Secure SQL Engine

D3.8

Project reference no. 653884

February 2018

European Commission

Horizon 2020
European Union funding
for Research & Innovation

Document information

Scheduled delivery	01.03.2018
Actual delivery	01.03.2018
Version	1.0
Responsible Partner	Cybernetica

Dissemination level

Public

Revision history

Date	Editor	Status	Version	Changes
05.02.2018	K. Tarbe	Draft	0.1	Initial version
13.02.2018	F. Maia	Draft	0.2	Revised version
20.02.2018	K. Tarbe	Draft	0.3	Revised version
28.02.2018	K. Tarbe	Final	1.0	Final version

Contributors

K. Tarbe (CYBER)
F. Maia (INESC-TEC)
V. Sokk (CYBER)

Internal reviewers

M. Pardal (INESC-ID)
H. Niedermayer (TUM)

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
ACID	Atomicity, Consistency, Isolation, Durability
MPC	Multi-Party Computation
OMID	Optimistically transaction Management In Datastores
REPL	Read-Eval-Print Loop
SQ1	Secure database server
SQ2	Secure multi-cloud database server
SQ3	Secure multi-cloud application server
SQL	Structured Query Language

Table of contents

Executive summary

This document gives a very brief overview of the three solutions in the secure queries layer of the SafeCloud project. This deliverable is a software deliverable and the main focus of this document is on the quickstart guide for the three solutions.

We do not link to the software from this deliverable, because all the solutions incorporate existing commercial components. The first two solutions: SQ1 and SQ2 can be obtained from SafeCloud Technologies Sàrl[1] and the third solution, SQ3 is offered by Cybernetica AS[2].

We refer readers to previous deliverables for more details about the solutions.

[1] https://safecloudtech.com/
[2] https://sharemind.cyber.ee/

1 Introduction

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

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

Figure 1: Secure queries solutions.

For each processing solution, a detailed architecture, the prototypes and the different sets of supported privacy-preserving techniques have been described in previous deliverables D3.1 to D3.7. The different architectures and privacy-preserving techniques used in each solution assume different trust models and, as a consequence, different privacy guarantees emerge for each solution.

This deliverable introduces the latest improvements and changes to the three secure queries layer solutions. In section 2 we summarise the initial goals of the secure queries layer of the SafeCloud project. After describing the goals we recap the deployment diagrams of the solutions and then proceed with quick start manuals for each solution.

2 Background

The Initial architecture of the secure queries layer solutions was detailed in D3.1. The cryptographic techniques used for the solutions were described in D3.2 and later refined in D3.5. Elasticity properties of the solutions were described in D3.6.

2.1 Goals

The most important goal of secure queries layer was to explore different techniques and trade-offs in enabling processing on encrypted data in a database setting. We achieved this goal by having three solutions, each with its own security model. The first solution, Secure database server, utilizes cryptographic techniques such as order-preserving encryption, deterministic and standard encryption and searchable encryption. The choice of the encryption scheme for particular database table column provides applications with a performance-security trade-off. The other solutions: Secure multi-cloud database server and Secure multi-cloud application server, both use multi-party computation based on additive secret-sharing, but they differ in the approach taken to ensure SQL capabilities. Solution 1 and solution 2 build a relational database on top of a key-value database HBase, which they extend with coprocessors to support cryptographic operators, while solution 3 uses a different approach, Sharemind, which does not rely on a component capable of transforming SQL queries into HBase queries. However, running SQL queries directly on top of the Sharemind MPC platform does come with a significant cost: SQL as supported by the Secure multi-cloud application server is not ANSI SQL standard compliant, because many advanced parts of the standard are not implemented, but all the important data transforming operations like filters, joins and aggregations are available.

2.2 Architecture

All three solutions follow a similar architecture, when looked at distance. But there are clear differences in the solutions.

Lets start with SQ1 – Secure database server. The deployment is pictured on Figure 2. We have a distinction between trusted deployment and untrusted deployment. In terms of security we protect data from the untrusted deployment.

