docker-compose -f ycsb.yml up

After executing these commands the YCSB benchmark should start issuing requests to the database and, after completing the benchmark, an output with the achievable latency and throughput for requests is displayed. Preliminary YCSB results for this prototype are detailed in deliverable D3.5.

5 Solution 2: Secure processing in multiple untrusted domains

5.1 Overview

Similarly to Solution 1, this solution also relies on a secure Key Value Store backend. However, Solution 2 considers multiple untrusted domains. Consequently, a single HBase deployment is no longer sufficient, as each untrusted domain must be treated as a completely independent entity. Nevertheless, the high level approach taken is similar to the one previously described for Solution 1, i.e., we do not modify HBase itself but rather externally add the necessary behavior via the appropriate embedded mechanisms.

For the particular case of Solution 2, we consider three independent untrusted servers, each with a separate HBase deployment. The added behavior empowers each HBase instance to provide secure operations 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 prototype is presented in Figure 14. As expected, this prototype considers three different remote deployments, where different HBase instances are installed. A client-side CryptoWorker, at the trusted site, is instantiated and configured with the appropriate multi-party computation parameters. In contrast with the previous prototype instantiated for deterministic encryption, backend-side CryptoWorkers take part in the process, and 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.

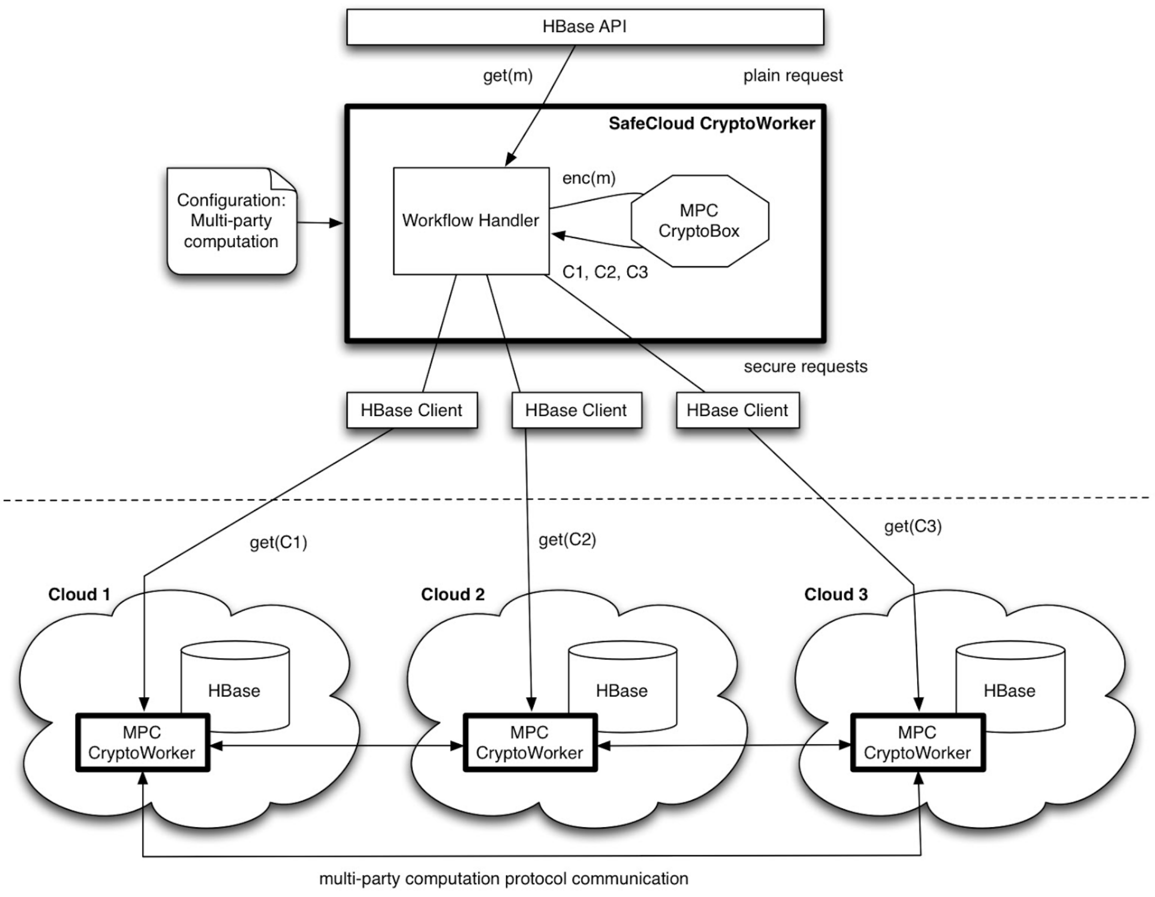


Figure 14 - Solution 2 prototype architecture.

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 the specific set of procedures to process it. This typically includes querying the local HBase instance for data, communicating with 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 initial demonstrator of Solution 2 follows the architecture described in the previous section. In more detail, the current 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 IO Sockets.

The implementation of this technique allows supporting the vanilla HBase operations *i.e.,* PUT, GET, SCAN, DELETE and filters. Again, the architecture and implementation used for this solution is highly modular so, any other security techniques that resort to multiple untrusted domains could be integrated with a minimal effort in our Solution 2.

5.3.1 Setup and Usage

The deployment of Solution 2 follows a very similar approach to Solution 1. 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 15 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 CrytpoWorker bundle, only a minor detailed must be changed. Instead of using the tag *aes* it must be used the tag *mpc* as can be seen in Figure 16.

```
cluster1:
        image: saferegions: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: saferegions: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: saferegions: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

```
ycsb:
      image: safecloud/ycsb:mpc
        net: ncwork
        hostname: ycsb
        container_name: ycsb
```

Figure 16 - YCSB deployment configuration

To deploy this solution, it is assumed that a docker engine and docker-compose 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 15 to a new file *hbase.yml*

- Write the contents of Figure 16 to a new file *ycsb.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 until the MPC CryptoWorkers establish a connection and the instances are operational.
- Deploy the ycsb image with the following command:

  docker-compose -f ycsb.yml up

After executing these commands the YCSB benchmark should start issuing requests to the prototype and after completing, the benchmark should produce a final output with the throughput and latency metrics for the NoSQL operations issued. Again, preliminary results for this prototype are detailed D3.5.

6 Conclusion

Along this document, we have described the current prototype versions for the non-elastic secure key value store. This implies two separate versions with distinct architectures. For the first solution prototype, there are two main goals to achieve until the end of the project. First, it is necessary to implement the remaining techniques and evaluate them. Second, optimizations should be studied and implemented in order to improve the data store performance and elasticity in order to achieve the use-cases non-functional requirements.

Solution 2 relies on multi-party algorithms that are known to be quite costly in terms of performance. The elasticity of this solution is also a point of future research work. Although it is not a main project goal we aim at providing some optimizations for this solution as well.

Finally, another important aspect that follows the work presented in this deliverable is integration. The prototype components we have described are part of a bigger picture and must be integrated with other components such as query engines and transaction management system in order to provide full functionality.

7 References

[APACHE16a] Apache HBase Team (2016). "Apache HBase ™ Reference Guide". (https://hbase.apache.org/book.html)

[APACHE16b] Apache Hadoop documentation (2016). (https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsUserGuide.html)

[APACHE16c] Apache Cassandra documentation (2016). (http://cassandra.apache.org)

[APACHE12] Apache HBase Team (2012). "Coprocessor Introduction". (https://blogs.apache.org/hbase/entry/coprocessor_introduction)

[DOCKER16] Docker containers web page (2016). (https://www.docker.com)

[JAVAC16] Java Crypto documentation (2016). (https://docs.oracle.com/javase/7/docs/api/javax/crypto/package-summary.html)

[PMP+16] Pontes R, Maia F, Paulo J, Vilaça R. 2016. SafeRegions: Performance Evaluation of Multi-party Protocols on HBase. 2016 IEEE 35th Symposium on Reliable Distributed Systems Workshops (SRDSW), 2016 . :31-36

[PRZ+11] Raluca Ada Popa, Catherine Redfield, Nickolai Zeldovich and Hari Balakrishnan. Cryptdb: protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 85-100. ACM, 2011.

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