Ref. Ares(2017)6389282 - 31/12/2017

Project title: Enforceable Security in the Cloud to Uphold Data Ownership
Project acronym: ESCUDO-CLOUD
Funding scheme: H2020-ICT-2014
Topic: ICT-07-2014
Project duration: January 2015 – December 2017

D1.5
Use Case Prototypes

Editors: Daniel Bernau (SAP)
Andreas Fischer (SAP)
Anselme Kemgne Tueno (SAP)
Reviewers: Stefano Paraboschi (UNIBG)
Pierangela Samarati (UNIMI)

Abstract

This deliverable gives a summary on each of the four ESCUDO-CLOUD modular use cases. To this end, the relevant background as well as the architecture and transfer of research will be illustrated per use case. Use Case 1 on Key Management in OpenStack introduces a flexible, hierarchical key management for the Swift object store that is allowing efficient key updates and secure deletion under data-at-rest-encryption. Use Case 2 on Secure Enterprise Data Management is leveraging Encryption and Oblivious Computation for processing over client-side encrypted data at an honest-but-curious Cloud Provider. Use Case 3 is on Data Protection as a Service (DPaas), a Cloud security service ensuring that data is protected by encryption and regulatory compliance requirements are met. Use Case 4 addresses an Elastic Cloud Service Provider where a middleware component is orchestrating a set of security services in an elastic Cloud storage service adjusting its capacity in a multi-Cloud environment without compromising on the security of the data.

Type	Identifier	Dissemination	Date
Deliverable	D1.5	Public	2017.12.31

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 644579. This work was supported in part by the Swiss State Secretariat for Education, Research and Innovation (SERI) under contract No 150087. The opinions expressed and arguments employed herein do not necessarily reflect the official views of the European Commission or the Swiss Government.

ESCUDO-CLOUD Consortium

1.	Università degli Studi di Milano	UNIMI	Italy
2.	British Telecom	BT	United Kingdom
3.	EMC Corporation	EMC	Ireland
4.	IBM Research GmbH	IBM	Switzerland
5.	SAP SE	SAP	Germany
6.	Technische Universität Darmstadt	TUD	Germany
7.	Università degli Studi di Bergamo	UNIBG	Italy
8.	Wellness Telecom	WT	Spain

Disclaimer: The information in this document is provided "as is", and no guarantee or warranty is given that the information is fit for any particular purpose. The below referenced consortium members shall have no liability for damages of any kind including without limitation direct, special, indirect, or consequential damages that may result from the use of these materials subject to any liability which is mandatory due to applicable law. Copyright 2017 by British Telecom, EMC Corporation, IBM Research GmbH, SAP SE, Wellness Telecom.

Versions

Version	Date	Description
0.1	2017.11.30	Initial Release
0.2	2017.12.14	Second Release
1.0	2017.12.31	Final Release

List of Contributors

This document contains contributions from different ESCUDO-CLOUD partners. Contributors for the chapters of this deliverable are presented in the following table.

Chapter	Author(s)
Executive Summary	Daniel Bernau (SAP)
Chapter 1: Use Case 1 OpenStack Framework	Mathias Björkqvist (IBM), Christian Cachin (IBM), Björn Tackmann (IBM)
Chapter 2: Use Case 2 Secure Enterprise Data Management in the Cloud	Anselme Kemgne Tueno (SAP), Daniel Bernau (SAP), Andreas Fischer (SAP)
Chapter 3: Use Case 3 Federated Secure Cloud Storage	Ali Sajjad (BT), Geŕy Ducatel (BT), Mark Shackleton (BT)
Chapter 4: Use Case 4 Elastic Cloud Service Provider	Sabine Delaitre (WT), Cristina de los Santos (WT), Andrew Byrne (EMC)
Chapter 5: Conclusion	All Partners

Contents

Executive Summary **9**

1 Use Case 1: OpenStack Framework **11**
1.1 Background . . . 11
1.1.1 Overview . . . 11
1.1.2 Context in ESCUDO-CLOUD . . . 12
1.1.3 Background on OpenStack Swift . . . 12
1.1.4 Availability . . . 13
1.1.5 License . . . 13
1.2 Solution . . . 13
1.2.1 Functionality and goals . . . 14
1.2.2 Architecture and components . . . 17
1.2.3 Getting started . . . 19
1.3 Summary . . . 24

2 Use Case 2: Secure Enterprise Data Management in the Cloud **25**
2.1 Background . . . 25
2.1.1 Overview . . . 25
2.1.2 Context in ESCUDO-CLOUD . . . 25
2.1.3 Background on Multi-User Information Sharing . . . 25
2.2 Solution . . . 28
2.2.1 Solution Approach . . . 29
2.2.2 Oblivious Order-Preserving Encryption . . . 32
2.2.3 OOPE Integration . . . 38
2.3 Summary . . . 51

3 Use Case 3: Federated Secure Cloud Storage **53**
3.1 Background . . . 53
3.1.1 Overview . . . 53
3.1.2 Context in ESCUDO-CLOUD . . . 53
3.1.3 Background on DPaaS . . . 53
3.2 Solution . . . 55
3.2.1 Functionality and goals . . . 55
3.2.2 Block Storage Encryption . . . 56
3.2.3 Object Storage Encryption . . . 62
3.2.4 Big Data Encryption . . . 63
3.3 Summary . . . 65

4 Use Case 4: Elastic Cloud Service Provider **66**
4.1 Background . . . 66
4.1.1 Overview . . . 66
4.1.2 Context in ESCUDO-CLOUD . . . 66
4.1.3 UC4: Core Technologies and Concepts . . . 67
4.2 Solution . . . 68
4.3 Use Case 4 Architecture . . . 69
4.3.1 Core Functional Blocks . . . 69
4.3.2 ECS Integration . . . 70
4.3.3 User Experience . . . 71
4.3.4 Data Structure . . . 71
4.3.5 REST API . . . 71
4.3.6 Architecture Summary . . . 73
4.4 Tools for Data Protection . . . 74
4.4.1 Client-side Encryption . . . 74
4.4.2 Key Management . . . 74
4.4.3 Shuffle Index . . . 75
4.4.4 Over-Encryption . . . 77
4.5 Tools for Compliance Checking . . . 77
4.5.1 RSA Archer . . . 78
4.5.2 RSA Archer Integration . . . 79
4.5.3 Use Case 4 Requirements and Policy Management . . . 79
4.5.4 Service Provider Assessment . . . 81
4.6 Summary . . . 83

5 Conclusion **85**

Bibliography **86**

List of Figures

1.1 Key hierarchy entity . . . 14
1.2 Key architecture, distribution and usage by different components . . . 15
1.3 Key rotation and secure deletion of object . . . 16
1.4 Key rotation and secure deletion of container . . . 17
1.5 Structure of a Cloud service with OpenStack Swift and Barbican . . . 18
1.6 The `rotating_keymaster` and `rotating_encryption` components in Swift. . 18

2.1 Illustration of how learning, encryption and encrypted query execution is split up among the parties in the aerofleet management use case . . . 26
2.2 Overall SEEED Architecture . . . 27
2.3 Onion before and after layer removal . . . 27
2.4 Client-Server Model (example database) . . . 29
2.5 User Interface Interaction (example database) . . . 30
2.6 Customer Application . . . 31
2.7 Server Application . . . 32
2.8 MRO Application . . . 33
2.9 Example initialization . . . 36
2.10 Oblivious OPE Protocol . . . 37
2.11 Customer's View . . . 40
2.12 Database Structure . . . 40
2.13 System Architecture . . . 41
2.14 DO Application Architecture . . . 41
2.15 DO Application Classes . . . 43
2.16 CSP Application Classes . . . 44
2.17 DA Application Classes . . . 45
2.18 Homomorphic Encryption Classes . . . 45
2.19 OOPE Classes for DO . . . 46
2.20 OOPE Classes for CSP . . . 47
2.21 OOPE Classes for DA . . . 47
2.22 Database Classes . . . 49
2.23 Decision Tree Classes . . . 50
2.24 OOPE Sequence Diagram . . . 51

3.1 High level view of the DPaaS solution design . . . 56
3.2 Architecture of the block storage encryption service component of the DPaaS solution . . . 56
3.3 Implementation reference of the block storage encryption service prototype . . . 57

3.4 Architecture of the object storage encryption service component of the DPaaS solution . . . 62
3.5 Implementation reference of the object storage encryption service prototype . . . 63
3.6 Architecture of the HDFS encryption component of the DPaaS solution . . . 64
3.7 Implementation reference of the Big Data encryption service prototype . . . 65

4.1 Overview of the Use Case 4 Trust Boundaries . . . 67
4.2 Elastic secure cloud storage architecture . . . 69
4.3 Use Case 4 user actions sequence . . . 72
4.4 Use Case 4 data structure . . . 73
4.5 Security Features . . . 74
4.6 Use Case 4 encryption sequence . . . 75
4.7 Use Case 4 decryption sequence . . . 76
4.8 Use Case 4 Shuffle Index implementation . . . 76
4.9 Use Case 4 Over Encryption (On Resource Mode) . . . 78
4.10 Archer Risk Assessment Framework . . . 79
4.11 Elastic cloud service provider architecture with policy checking . . . 80
4.12 Sample mapping between UC requirements and Gold, Silver, Bronze services . . . 82
4.13 Archer Risk Assessment Dashboard . . . 83

Executive Summary

This deliverable provides an integrated view of the contributions and findings that were produced while working on the Work Package 1 (WP1) use cases. WP1 is devoted to four use cases considered by the ESCUDO-CLOUD project. Use cases were designed along requirements that have been introduced in D1.1 and reshaped in D1.2.

The final outcome is a collection of modular, compatible, and inter-operable, tools representing the ESCUDO-CLOUD framework. The tools address four security dimensions. *Security properties*, represented by Confidentiality, Integrity and Availability. These encompass protecting outsourced data, validating authenticity and integrity, and evaluating whether Cloud providers meet Service Level Agreements. *Access requirements*, represented by protection of actual data, computations over data and their results, and access patterns of data in the Cloud. *Sharing requirements*, addressing the need for differentiating access control between data owners and other users in the Cloud. *Cloud architectures*, focusing on the assessment and enforcement of security standards among multiple Cloud providers.

In brief, the four use cases developed tools addressing the security dimensions and technical challenges through the following scenarios and goals.

Use Case 1: OpenStack Framework. Use Case 1 describes the data-at-rest-protection and key-management solution for OpenStack Swift developed in the context of ESCUDO-CLOUD. Swift is the object-store component of the open-source cloud-service framework OpenStack. We developed a flexible, *hierarchical key management* in which each object stored in Swift is protected with an individual key, and these object keys are wrapped by keys that are on higher levels in the hierarchy. Besides the fact that this structure aligns well with Swift's hierarchy of accounts, containers, and objects, one main advantage of this approach is its flexibility. In particular, it allows for *updating the keys* in some branch of the hierarchy, while leaving the remaining part untouched. This efficient key-update operation is instrumental in implementing *secure deletion* in our data-at-rest protection in Swift.

Use Case 2: Secure Enterprise Data Management in the Cloud. Use Case 2 considers the use case of outsourcing supply chain interactions in the aerospace engine maintenance industry. The technological challenge of this use case is to develop new supply chain cooperation systems, based on encrypted database technology in the Cloud. This technology is based on the search and aggregation of encrypted data. Supply chain collaboration is the alignment of individual plans and strategies of the involved parties. The stronger coordination and the overcoming of information asymmetries among involved parties shall conduce to improvements of the supply chain performance.

Use Case 3: Federated Secure Cloud Storage. Use Case 3 has the objective to ensure the confidentiality and integrity of the customer's data when it is outsourced for storage to different Cloud storage services. The primary challenge addressed by Use Case 3 is to offer the key

management and policy-based access control capabilities to customers through a Cloud service store. Each customer should be able to apply key release rules and conditions, taking into account user access rights. These objectives are achieved by building a prototype solution called Data Protection as a Service (DPaaS). In the context of Use Case 3, three types of storage services are addressed: block storage, object storage, and Big Data (HDSF) storage services. The DPaaS solution also addresses aspects of controlling access to the outsourced data through policy-based security rules that enforce the release of encryption keys.

Use Case 4: Elastic Cloud Service Provider. Use Case 4 considers the implementation of an elastic Cloud provider, targeting a data storage service. The resulting reference architecture and prototype is of particular relevance to Cloud service brokers and private Cloud providers that offer secure, accessible, and scalable storage in the Cloud. The architecture is also applicable to organizations hosting local datacenters that intend to employ Cloud bursting to exploit relatively cheap storage resources of the Cloud. These objectives present a data protection challenge whereby end users must entrust third parties (i.e., the Cloud provider) with the protection of their data. To mitigate such risks, UC4 uses client-side encryption techniques, and secure authentication and key management processes enabling the data owner to remain in complete control of the security protecting their data.

The use case implementations were fed from selected results of WP2, WP3, and WP4 that performed in-depth research on basic protection, sharing and multi-clouds to guarantee security properties and access and sharing requirements considering different cloud architectures. Furthermore, use case implementations were driven by the industrial partners. Use Case 1 by IBM, Use Case 2 by SAP, Use Case 3 by BT, and Use Case 4 by EMC and WT.

The remainder of this document is organized as follows: Chapter 1 introduces the solution for Use Case 1. Chapter 2 describes the solution for Use Case 2. The solution for Use Case 3 is presented in Chapter 3. The Use Case 4 tools are stated in Chapter 4. Lastly, Chapter 5 formulates an overall conclusion.

1. Use Case 1: OpenStack Framework

1.1 Background

1.1.1 Overview

Cloud services have turned remote computation and storage into a commodity and conveniently allow clients to seamlessly scale resources on demand. However, the fact that client data is transferred to, and stored by, the Cloud service provider introduces new security risks.

One important aspect of data security in the context of Cloud storage is data-at-rest protection, meaning that the data is encrypted before being written to persistent storage media. This aspect is complementary to data-in-transit protection, where the data is encrypted while it travels over a network. In data-at-rest protection, the customer's plaintext data is encrypted in the data center of the Cloud service provider, who therefore in principle has access to the plaintext data. The access to the plaintext data is, however, restricted to those systems in the data center that handle the encryption, while the actual storage systems only deal with encrypted data. Encryption of data at rest and secure data deletion are important requirements especially for enterprise-level Cloud storage services. Encryption of data at rest is required to handle the case of stolen or improperly-decommissioned storage media (e.g., disks, tapes) by minimizing the exposure of plaintext data. Secure data deletion is required to delete large quantities of customer data and must be time-efficient. Overwriting-based deletion techniques are ineffective on certain media and vastly inefficient in general, and the physical destruction of storage media permits for very little flexibility when it comes to the selection of which files, objects or volumes are required to be deleted. By contrast, cryptography-based secure deletion offers a flexible and efficient alternative: if data is encrypted and the encryption keys are properly managed and carefully destroyed, large subsets of data are instantaneously no longer accessible.

OpenStack is the leading open-source cloud-service framework, and is widely used in industry. In fact, a wide subset of IBM's storage and Cloud storage offerings make use of the OpenStack framework, and extend it to offer enterprise-grade services to its customers. Swift is the object-store component of OpenStack. The deployed version of Swift provides data-at-rest protection by encrypting the objects before writing them to disk. The object encryption is performed using keys derived from a single master key that is set up in the Swift configuration. This scheme is simple, but has severe restrictions in terms of its flexibility. For instance, rotating any key in the entire Cloud storage, such as when a client terminates the service agreement and wants all her data to be destroyed, would require to re-encrypt all encrypted objects, which is not feasible in large-scale deployments.

This document describes the extension of OpenStack Swift with a flexible key-management technique that implements enterprise-level features such as the rotation of individual keys and uses this to provide selective secure deletion.

1.1.2 Context in ESCUDO-CLOUD

The main focus of this document is within Work Package 1 of ESCUDO-CLOUD, and describes the solution developed in the context of Use Case 1. The techniques are based on results from Work Package 2, which focuses on protection techniques for outsourced data. This encompasses the encryption of data at rest in Task 2.1, with the goal to guarantee to the data owner that the confidentiality and integrity of her data are adequately protected, especially when the outsourced data collection includes sensitive information. The main contributions used in the solution described in this document relate to Task 2.2, which develops flexible key-management techniques targeted toward their use in data-at-rest protection. The solution presented in this document implements enterprise-level security guarantees for data outsourcing, which is an explicitly stated goal of ESCUDO-CLOUD. The solution is implemented within OpenStack Swift.

1.1.3 Background on OpenStack Swift

OpenStack Swift, also referred to as OpenStack Object Storage, is an open-source object storage system and part of the OpenStack Cloud platform (`http://swift.openstack.org`). OpenStack Swift is best suited to backup and archive unstructured data, such as documents, images, audio and video files, email and virtual machine images. Clients access the system through a REST HTTP API.

OpenStack Swift is highly scalable, runs on standard hardware, and implements data resilience and redundancy in software. Objects and files are written to multiple disk drives spread throughout servers in the data center, with Swift responsible for ensuring data replication and integrity across the cluster. By default, Swift places three copies of every object in separate locations that are as different as possible—in terms of region, zone, server, and drive. If a server or hard drive fails, Swift replicates its content from active nodes to new locations in the cluster.

Storage clusters scale horizontally simply by adding new servers. Should a server or hard drive fail, OpenStack replicates its content from other active nodes to new locations in the cluster. Because OpenStack uses software logic to ensure data replication and distribution across different devices, inexpensive commodity hard drives and servers can be used.

Many descriptions of the architecture of Swift are available, for example, on the OpenStack website at `http://docs.openstack.org/developer/swift/overview_architecture.html`.

Data-at-rest encryption

The need to support encryption for data-at-rest protection in Swift has been recognized for a long time by now. During 2014 an initial "blueprint" and a corresponding prototype was released by IBM Research – Zurich (`https://blueprints.launchpad.net/swift/+spec/swift-enc-proxy`). This triggered a design process in close connection with the core development team of Swift. Starting from the initial design specification, a team at HP, IBM (mainly IBM Corp. and with inputs from IBM Research – Zurich), and SwiftStack, has implemented the basic encryption functionality.

For an overview and introduction to the architecture of the data-at-rest encryption feature, see the online documentation at `https://docs.openstack.org/developer/swift/overview_encryption.html`.

Key management

The *keymaster* component in the first release of the data encryption feature has limited functionality in the sense that it supports only *one* shared key for the whole Swift cluster, and this key must be stored on every proxy server in the configuration data.

In this document, the design and an implementation for a more flexible key management module are described. The high-level goals stated before make the encryption feature more flexible: for example, towards supporting common security policies that require periodic key rotation, and for providing a secure deletion feature, whereby data can be provably forgotten (assuming that a master key can be rotated and securely deleted).

The design is based on a "SmartKeyMaster" and on a prototype of a file system supporting policy-based secure deletion, as described by Cachin et al. [CHHS13].

1.1.4 Availability

All code contributed to OpenStack in the context of use case 1 is available as open source at `https://github.com/ibm-research/swift-keyrotate`.

1.1.5 License

The code is provided under the following license.

> Copyright (c) 2017 IBM Corp.
>
> Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at
>
> `http://www.apache.org/licenses/LICENSE-2.0`
>
> Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

1.2 Solution

OpenStack Swift consists of client-facing (stateless) *proxy* nodes to which clients connect and that handle a large part of its functionality. The proxy nodes access *object*, *container*, and *account* servers, which are responsible for persistently storing object, container, and account-related data. The object, container, and account servers exist in multiple redundant instances. Data replication, for tolerating failures of one or more of the storage servers, is handled by the proxy nodes.

The main contribution described in this document is part of a *keymaster* WSGI middleware, which is one of many middleware components of the Swift proxy server pipeline. The *keymaster* middleware performs key-management tasks and provides encryption keys for other middleware, in particular for the *encryption* middleware. The encryption middleware encrypts and decrypts user and system metadata, as well as the actual user data that is stored in the Swift object storage system.

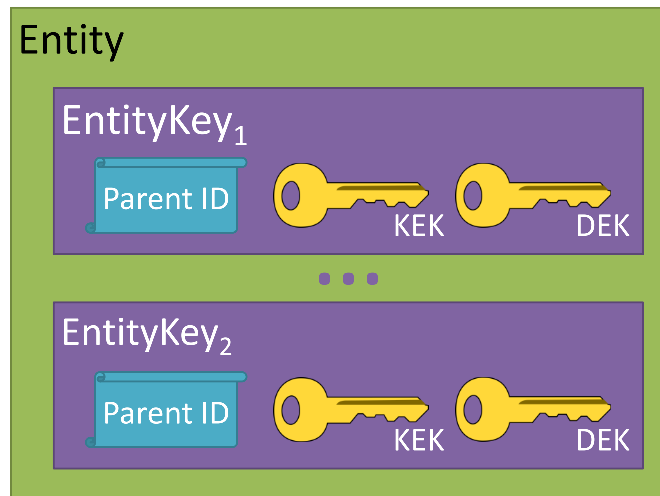


Figure 1.1: Key hierarchy entity

1.2.1 Functionality and goals

Structure. Swift organizes and manages data in terms of *accounts*, *containers* and *objects*. The *keymaster* abstracts Swift accounts, containers, and objects as *entity* objects. Swift objects belong to containers, and containers belong to accounts. Swift users authenticate to the system using a separate service such as OpenStack Keystone, and Swift requests are identified as belonging to a certain account, and possibly also to a container and an object. Which keys need to be retrieved, created and/or provisioned by the keymaster depends on the type of entity that the request operates on.

Our *keymaster* operates on Entities, which contain *key-encryption keys (KEKs)*, *data encryption keys (DEKs)*, and the ID of the parent Entity whose KEK is used to wrap the current Entity's KEK, as shown in Figure 1.1. The KEKs are only used internally in the keymaster to wrap other KEKs, as well as DEKs. The DEKs are provided to other middleware, in particular to the decrypter and encrypter middlewares, which in turn use the DEKs to decrypt and encrypt user and system metadata, as well as user data.

Figure 1.2 illustrates the different entities and the provisioning of keys. The figure shows the top-level Master key stored in Barbican, and the Master key is used to wrap the Account KEK. The Account KEK is used to wrap both the corresponding Account DEK, as well as the Container KEK. In the same way, the Container KEK is used to wrap the Container DEK and the Object KEK, and finally the Object KEK is used to wrap the Object DEK.

The master key(s) are generated by, and stored in, Barbican. The requests for generating the keys and storing them in Barbican are sent by the user directly to Barbican. The Swift keymaster retrieves the user's keys from Barbican using the user's authentication token, which is passed in with regular Swift requests. The fact that the user's authentication token is required for accessing the keys from Barbican means that any key management operation that requires access to the master keys from Barbican, can only be performed as part of a user request, and, e.g., not as regularly scheduled background tasks.