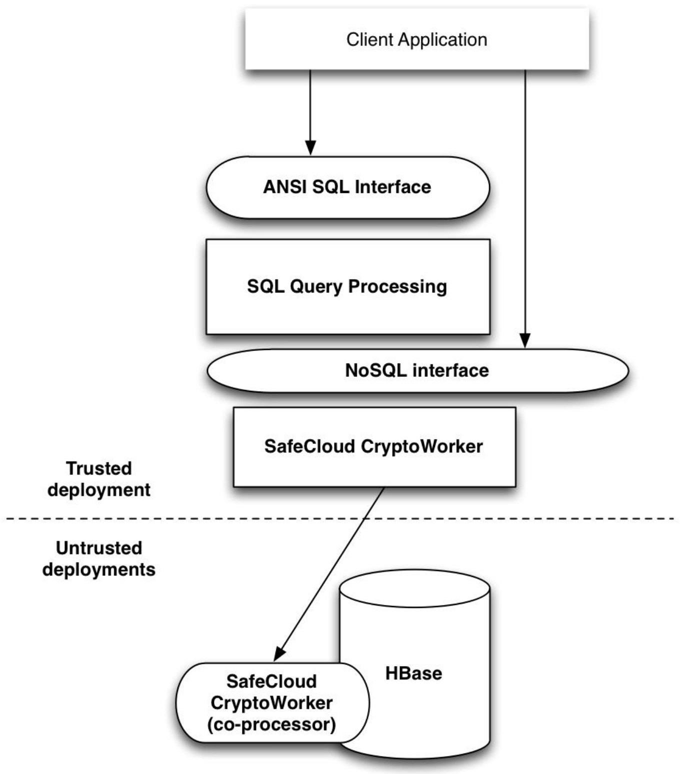


Figure 2: Concrete SafeCloud deployment for Solution 1: Secure database server

In SQ2 – Secure multi-cloud database server - the untrusted deployment as in SQ1 is separated into 3 untrusted deployments that in addition must not collude. The deployment can be seen on Figure 3.

In SQ3 – Secure multi-cloud application server- we have more deployment domains, because SQ3 separates data between multiple users. An example is shown on Figure 4.

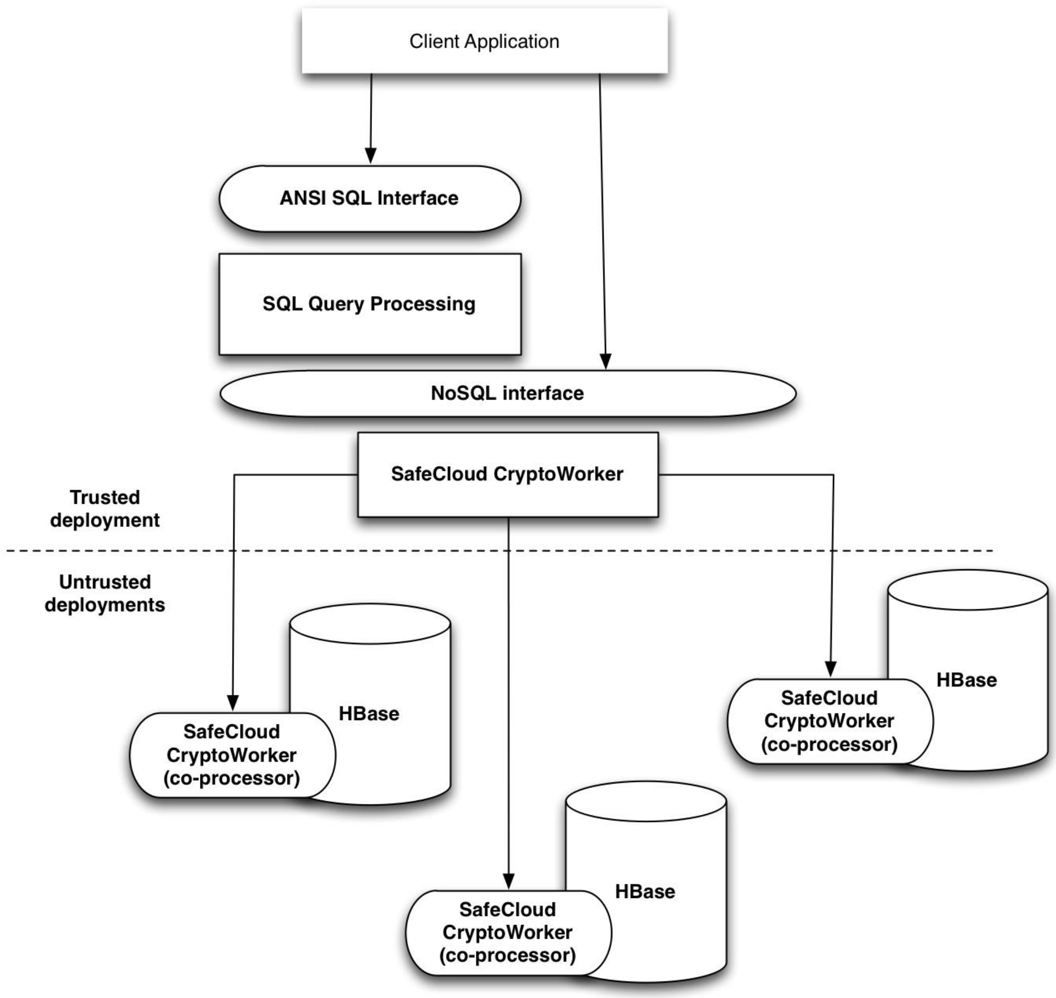


Figure 3: Concrete SafeCloud deployment for Solution 2: Secure multi-cloud database server

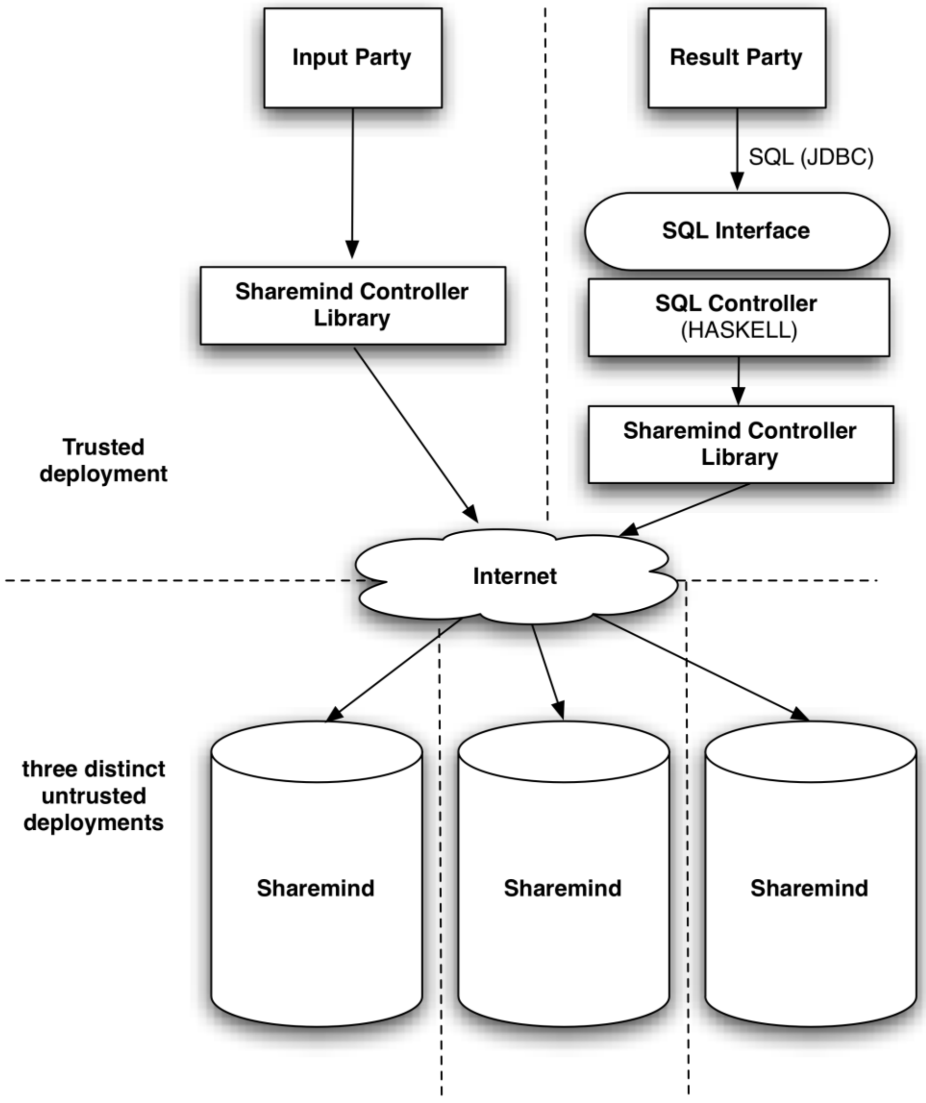


Figure 4: Solution 3: Secure multi-cloud application server deployment

3 Secure database server and Secure multi-cloud database server

3.1 Brief description of the solution

Secure queries solutions 1 and 2 share a similar high-level architecture and components. Although their deployment settings are different, from the point of view of the client application, they export the same APIs and have similar setup procedures.
Briefly, both solutions expose a SQL API comparable to those of traditional relational database management systems. With the exception of potential SQL dialect differences and minor incompatibilities, which are easily addressed should they arise, any application can use our Secure Database Server and our Secure Multi-Cloud Database Server interchangeably with any commercially available relational database system.
Naturally, the actual deployment of our databases will differ with respect to traditional systems. In order to provide protection against untrusted Cloud infrastructures our database systems are installed in two different sites: untrusted site (Cloud provider(s)) and trusted site (on premises or trusted Cloud(s)). The goal of our database systems is to ensure that data uploaded to the Cloud provider (or Cloud providers for SQ2) will be protected at all times. Only the client, in the trusted site, will have access to such data.
The enforcement of this privacy-by-design approach differs for SQ1 and SQ2 as they consider different security models. SQ1 uses a single untrusted deployment and guards the data from it, while SQ2 uses multiple untrusted deployments. These security models are extensively discussed in previous deliverables such as D3.2 and D3.5.
In this Section, we describe how these two database systems can be installed and configured to be used by any application. In particular, we provide an example of how the system can be configured to use specific privacy-preserving techniques.