Key rotation and secure deletion. According to commonly accepted security practice, encryption keys must be periodically rotated, for various reasons that are best described in the relevant

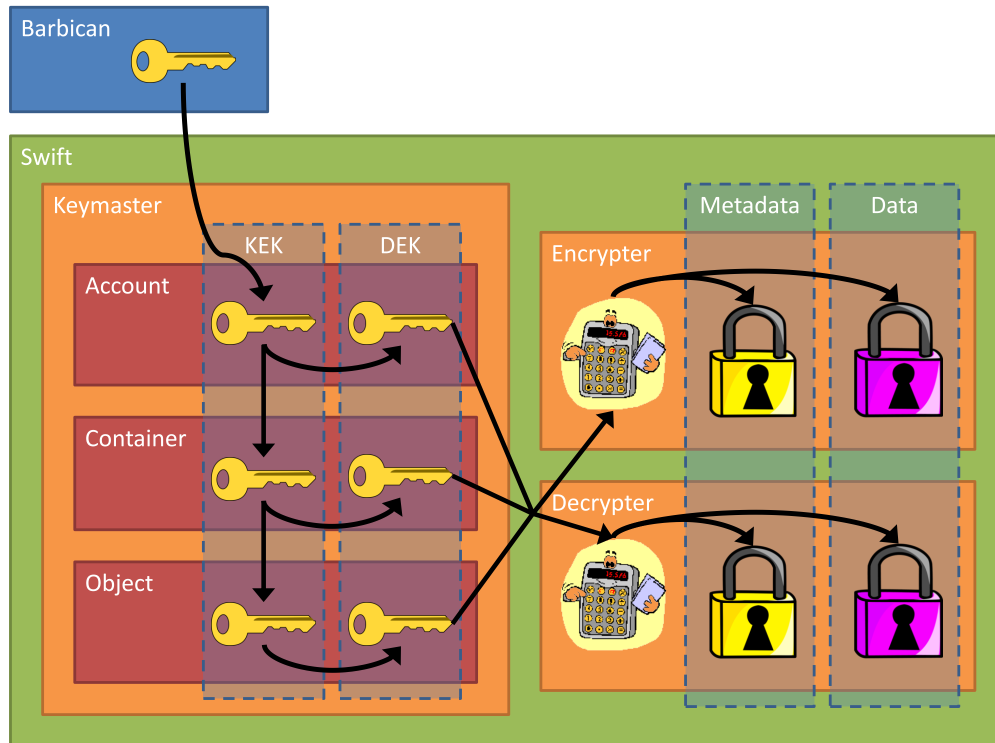


Figure 1.2: Key architecture, distribution and usage by different components

guidelines of NIST [Bar16, BBB+05, BD15]. Industry has supported this requirement through systems for key-lifecycle management [BCH+10].

As a first step, one could consider a scheme where only the key at the top level (the account key) can be rotated. Yet, this provides only partial protection because an adversary could have locally stored some key that lies below the account key in the hierarchy. This would allow the adversary access to all underlying data. Instead, in the design described here, key rotation is performed on all levels; this is necessary, among other reasons, for supporting secure cryptographic deletion.

For cryptographically deleting an object o, all the parent KEKs of the object to be deleted are rotated (e.g., its container key, the account key, and the master key), and then all the KEKs whose parent KEK changed are re-wrapped. Once the old master key is securely deleted, the object o will be cryptographically deleted, i.e., the encrypted data might still be accessible, but the DEK required to decrypt the data might no longer be available. Obviously the adversary could have simply stored a copy of the plaintext data to be securely deleted; this is an attack that cannot be prevented, therefore it is not considered in this design.

Figures 1.3 and 1.4 show the key rotation process when securely deleting an object and a container, respectively. The initial state is shown as yellow keys - in both figures, there are two containers, C_1 and C_2, each containing two objects, O_{11} and O_{12} in container C_1, and O_{21} and O_{22} in container C_2. The new, re-keyed, keys are shown in red, and old keys that are deleted as part of the key rotation process are shown with red X symbols.

In Figure 1.3, the user wants to delete object O_{22}, corresponding to KEK $O_{22}K_1$. The Swift delete operation may already have been executed, in which case the object data is no longer accessible through the regular Swift APIs, but low-level access to the physical storage medium may still reveal the encrypted data of the deleted object. As long as the keys needed to decrypt the

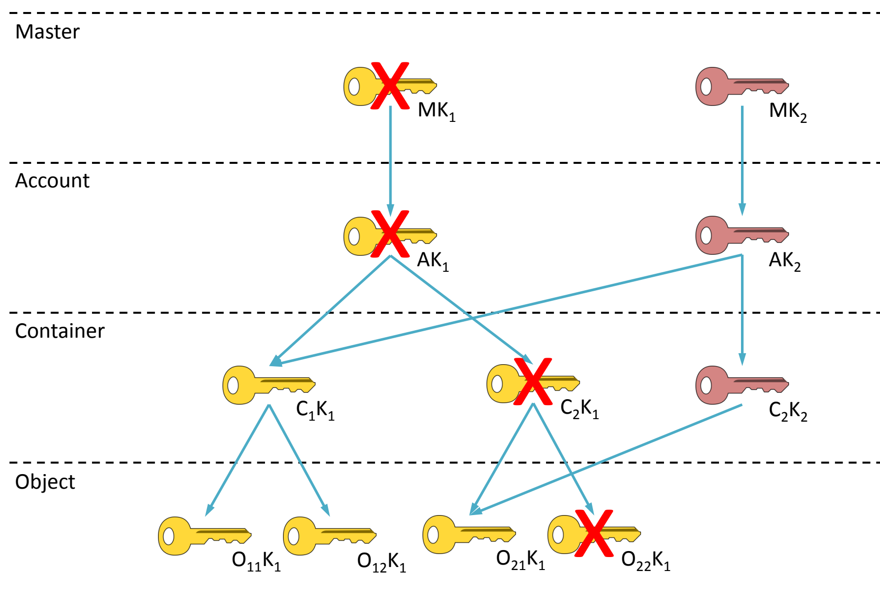


Figure 1.3: Key rotation and secure deletion of object

object exist, from the Swift encryption body key, to the DEKs and KEKs of the key hierarchy all the way up to the master KEK, it may be possible for an attacker to recover the data. To securely delete the data, one or more keys in the key hierarchy shall be securely deleted. To this end, key rotation is performed, where the keys for the entities that are to be kept are rotated, while the keys for the entities that are to be securely deleted are not rotated. When securely deleting object O_{22} in Figure 1.3, the KEKs of the parent container C_1 and the account, as well as the master key, are re-keyed. Subsequently, the KEKs of all entities whose parent KEKs have changed, are rewrapped with the new parent key. In this case, the KEK of object O_{21} is rewrapped with the new container key C_2K_2, and the KEK of container C_1 is rewrapped with the new account key AK_2. Once the old keys are purged from the system, object O_{22} is effectively cryptographically deleted. The other object in container C_2 can still be accessed using its old KEK $O_{21}K_1$, and the other container C_1 can still be accessed using its old KEK C_1K_1.

Figure 1.4 shows a similar scenario, but here the user wants to securely delete an entire container C_2, instead of just a single object within the container. In this case, the KEKs of all parent entities of the container C_2 (i.e., AK_1 and MK_1) are re-keyed and the KEKs of all entities whose parent KEK has changed (i.e., container C_1) are re-wrapped. Once the old keys are deleted, up to and including the old master key MK_1, the container C_2 and all objects stored in it are securely deleted.

Performance-wise the key rotation process can be quite expensive, and when deleting multiple objects it may be desirable to postpone the key rotation until all objects in the current deletion batch have been removed. Another argument for delaying the key rotation process is to implement a “grace period”, where a user has a possibility to undo the delete operations within a certain period of time, e.g., a day, before the operation is committed by performing key rotation. To support the above scenarios, it is possible to decouple the deletion operation from the key rotation process, and perform them separately. Objects and containers can then be deleted using the regular delete operations, which do not involve any key rotation. Once the operations are to be committed, the

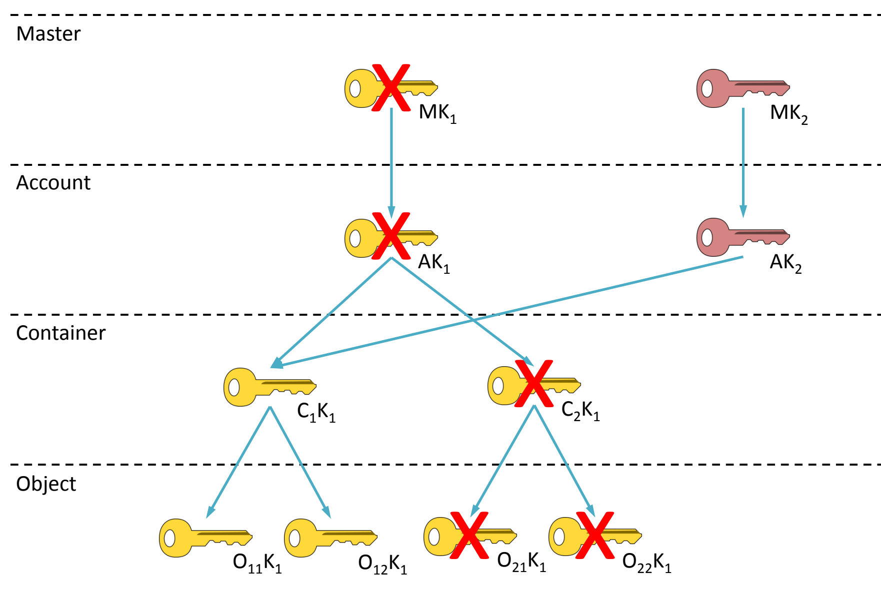


Figure 1.4: Key rotation and secure deletion of container

user performs a key rotation on the parent entities of all entities that have been deleted, i.e., for deleted objects the parent containers are re-keyed, and for deleted containers, the parent account is re-keyed.

1.2.2 Architecture and components

The high-level structure of the implementation can best be described via Figure 1.5. To start with, the client first needs to create a master secret in Barbican. For regular operations (e.g., GET, PUT), the client sends a REST request for reading or writing an object to the proxy server. The proxy server retrieves the master secret from the Barbican key server. It then obtains the account data from the account server, the metadata that is associated with the account contains the account keys, which are wrapped with the master key. The corresponding steps are also performed to obtain the container and object metadata from the container server and the object server, respectively. The object metadata then contain the object data encryption key which is used to wrap the object body key, which is finally used to decrypt (or encrypt) the actual object data.

Internally, the Swift proxy server is structured as a pipeline of modules that process the request. Each request first passes the pipeline in one direction, where the final module performs the actual write to, or read from, the servers that actually store the data, and passes the result back through the pipeline. The data-at-rest protection is implemented in two modules which are inserted into the pipeline as

```
keymaster encryption
```

which ensures that the proper cryptographic keys are prepared by the `keymaster` before the `encryption` module processes a write- or read-request. The enhanced key-management techniques developed for this use case are implemented as an alternative `keymaster` component, and as a slightly modified `encryption` component. The high-level design is depicted in Figure 1.6.

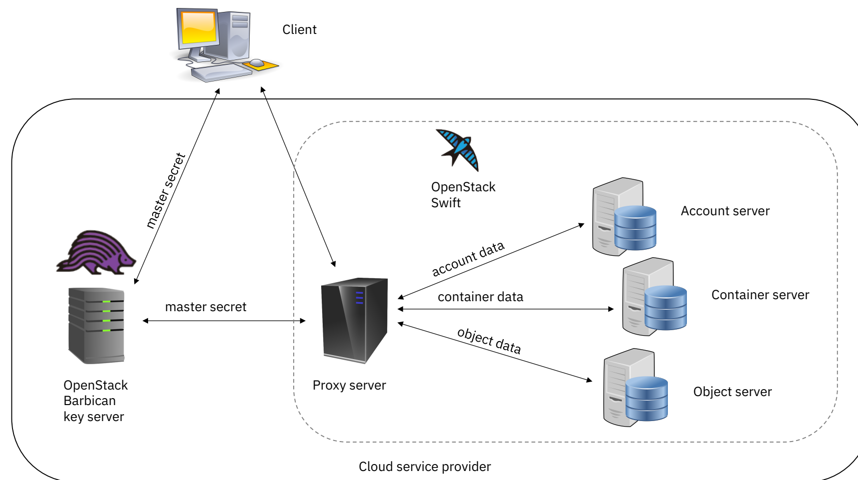


Figure 1.5: Structure of a Cloud service with OpenStack Swift and Barbican

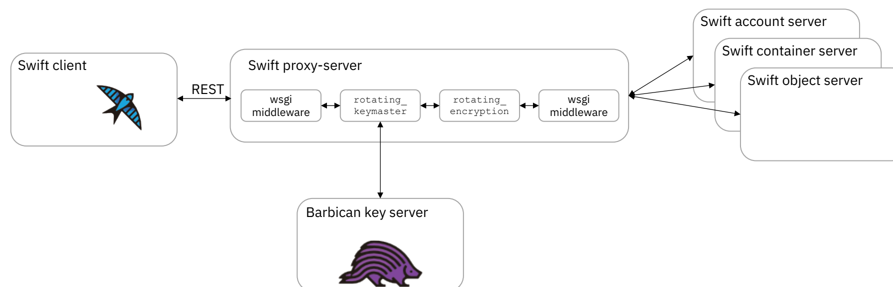


Figure 1.6: The `rotating_keymaster` and `rotating_encryption` components in Swift.

The new keymaster component, called `rotating_keymaster`, accesses the master secret stored in Barbican and performs all computations on the key hierarchy. To do so, it retrieves the metadata for the respective account, container, and object in sub-queries to the respective Swift services, and unwraps the respective keys stored in the metadata. Finally, it provides access to the object-encryption key to the `rotating_encryption` component via a callback interface.

A client can request services of the key-management component using specific header fields in the REST query, and in particular, POST operations. The *Rekey* header instructs the keymaster to re-key the entity that is the target of the POST operation, whereas a *Rewrap* header instructs the keymaster to re-wrap the entity that is the target of the POST operation. Re-keying involves generating a new KEK for the entity, and wrapping it with a specific KEK of the entity's parent by explicitly specifying it as the value to the Rekey header, or simply using the latest KEK of the entity's parent. Re-wrapping is the same as re-keying, except that no new KEK is generated—the existing KEK is merely re-wrapped with the KEK of a specific parent entity, or if no parent entity is specified, the KEK of the latest parent entity. Re-keying can be performed on accounts and containers, and re-wrapping can be performed on containers or objects. Re-keying and re-wrapping results in the keys associated with the corresponding entity to be changed. The concept of *key rotation* involves one or more re-keying and re-wrapping operations. When rotating an account key, all containers keys that are wrapped by the account key are re-wrapped with the new account key, and when rotating a container key, all object keys that are wrapped by this key are re-wrapped.

The standard Swift client has been extended to support re-keying and secure deletion. Re-keying simply sets the corresponding flag in the REST header. Secure deletion of an entity deletes the respective entity, and then re-keys the entity on the next-higher level. This guarantees that even if a copy of the original object were still to be found, it would be impossible to decrypt because even knowledge of the current master key does not allow to unwrap the key that protects the object.

1.2.3 Getting started

This section describes two ways of configuring an OpenStack Swift setup with the hierarchical key management keymaster and encryption features. The first paragraph below assumes that the OpenStack Swift, Keystone and Barbican services are already installed, and describes how to configure Swift to use the new keymaster. The second paragraph describes how an entire development environment can be set up from scratch using Vagrant and VirtualBox virtual machines (VMs).

Installation requirements and procedure

The first step is to set up an OpenStack environment including the Barbican service. We refer to a standard OpenStack manual for this process. Swift also needs to be configured to use Keystone for authentication, and not e.g., *tempauth*. This is because the user's authentication token is used by Swift to retrieve the user's root encryption secrets from Barbican, so the two services shall use the same authentication and authorization service.

Next, the hierarchical keymaster needs to be installed on the Swift proxy node(s). This is done by first cloning the swift-keyrotate repository:

```
$ git clone https://github.com/ibm-research/swift-keyrotate.git
```

Thereafter, the requirements and the hierarchical keymaster middleware itself are installed:

```
$ cd swift-keyrotate/swift
$ sudo pip install -r requirements.txt
$ sudo python setup.py develop
$ cd -
```

The data-at-rest encryption and the advanced key management are configured by adding

```
rotating_keymaster rotating_encryption
```

into the pipeline of the proxy server. The `rotating_keymaster` and `rotating_encryption` filters are configured by adding sections

```
[filter:rotating_keymaster]
use = egg:swiftkeyrotate#rotating_keymaster
keymaster_config_path = /etc/swift/rotating_keymaster.conf

[filter:rotating_encryption]
use = egg:swiftkeyrotate#rotating_encryption
```

to the configuration of the proxy server, and the file `/etc/swift/rotating_keymaster.conf` which contains the following configuration directives:

```
[rotating_keymaster]
auth_endpoint = http://<keystone-IP>/identity/v3
api_class = swiftkeyrotate.keyrotate_key_manager.KeyrotateKeyManager
```

where the keystone-IP placeholder is replaced by the proper value according to the installation.

To facilitate the user of the new hierarchical key management features, also set up the modified python Swift client located in the same git repository as the Swift module:

```
$ cd swift-keyrotate/python-swiftclient
$ sudo pip install -r requirements.txt
$ sudo python setup.py develop
$ cd -
```

The new commands are *swift rekey* and *swift secdel*. See the paragraph on the demonstration workflow below for more details on the usage of the new commands.

Installation of a test environment using Vagrant

An easy set-up for testing purposes can be achieved using the pre-configured Vagrant environment included in the repository. This setup requires VirtualBox (available at `http://www.virtualbox.org`) and Vagrant (available at `http://www.vagrantup.com`) to be installed on the host system. To set up the test environment with the modified Swift instance, the following procedure is used: First clone the `swift-keyrotate` repository via

```
$ git clone https://github.com/ibm-research/swift-keyrotate.git
```

on the local hard drive. This repository contains all data to set up a local installation of Swift.

Second, provision the virtual machines. This will install one `swift-services` VM that runs the base services (Keystone and Barbican), and one `swift` VM that runs the Swift server.

```
$ vagrant up
```

The process will take several minutes, since virtual-machine images will be downloaded and provisioned with the OpenStack software.

Finally, log in to the Swift VM and load the credentials of the Swift user that is pre-installed by the scripts.

```
$ vagrant ssh swift
$ source ~/openrc.swiftuser
```

Now the `swift-keyrotate` functionality is available, as described in the subsequent paragraphs.

Completing the setup

As described in the previous paragraphs, the master secret is stored in the Barbican key server. Using the OpenStack command-line tools, we can list the root secrets in Barbican:

```
$ openstack secret list
```

If no secrets exist in Barbican, use the *rekey* command in the swift command-line client to create one, along with an account key of the present user.

```
$ swift rekey
```

Using the new features

The modified Swift client introduces two new commands to the `swift` command line client, for re-keying containers and for securely deleting containers and objects.

1. For rekeying a container, specify the `rekey` sub-command along with the identifier of the container (here: `cont1`).

   ```
   $ swift rekey cont1
   ```

2. Secure deletion is implemented via the `secdel` sub-command. To securely delete object `obj1` from container `cont1`, issue the following commands.

   ```
   $ swift secdel cont1 obj1
   ```

   Containers can be securely deleted analogously, by only specifying the identifier of the container that shall be deleted.

A demonstration workflow

The following workflow shows the use and the effects of the rekeying and secure deletion functionalities.

Preparation. The first step is to create objects in the Swift object store that can later be used to demonstrate the actual functionality.

1. Create some temporary files to upload.

   ```
   $ echo obj11 > obj11
   $ echo obj12 > obj12
   $ echo obj13 > obj13
   $ echo obj21 > obj21
   ```

2. Upload files; three in container `cont1`, one in container `cont2`.

   ```
   $ swift upload cont1 obj11
   $ swift upload cont1 obj12
   $ swift upload cont1 obj13
   $ swift upload cont2 obj21
   ```

3. Show the account metadata, indicating that the account key is wrapped with the root encryption secret in Barbican.

   ```
   $ swift stat
   ```

4. Show the container metadata, indicating that the container keys of `cont1` and `cont2` have been wrapped with the account key.

   ```
   $ swift stat cont1
   $ swift stat cont2
   ```

5. Show the object metadata of `obj12` in container `cont1`, showing that the key is wrapped with the container key of `cont1`.

```
$ swift stat cont1 obj12
```

6. Download an object to verify that it works.

```
$ swift download cont1 obj12
$ cat obj12
```

Secure deletion. The second part of the demonstration workflow shows the secure deletion of objects.

1. Securely delete `obj11` from container `cont1`.

```
$ swift secdel cont1 obj11
```

2. Show the object metadata of `obj12` in container `cont1`, showing that the key is wrapped with the new container key of `cont1`. The key ID of the object key is still the same, since it was only rewrapped, not rekeyed.

```
$ swift stat cont1 obj12
```

3. Show the container metadata, indicating that the container keys of `cont1` and `cont2` have been wrapped with the new account key. Also note that the key ID of `cont2` is the same as before - the old key was merely rewrapped as part of the secure deletion process.

```
$ swift stat cont1
$ swift stat cont2
```

4. Show the account metadata, indicating that the new account key is wrapped with the new root encryption secret in Barbican.

```
$ swift stat
```

5. List the root encryption secrets in Barbican. Note that the old root encryption secret(s) have been deleted as part of the secure deletion process.

```
$ openstack secret list -c "Secret href" -c "Created"
```

The process in more detail. We now demonstrate the implementation of secure deletion by combining a regular deletion of object with an explicit rekey; together these operations securely delete the object.

1. Show the container metadata for `cont1`.

```
$ swift stat cont1
```

2. Delete an object as usual.

```
$ swift delete cont1 obj12
```

3. Show the metadata of the container. Since it has not changed, if the content of the deleted object were to be retrieved (e.g., from a backup), it would still be possible to decrypt and read the object.

```
$ swift stat cont1
```

4. Explicitly rekey the container. This generates new container, account, and root encryption keys. Once the old ones have been deleted, the deleted object `obj12` has effectively been securely deleted.

```
$ swift rekey cont1
```

5. Show the metadata of the container to see that the key has changed.

```
$ swift stat cont1
```

6. For demonstration purposes, we show secure deletion of object (rekeying), while still retaining the actual object data. First, show the metadata of the object

```
$ swift stat cont1 obj13
```

7. Securely delete an object using the demo only-flag `-retain`, which only rekeys/rewraps parents/siblings, but does not actually delete the object.

```
$ swift secdel --retain cont1 obj13
```

8. Show the metadata of the object. The etag is now basically garbage, since it is encrypted but could not be decrypted, since the keys no longer exist.

```
$ swift stat cont1 obj13
```

9. Trying to download the object fails with 403 Permission Denied. This error is generated by the Swift server because the object cannot be decrypted.

```
$ swift download cont1 obj13
```

10. Regular deletion of object still succeeds, since this operation does not need encryption keys.

```
$ swift delete cont1 obj13
```

1.3 Summary

OpenStack is the leading open-source Cloud platform. Prior to ESCUDO-CLOUD, however, its object-store component Swift was lacking enterprise-grade data-at-rest protection. In the context of Tasks 2.1 and 2.2, we developed a method for data-at-rest encryption for Swift, along with enterprise-grade key-management functionality that supports flexible key-rotation and efficient secure deletion of objects and containers. In WP1, Use Case 1, we implemented the cryptographic methods within Swift as filters in the web-service pipeline.

Through collaboration and synchronization with the OpenStack project, solutions produced in ESCUDO-CLOUD have been integrated into the main OpenStack distribution. In particular, the basic data-at-rest encryption has been adopted by the OpenStack project as part of the recent Newton release of Swift. Parts of the advanced key-management functionality are currently under review for being integrated into the main OpenStack distribution. The complete code is available as open source.

As encryption and key management are performed at the server, client software does not have to be adapted to benefit from the improved security brought by encrypting the data. Secure deletion of objects or containers, which is invoked by the client, is accessible through an extension of the standard Swift API. In summary, through the contributions of ESCUDO-CLOUD, Cloud service providers using OpenStack Swift as their Cloud object store service now have access to enterprise-grade data-at-rest protection.