3.2 New Features

In this final version of the Secure Database Server and the Secure Multi-Cloud Database Server we added support for transactions. To this end we leverage the Apache Omid system [APACHE17c] to provide full ACID (Atomicity, Consistency, Isolation, Durability) semantics.

3.3 Quickstart user manual

Solution SQ1 and Solution SQ2 follow the same lifecycle of initialisation and termination. They both provide an extensible configuration to enable adaptation to user requirements. In addition, their configuration flexibility allows to update the system software without disrupting working database deployments.
A distribution of these systems is made available through a compressed file with the necessary binaries for initiating the system, runtime Java and C dependencies, a folder with example configurations and this guideline that helps users to correctly install, configure and use the software. This guideline focuses on the configuration of the trusted domain part as it is expected that the untrusted domain has a standard HBase deployment in one or three clusters depending on the solution.

3.3.1 Distribution Structure

The distribution file contains the following structure which is exemplified on Figure 5:

- *bin*
- *lib*
- *lib.hb*

- *cryptoboxes*
- *etc*
- *Readme*
- *Manual*

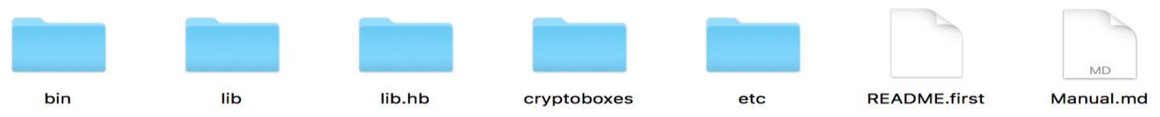


Figure 5: Example Distribution folder structure.

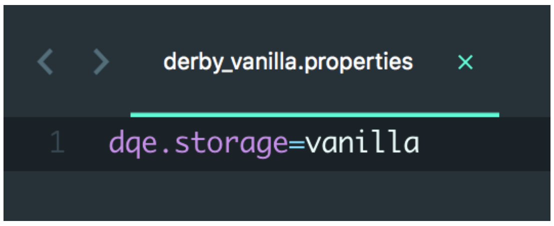
The bin folder contains the scripts to start and stop the database on either standalone mode or distributed mode. The standalone mode is used for development and testing while the distributed mode is intended for production purposes. In either mode, SQ1 is configured in the same way. Standalone mode is, however, not currently available for SQ2.
The bin folder also contains a client interface to remotely access an active database. The *lib*, *lib.hb* and *cryptoboxes* contain all of the dependencies required for system. In detail, the *lib* and *lib.hb* folder contain Java dependencies while the *cryptoboxes* folder contains C dependencies. The Java dependencies are loaded when starting the database system and no additional step needs to be taken. However, the *cryptoboxes* are C shared libraries and need to be installed on the trusted domain. These libraries only support a Linux operating system. As any other C library, *cryptoboxes* are installed by adding the shared libraries (.so) to the OS library folders such as */usr/lib* or */usr/local/lib* and by updating the dynamic library variables. These libraries have additional dependencies which are specified on a Readme document inside the folder. In the same folder there is also a shell script "*install.sh*" that requires administrator privileges and installs the dependencies and the *cryptoboxes*.
On the untrusted domain, no dependencies need to be manually installed as the database relational engine can dynamically load the necessary dependencies to the remote HBase servers.

3.3.2 Distribution configuration

The most important step before starting the relational database engine is the configuration of three property files in the *etc* folder. The three main files that need to be configured are the "*derby.properties*", "*hbase-site.xml*" and "*schema.xml*". In "*derby.properties*" it is possible to configure the properties of the relational database engine. The *"hbase-site.xml"* file contains the information about the connection to the remote untrusted domain NoSQL databases and whether data will be protected or not before being stored. The "*schema.xml*" file contains a description of the database schema and which cryptographic schemes should be used to protect sensitive data.

3.3.3 Derby.properties

This property file supports all of the default Derby properties and already has a set of standard predefined configurations which can be safely used for a correct and efficient deployment of the relational engine. The only new property that must actually be considered by users is if the system is using a transaction manager or not. The default configuration assumes that the system is using the OMID transaction manager and, if this is the desired behaviour, then no steps need to be taken. However, if a transaction manager is not necessary, "*derby.properties*" should contain the property "*dqe.storage*" set to "vanilla" as presented in Figure 6.



```
dqe.storage=vanilla
```

Figure 6: Derby configuration without transaction manager.