2. Use Case 2: Secure Enterprise Data Management in the Cloud

2.1 Background

2.1.1 Overview

This chapter describes the technology and architecture of ESCUDO-CLOUD Use Case 2. Use Case 2 tools are designed for a supply chain optimization scenario of maintenance (respectively demand) forecasting between multiple parties in the Cloud. The parties are represented by Customers (airlines) who outsource their data to a common Cloud Service Provider (CSP) so that Maintenance Provider (MRO) can perform efficient maintenance scheduling by secure collaborative forecasting. The Cloud Service Provider does not have access to customers' data in plain text. In such a scenario there is the need to enforce possible access restrictions on data stored in the Cloud according to specified access restrictions. Thus, users are allowed to access different views of data, and to modify different portions of such data. Such techniques should be non-bypassable and guaranteed not to open the door to possible violations of the confidentiality or integrity (e.g., by the provider in isolation or in collusion with users of the system) provided by underlying protection techniques. The problem of ensuring confidentiality of policies that are evaluated between the parties (i.e., decision trees for forecasting airplane spare parts) are also considered. Our techniques are designed considering the trade-off between protecting confidentiality on one side, and the need to guarantee efficient access execution on the other side, meaning they are server-agnostic and directly applicable to common cloud interfaces [DFLS16].

2.1.2 Context in ESCUDO-CLOUD

Use Case 2 tools utilize encrypted query processing as investigated from research performed under WP3. Based on the research concepts, Use Case 2 implementation is also leveraging oblivious order-preserving encryption (OOPE) to preserve the privacy of both Customer and MRO. The outlined prototype extends existing work of SAP Security Research Karlsruhe project SEEED, which is also featured in the EU funded projects PRACTICE and TREDISEC. The extensions in ESCUDO-CLOUD are, for example, represented by Oblivious Order-Preserving Encryption and our encrypted query processing in multi-user applications.

2.1.3 Background on Multi-User Information Sharing

The aerofleet management use case (Use Case 2) considers a scenario in which multiple customers (i.e., airlines) and a maintenance provider (i.e., MRO) share data via a secure computation approach to enable a CSP to execute an encrypted predictive model. The participants and a description of their roles in this use case, as well as the flow of information, are provided in Figure 2.1.

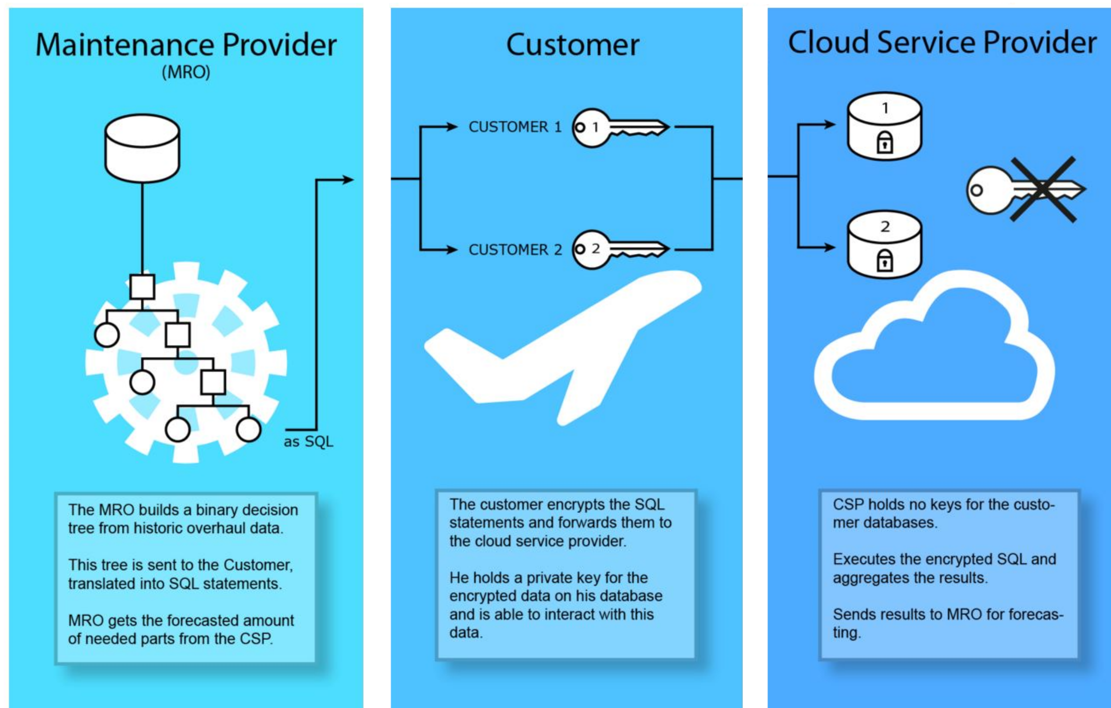


Figure 2.1: Illustration of how learning, encryption and encrypted query execution is split up among the parties in the aerofleet management use case

SEEED

SEEED (Search over Encrypted Data) allows data held in a relational database to be outsourced to a *Honest-but-Curious* CSP using encryption. SEEED follows the *privacy-by-design* principle and does not require encryption keys to be shared with untrusted parties. Instead, it makes use of property-preserving encryption schemes allowing the Database Management System (DBMS) to perform meaningful computations on ciphertexts.

The overall architecture of SEEED is illustrated in Figure 2.2. End users establish a connection to a trusted application server using a secure protocol such as HTTPS. The unmodified application sends plaintext SQL queries to the SEEED JDBC driver, which translates them into queries on encrypted data. The untrusted DBMS at the CSP (e.g., SAP HANA or MySQL) executes the translated query and returns the final result to the SEEED JDBC driver for decryption. In order to do so, the SEEED JDBC driver has access to the encryption keys stored at the application server.

Adjustable Encryption

Popa et al. [PRZB11] offer an intriguing solution to the encryption type selection problem. They exploit the fact that the selected Order-Preserving Encryption (OPE) scheme supports a superset of queries supported by Deterministic Encryption (DET). DET in turn supports a superset of queries supported by Randomized Encryption (RND). These properties are used to store values in layered ciphertexts called onions. For each data item x several onions are computed and stored in different database columns.

The first onion defined by Popa et al. is constructed by applying the available encryption

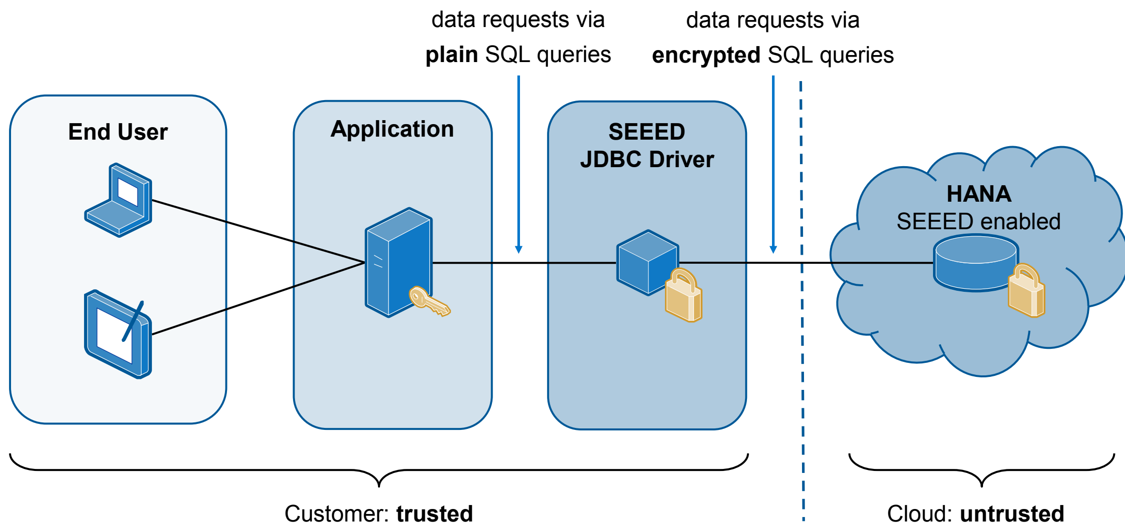


Figure 2.2: Overall SEEED Architecture

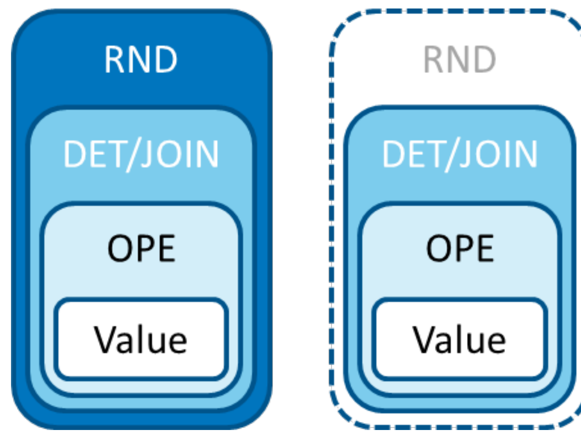


Figure 2.3: Onion before and after layer removal

schemes in the following sequence:

$$E_{RND}(E_{DET}(E_{OPE}(x)))$$

At first, this onion only allows data retrieval because *RND* does not support any computations on the ciphertext. The *RND* layer is removed when the client encounters a query that requires deterministic encryption, for example a selection using equality. To remove the layer, the client sends the decryption key for $D_{RND}()$ to the DBMS. The DBMS invokes a user-defined function to perform an update such that $E_{DET}(E_{OPE}(x))$ is stored in the database. Now, the equality query can be executed. The same procedure is applied in case a query requires order-preserving encryption. Figure 2.3 illustrates the onion structure and the onion removal process.

Additive homomorphic encryption (*HOM*) is handled in a separate onion and stored in a separate column. This separate column enables aggregation operations, but does not harm security, since *HOM* provides the same security level as *RND*. Layering is not possible in this case as it would destroy the homomorphic property provided by *HOM*.

The combination of multiple onions with their layered encryption schemes result in a database adjustment mechanism enabling the support of a wide range of queries. This mechanism allows

	ONION 1	ONION 2	ONION 3	ONION 4
LAYER 1	DET	OPE	OPE	HOM
LAYER 2	RND	RND	JOIN	
LAYER 3			RND	
TYPES	Integer, String, Date, Decimal	String	Integer, Date, Decimal	Integer, Decimal
USAGE	Retrieval, Group By	Joins with Strings, Range queries with strings	Joins, Equality, Range queries with numbers	Aggregation

Table 2.1: Onion structures used by SEEED

the database to be adjusted dynamically without knowing all possible SQL queries in advance. We call such a database *adjustably encrypted*. The adjustment is unidirectional: once decrypted to deterministic or order-preserving encryption, it is not necessary to return to a higher encryption level. This is the case because all inner layers strictly support the functionality of outer layers. Furthermore, security against the CSP has already been weakened, because the less secure ciphertexts have been revealed at some point and thus have to be assumed available for cryptanalysis.

SEEED employs four onions to support multiple SQL operations within the same query. For example, an SQL query may perform aggregation while also containing a range condition. Moreover, we provide separate onion structures for integers and strings. We refer to Table 2.1 for a detailed overview. The *types* row contains plaintext data types to which the corresponding onion can be applied. For example, to allow for aggregation of values, integers require the additive homomorphic encryption scheme, but strings do not. For sorting items using *OPE*, different paddings are used for integer and string values. ONION 1 is used for data retrieval and GROUP BY selections only because having an onion solely relying on AES-based encryption schemes allows for very efficient decryption of result sets on the client.

MySQL

MySQL is an open source DBMS owned by Oracle. It is used for web-based applications like Facebook and Twitter, but also Cloud services can be built on it. It supports SQL and can easily be accessed from an application. SQL stands for Structured Query Language which is the most used language to access relational databases. The MySQL database is published under the GPL (General Public License). It is a relational database storing the structure in files to optimize speed.

2.2 Solution

This section presents the solution approach of our client-server application that solves the problem described in the previous section.

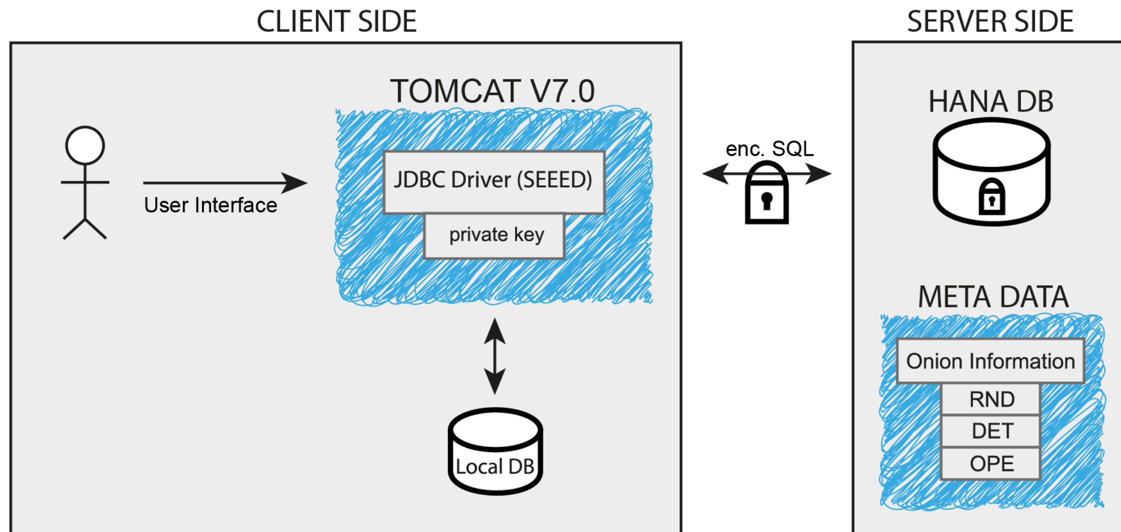


Figure 2.4: Client-Server Model (example database)

2.2.1 Solution Approach

First, we describe a core solution that only provides privacy for customers and will be extended in section 2.2.3 to also provide privacy for the maintenance provider.

System Architecture

The Client-Server Model. Since the application utilizes SEEED to search over encrypted data, its architecture must also follow the client-server model as illustrated in Figure 2.4. The untrusted CSP hosts a database server that contains encrypted customer's data. Moreover, the database server stores some metadata, which contain information for the onion encryption. In the current implementation of the prototype, each customer has her own tables, where her data is stored encrypted with her private encryption keys. The encryption keys are stored on the client side in a local database and never leave this trusted environment. The client application is a web application that interacts with a local Tomcat v7 web server application. The SEEED driver is the central component on the web application server that is used to connect and send queries to the database server. The user enters queries in plaintext on the client user interface, which sends them to the web server application. The web server uses the SEEED driver to transform the query in an encrypted form, that can be executed on the encrypted data. After the query is executed, the web server application received the encrypted results, decrypts them, and sends the plain result to the client application.

User Interface. The user interface of the client application is realized with SAPUI5, a design framework that uses JavaScript and a model-view-controller architecture. The logic is implemented in Java classes. Some of these Java classes are servlets that interface the communication between the client and the database server. Figure 2.5 illustrates how this interaction is performed. A user interacts with the JavaScript frontend and triggers a GET or POST request, which is processed by a servlet on the web server application. The servlet makes use of other Java classes to connect to the encrypted database via the SEEED driver. After the encrypted result is decrypted

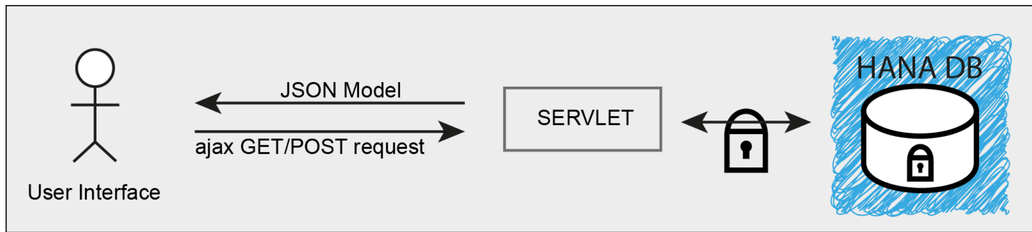


Figure 2.5: User Interface Interaction (example database)

the data is transformed into the data exchange format JSON and loaded by a JSON model in a controller of the client application. This JSON model is linked to a displaying element, in this example a table, and lets it display the desired data. It should be noticed that the result of the request is only decrypted if the SEEED driver finds the appropriate decryption keys in the local key storage. Hence in the views of the MRO or the CSP customers' data are always displayed in encrypted form.

Application overview

The solution consists of three distributed applications, one for each actor in the system. The Customer's application encrypts the data and stores them on the database server. The MRO application constructs a decision tree and transforms it to SQL requests that are encrypted by customers and later executed on the encrypted databases. The CSP application receives encrypted SQL queries from customers, executes them and sends the result to the MRO. In our prototype implementation there is only one frontend with one view for each actor. In a production implementation the views will be completely separated in different applications to support the novel concept of oblivious order-preserving encryption (see Section 2.2.3). Each view interacts with a corresponding servlet to access the data on the database server. In these interactions Ajax GET requests are triggered by user actions on the frontend and submitted with a user identification number. The servlets process these requests and later transform the resulted data into a JSON array, which is forwarded to the view.

Customer Application. This application consists of the *customer servlet* and the *database accessor*.
The *customer servlet* checks which customer is active and triggers a GET request with the customer ID attached. After the request is processed, the servlet builds a JSON array out of the data and sends it back to the view where it is displayed. For this display the SEEED driver is needed to decrypt the data, which is stored in the cloud database.
The *database accessor* sets up the connection between the prototype application and the HANA database, using the SEEED Driver or a simple plain connection. Furthermore, it provides the SEEED driver with several meta data tables and user information about the database in use. In the current prototype, there is only one database accessor component for all parties, but there would be an accessor for every role that needs a database connection.

Server Application. In contrast to the *customer servlet*, the *cloud servlet* forwards data in an encrypted format. This ensures, that the CSP cannot read any plaintext from the customer tables within her database.

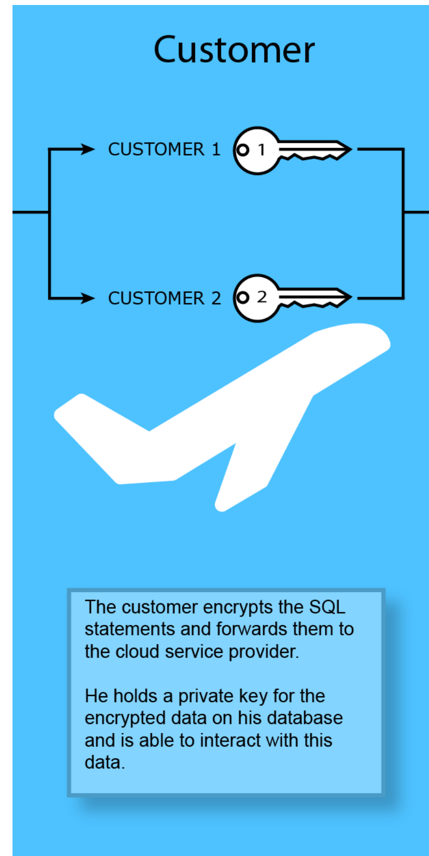


Figure 2.6: Customer Application

MRO Application. This application consists of the *MRO servlet*, the *decision tree*, the *binary decision tree*.
The *MRO servlet* provides the evaluation data. Furthermore, it provides the data from which the binary tree is visually generated. For that, two Java classes are needed. First, the MRO Class which builds a JSON object and fills it with the data of the decision tree: attribute names of the nodes, number of nodes, executed operation. Second, the aggregated probability of a upcoming overhaul, which results out of the given path of the tree.
The Java class *decision tree* builds up the binary decision tree from the given historical data of the service provider. Furthermore, it generates a SQL query that describes the tree. This SQL statement is forwarded to the customer who has the key to her database and is therefore able to encrypt the SQL the same way the data in her cloud database is. In that way, after forwarding the needed SQL to the CSP, the CSP is able to use the encrypted SQL for a search on the encrypted data without awareness of the SQL request or the requested data itself.
The *binary decision tree* is a Java class, which is essential for forecasting. For the prototype the tree is built manually, but in a production environment historical overhaul-data should be used to build it. It consists of the following information: The attribute to be tested, the operation that has to be applied (<,>,=), the threshold of said attribute which defines whether a part should be exchanged or repaired.

The live data of the attribute of a part from the customer database is compared with the threshold in coherence with the operation. Each node contains such a statement that could be true or false. After computing the whole tree for a specified part, a repair/replace probability for each leaf

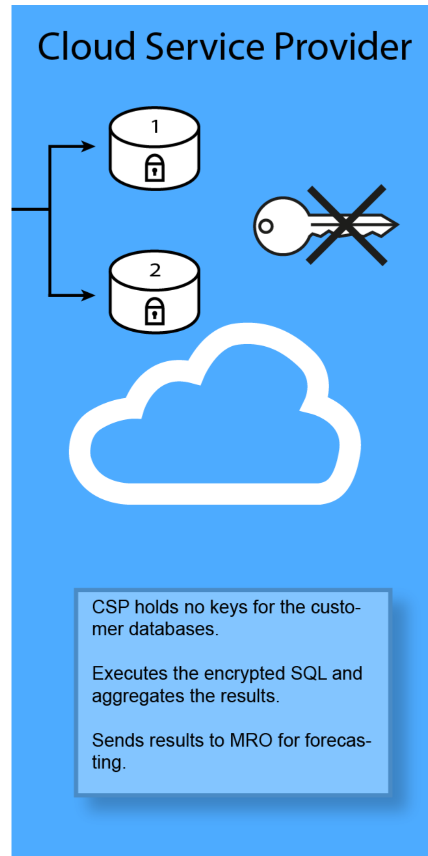


Figure 2.7: Server Application

is computed.

2.2.2 Oblivious Order-Preserving Encryption

This section presents an oblivious order-preserving encryption protocol, which allows the MRO to encrypt the decision tree without revealing any information on it to the customer. At the end of the protocol the MRO will receive a tree encrypted with the customer's private key. No further information will be revealed to any party participating in the protocol.

Intuition

So far we have been concerned about the privacy of customer data (i.e., customer's privacy). However, there is another security property that we want to ensure. The application does not give access to customer data in plain form to the MRO. Hence to run the classifier on encrypted data the MRO must first send the decision tree to the customer, who can then compute a corresponding encrypted version of the tree. However, assume the tree itself contains sensitive information. After all, the MRO has accomplished some amount of work to produce the tree and may not want to reveal any information about it. Therefore, one should also consider the MRO's privacy.
A naive way for the MRO to classify customer data with her decision tree while preserving both customer's and MRO's privacy is to use a private decision tree classifier. This allows to traverse the decision tree using the customer's input vector x such that the MRO does not learn the input x, and the customer does not learn the structure of the tree and the thresholds at each node [BPTG14]. In

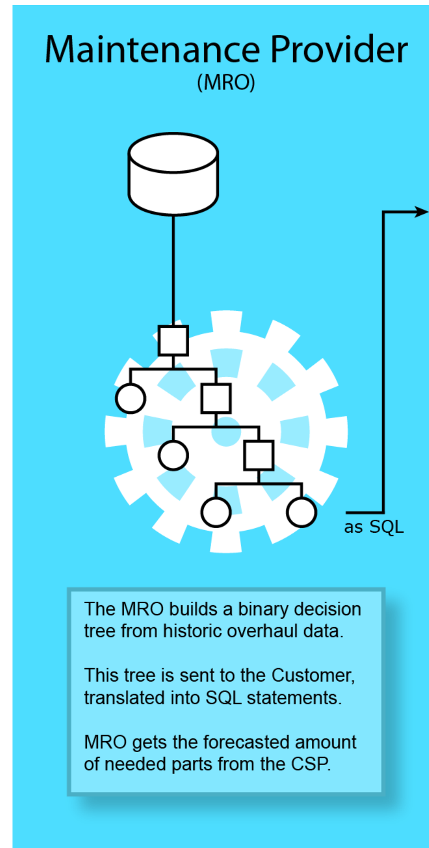


Figure 2.8: MRO Application

particular, the customer should not learn the path in the tree that corresponds to x since the position of the path in the tree and the length of the path leak information about the model. Assume that input x is a data record with n fields. While traversing the tree, the MRO and the customer can successively compute a garbled circuit for comparison where the customer garbles the circuit and the MRO evaluates it. The inputs to the garbled circuit are respectively a threshold in the decision tree for the MRO and the corresponding data field for the customer. Although current implementations of garbled circuits are efficient, the computation costs to classify a whole database is assumed too high. Furthermore, since garbled circuits require sending garbled tables and inputs for each new computation over the network the communication costs will also be very large.