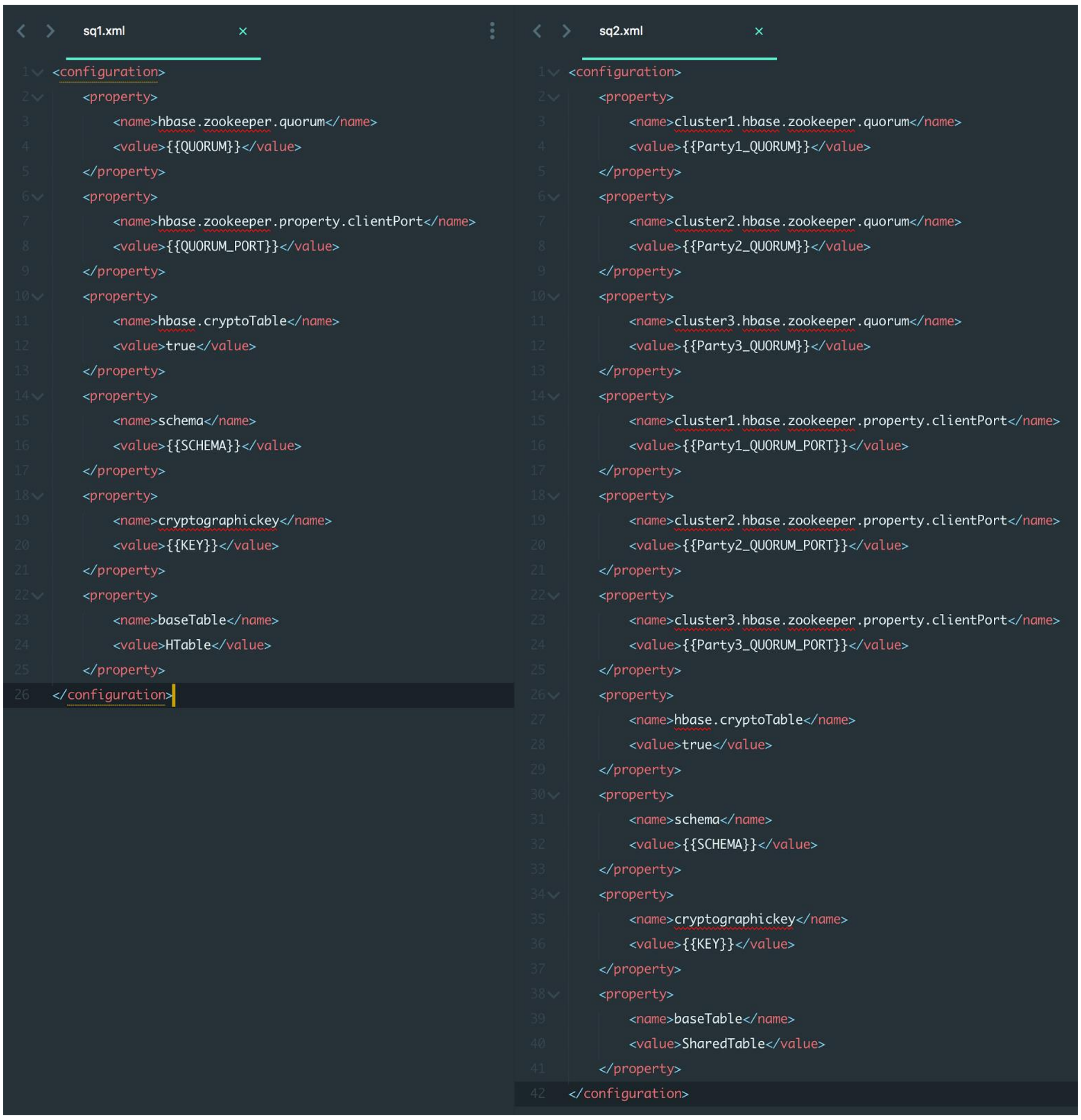
3.3.4 HBase-site.xml

The HBase-site.xml has the HBase client properties necessary to connect to the HBase deployment to the untrusted domain. Solution 1 and 2 have specific sets of properties and values, however, they are designed in such a way that both solutions share a set of common properties to minimise and simplify the configuration process. In fact, the configuration of SQ1 can be seen as a subset of the configurations of SQ2. An example of the configurations is presented in Figure 7.

An SQ1 deployment requires 6 essential configurations, the first two are "**hbase.zookeeper.quorum**" and "**hbase.zookeeper.property.clientPort**" that point the relational engine to the location of the untrusted domain HBase database. The value of the first property must contain the hostname of the HBase Zookeeper[3] location and the second property value must have the Zookeeper listening port (default: 2181). The third property is "**hbase.cryptotable**" which accepts a boolean value to activate the security mode that protects confidential data. If the value is set to "*false*" the system does not protect client requests and processes them as any other database without security guarantees. The value of the fourth property, "**cryptographickey**", must point to a file in the "*etc"* folder that contains the cryptographic key used to protect the confidential data. A sample key file is available on the distribution but it should be changed for security reasons. The fifth property is the "**baseTable**" that allows the user to choose between Solution SQ1 and SQ2. For the SQ1 solution, this property should have the value set to "*HTable*" so an underlying standard HBase table is used. In Solution SQ2, this property should have the value "*SharedTable*" to use our own implementation of an HTable that abstracts the distribution of data over three distinct HBase databases. Apart from this, SQ2 requires the user to specify the location of the remote NoSQL databases. It follows a similar approach to SQ1 where the zookeeper location and port must be specified, but the properties must follow the following syntax "**cluster<*ID*>.hbase.zookeeper.quorum**" and "**cluster<*ID*>.hbase.zookeeper.property.clientPort**" where the ***ID*** is a value from 1 to 3 that identifies and distinguishes the remote databases.