Another way to privately evaluate the decision tree is to express it as a polynomial P whose output is the result of the classification, the class predicted for x. In this interpretation, the MRO and the customer privately compute inputs to this polynomial based on x and the thresholds. Finally, the MRO evaluates the polynomial P privately [BPTG14]. This solution also suffers from the same problems as the garbled circuits approach, since it can only classify one input at a time and the customer must be involved in the classification protocol.

To reduce the communication costs one can consider a solution based on fully homomorphic encryption (FHE), where the customer encrypts her data with a public-key FHE scheme before outsourcing them to the cloud. In this case, the MRO can encrypt the thresholds of the decision tree using the public-key FHE scheme and then use the homomorphic property to compute the comparison. Since FHEs return encrypted results, a customer still has to decrypt the result of the comparison. Also this approach will have a large overhead as only one data record can

be classified at a time and FHE schemes are still not efficient enough for practical applications. Furthermore, with FHE there is another problem, namely the fact that a malicious MRO can use the FHE evaluation algorithm in another function instead of the comparison. For example it can homomorphically evaluate the addition of a ciphertext of an unknown plaintext m with a random ciphertext of zero to learn m after decryption.
As already mentioned in the previous section, we will rather encrypt the customer's dataset with an order-preserving encryption. Afterwards, the decision tree is encrypted with the same order-preserving encryption using the same key, and finally transformed into a set of SQL-queries that can be executed directly on the encrypted data. To preserve the MRO's privacy, oblivious OPE allows the MRO to encrypt the thresholds of the decision tree without revealing any information to the customers. Customers are only involved during the encryption of the tree. The MRO sends SQL-queries directly to the database server during classification.

Building Blocks

Mutable OPE (mOPE). Order-preserving encryptions leak the order of the plaintexts because they allow to perform comparison directly on the ciphertexts. Hence the ideal security guarantee that can be achieved with OPEs is that no information about the plaintexts besides the order is revealed. In [PLZ13] Popa et al. showed that the ideal security of OPE is possible by relaxing the definition and allowing some ciphertexts to mutate during the encryption process. In the following, we briefly review this scheme and an improved variant [KS14] that are relevant for oblivious OPE.
Popa et al.'s scheme (mOPE$_1$). In [PLZ13] Popa et al. presented the first OPE that achieves ideal security by using the fact that most OPE applications only require a relaxed OPE interface which is less restrictive than the interface of encryption schemes. The first aspect of this relaxed interface is interactivity: the encryption scheme is implemented as a protocol running between a client that also owns the data to be encrypted and an honest-but-curious server that stores the data. The second aspect is the mutability property which allows some ciphertexts of already-encrypted values to change over time as new plaintexts are encrypted. The basic idea of Popa et al.'s scheme is to have the encoded values organized at the server in a binary search tree (*OPE-tree*), where each node is encrypted with an encryption scheme (AES). A binary search tree is a tree in which for each node v, all the nodes in the left subtree of v are strictly smaller than v and all the nodes in the right subtree of v are strictly larger than v. To encrypt a value x the client and the server traverse the tree, where the client receives the current node v of the search tree, decrypts and compares it with x. If x is smaller (resp. larger) then they continue with the left (resp. right) child node of v. If the chosen childnode v' is null then x is inserted at this place otherwise v' must be decrypted and compared by the client. The OPE encoding of x is then the path from the root of the tree to x, where an edge to the left (resp. to the right) is encoded as 0 (resp. 1), padded with $10\cdots0$ to the same length l. The server also maintains a table (*OPE-table*) containing the AES ciphertext and corresponding OPE encoding for each encrypted plaintext. Before encoding a value x the server first looks in the OPE table. To ensure that the length of OPE encodings do not exceed the defined length l, the server must occasionally perform balancing operations, which require updates of the OPE table (i.e the OPE encodings of some already encrypted values mutate to another encoding).
Kerschbaum & Schröpfer's scheme (mOPE$_2$). The insertion cost of Popa et al's scheme is high because server and client must traverse the tree together. To tackle this problem, Kerschbaum & Schröpfer proposed in [KS14] another ideal secure, but significantly more efficient, OPE scheme. Both schemes use binary search and are mutable, but the main difference is that in the scheme of

[KS14] the state is not stored on the server but on the client. Hence the client does not need AES encryption as in Popa et al.'s scheme. Instead the client maintains for each plaintext x a pair $\langle x, y \rangle$ in the client state, where y is the OPE encoding of x. To insert a new plaintext the client finds two pairs $\langle x_i, y_i \rangle$, $\langle x_{i+1}, y_{i+1} \rangle$ in the state such that $x_i \leq x < x_{i+1}$ and computes the OPE encoding as follows:

- if $x_i = x$ then the OPE encoding of x is $y = y_i$.
- if $y_{i+1} - y_i = 1$ then the client balances the search tree.
- if $y_{i+1} - y_i > 1$ then the OPE encoding of x is $y = y_i + \left\lceil \frac{y_{i+1} - y_i}{2} \right\rceil$.

The encryption algorithm is keyless and the only secret information is the state, which is not pre-generated but rather grows with the number of encryptions. The client uses dictionary compression to keep the state minimum and hence does not need to store a copy of the data.

Yao's Garbled Circuit. In garbled circuit protocols [LP07, LP09, PSSW09] a party called Generator garbles the Boolean circuit of the function to be computed and sends it to a second party, Evaluator, that evaluates and outputs the result. To garble the circuit the generator chooses randomly for each input/output wire one secret key representing 0 and another secret key representing 1. Then, it replaces the bits by the corresponding secret keys in the truth table for each gate of the circuit and uses the input keys to encrypt the output keys. The result is called Garbled Table. Assume outputs of gates G_1 and G_2 are inputs of a gate G_3 and that G_1, G_2 have already been garbled. Then, to garble G_3 the generator randomly chooses two output keys and uses the output keys of G_1 and G_2 as input keys for G_3. The garbling procedure results in a set of garbled tables that are sent to the evaluator as well as the keys corresponding to generator's input. Finally, both parties engage in an oblivious transfer protocol that allows the evaluator to learn the keys corresponding to her input without revealing any information on the actual input to the generator.

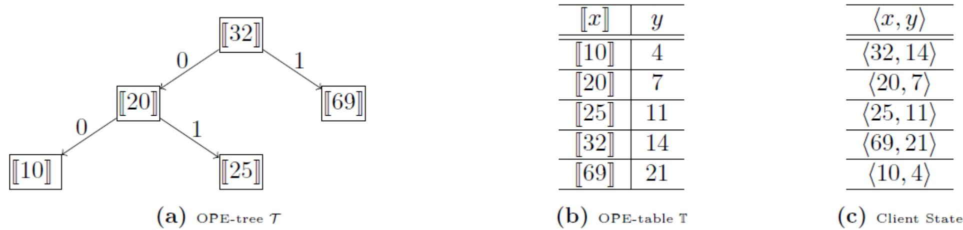
Homomorphic encryption. A homomorphic encryption scheme is an encryption scheme that allows computations on ciphertexts by generating an encrypted result whose decryption matches the result of operations on the corresponding plaintexts. With fully-homomorphic encryption schemes one can compute any efficiently computable function, but with the current state of the art, their computational overhead is still too big for practical applications. Efficient alternatives are additive homomorphic encryption schemes which allow specific arithmetic operations on plaintexts, by applying an efficient operation on the ciphertexts. To illustrate the concept of oblivious OPE protocol we will use the additive homomorphic encryption scheme of Paillier. It is a public-key scheme and has semantic security.

Oblivious OPE Protocol

An OOPE protocol is a three-party protocol. The first party, i.e., the Data Owner (DO), encrypts her data with an order-preserving encryption as described in the previous section and stores the encrypted data in a cloud database hosted by the second party, i.e., the CSP. The third party, Data Analyst (DA), needs to execute analytic queries, like how many values are in a given range, on the data owner's encrypted data. However, the DO's data is encrypted with a symmetric key and the DA's queries contain sensitive information. Therefore, the DA interacts with the DO and the CSP to order-preserving encrypt the sensitive queries values without learning anything else or revealing

any information on the sensitive queries values.

Initialization. We assume the DO's dataset is fixed and does not change, i.e., the DO does not insert new values during the OOPE protocol. This is a realistic assumption since the decision tree classification operates on fixed dataset. Let $\mathbb{D} = \{x_1 \cdots x_n\}$ be the unordered DO's dataset. First, the data owner uses mOPE$_2$ by Kerschbaum & Schröpfer to encrypt the data to a set of pairs $\langle x_i, y_i \rangle$ and sends the encrypted results y_i to the cloud as in [KS14]. Second, the DO uses mOPE$_1$ with Paillier instead of AES to generate the OPE-tree and the OPE-table as in [PLZ13]. Let $[\![x]\!]$ denote the Paillier encryption of x then the OPE table is a list of pairs $\langle [\![x_i]\!], z_i \rangle$, where z_i is the OPE encoding as in [PLZ13], i.e., the binary encoding of the path from the root of the tree to x padded with $10 \cdots 0$ to the length l of the OPE encoding. Then, the DO replaces each pair $\langle [\![x_i]\!], z_i \rangle$ of the OPE-table with the pair $\langle [\![x_i]\!], y_i \rangle$ and sends the OPE-tree and the OPE-table to the CSP. The reason of replacing z_i by y_i is that the DA will receive the order of her input after the OOPE protocol. However, z_i always reveals the corresponding path in the tree, allowing the DA to infer more information from the protocol than required. In contrast, mOPE$_2$ allows the DO to choose not just the length of the OPE encoding, but also the ciphertext space like $0 \cdots M-1$. If $log_2 M$ is larger than the needed length of the OPE encoding and M is not a power of 2, then y_i does not reveal the position of $[\![x_i]\!]$ in the tree.
For example, assume the data set $\mathbb{D} = \{10, 20, 25, 32, 69\}$, $l = 4$, $M = 28$ and the insertion order is $32, 20, 25, 69, 10$. The aforementioned initialization procedure generates the OPE-tree $\mathscr{A}$, the OPE-table $\mathbb{T}$, and the client state as depicted in Figure 2.9. Then the OPE-tree and the OPE-table are sent to the CSP as initial server state.



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

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

Figure 2.9: Example initialization

Algorithm. Let $\bar{x}$ be the input of the DA and $\bar{y}$ the corresponding OPE encoding that is to be computed. To compute $\bar{y}$ the three parties traverse the OPE-tree obliviously as depicted by Figure 2.10. Let $[\![x]\!]$ denote the current node (at beginning the root node) of the OPE-tree and $\langle [\![x]\!], y \rangle$ the corresponding pair in the OPE-table, then the parties execute an oblivious comparison protocol to compare $\bar{x}$ and x.

First, the CSP retrieves the current node $[\![x]\!]$ of the tree, chooses a random integer r, homomorphically computes $[\![x+r]\!]$, sends $[\![x+r]\!]$ to the DO, and sends r to the DA. Second, the DA and the DO engage in a Yao protocol for comparison with input $\bar{x}+r$ and $x+r$ respectively, where the garbled circuit has two output bits and is defined according to [KS08, KSS09]. The first bit b_g tells if x is larger than $\bar{x}$ and the second bit b_i tells if the inputs are unequal. Since the result of the comparison leaks information, the DA and the DO also input masking bits b_a and b_o respectively in the garbled circuit. Therefore, the actual outputs of the garbled circuit are $b_g \oplus b_a \oplus b_o$ and $b_i \oplus b_a \oplus b_o$, where $\oplus$ denotes the XOR operation.

The CSP receives the results of the comparison protocol, reconstructs b_g and b_i, and decides

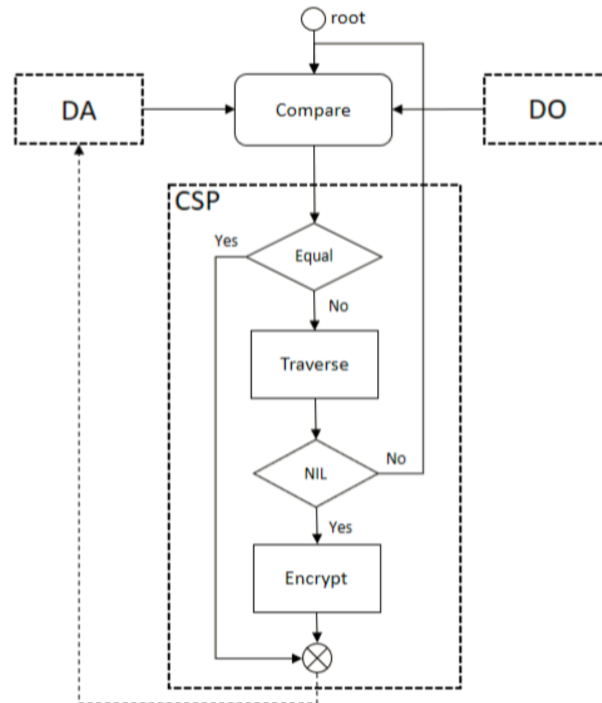


Figure 2.10: Oblivious OPE Protocol

what to do next:

- If $b_i = 0$ (i.e., $x = \bar{x}$) then the CSP looks for the pair $\langle [\![x]\!], y \rangle$ in the OPE-table and sets $\bar{y} = y$.
- If the input values x, $\bar{x}$ are unequal (i.e., $b_i = 1$) then the next node to consider is either the left child node or the right node child node of $[\![x]\!]$ depending on $b_g = 0$ or $b_g = 1$ respectively.
  - If the next node is not null, then the CSP randomizes again and sends the values for a new comparison step.
  - If the next node is null and $b_g = 0$ holds, then there exists $\langle [\![x']\!], y' \rangle$ in the OPE-table such that $x' < \bar{x} < x$: $\bar{y} = y' + \left\lceil \frac{y-y'}{2} \right\rceil$.
  - If the next node is null and $b_g = 1$ holds, then there exists $\langle [\![x'']\!], y'' \rangle$ in the OPE-table such that $x < \bar{x} < x''$: $\bar{y} = y + \left\lceil \frac{y''-y}{2} \right\rceil$.

In the last step, the CSP sends $\bar{y}$ to the DA, which sends $[\![\bar{x}]\!]$ back to the CSP. Notice that, even if $x = \bar{x}$ holds, the honest CSP must require the DA to send back the ciphertext $[\![\bar{x}]\!]$. This prevents the DA from inferring that in the last comparison the equality of inputs to the garbled circuit held.

Security. The security of the protocol depends on randomization of inputs, the security of Yao's protocol and the honesty of the CSP. If the CSP is honest and do not collude with any other party, then the view of the DO contains a series of randomized plaintexts data and her view of Yao's protocol, which does not even allow to learn the result of the comparisons because of the masking bit. The DA observes a series of random integers and her view of Yao's protocol, which is completely random as well. The CSP receives the ciphertexts $[\![x_i]\!]$ of Paillier's scheme, which is semantically secure, as well as the OPE encoding of mOPE_2, which is ideal secure.

SCAPI - Secure Communication API. We implemented the OOPE protocol described above with the SCAPI library. SCAPI is an open-source Java library for implementing secure two-party and multiparty computation protocols. It provides a reliable, efficient, and highly flexible cryptographic infrastructure [EFLL12]. The main idea of SCAPI is to offer a practical and applicable solution for secure computation. The difference between SCAPI and previous secure computation projects is that SCAPI does not want to solve a special problem. It is more general and independent of a specific environment. Secure computation can be useful for electronic voting, privacy-preserving data mining, or private database queries [EFLL12]. SCAPI provides different encryption primitives, among which oblivious transfer and garbled circuit are relevant for our application. Furthermore, it provides a communication layer for multiparty communication [EFLL12].

2.2.3 OOPE Integration

To preserve privacy for both MRO and Customer (airline), this section describes the integration of OOPE (from section 2.2.2) into the core solution (from section 2.2.1). To this end, the application is distributed among the involved parties which leverage OOPE to order-preservingly encrypt MRO's inputs while revealing only the corresponding order information to the MRO and nothing else. As before, the CSP also learns only the order information such that it can still execute queries from MRO. However, Customer remains oblivious on any input coming from the MRO.

Database Design

In UC2 the DO which is an airline stores information about its airplanes. This information consists of the following data:

- *engineid*: Engine ID,
- *enginesn*: Engine Serial Number,
- *enginepn*: Engine Part Number,
- *fh_limit*: Flight hour limit,
- *fc_limit*: Flight cycles limit,
- *tbsv*: Time before shop visit,
- *fhsn*: Flight hours since new,
- *fcsn*: Flight cycles since new,
- *tslsv*: Time since last shop visit.

The database contains a corresponding table, which we will refer to as the *data table*. The corresponding view from the application is illustrated in Figure 2.11 showing the decrypted data to the customer. Each column will be order-preserving encrypted allowing range queries to be executed on all columns without decryption. OOPE requires the underlying OPE scheme to be stateful. The state consists of an OPE-table and an OPE-tree. As the OPE-table can be derived from the OPE-tree, only the OPE-tree is stored in the same database as the data table. Each node of the tree consists of a homomorphic ciphertext and the corresponding order. Additionally, we store for each node some meta-information resulting in a database table with the following attributes:

- *hCipher*: Homomorphic ciphertext,
- *oCipher*: Order ciphertext,
- *isRoot*: Boolean value that is true if the node is the root of the tree,
- *pLchild*: Pointer to the left child of the node,
- *pRchild*: Pointer to the right child of the node,
- *oCipherRep*: Auxiliary attribute for rebalancing the tree.

In the following, we refer to database tables storing the state of the OPE scheme as the *OPE-table* or *state table*. For each column of the data table that is stored in an order-preserving manner, such a state table is needed to represent the OPE state for that column. In the application, the tables are named “opetree_X”, where “X” is the name of the column. Because there is only one data table in our application, the data table is not specified in the name, otherwise the data table name should also be part of the OPE-tables’ names.

In the UC2, all columns of the table are stored order-preserving. For the new application, the database structure is displayed in Figure 2.12. Table *airplanes_enc(engineid, enginesn, enginepn, fh_limit, fc_limit, tbsv, fhsn, fcsn, tslsv)* stores the encrypted data table. All attributes of this table are foreign keys referring to the order encryption in the corresponding OPE-tree. Table *metadata(table, column, isOPE)* stores metadata as mentioned above. Finally, tables *opetree_<column>(hCipher, oCipher, isRoot, pLchild, pRchild, oCipherRep)* store the OPE state of the corresponding column.

The SEEED driver from Section 2.1.3 encrypts plaintexts in onions with different encryption schemes as layers. For our supply chain application only the order-preserving encryption scheme is relevant. Hence, to integrate OOPE in the web application, a database management system is sufficient. However, a combination of OOPE with SEEED is desirable, if the application has to be integrated in a larger context that requires more than OPE. The integration is actually simple, but tedious. Therefore, we have implemented the new web application with only MySQL for simplicity. Hence, a standard MySQL Server as described in section 2.1.3 will be used. This is underlining that oblivious order-preserving encryption can be retrofitted into a standard DBMS with small effort.

System Architecture

Each involved party - DO, MRO and CSP - is implemented as an independent application following the client-server model. The DO stores her encrypted data on a MySQL database server, hosted by the CSP. Additionally, all tables needed for order-preserving encryption are stored at the CSP as well. The DO stores the private encryption and decryption key on the client side in a trusted area. The applications are deployed on different Tomcat v7 web servers and can be called via HTTP using a client browser. During the OOPE protocol the applications communicate via a socket connection. The database connection is a JDBC connection, the SQL queries have to be encrypted before being transferred to the database server. Otherwise they cannot be executed. In the architecture displayed in Figure 2.13, the database and the CSP application are hosted by the CSP.