The final property is the "**schema**" which must have the value set to a full path to a configuration file that describes the database schema and the encryption techniques that should be used to protect the database. This configuration file is presented in the following subsection.

[3] Apache Zookeeper - a system for distributed coordination



```xml
<configuration>
    <property>
        <name>hbase.zookeeper.quorum</name>
        <value>{{QUORUM}}</value>
    </property>
    <property>
        <name>hbase.zookeeper.property.clientPort</name>
        <value>{{QUORUM_PORT}}</value>
    </property>
    <property>
        <name>hbase.cryptoTable</name>
        <value>true</value>
    </property>
    <property>
        <name>schema</name>
        <value>{{SCHEMA}}</value>
    </property>
    <property>
        <name>cryptographickey</name>
        <value>{{KEY}}</value>
    </property>
    <property>
        <name>baseTable</name>
        <value>HTable</value>
    </property>
</configuration>
```

```xml
<configuration>
    <property>
        <name>cluster1.hbase.zookeeper.quorum</name>
        <value>{{Party1_QUORUM}}</value>
    </property>
    <property>
        <name>cluster2.hbase.zookeeper.quorum</name>
        <value>{{Party2_QUORUM}}</value>
    </property>
    <property>
        <name>cluster3.hbase.zookeeper.quorum</name>
        <value>{{Party3_QUORUM}}</value>
    </property>
    <property>
        <name>cluster1.hbase.zookeeper.property.clientPort</name>
        <value>{{Party1_QUORUM_PORT}}</value>
    </property>
    <property>
        <name>cluster2.hbase.zookeeper.property.clientPort</name>
        <value>{{Party2_QUORUM_PORT}}</value>
    </property>
    <property>
        <name>cluster3.hbase.zookeeper.property.clientPort</name>
        <value>{{Party3_QUORUM_PORT}}</value>
    </property>
    <property>
        <name>hbase.cryptoTable</name>
        <value>true</value>
    </property>
    <property>
        <name>schema</name>
        <value>{{SCHEMA}}</value>
    </property>
    <property>
        <name>cryptographickey</name>
        <value>{{KEY}}</value>
    </property>
    <property>
        <name>baseTable</name>
        <value>SharedTable</value>
    </property>
</configuration>
```

Figure 7: Example configuration of SQ1 and SQ2 HBase-site.xml

3.3.5 Schema configuration (Schema.xml)

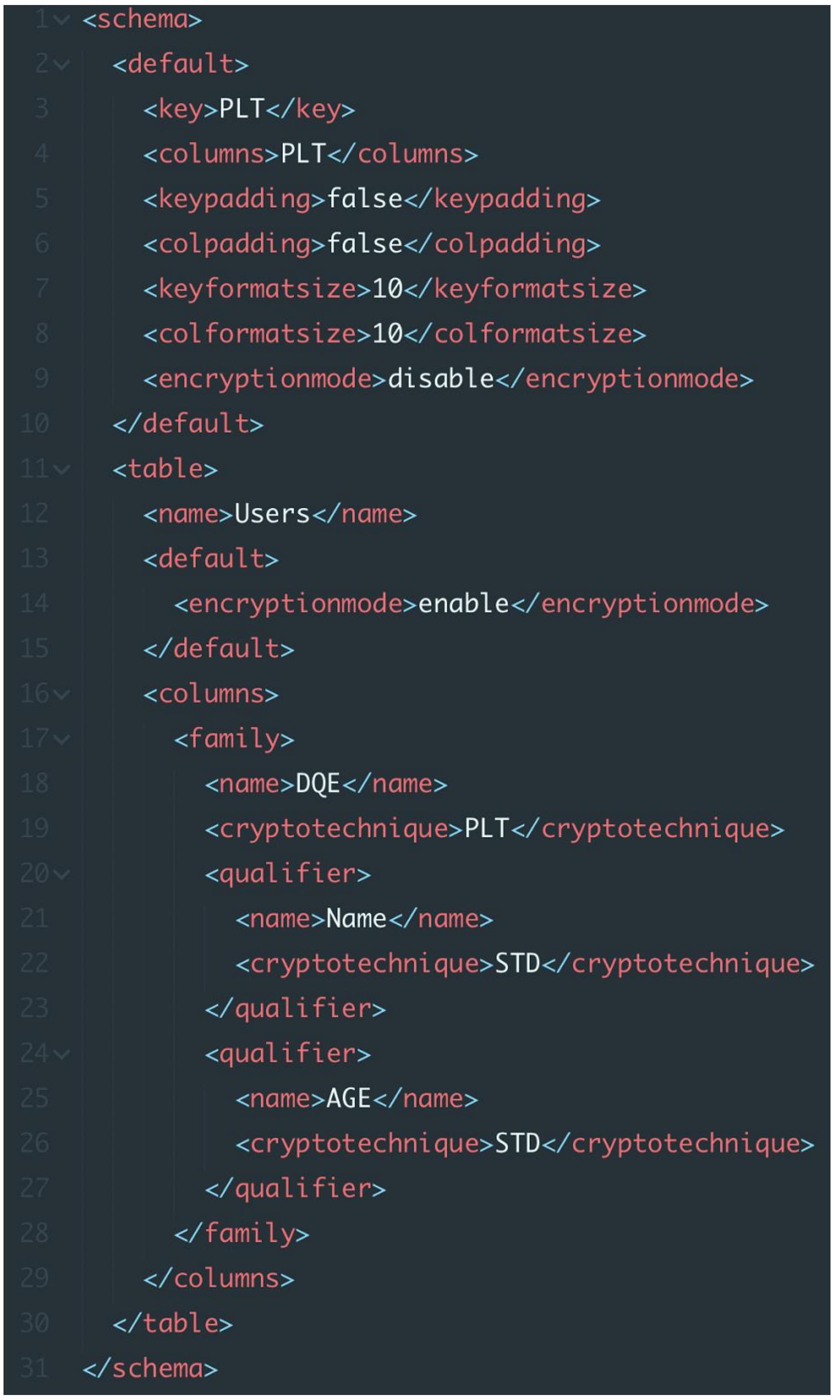
The final configuration file is the schema defined in the property "**schema**" on the HBase-site.xml file. This configuration is an XML file that has a set of common elements that define the protection mode that should be used to protect database tables and columns. The protection modes available are the following:

- DET - Deterministic Encryption
- OPE - Order-preserving Encryption
- STD - Standard Encryption
- OTP - One-Time Pad
- ISMPC - Integer Secure Multi-party Protocols
- LSMPC - Long Secure Multi-party Protocols
- PLT - Plaintext, data is not sensitive and is left unprotected

This file, as any other XML file, has tree data structure with a root node and multiple children. The file starts with the root element with the tag name "*schema*" which has at least two children: a default element with the name tag "*default*" and at least a table element with the tag "*table*". The "*default*" sets the default protection mode that should be used to protect data on the level in which it is defined. For instance, if this element is defined as a child of the root "schema" element it defines the defaults for every table but if defined within the context of a "*table"* element, it defines the defaults for that specific table. This element cannot be defined as a child of any other element. The properties that can be defined inside the element "default" are the following:

- *key* - Defines the default cryptographic scheme used to protect table identifiers.
- *column* - Defines the default cryptographic scheme used to protect table columns.
- *keypadding* - Flag value to determine if a binary padding is necessary on the table identifiers.
- *columnpadding* - Flag value to determine if a binary padding is necessary on the table columns.
- *keyformatsize* - Fixed identifier size that should be used if identifier padding is active.
- *columnformatsize*- Fixed column size that should be used if column padding is active.
- *encryptionmode* - Flag value to determine if cryptographic schemes should be used.

If a table identifier or column is protected with DET or OPE, it is necessary to apply padding in order to ensure that the values all have the same size and the system behaves correctly.
Besides the "*default*" element, there can be one or more "*table*" elements that have at most three child elements: an inner tag "*name*" that defines the table name, an optional "*default*" element that takes precedence over the schema "*default*" element, and a "columns" element which contains the information relevant to the table columns. The "columns" element has only a single child, the "*family*" element. Inside the "*family*" family element there is an element "*name*" which must always have the value "DQE", a "*cryptotechnique*" element that defines the default cryptographic technique used to protect every column qualifier and an optional "*qualifier"* element. Columns that need to be protected with specific schemas can specify a "*qualifier*" element, which has child nodes with the same properties as the "*default"* element. Any property defined in this "qualifier" element takes precedence over the defaults defined by the "*default" element at* table level, or schema level.
As an example, consider a relational table schema with the name Users that has two columns, Name and Age. These two columns are mapped to an HBase table in the column family "DQE" and the qualifiers "Name" and "Age". Furthermore, the columns are protected with STD and the table identifiers are left unprotected as PLT. A sample schema file of this example is shown in Figure 8.



```xml
<schema>
  <default>
    <key>PLT</key>
    <columns>PLT</columns>
    <keypadding>false</keypadding>
    <colpadding>false</colpadding>
    <keyformatsize>10</keyformatsize>
    <colformatsize>10</colformatsize>
    <encryptionmode>disable</encryptionmode>
  </default>
  <table>
    <name>Users</name>
    <default>
      <encryptionmode>enable</encryptionmode>
    </default>
    <columns>
      <family>
        <name>DQE</name>
        <cryptotechnique>PLT</cryptotechnique>
        <qualifier>
          <name>Name</name>
          <cryptotechnique>STD</cryptotechnique>
        </qualifier>
        <qualifier>
          <name>AGE</name>
          <cryptotechnique>STD</cryptotechnique>
        </qualifier>
      </family>
    </columns>
  </table>
</schema>
```

Figure 8: Sample Schema file for User table.