Each application’s user interface is realized in SAPUI5, a design framework that uses JavaScript and a model-view-controller architecture. The interface is connected to a servlet and some Java

Customer View

random Input initialize

Engine ID	Engine Serial Nr.	Engine Part Nr.	Flight Hours Limit	Flight Cycles Limit	Time between Shop Visits	Flight Hrs. since new	Flight Cycles since new	Time since last Shop Vi...
0	1981285515	41676503	5000	2500	500	1446	2482	234
1	583291957	495997044	5000	2500	500	3191	1639	370
2	1351516016	2060017004	7000	3500	500	3502	2281	59
3	1440654831	491824205	7000	3500	500	4487	3115	111
4	766351737	928176859	6000	3000	500	11	12	123
5	308214088	1085265234	8000	4000	500	6483	2557	274
6	803062969	97338398	9000	4500	500	3318	4227	406
7	2035983662	1170748968	7000	3500	500	1596	3072	299
8	196186130	431411458	8000	4000	500	2384	665	372
9	1502353669	456430051	5000	2500	500	531	576	241

Add Engine

Figure 2.11: Customer's View

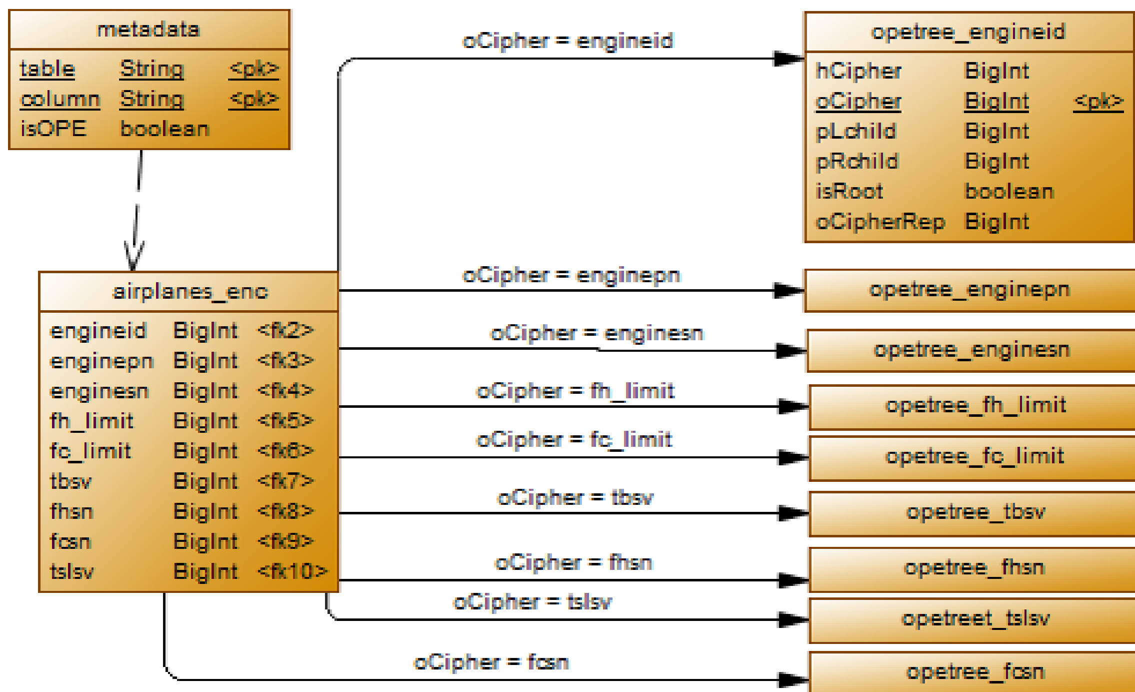


Figure 2.12: Database Structure

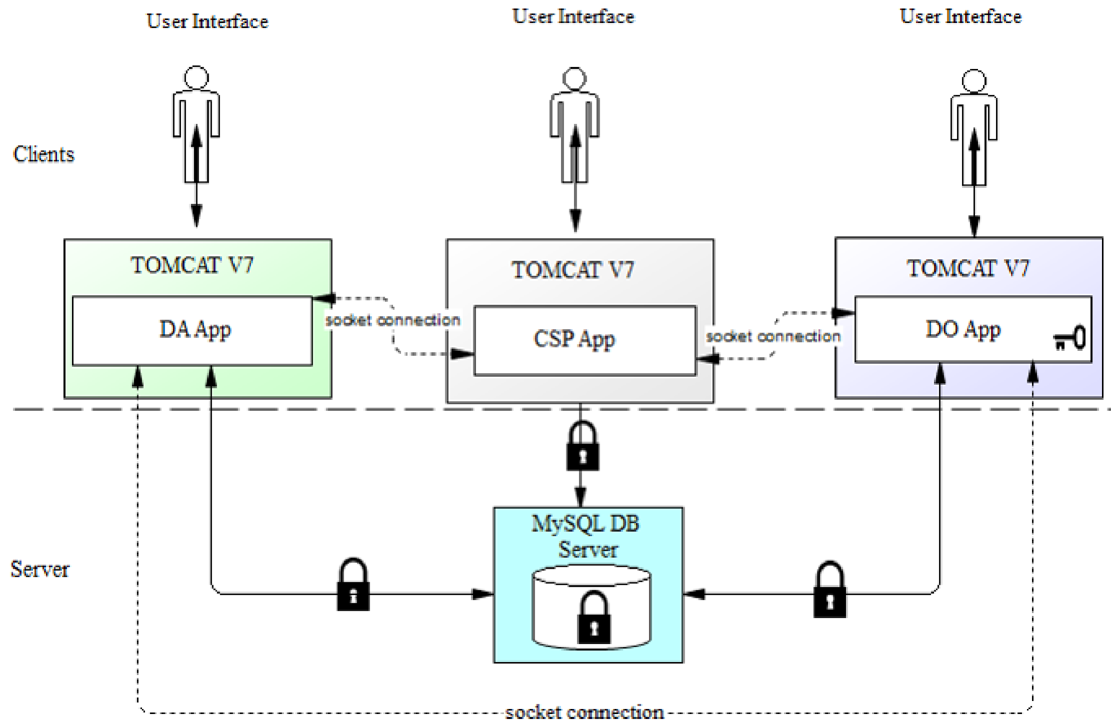


Figure 2.13: System Architecture

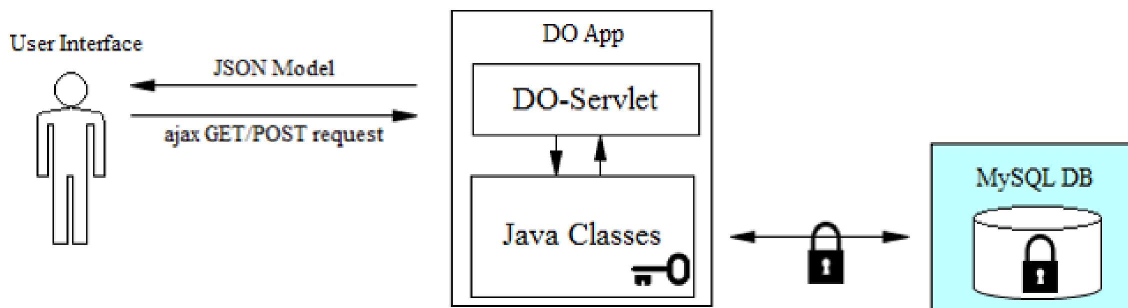


Figure 2.14: DO Application Architecture

classes implementing the programming logic, see Figure 2.14. All activities are triggered by the user via the frontend and will then be handled by the web server. The frontend calls the servlet on the server using HTTP GET and POST requests. For example, when the user in the DO application adds a new engine, the frontend sends the data via POST request to the servlet. There, the data is received, encrypted and inserted into the database. If the POST request is successful, the frontend sends a GET request to the web server to get the updated data table with the new created engine. The web server gets the relevant data from the database, decrypts it, transfers it in a JSON object, and responses to the client.

Application Structure

The UC2 prototype with OOPE consists of three independent applications, which have common components. These common components are first described before discussing the particular specifications for each application.

OOPE Classes. One of our non-functional requirement is the expandability of the applications toward homomorphic encryption and order-preserving encryption schemes. For this purpose, an abstract class for each actor is defined that implements all methods needed for any type of oblivious order-preserving encryption by that party. This abstract class is called *oblivDO_Parent* in the DO application and correspondingly *oblivDA_Parent* and *oblivCSP_Parent* in the DA and CSP application. Every inheriting class needs to implement the (oblivious) order-preserving encryption protocol and some additional methods to fulfill the functional requirements.

DO Application The DO application shows the plain data of the DO, in our use case the data of airplanes. Furthermore, a user can add new records. The displayed data is stored in a cloud database at the CSP. For security reasons, the data is encrypted before being sent to the CSP. The DO uses order-preserving encryption, thus DO can execute range queries on the encrypted data. Also, the DO application is involved in the OOPE protocol, helping MRO to encrypt data without revealing any information about this data. The abstract super class *oblivDO_Parent* and the extended class *DataOwner* implementing the encryption algorithm are displayed in Figure 2.15. The following listing gives an overview over the most important methods of each class.

- *oblivDO_Parent(homEncType)*: The constructor of the class gets the homomorphic encryption type as parameter. In the method, the private attribute *homEncType* is set and depending to this parameter an object of the encryption class is created and saved in the private attribute *homEnc*.
- *oblivOpeInit()*: This method initializes the OOPE protocol for the calling party. First, the connections to the other parties are established, then the DO is set as the Sender for the oblivious transfer and a Boolean circuit for comparison and equality check is generated.
- *homEncrypt(Object), homDecrypt(Object)*: These two methods call the corresponding method for homomorphic encryption and decryption on the attribute *homEnc* but cast the input and output value accordingly.
- *oblivOpeExec()*: This is an abstract method, which needs to be implemented by the extending class. This method implements the protocol part for the corresponding party.

The actual implementation of order-preserving encryption is realized in the class *DataOwner* which extends *oblivDO_Parent* by the following methods:

- *DataOwner(homEnc)*: The constructor. Here, the super class constructor is called to set the homomorphic encryption type.
- *addEngine(Engine)*: The method is called from the servlet if the Customer inserts a new data record into the database by adding a new airplane engine. The parameter object contains all information about the new engine as plaintext. All attribute values of the engine are order-preserving encrypted and then inserted in the database. The DO has a direct connection to the cloud database, so the CSP is not involved in the process.
- *getPlainTableAsJson()*: The method returns the current data table with all engines in plaintext as a JSON object. It is called from the frontend when the application is started and after a new engine has been added. The data is read from the database, decrypted, and put in the JSON object.

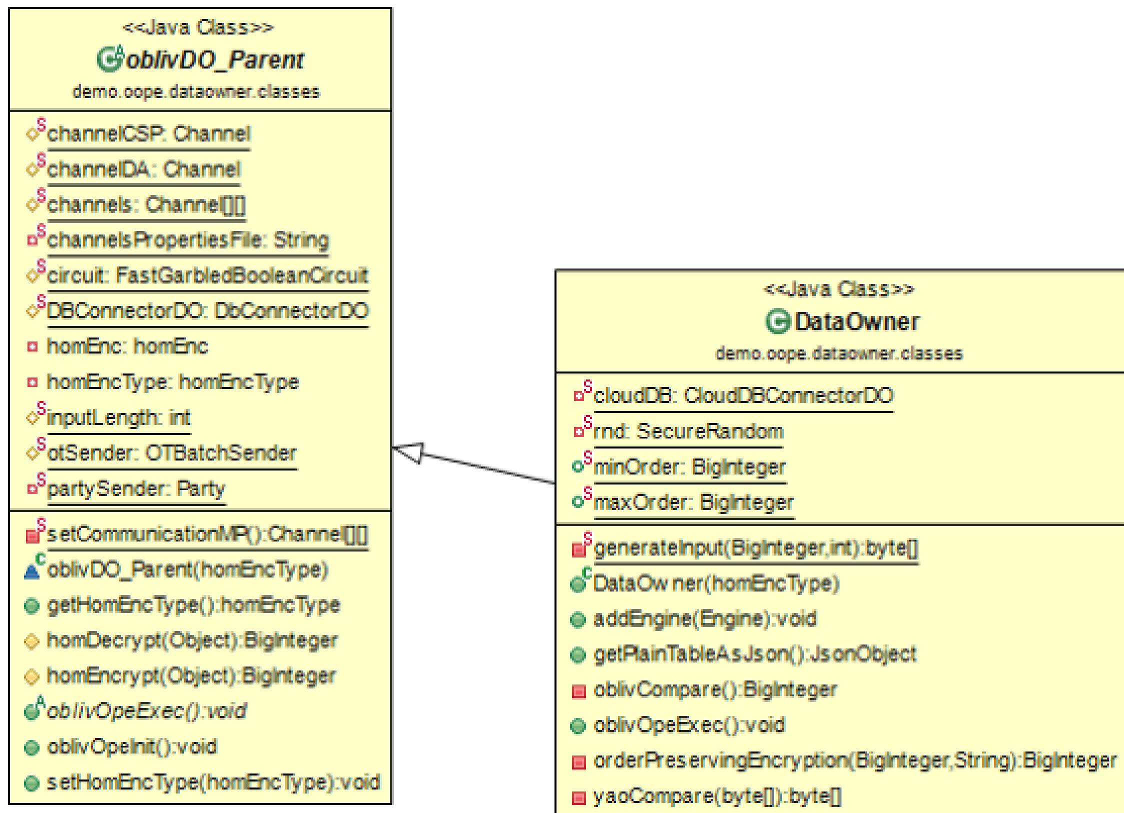


Figure 2.15: DO Application Classes

- *oblivOpeExec()*: The method implements the OOPE protocol for the DO invoking many times the private method *oblivCompare()* to execute an oblivious comparison. Each oblivious comparison requires generating some random input using *generateInput()* before invoking *yaoCompare()* to execute Yao's garbled circuit protocol.

CSP Application. The CSP application displays the encrypted data of the DO, but there is no activity that can be triggered by the user. Nevertheless, the CSP application also takes part in the OOPE protocol. Class *oblivCSP_Parent* does nearly the same as *oblivDO_Parent* in the DO application. However, the CSP does not need the method *homDecrypt()*, since homomorphic decryption requires the private key known only to the DO. The new method *traverse()* is called during the OOPE protocol. As input it gets a byte, which contains the result of the comparison, a node of an OPE-tree, and the name of the OPE-tree. Depending on the input byte, it returns the right or left node of the input node. The OOPE algorithm is implemented in subclass *CloudProvider*. All methods except the constructor and *getEncryptedTable()*, are used in the OOPE-protocol. Method *getEncryptedTable()* returns a JSON object that contains all data stored on the database, and is called by the servlet to send the data to the client.

DA Application. The DA application displays a decision tree to analyze data on its user interface. DA strives to evaluate the repair and replace probability of an airplane. By selecting one leaf, the corresponding path is converted into an SQL query, which is then encrypted during the OOPE protocol. Class *oblivDA_Parent* owns more methods than the super classes of the other applications. All the new methods handle the decision tree. To display the tree, the servlet calls

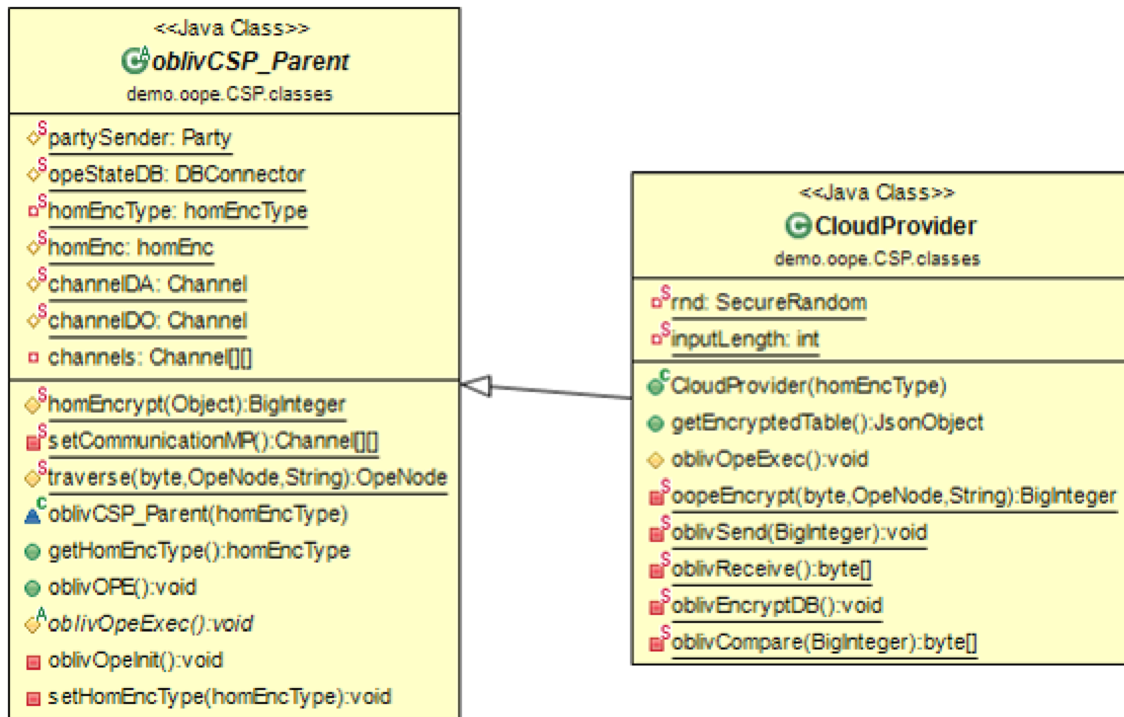


Figure 2.16: CSP Application Classes

the method *getDecisionTreeAsJson()* and gets back a formatted JSON object. How the decision tree is stored and created is explained in section 2.2.3. The next action triggered by the user is the evaluation of the tree or a leaf. In this case the methods *getDecisionTreesEvaluationAsJson()* or *getLeafAsJson()* are called. They start the OOPE protocol. First *oblivOpeInit()* establishes the connections to the CSP and DO and prepares the execution of the protocol, afterward *oblivOpeExec()* is called. The parameters of this method deliver two lists containing the plaintexts of all selection attributes that need to be order-preserving encrypted and the names of the corresponding columns. It returns an array with all order encodings in the same order as the plaintexts in the input list. The computation of the order encoding is implemented in the class *DataAnalyst*.

Homomorphic Encryption. For OOPE, homomorphic encryption is needed. In our prototype application we implemented Paillier encryption. For the future, we plan to use another homomorphic encryption that also supports addition. Therefore, the connection between the *obliv<Party>_Parent* class and the encryption class should be implemented as general as possible. Hence, an interface named *homeEnc* has been defined providing the methods *generateKey(), encrypt(), decrypt() and add()*. Class *Paillier* implements the methods, see Figure 2.18. Paillier Scheme is an asymmetric encryption, therefore there are two calling modes: Public and Private. When a *Paillier* object is created, the mode and the corresponding key file are given as parameters. All implemented homomorphic encryption types are listed in enumeration *homEncType*. The classes are the same for the DO and the CSP application, the DA does not need the homomorphic encryption.

Multiparty Computation. The communication during the OOPE protocol is realized by socket connections between DO, DA and CSP. For this purpose, all applications use the *Channel* interface and the class *Party* from the SCAPI library, see Section 2.2.2. The CSP application also uses the

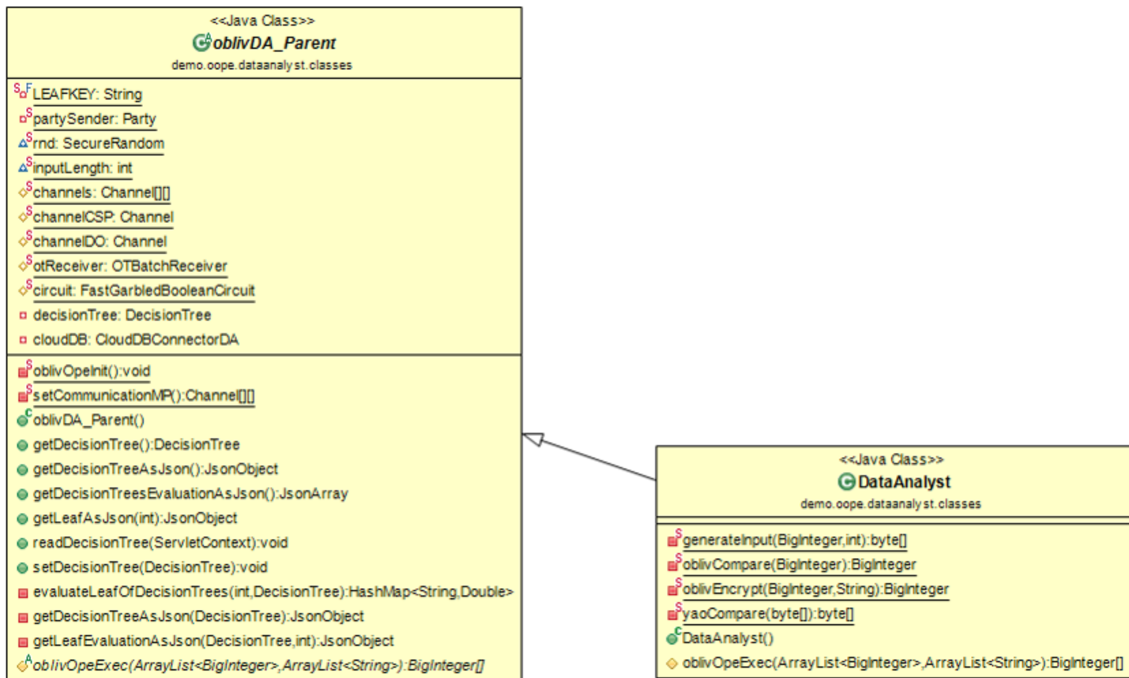


Figure 2.17: DA Application Classes

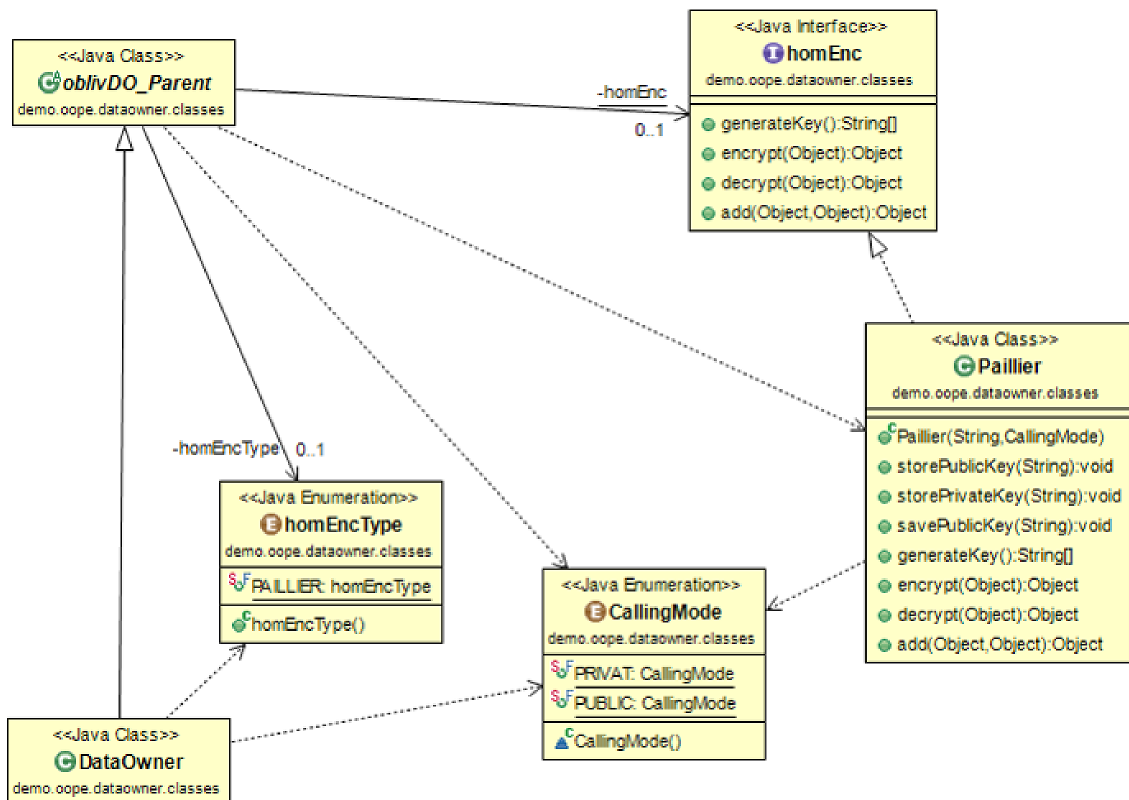


Figure 2.18: Homomorphic Encryption Classes

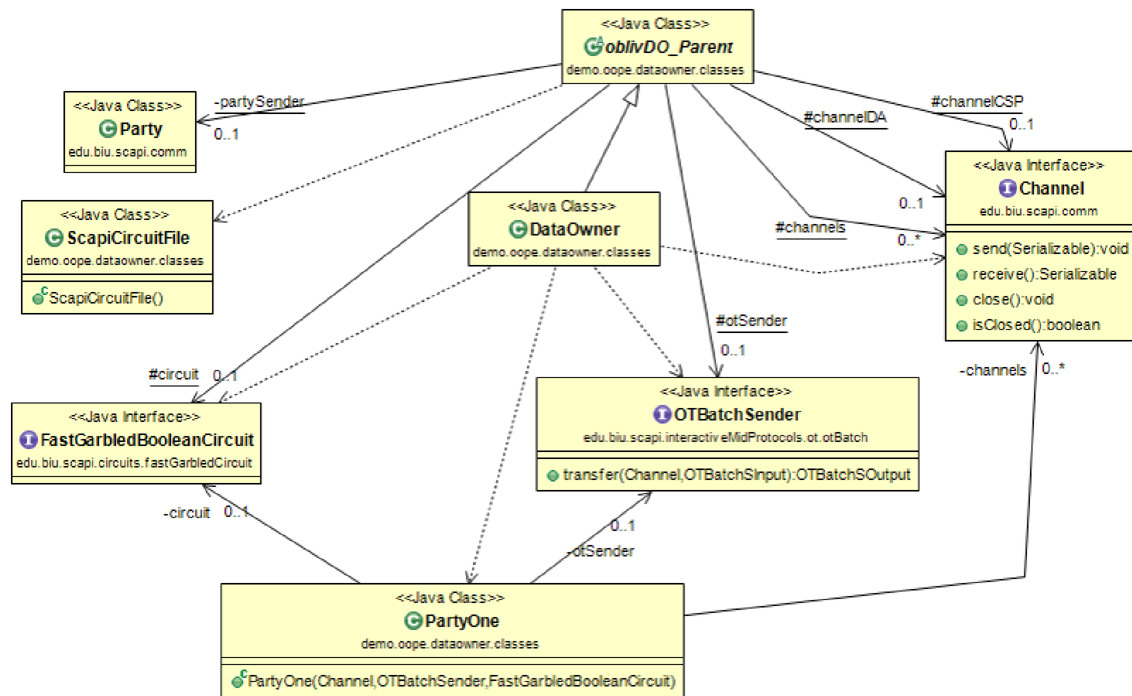


Figure 2.19: OOPE Classes for DO

class *ReceiverThread* to enable multi-threading while waiting for the results of the garbled circuit computation. The CSP application's classes are illustrated in Figure 2.20. The garbled circuit protocol runs only between the DO as generator and the DA as evaluator. Correspondingly, in the oblivious transfer protocol the DO is the sender and uses therefore an object that implements the interface *otBatchSender* to play this role. This object gets an object of type *Party* as parameter which contains connection data as IP-addresses and port numbers relevant to interact with other parties in the Multiparty Computation. This connection data is read from a configuration file that needs to be created manually before starting the application. The DO uses the class *PartyOne* to implement its generator's role of the Yao protocol. This class gets the channels to the evaluator (the DA), object OT sender and the Boolean circuit as parameters. The Boolean circuit is created in class *ScapiCircuitFile*, which is not a built-in SCAPI class and is only used to generate the comparison circuit for our application in a SCAPI-format. The generated circuit is stored in a text file that is read when creating the *FastGarbledBooleanCircuit* object. During protocol execution, the DO first invokes the method *garble* of the class *FastGarbledBooleanCircuit* to garble the Boolean circuit and both parties (the DA and the DO) execute an oblivious transfer by invoking simultaneously the method *transfer* of the class *otBatchSender* (resp. *OTBatchReceiver*). Then, the DO uses the *Channel* to the DA to send her garbled input and garbled table and translation table of the circuit, which are used by the DA to initialize its local *FastGarbledBooleanCircuit* object. Finally, the DA invokes the methods *setInputs*, *compute* and *translate* of the *FastGarbledBooleanCircuit* class to set the input, evaluate the garbled circuit, and translate the output.

The structure of the OOPE classes for the DA application is similar, see Figure 2.21. The fact that the DA is the receiver and not the sender makes the difference between the two applications concerning the garbled circuit and the oblivious transfer protocols. As a consequence, the interface *OTBatchReceiver* and the class *PartyTwo* are used instead of *OTBatchSender* and *PartyOne*.

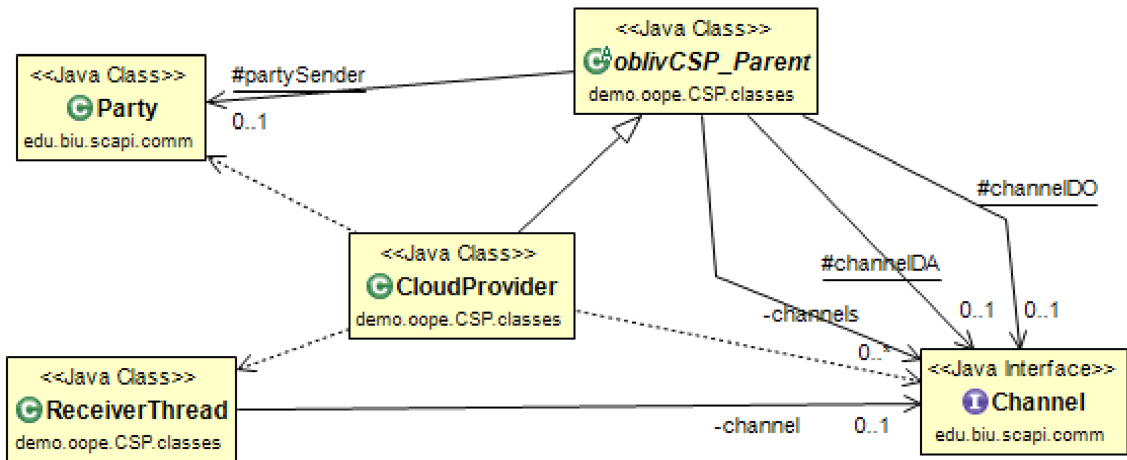


Figure 2.20: OOPE Classes for CSP

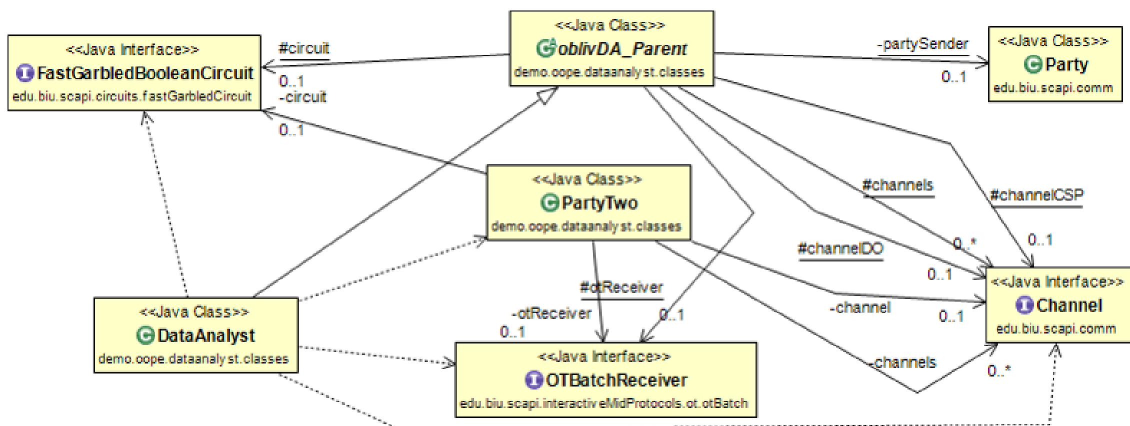


Figure 2.21: OOPE Classes for DA

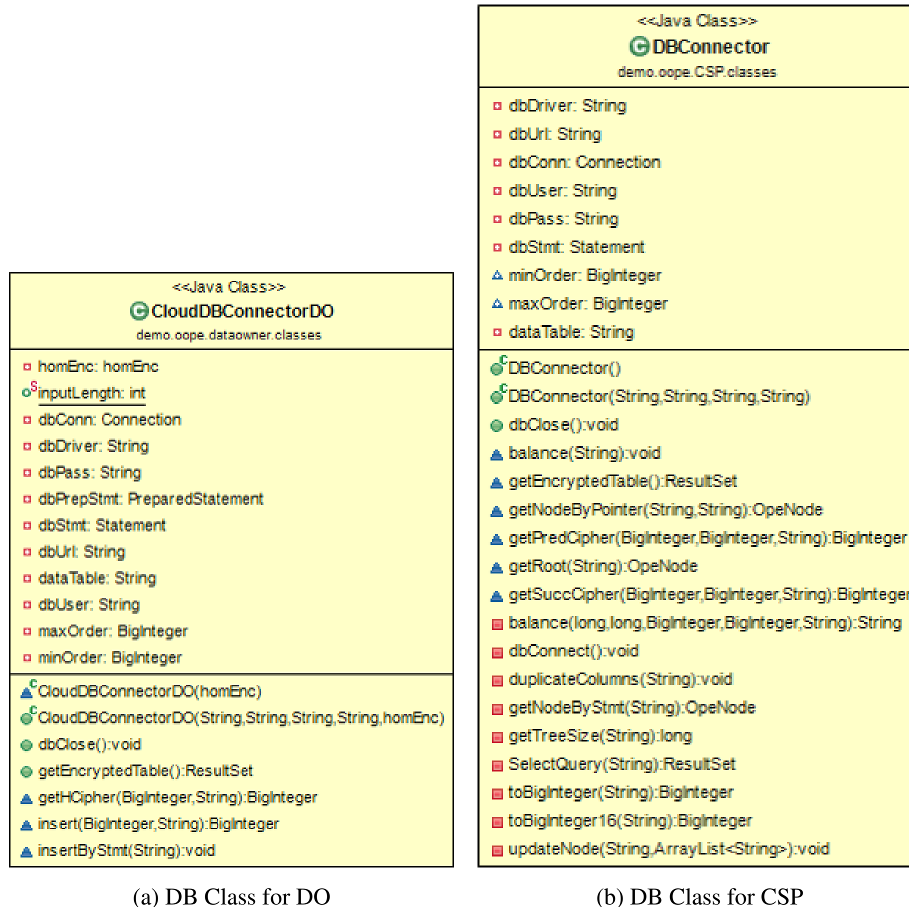
Database Connection

DO, DA and CSP applications use a connection to the cloud database, but the access privileges differ. The DO application has full access to the data. It can modify and update exiting data or add new entries. In the prototype application, the functionality is limited to adding new entries. Other functionalities shall be implemented in a later version. The CSP application has full access to the encrypted database, but cannot decrypt. It is assumed that the CSP does not manipulate the data in the cloud database. The main difference between database operations for CSP and DO is that the CSP application may perform rebalancing operations for an OPE-tree. All other database operations are reading operations. The DA application has restricted access to the database. It is only allowed to execute range queries that return an aggregated result, like count queries. In the prototype application, the only query executed by the DA application is: `SELECT COUNT(*) FROM <data table> WHERE <conditions>`. Each application owns a database connector class. In the DO and CSP applications the class contains the logic for the corresponding database operations. If another OPE scheme is chosen, the database accessor classes must be adjusted. The DA application's class has only one method *selectQuery()* to execute the SQL statement.

DB Connection for DO. All methods that access the cloud database are summarized in class *CloudDBConnectorDO*, see Figure 2.22a. When creating an instance of this class, the current homomorphic encryption object is passed as a parameter to enable encryption and decryption. The class does not only establish a connection to the database, it also contains some database operations. The method *getEncryptedTabe()* returns the *ResultSet* for the SQL-query `Select * From <datatable>` and is called by the *DataOwner* class when the frontend loads. Function *getHCipher()* returns the homomorphic encryption of the given order encryption in the given OPE-tree. Method *insert()* gets a new value and the corresponding column of the data table as parameters. It computes the order encryption for the new value and inserts it, if necessary, into the OPE-tree of the given column. It returns the order encryption for the new inserted value. However, it does not insert the value into the data table. This is done by calling method *insertByStmt()* specified in the *DataOwner* class after all attributes of an engine are order-preserving encrypted.

DB Connection for CSP. The database connection for the CSP differs from the one of the DO application because the CSP does not insert new values into the database. It reads the data or rebalances the OPE-trees. As shown in Figure 2.22b, class *DBConnector* does not contain public methods because it is only called from the *CloudProvider* class. The most complex methods are the following.

- *balance(String)*: The method rebalances the OPE-tree for the column given as parameter. It is called in the OOPE protocol whenever the interval between the minimum and maximum order encryption is smaller than two.
- *getPrepCipher() / getSuccCipher()*: The methods return the order encoding of the closest smaller / greater value in the OPE-tree compared to the first parameter. The other two parameters are the minimum / maximum order encoding and the name of the OPE-tree where to search. If there is no smaller / greater value, then the method returns the minimum / maximum order encoding value. The method is called when the right place for a new value in the OPE-tree is found and the order encryption must be computed.



(a) DB Class for DO (b) DB Class for CSP

Figure 2.22: Database Classes

Decision Tree. The DA application displays a decision tree which is important for the forecasting. For the prototype, the tree is stored in text files which are read by the method *read()* of the class *DecisionTreeReader*. Then, a *DecisionTree* object is created which itself creates *DecisionTreeNodes* and *DecisionTreeLeafs*. A node consists of three main information:

- *attributeName*: The attribute/column to be checked,
- *splitAttribute*: The threshold,
- *operation*: $<$ or $\geq$.

Each node can be converted to a condition of an SQL query. Method *getSQLStatement()* returns the SQL condition of the current selected node. Method *getCompleteSQLStatement()* calls *getSQLStatement()* for all predecessor nodes and returns a complete `SELECT` query with the table name and all conditions for the current selected node. A leaf is a node with both *attributeName* and *splitAttribute* empty. Leaf nodes extend the *DecisionTreeNode* with the attribute probability. A leaf node stores the repair probability of an engine that fulfills all criteria of the path to that leaf. The replacement probability is the converse probability of the repair probability.

Algorithmic Design

This section shows how the DO, DA and CSP applications work from the algorithm perspective.

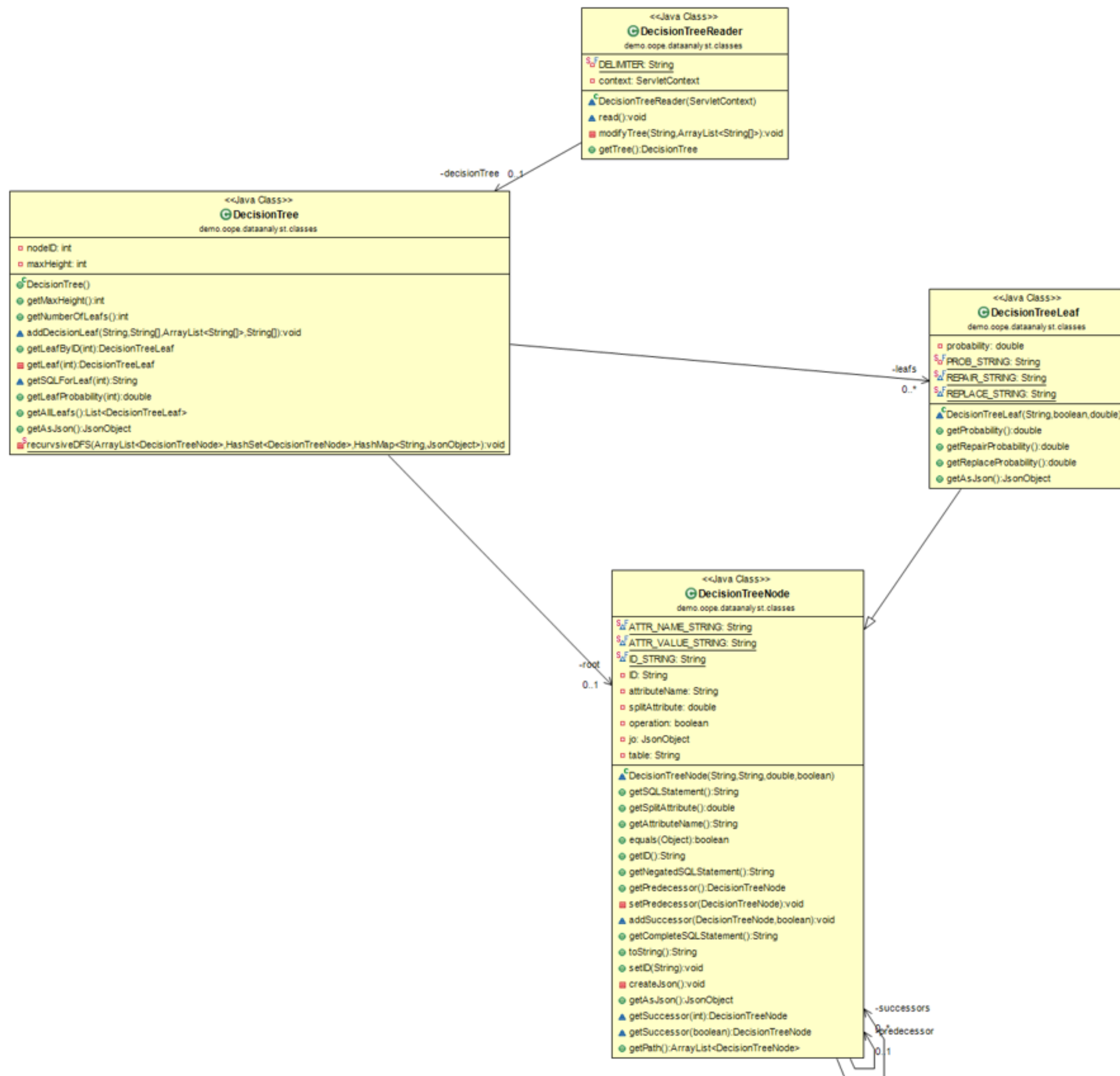


Figure 2.23: Decision Tree Classes

Prerequisites. All three applications are deployed using a Tomcat v7 web application server and can be accessed on the client browser. A MySQL database server is running and hosts a database that contains all tables defined in section 2.2.3.

Process. The OOPE is started in the DA application by clicking on a leaf of the decision tree. The frontend controller sends AJAX requests to the DO, DA and CSP servlets. A GET request to the DA application's servlet with the leaf-ID as parameter and POST requests to the servlets of the DO and CSP application with the string-parameter *"obliv"*. The DA servlet reads the incoming leaf ID and prepares the return object by setting the repair probability and SQL-statement for the path of the corresponding leaf node. Finally, it calls the method *getLeafAsJson()* on the *DataAnalyst* object, which returns estimated number of repairs and replacements in a JSON object: *"leafIdKey":4,"repair":"1.4","replace":"0.6"*. In the next step, the call is forwarded to method *evaluateLeafOfDecisionTrees()*, which takes the leaf ID and the decision tree as input. The method computes the path to the given leaf and puts all *splitAttributes* and *attributeNames* of the nodes in two lists. Then, the DA application is ready to start the OOPE protocol by calling the method

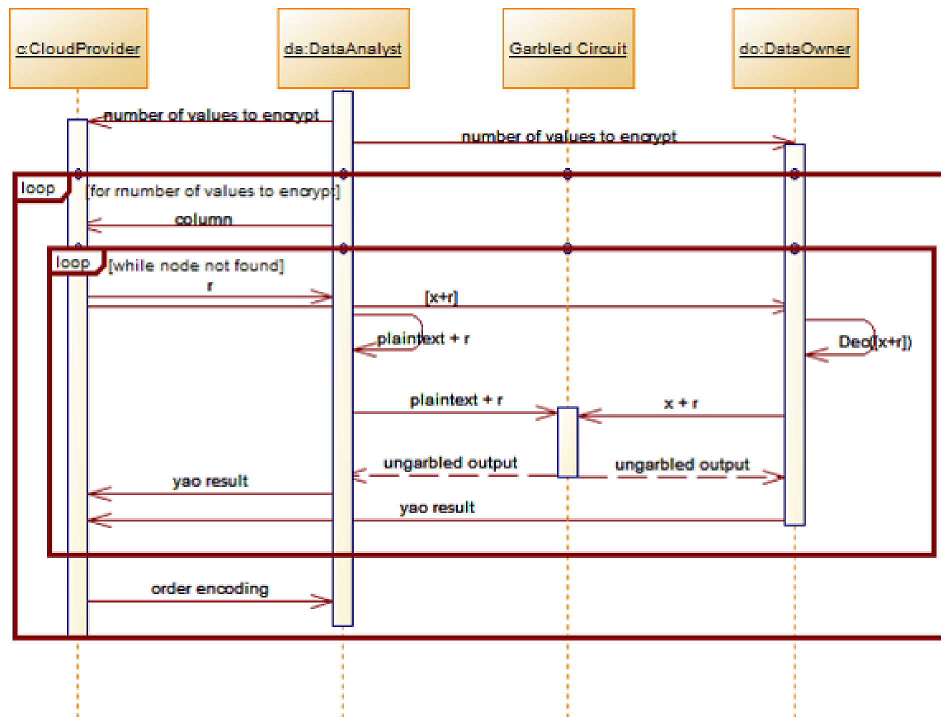


Figure 2.24: OOPE Sequence Diagram

oblivOpeInit(). The other two applications (DO and CSP) do not need to compute anything before the OOPE protocol is started. Their servlets directly call the *oblivOpeInit()* method. A socket connection between each application is established with the other applications and all initialization work for the protocol is done. In all three applications, the OOPE protocol is carried out by method *oblivOpeExec()*. The sequence diagram in Figure 2.24 shows the interaction between the three parties during the protocol. In the DA application, the method returns an array that contains the order encryption for all *splitAttributes* of the inserted list. Next, the SQL statement for the leaf is generated and all *splitAttributes* are replaced by the corresponding order encryption. Then, the encrypted SQL query is executed on the database and -for the airline scenario considered by UC2 - the amounts of repairs and engine replacements are computed. The result is returned as a JSON object. The servlet aggregates it with the SQL-query, the leaf ID and the repair probability in another JSON object and answers the GET request with this object.

2.3 Summary

This chapter outlined the final prototype for information sharing in a single cloud architecture between multiple users. Specifically, we evaluated how Use Case 2, an airline business scenario, is realized between a cloud service provider, customers (airlines) and an MRO (Maintenance, Repair and Overhaul) provider. We also discussed existing secure computation approaches as well as property-preserving encryption schemes. To achieve privacy for the MRO as well as the customers, the prototype is based on two technological foundations which are integrated into the SAP research project SEEED. First, adjustable, property-preserving encryption schemes to ensure customer privacy against the CSP. However, order-preserving encryption schemes are symmetric, which means that either the customer must reveal her private key to the Maintenance, Repair

and Overhaul provider (MRO) or the MRO must reveal the plaintext of the decision tree to the customer. Thus, a second building block was formulated: secure computation for enforcing MRO privacy while preserving customer privacy. From an architectural perspective, our prototype for search over encrypted data in a cloud architecture utilizes a Client Server Model and is integrated into a relational SQL database. Demo access is provided via a central web front-end. Within ESCUDO-CLOUD Use Case 2 has the objective to formulate a state-of-the-art Cloud solution for information sharing between business who transition to the Cloud. We see the presented prototype as an important milestone for enabling significantly more secure information sharing between multiple parties without requiring business developers to have much cryptographic knowledge.

3. Use Case 3: Federated Secure Cloud Storage

3.1 Background

3.1.1 Overview

The core responsibility of Use Case 3 is to ensure the confidentiality and integrity of the customer's data when they are outsourced to different Cloud services for storage. This security is enforced by using client-controlled encryption approaches. In addition to this core responsibility, the access and ease-of-use of the key management and access control features is also very relevant to Use Case 3 security properties. The key challenge addressed by Use Case 3 here is to offer key management and policy-based access control as a service through a Cloud service store. The instance of a key management service and the access control service have to be tightly coupled for each customer, which allows the customers to specify key release rules that are valid only under specific conditions. The data owner, or the customer of the Cloud storage service, is in control of the data encryption process and the data can be encrypted on different types of storage media on multiple CSPs.

3.1.2 Context in ESCUDO-CLOUD

The main focus of this chapter is within Work Package 1 of ESCUDO-CLOUD, in describing the solution developed in the context of Use Case 3. It describes the scope and granularity of some of the main components of the solution, especially those related to offering data protection for block, object and Big Data storage services on different Cloud platforms.

The main outcome of Use Case 3 is the design and development of the BT Data Protection as a Service (DPaaS) solution, based on results from Work Package 4, which focuses on the development of models and techniques that provide data security advantages to a user utilizing the services of multiple Cloud platforms. This encompasses the encryption of data-at-rest described in Task 4.3, with the goal to guarantee to the data owners that the confidentiality and integrity of their data is adequately protected, especially ensuring interoperability with the storage services offered by different IaaS providers.

3.1.3 Background on DPaaS

The core components of the DPaaS solution are the agents, tools and utilities providing the core data encryption and decryption mechanisms, the access control service and the key management service. The customers are able to use these services via a service store that integrates the data protection solution and its components with multiple independent CSPs. Therefore, the service store's orchestration capability is the central component of the data protection service. The purpose and application of these components in Use Case 3 are summarized below.