4 Secure multi-cloud application server

4.1 Brief description of the solution

The third solution is built on top of the Sharemind framework [SHAREMIND+08]. It uses multi-party computation (MPC) to process the queries on the untrusted domains. This requires very little resources on the trusted deployment and can also be used with data from multiple data owners without revealing it to other data owners.

The solution uses a simple client side component which secret-shares the literals used inside the query and then sends the classified literals and the query to the Sharemind servers. Inside the servers there is the new mod_sql module, which translates the SQL statements into Sharemind Analytics Engine operations and executes the operations. At the end, the results of the query are declassified and returned to the client. However, this approach is not fully secure: it does not protect the data owners' data from people that are allowed to execute SQL queries. To allow multiple non-trusting data owners, then one needs to use custom SecreC[4] [BLR14] programs to allow only specific queries that are whitelisted to only give aggregation results that do not leak private data.

4.2 New Features

We have made improvements since the initial release of the SQ3 in D3.4 – Non-elastic restricted Secure SQL Engine. We had the SQL statements translation to Sharemind Analytics Engine operations done on the client side in the trusted deployment, but we figured that it would be beneficial to move the translation into a Sharemind module called mod_sql. For this deliverable we did that refactoring and we gained a number of benefits:

- **Better audit logs** – Previously the logs contained only the Analytics Engine operations, but now we can also log SQL statements directly.
- **Less latency –** For one SQL statement many Analytics Engine operations were run as separate Sharemind program invocations. Now one SQL statement can be run inside one Sharemind program invocation and we got rid of the additional overhead of setting up multiple Sharemind processes for all the operations needed to process the statement.
- **Embedding SQL statements in SecreC –** Now we can embed SQL statements directly in the SecreC code. This simplifies access control and should be very convenient when writing Sharemind applications using SecreC.
- **Better access control –** We had a problem that we did not know, when it was okay to declassify some results. Now we have the full SQL query on the server side and in future work we might be able to automatically deduce that some query results are safe to publish without infringing on the privacy of data owners. Also we can now explicitly whitelist queries known to not leak private data.

[4] SecreC is a programming language for the Sharemind Framework.

4.3 Quickstart user manual

4.3.1 Installing

Compile **sql_server.sc** in the SecreC directory. Pass the SecreC directory as an include path to the SecreC compiler. Copy the compiled **sql_server.sb** program to the Sharemind servers scripts directories.

To load mod_sql in the Sharemind server, add the following lines to the server configuration file:

```
[Module sql]
File = libsharemind_mod_sql.so
```

4.3.2 Running

sharemind-sql-client needs to be configured to connect to the Sharemind servers. You will need the client library configuration file, public and private keys of the client and public keys of the servers. By default, *sharemind-sql-client* looks for the **client.conf** file in the following directories (in that order):

1. **~/.config/sharemind**
2. **/etc/xdg/sharemind**
3. **/etc/sharemind**

If your client.conf is in one of those directories, just executing *sharemind-sql-client* will work. You can also supply the path to the configuration file with the --config-file (or -c) argument.

Before using SQL for the first time, you should run the following command to create tables that SQL uses internally:

```
sharemind-sql-client --initbackend
```

To interpret SQL scripts pass the filenames as arguments to sharemind-sql-client:

```
sharemind-sql-client script1.sql script2.sql
```

If no filenames are supplied the client enters the read-eval-print loop (REPL).

4.3.3 Whitelisting

The *sharemind-sql-client* program declassifies the output of all SELECT queries and is not safe.

To only allow specific SQL programs, execute them from a SecreC script. Using the SQL module as a SecreC library is described in **docs/secrec-tutorial.md**. If the **sql_server.sc** program is not installed then we rely on the same model as SecreC programs – whoever compiles and installs a SecreC program on the server verifies its source code (including the embedded SQL programs) prior to installing.

5 Conclusion

This deliverable gave a quick overview of the solutions in the Secure queries layer of the SafeCloud project. It also contains quick instructions on how to configure and run these solutions, but more detailed information is available in other SafeCloud work package 3 deliverables[5].

[5] http://www.safecloud-project.eu/results/deliverables

6 References

[APACHE17c] Apache Omid (Optimistically transaction Management In Datastores) documentation. (https://omid.incubator.apache.org), 2017

[BLW08] Dan Bogdanov, Sven Laur, Jan Willemson. Sharemind: a framework for fast privacy-preserving computations**.** In Proceedings of 13th European Symposium on Research in Computer Security, ESORICS 2008, LNCS, vol. 5283, pp. 192-206. Springer, Heidelberg. 2008.

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