Key Management Service

The main purpose of the BT Key Management Service is to allow customers to generate, store and manage their encryption keys and certificates securely. It is a centralized, high-availability and standards-based key management solution. It is provisioned and managed via the BT Cloud service store's Service Orchestrator, which enables each customer to create isolated and compartmentalized key management domains, so that the customers can store and manage their keys used in multiple Cloud platforms in a secure multi-tenant environment. More details of its features and capabilities are available in [TZS17].

One of the core requirements of Use Case 3, and also in the broader context of ESCUDO-CLOUD, is to empower end-users with maximum control of their data in the Cloud eco-system. This is realized in Use Case 3 by the customer-based management of the cryptographic keys, so that only the customers are able to access the their key store and only the customers are able to authorize the release of their keys to trusted encryption agents. The BT service store and the key management service administrators have no view or control of the customers' keys and other security credentials, even if the key management service is deployed on BT Cloud platform.

Access Control Service

The main purpose of the BT Access Control Service is to allow the customers to regulate access to their encrypted data stored on multiple Cloud platforms through a policy based enforcement of rich access control attributes. Thus, its core value is to give the customers the assurance that their data remains protected from the untrusted or curious Cloud service providers. More details of its features and capabilities are available in [TZS17].

In the context of Use Case 3, the Access Control Service enables the creation of a strong separation of duties between privileged Cloud service administrators and data owners outsourcing their data on these Cloud platforms. When the data owners or authorized users want to access their data, the Access Control Service lets them achieve this seamlessly by releasing the correct keys to the trusted encryption agents or gateways.

Data Encryption Agents

The main purpose of the BT Data Encryption Agents is to offer capabilities for data-at-rest encryption of sensitive data stored on different types of Cloud storage services, as well as enforce privileged user access control on that data. The agents can encrypt and protect data residing in physical, virtualized, object, and big data storage environments. More details of their features and capabilities are available in [TZS17].

One of the core requirements of Use Case 3 is that the customers should be able to cache their keys on trusted virtual machines or gateways in order to outsource or improve performance of the encryption and decryption process. The BT Data Encryption Agents are the components that are considered trustworthy in the scope of Use Case 3 as they are provisioned and deployed by the BT Service Orchestrator in a secure process on customer's Cloud hosted virtual machines.

Service Orchestrator

The main purpose of the BT service store's Service Orchestrator is to provision the data protection service to the customers and manage its life-cycle. Each customer gets a compartmentalized access to the data protection service, and is able to define the access control and key release policies via

the service store or the key management service interface and associate keys with those policies. The service store also has the ability to install and configure the data encryption agents on virtual machines and gateways on different supported Cloud platforms.

3.2 Solution

The DPaaS solution allows BT customers to protect and control their confidential and sensitive information, with a user-friendly interface. Customers are able to store their data on multiple Cloud vendors and platforms, and are able to manage the security related aspects of their stored data via the federated protection service. Only the customers (or a trusted third party designated by the customers) have the access and control of the cryptographic keys, giving the customers the freedom to decrypt data on-demand and in real-time.

The main challenge being addressed by this solution is the federated security and management of data that is hosted on different types of third party storage services, for example in the form of block/file-system storage, data backups, or databases. This problem is further complicated in the Cloud computing environment as data can be replicated and moved automatically to cater for the scalability and reliability needs of the CSPs' customers, thus increasing the risk of a security compromise. In addition to the data security concerns, most customers also have to abide by their company's data protection policies and governmental regulatory compliance (e.g., FIPS [NIS01], HIPAA [HIP03], HITECH [Blu10], Sarbanes-Oxley [CDL08], PCI DSS [MR08] and GDPR [Man13] etc.).

3.2.1 Functionality and goals

The DPaaS solution is designed around the core functional requirements of a multi-tenant customer scenario that aims to protect different types of data assets on multiple Cloud storage services. The main component that addresses the administration and management of multiple tenants/customers, multiple Cloud platforms and multiple Cloud storage services is the Service Orchestrator component described in the previous section. This is the central managerial component of the BT Service Store that is used to provision all the DPaaS capabilities to the BT customers, as shown in Figure 3.1. Each customer gets a compartmentalized access to the service store and the data protection service, as discussed in the previous section, and is able to define the access control and key release policies via the service store or the Key Management Service (KMS) interface. The BT Service Store also has the ability to install and configure the data encryption agents, plug-ins and gateways on virtual machines on different supported Cloud platforms.

The main benefit of this design is that a centrally managed data protection service is used to federate the disparity of different Cloud storage services on multiple Cloud platforms. In ESCUDO-CLOUD Use Case 3, as depicted in Figure 3.1, we focus on three different storage mediums, which are commonly requested by BT customers, to store and process data on current Cloud platforms. We manage this diversity of storage technologies as three different branches or three sub-use cases, which are block, object and Big Data storage. The core technical use case is to offer data protection as a service in a multi-Cloud environment to BT customers, whereas the service store provides the interface for the customers of the DPaaS to access and manage these storage services. As a result of this structure, the sub-use cases will inherit a common encryption, key management and access control system, but will have different abstractions and interfaces to cater for the provisioning, management and operation of underlying storage mediums. The archi-

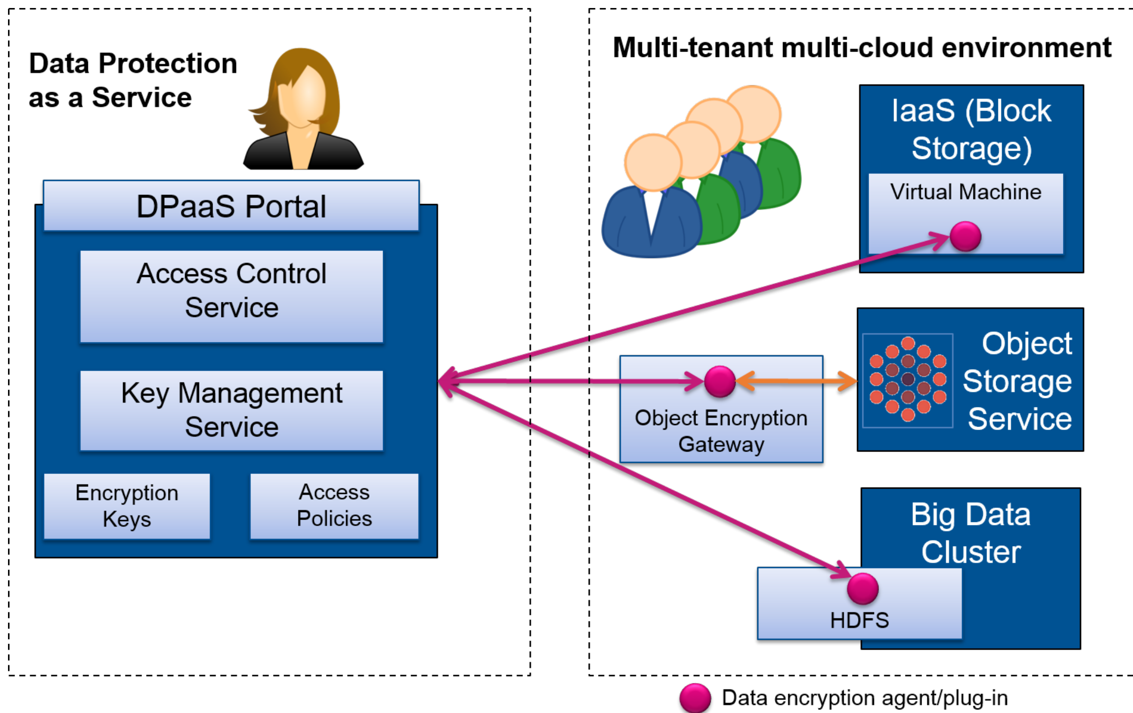


Figure 3.1: High level view of the DPaaS solution design

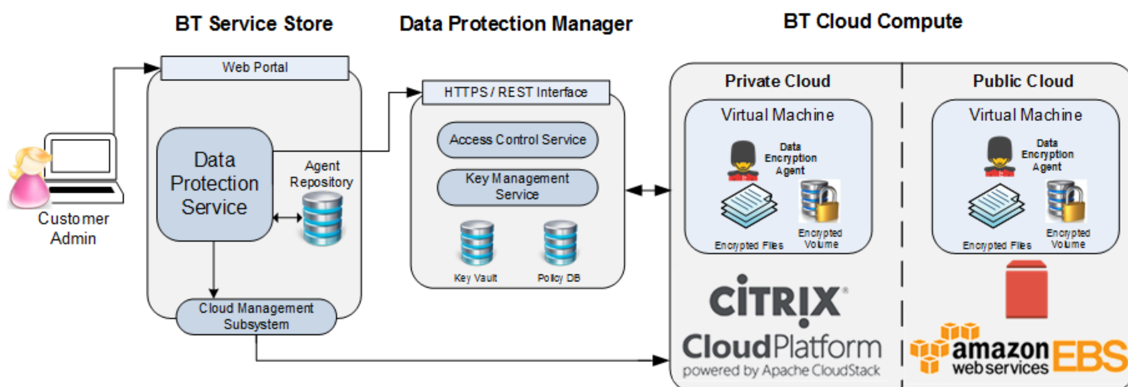


Figure 3.2: Architecture of the block storage encryption service component of the DPaaS solution

tectural and implementation details of these three sub-use cases will be described in the following sections.

3.2.2 Block Storage Encryption

Architecture

The architecture for the block storage encryption component of the DPaaS comprises of three main modules, and is shown in Figure 3.2.

The first module is the BT Service Store which is used to provision and manage the life-cycle of the component's service to the customers or tenants. Each tenant gets a compartmentalized view of the service store and the data protection service, as discussed in the previous section (3.1). The BT Service Store also has the ability to install and configure the data encryption agents on

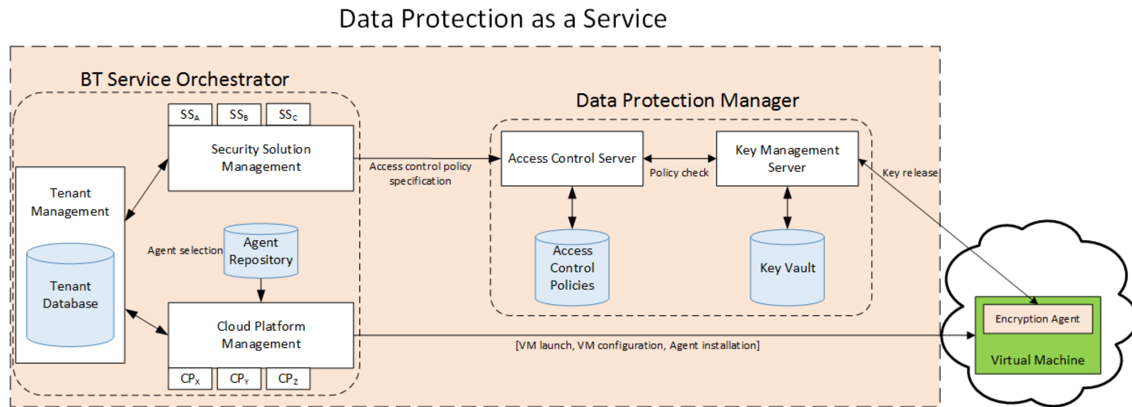


Figure 3.3: Implementation reference of the block storage encryption service prototype

virtual machines on different supported Cloud platforms. These agents are stored in a software repository on the BT Service Store.

The second module is the Data Protection Manager, which enables the customers to define the access control and key release policies via the BT Service Store or its own Web/REST interface. The Data Protection Manager also contains secure vaults where the customers can import, export and manage their encryption keys and access control policies. The Data Protection Manager is able to communicate with the data encryption agents running on the virtual machines on different Cloud platforms over secure communication channels like TLS.

The third, and the last module, is the data encryption agent that is provisioned on the target virtual machine by the BT Service Store's Service Orchestrator (SO). After successful provisioning, the agents on the virtual machines use PKI-based authentication to identify themselves to the Key Management Service which is being managed by the Data Protection Manager module. If the agent successfully passes the authentication phase, the Key Management Service issues the keys necessary to encrypt the files and data volumes present on the block storage attached with the virtual machine. After the completion of the encryption process, the access to the protected files and volumes is enforced by the agent as well. Upon receiving a data access request, the agent checks the Access Control Service for the policy associated with the protected file or volume. The agent then submits an encryption key release request to the Key Management Service if the access request is approved by the Access Control Service. Once the Key Management Service issues the encryption key, the agent can use the key to decrypt the data requested.

Implementation

The reference implementation design of the block storage encryption component of DPaaS is shown in Figure 3.3. It consists of four main components: BT Service Orchestrator, Access Control Service, Key Management Service and Data Encryption Agents. The BT Service Orchestrator itself is composed of the following sub-components.

Tenant Management (TM): The TM sub-component is in charge of managing customers registering to use the data protection service. In addition, it maintains information of the users' Cloud platforms and security solutions that can be embedded in the data protection service.

Cloud Platform Management (CPM): To support a multi-cloud environment, this sub-component provides an interface consisting of APIs for communications with the supported cloud platforms. For each cloud platform the framework supports, a cloud management plug-in is implemented and

attached to this sub-component. Among APIs defined in the interface, some of them are mandatory while others are optional for implementation. For example, it is required to implement the APIs that connect to the Cloud platform for virtual machine deployment and virtual machine termination because these operations are involved in the encryption agent installation and un-installation actions.

Security Solution Management (SSM): Similar to the CPM sub-component, to support different security solutions, the SSM sub-component defines an interface with APIs for communications with the security solution servers and for each security solution the framework supports, a security plug-in is needed. Basic APIs in this interface include access control policy definition and data encryption/decryption requests.

As shown in Figure 3.3, while the TM sub-component interacts with both the CPM and the SSM sub-components, the CPM and SSM sub-components are independent of each other. Besides, even though the TM sub-component requires interaction with the CPM and SSM sub-components, this interaction is loosely coupled. Basically, based on the registered information of the users and depending on the specific requests of users, the TM sub-component will directly trigger the corresponding plug-ins inside the CPM and SSM components. For example, if a user chooses to protect data in a virtual machine deployed in the Amazon EC2 platform and she chooses to employ a security solution from Trend Micro, the Amazon EC2 plug-in and Trend Micro plug-in will be called by the TM sub-component. Next time, if the user chooses to protect data in a virtual machine deployed in the CloudStack platform and he chooses to employ a security solution from SafeNet, the CloudStack plug-in and SafeNet plug-in will be called by the TM sub-component. This design provides the scalability for the framework to support several Cloud platforms and security solutions.

The implementation is supported by four databases: the tenant database, the access control policies database, the key store/vault, and the agent repository. Among these four databases, only the first one, which is the tenant database, is located inside the BT Service Store framework. The other three databases are situated outside the Service Store framework and have separate interfaces for policy management and key management. In this way, fine-grained access control policies can be defined and users can customize the policies via the external interfaces without going through the Service Store framework. However, by having these three databases outside the Service Store framework, we need to also define interfaces to connect them with the SSM sub-component. At the moment, the SSM employs a simple REST interface to set-up default basic access control policies for users and leave users the freedom to add-in or modify the policies later through the external interfaces.

Customer Journey

In this section, a typical customer's experience of interacting with the block encryption component of DPaaS is showcased.

Set-up Phase

As a pre-requisite of the typical customer journey scenario, the BT Service Orchestrator needs to be up and running. This is a one-time process and does not need to be repeated. The main steps for setting up the BT Service Orchestrator, as performed by the BT Service Store administrator, are:

1. Provision a VM (can be either Windows or Linux)
2. Install Puppet server and Apache Tomcat on the VM
3. Configure the Puppet and Apache Tomcat servers
4. Deploy the Puppet configuration scripts on the Puppet server
5. Deploy the WAR file on the Apache Tomcat server

At this point, the BT Service Orchestrator is ready to deal with customer registrations. The WAR file is deployed on the BT Service Orchestrator in `/etc/puppet/java/btappcara` folder. A configuration file (`dpm.configuration.properties`) is used for submitting parameters to the jar library. The file is a Java .properties file containing generic information on how to access the Data Protection Manager with the correct information. It has to be configured correctly only once during the deployment of the BT Service Orchestrator. Its template is as follows:

```
# Access to the Data Protection Manager
dpm.address=''<IP address of the Data Protection Manager>''
dpm.DNS=''<DNS name of the Data Protection Manager>''
dpm.port=''<Port number of the Data Protection Manager>''

# Access to the Web Service which provides support functionalities
# to the Puppet server
dpm.tomcat.address=''<IP address of the tomcat server>''
dpm.tomcat.port=''<Port of the tomcat server>''

# Configuration of the System Administrator, the account used to
# create and delete domains and users on DPM
dpm.login.name=''<Login name of the System Administrator>''

# Configuartion for the default symmetric key created for the
# encryption
dpm.symmetricKey.algo=''AES-128 || AES-256 || 3DES''
dpm.symmetricKey.name=''<Name of the key (can be empty)>''
dpm.symmetricKey.type=''[StoredOnServer||CachedOnHost]''

# Policy used for the re-keying phase; this policy is applied when a new
# Guard Point is defined, the enforcing of this policy allows for the
# encryption of any content already existing in the folder before the
# creation of the GP itself.
os.rekeying.policy.file=''<Path to the rekeying policy XML>''
os.rekeying.policy.name=''<Name of the rekeying policy>''

# Policies used for the Encryption of the GP, this policy is
# automatically replaced in the Guard Point during the Guard Point
# provisioning. At the moment there are 2 different policies for
# Windows and Linux.
windows.encryption.policy.name=''<Path to the encryption policy XML>''
```

```
# linux.encryption.policy.name=''<>''
windows.encryption.policy.file=''<Name of the encryption policy>''
# linux.encryption.policy.file=''<>''

# Policies used for the Encryption of the GP, this policy is replaced
# in the Guard Point during the Volume/Guard Point de-provisioning.
# At the moment there are 2 different policies for Windows and Linux.
windows.decryption.policy.file=''<Path to the decryption policy XML>''
# linux.decryption.policy.file=''<>''
windows.decryption.policy.name=''<Name of the decryption policy>''
# linux.decryption.policy.name=''<>''
```

Operation Phase

A typical customer's lifecycle, in the context of this solution and use case scenario, consists of the following steps:

1. Account registration
2. VM provisioning
3. Guard point[1] creation
4. Guard point de-provisioning
5. VM de-provisioning
6. Account removal

These six steps are explained in more detail below.

1. Account registration: The main actions undertaken by the BT Service Orchestrator in this step are below.
   (a) Create a new domain on the Data Protection Manager for the customer.
   (b) Create a new admin account for the customer on the new domain.
   (c) Create three default policies for the domain:
      - *Encryption re-key policy*: To encrypt existing data in guard points.
      - *Default Guard point policy*: To protect data in guard points.
      - *Decryption re-key policy*: To decrypt data in guard points.
2. VM provisioning: The main actions undertaken by the BT Service Orchestrator in this step are below.
   (a) Provision a VM (with pre-installed Puppet agent) on a Cloud platform, and attach one or more block storage devices to it.
   (b) Create a configuration file for the VM on the Puppet server.

[1]BT refers to a file, folder or block device (e.g., /dev/sda), which is to be protected by encryption and access control, as a Guard point

(c) Register the VM with the Puppet server and the Data Protection Manager.

(d) After the VM is registered with Puppet, the puppet agent will perform the following actions at the next heartbeat:

- Install Python.
- Download and install appropriate data encryption agent.
- Restart the VM (only in case of Windows).
- Register the data encryption agent with the Data Protection Manager.

3. Guard point creation: The main actions undertaken by the BT Service Orchestrator in this step are below.

   (a) Create a guard point with *encryption re-key* policy.

   (b) Update the configuration file for the VM at the Puppet server.

   (c) At the next heartbeat, the Puppet agent will download the updated configuration file to perform the following actions:

   - Execute the *dataXform* operation to re-encrypt any existing data in the guard point.
   - Call an Access Control Service API that applies the default access control policy associated with this customer.

4. Guard point de-provisioning: The main actions undertaken by the BT Service Orchestrator in this step are below.

   (a) Remove the existing access control policy from the guard point.

   (b) Create a new guard point with the *Decryption re-key* policy.

   (c) Update the configuration file for the VM on the Puppet server.

   (d) At the next heartbeat, the Puppet agent will download the updated configuration file to perform the following actions:

   - Execute the *dataXform* operation to decrypt any existing data in the guard point.
   - Remove the guard point created in Step 2.

5. VM de-provisioning: The main actions undertaken by the BT Service Orchestrator in this step are below.

   (a) Un-register the VM from the customer's Data Protection Manager domain.

   (b) Update the configuration file for the VM on the Puppet server.

   (c) At the next heartbeat, the Puppet agent will download the updated configuration file to perform the following actions:

   - Un-install the data encryption agent from the VM.
   - Restart the VM (only in case of Windows).

6. Account removal: The main actions undertaken by the BT Service Orchestrator in this step are below.

   (a) Remove all guard points and un-register all VMs from the customer's Data Protection Manager domain.

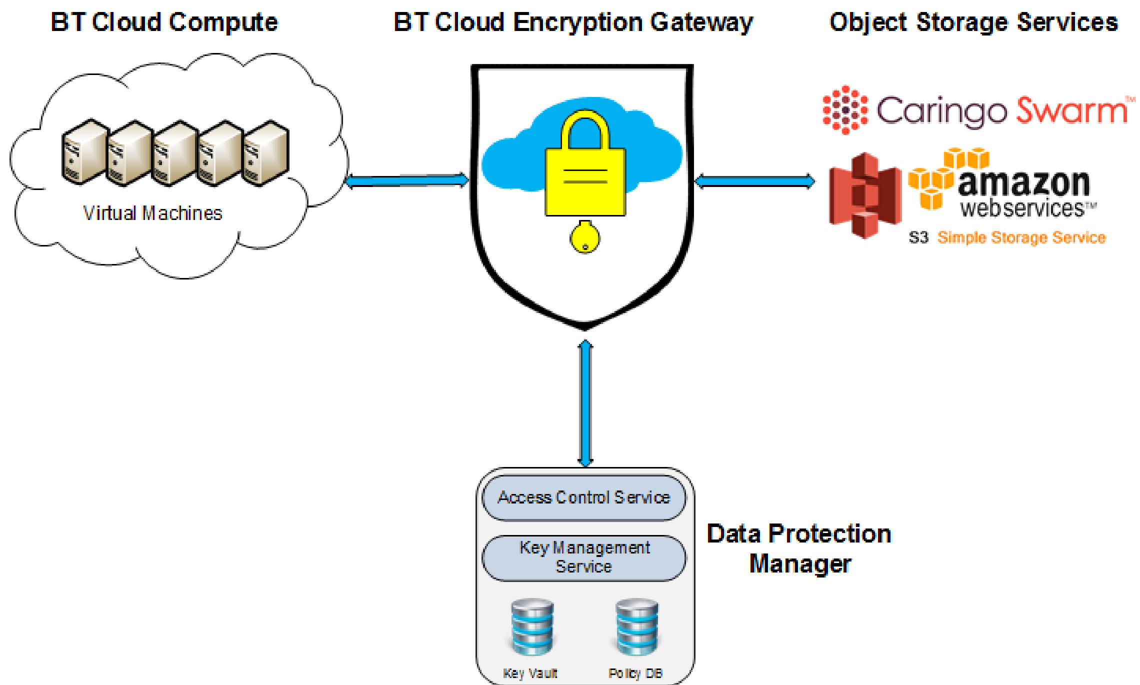


Figure 3.4: Architecture of the object storage encryption service component of the DPaaS solution

(b) Remove all access control policies and keys from the domain.

(c) Remove all user accounts of the customer from the domain.

(d) Remove the domain from the Data Protection Manager.

3.2.3 Object Storage Encryption

Architecture

The architecture for the object storage encryption component of the DPaaS comprises of one main module, i.e., the BT Cloud Encryption Gateway, and is shown in Figure 3.4.

The gateway solution relies on the Data Protection Manager, described in the previous section, for encryption key and policy management. As a result, customers never need to relinquish control of cryptographic keys to the service provider and data never leaves the virtual machines unencrypted or unaccounted.

The main role of the BT Cloud Encryption Gateway is to act as a proxy that can be used to intercept objects being sent to the object storage services and transparently encrypt them during transfer. The proxy can be deployed as a gateway in either the customers' premises as a forward proxy or in the Cloud environment as a reverse proxy. In addition to the proxy, the Cloud Encryption Gateway also implements a basic key-value database to store and track the state of the objects being encrypted in the gateway, connectors for the supported Cloud object storage services (S3, Caringo etc.), and the data encryption agent performing the object encryption and decryption operations.

Implementation

The reference implementation design of the object storage encryption service of DPaaS is shown in Figure 3.5. It uses most of the same components as the implementation described in the previous

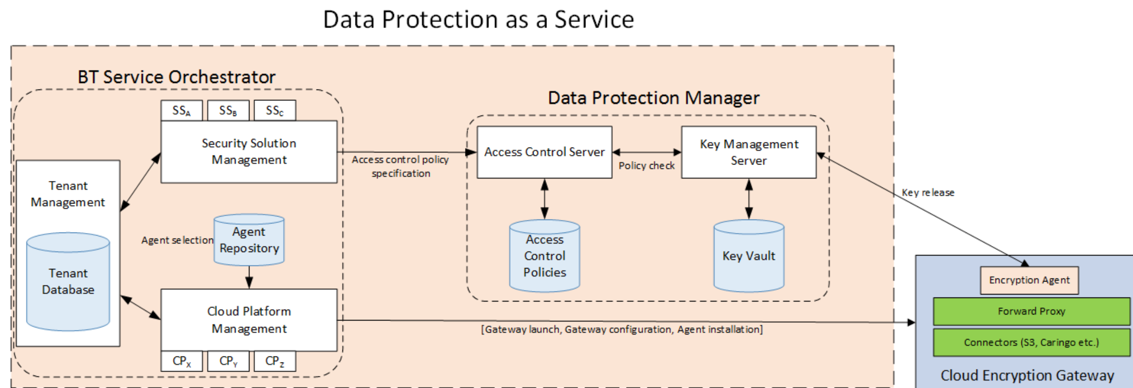


Figure 3.5: Implementation reference of the object storage encryption service prototype

section (3.1). The only main addition in this case is the BT Cloud Encryption Gateway. The Cloud Encryption Gateway can be deployed in either the customers' premises as a forward proxy or in the cloud environment as a reverse proxy, and implements the following main functions:

The main feature of the Cloud Encryption Gateway is to act as the proxy server for all the object storage requests and responses coming to and from the S3 and Caringo object storage services. This proxy server intercepts these objects being sent to the object storage services and transparently encrypts them during the caching phase. In addition to the basic proxy functionality, the Cloud Encryption Gateway also implements a basic key-value store using MongoDB to keep track of the state of the objects being encrypted in the gateway.

Similar to the CPM sub-component described in the block storage encryption implementation, in the object storage encryption solution makes use of different object storage connectors that provide the interfaces to interact with different Cloud object storage services. So far, it makes use of Amazon S3 and Caringo Swarm connectors. The customers provide the appropriate object services' credentials to these connectors after the Cloud Encryption Gateway has been set-up and configured by the Service Store. The proxy server uses these connectors to upload and download objects from the respective object store.

3.2.4 Big Data Encryption

Architecture

The architecture for the HDFS encryption component of the DPaaS comprises of three main modules, and is shown in Figure 3.6.

The first module is the BT Service Store that is used to provision and manage the life-cycle of the HDFS cluster to the customers or tenants, usually through a Hadoop cluster management and monitoring service like Apache Ambari [AMB17]. Each tenant gets a compartmentalized view of the Service Store and the Data Protection service. The BT Service Store also has the ability to install and configure the Data Encryption Agents on HDFS NameNode and DataNodes, on different supported Cloud platforms. These agents are stored in a software repository on the BT Service Store.

The second module is the Data Protection Manager, which enables the customers to define the access control and key release policies via the BT Service Store or its own Web/REST interface. The Data Protection Manager also contains secure vaults where the customers can import, export and manage their encryption keys and access control policies. The Data Protection Manager is

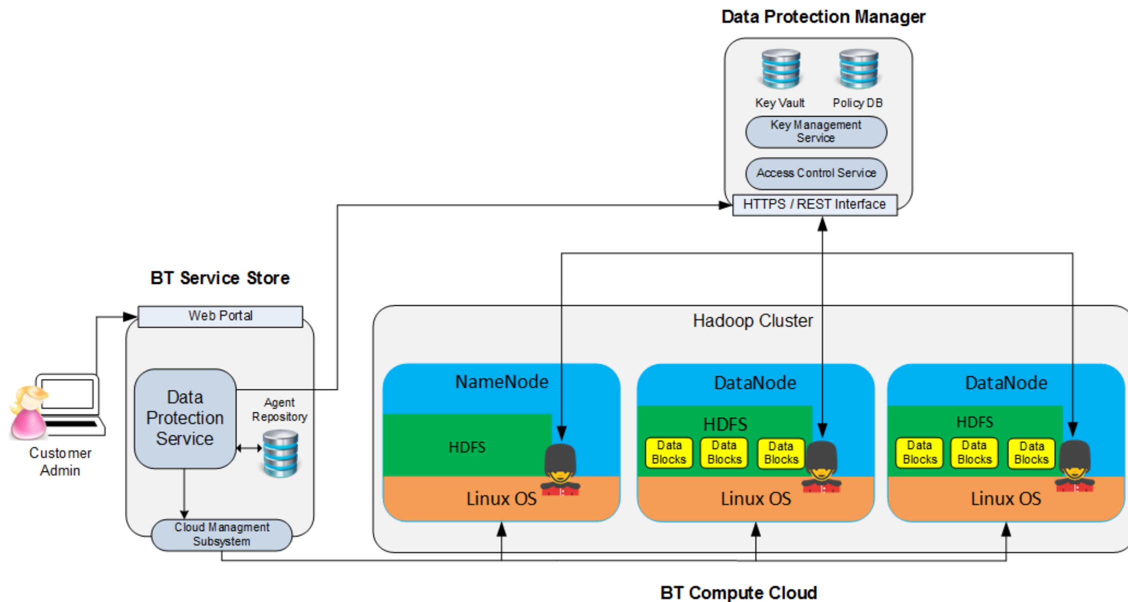


Figure 3.6: Architecture of the HDFS encryption component of the DPaaS solution

able to communicate with the Data Encryption Agents running on the NameNode and DataNodes of the customer's HDFS cluster, over secure communication channels like SSL/TLS.

The third and last module are the Data Encryption agents that are provisioned on the target virtual machines, which are the hosts of the customer's HDFS cluster, by the BT Service Store's SO. After successful provisioning, the agents on the HDFS NameNode and DataNodes use PKI-based authentication mechanism to identify and authenticate themselves with the KMS, which is being managed by the Data Protection Manager module. If the agents successfully pass the authentication phase, the KMS issues them the DEKs necessary for encrypting the data blocks stored on the HDFS storage. After the completion of the encryption process, the access to the protected data blocks is enforced by these agents as well. Upon receiving a HDFS data access request, the agents check the Access Control Service for the policy associated with the protected HDFS Guard point. If the access request is approved by the Access Control Service, the data encryption agents submit an encryption key release request to the KMS. Once the KMS issues the DEK, the agents can use it to decrypt the HDFS data blocks requested by the end-user.

Implementation

The reference implementation design of the Big Data encryption service of DPaaS is shown in Figure 3.7. It uses most of the same components as the implementation described in the previous section. As in the block storage encryption implementation, the CPM sub-component is responsible for launching and configuring the base VMs, which will constitute the cluster nodes of the HDFS cluster, on the selected Cloud platform. It also provisions and configures the Data Encryption Agents on all the cluster nodes, be they either NameNodes or DataNodes.

The main difference between this reference implementation and the block storage encryption implementation is the use of the Apache Ambari service to deploy and configure relevant Hadoop services on the cluster, as chosen by the customer organization. The Ambari service is also used to perform part of the encryption agents' configuration so that they are able to understand the HDFS virtualisation and perform encryption/decryption and policy enforcement operations at the HDFS

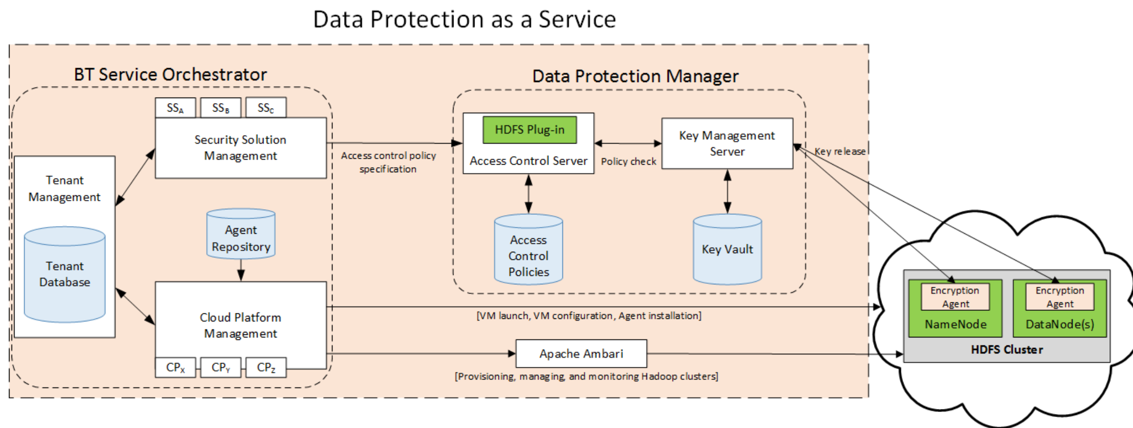


Figure 3.7: Implementation reference of the Big Data encryption service prototype

level.

Similar to the CPM sub-component described in the object storage encryption implementation, in the Big Data encryption solution we make use of the HDFS Plug-in that provides the capability and interface to define encryption policies for the HDFS files and directories stored in HDFS cluster's DataNodes. Due to this component, the customers can even selectively encrypt HDFS folders with different keys, providing multi-tenancy support. Additionally, the customers can also define user-based access control rules for HDFS files in their HDFS cluster.

3.3 Summary

CSPs offer storage services of different types to their customers. However, they maybe also offer heterogeneous and non-standard APIs, specialized functions, and security functionalities for the consumption of their users. Therefore, ESCUDO-CLOUD Use Case 3 offers a solution that provides data protection services to its customers for Cloud-based block, object and Big Data storage services, and which can work seamlessly across multiple Cloud platforms. Customers retain control of the key management, thus have more flexibility in meeting regulatory and privacy requirements and ensuring data confidentiality and secure access. Thus, BT Data Protection as a Service provides full automation of data protection services to its customers as a complete life-cycle, from data encryption to data decryption.

4. Use Case 4: Elastic Cloud Service Provider

4.1 Background

4.1.1 Overview

Use Case 4, as a subset of the overall ESCUDO-CLOUD use case strategy, is tasked with identifying and developing mechanisms for the protection of user data where the host i.e., the Cloud Service Provider (CSP) is considered untrusted. The provider is not necessarily malicious, but may be vulnerable to accidental exposure of sensitive data (or cryptographic keys), or attacks from malicious parties. Therefore, the approach pursued in developing the Use Case 4 architecture was to remove dependency on the CSP to implement the security mechanisms. Instead, the onus is on data owner to implement data protection mechanisms on the client-side, thus preventing the provider from affecting the security level of the data.

The architecture and prototype were jointly defined and developed by WT and EMC. The architecture comprises of a newly-developed middleware component orchestrating a number of open-source and proprietary services resulting in an elastic Cloud storage service that can adjust its capacity in a multi-Cloud environment without compromising on the security of the data. With the ESCUDO-CLOUD middleware, users have secure access to their data hosted by the Cloud provider, and can manage the security and storage configurations enforced on their data. Access to the service is possible via a web browser or an agent installed in the user devices. The service is compatible with public Cloud services such as AWS S3 and Google Drive, as well as hybrid Cloud solutions, such as Dell EMC's Elastic Cloud Storage (ECS) platform. Implementing this ESCUDO-CLOUD architecture, users will be able to leverage Cloud services in a more cost effective way, with a higher level of security, access control and assurance.

4.1.2 Context in ESCUDO-CLOUD

ESCUDO-CLOUD project considers across the four use cases four *dimensions* for secure data sharing in the Cloud. In tandem with this, the trust boundaries of each use case were clearly defined in order to appropriately target different aspects of those dimensions in each use case. Figure 4.1 illustrates the trust boundaries for UC4, with the data owner and authorized users identified as the only trusted entities. Data stored in across the multiple Cloud providers, is protected (i.e. encrypted) by the ESCUDO-CLOUD solution with only the trusted entities able to access the data unencrypted.

UC4 has provided the platform for the evaluation of several of the novel mechanisms and advances in data protection techniques that have resulted from the technical work packages (WP2-4). The Shuffle Index reported initially in D4.2 was implemented and tested on the ECS component of UC4, with the results documented in D4.3. The concept of *over-encryption* was first introduced in D3.1, with an initial implementation for policy evolution analysed in D2.2. The concept of

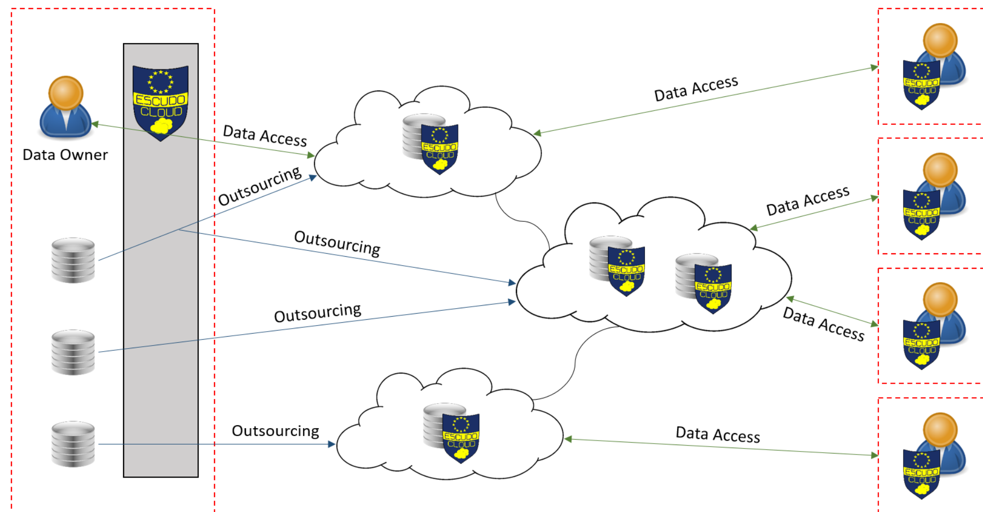


Figure 4.1: Overview of the Use Case 4 Trust Boundaries

over encryption and the flexibility of the mechanism demonstrated in D2.2 proved its potential for ECS, making use of the native encryption and key management capabilities. Following the model of implementing Base Encryption Layer (BEL) and Surface Encryption Layer (SEL) for access control, the architecture can be adapted such that the existing client-side encryption is the BEL, while ECS encryption can be applied as the SEL enabling a convenient mechanism for the revocation of access rights to users.

4.1.3 UC4: Core Technologies and Concepts

Elastic Cloud

Cloud computing was a massive disruptor in how the IT industry thought about their infrastructure and deployment models for services and applications. The impacts of this are still being felt across the industry as Cloud continues to evolve and new applications in areas such as IoT, Big Data, Edge Compute and 5G come into focus. Cloud was initially viewed as a low cost and scalable solution for IT infrastructure. However, studies have shown that Cloud may not be the best deployment model for some applications. The impact of latency, and the cost to move data (typically charge *per transaction*), combined with the reduce cost of establishing in-house storage facilities (due to lower cost of capacity on new drive technologies) has contributed to this evolution. Cloud still has a major role to play in IT infrastructure, however it needs to be flexible in order to meet the requirements of a wide array of use cases.

An elastic Cloud model offers flexibility and scalability, combining on-premises resources with public (or private) Cloud provider to achieve a saving in the Total Cost of Ownership (TCO) for the customer. UC4 addresses this aspect of a Cloud storage platform with the inclusion of EMC's Elastic Cloud Storage (ECS) platform (discussed in further detail in Section 4.3.2). Offering object storage capability, end user data can be stored more efficiently than traditional *hierarchical tree structure* file systems. This enables a private Cloud provider, offering elastic storage resources, to scale up their on-site and off-site storage pools to meet demand. Crucially, due to the data protection mechanisms implemented on the client, the data remains secure even if offloaded to third party providers.

Compliance in the Cloud

Finally, UC4 addressed the challenge for determining the compliance level of a provider, or chain of providers, with the security requirements of the data owner. There are many frameworks available that tackle this issue, such as the CSA Cloud Control Matrix[1] that was designed to provide guidance for Cloud vendors and customers in assessing the security risk of a Cloud provider. Another initiative by CSA is the Security, Trust and Assurance Registry (STAR)[2] that offers certification to providers indicating the security level of a Cloud provider. Outside of CSA, the FP7 project SPECS[3] proposed and developed a negotiation, enforcement, and monitoring platform that integrated with the service providers to implement the specifics of an agreed Security SLA between the provider and the customer. In contrast to the approach of CSA, the SPECS approach offered real-time monitoring of security levels. The benefits of this approach are significant but the ongoing technical integration costs as features and APIs are added, removed or modified for Cloud services is a potential barrier for the adoption of such a service.

The approach taken in ESCUDO-CLOUD for UC4, as discussed in Section 4.5, is more in line with that of the CSA. Using the RSA Archer tool for compliance checking[4], a template for a Cloud storage service was developed based on the use case requirements collected in D1.1. Requirements were included from all four use cases, enabling the template to be used in a range of configurations beyond those following the UC4 architecture. This template feeds into an online questionnaire portal that maps provider features to the security metrics, offering Cloud customers to assess the security posture of the provider.

4.2 Solution

The elastic Cloud solution developed by WT, integrating EMC's ECS service to provide the external Cloud storage infrastructure, aims to provide data protection in Cloud and multi-Cloud environments which enables users to protect their data without any involvement of the Cloud Provider on the fulfillment of data protection mechanisms. This solution enables end users to have control over their data while providing flexibility of the services provided by CSPs.

Use case 4 will promote trust in an elastic Cloud solution by the implementation of client-side encryption for sensitive data. Due to the inherent risks to data stored and processed in the Cloud, such as the threat to keys actively used in the Cloud, encryption at the source is a vital mechanism to be included as part of an organization's security strategy. Trust in the Cloud requires accountability and transparency from the CSP, enabling data owners to have visibility of how their data is managed. Through encryption and proper key management, data owners can retain their keys protecting their data stored in the Cloud. In this way, the data owner has the ultimate control over how their data are protected in the Cloud, how it is accessed and who may access it. Thus, this approach also facilitates the inherent secure usage of third party CSPs, so that a CSP can employ *Cloud bursting* techniques to expand the resource capacity of the service it is providing without additional risks to the data.

The objective of techniques or tools provided for Use Case 4 is to offer certifiable secure data management in the Cloud by including the following main features:

[1] https://cloudsecurityalliance.org/group/cloud-controls-matrix/
[2] https://cloudsecurityalliance.org/star/
[3] http://specs-project.eu/
[4] https://www.rsa.com/en-us/products/governance-risk-and-compliance

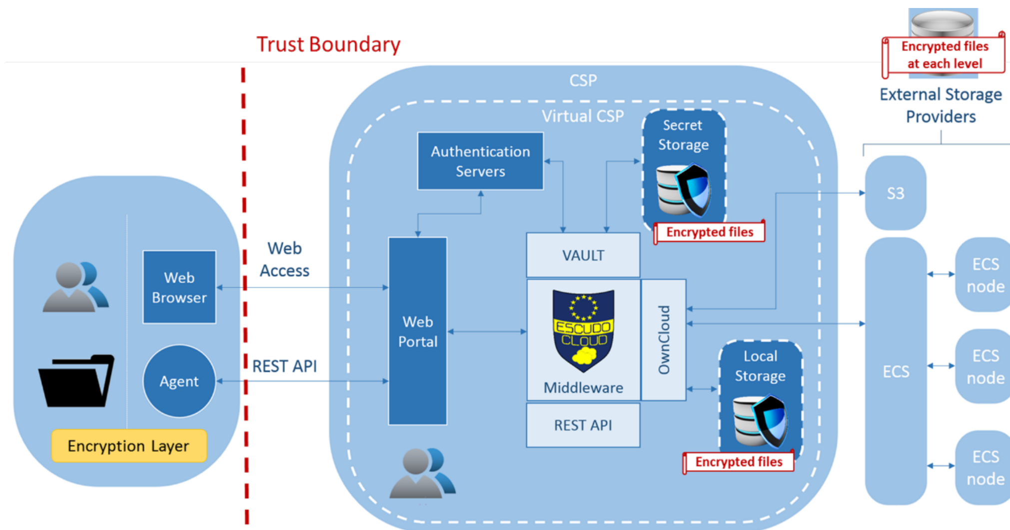


Figure 4.2: Elastic secure cloud storage architecture

- Encryption of the communication channel between the client and the CSP,
- Encryption features for file storage in the cloud,
- Access management by establishing and defining clear roles,
- Enabling file access through a web portal or agent.

4.3 Use Case 4 Architecture

The client-side encryption model adopted in Use Case 4 places the trust boundary as close as possible to the data owner. Figure 4.2 shows the general architecture used for the implementation of the *Elastic Secure Cloud Solution*. There exist three stakeholders in this scenario, namely: the *end user* (i.e. the data owner), the *agent* and the *CSP*. Communication between these actors is always the same: the end user will be able to manage their files hosted in the Cloud through the ESCUDO-CLOUD middleware. By using this middleware, the data is protected by client-side encryption, and only the end user can access the management of data protection mechanisms. In this architecture, the CSP is not involved in the encryption/decryption process, nor does it have access to the key material. Once the user is logged in via the EC middleware, encryption is performed transparently through the agent allowing the end users to access their data seamlessly, without any additional software requirements.

4.3.1 Core Functional Blocks

The core functional blocks of the architecture are the ESCUDO-CLOUD middleware, and the web portal. The *ESCUDO-CLOUD middleware* is the core module and is in charge of the system control. This module includes:

- **Vault framework**[5]: secures, stores, and tightly controls access to tokens, passwords, certificates, API keys, and other secrets in modern computing. Vault handles leasing, key revo-

[5]https://vaultproject.io

cation, key rolling, and auditing. Vault presents a unified API to access multiple backends: HSMs, AWS IAM, SQL databases, raw key/value, and more.

- **Owncloud platform**[6]: is a self-hosted file sync and share server. It provides access to your data through a web interface, sync clients or WebDAV while providing a platform to view, sync and share across devices easily – all under your control. Owncloud's open architecture is extensible via a simple but powerful API for applications and plugins and it works with any storage:
  - Local storage: CSP storage cluster integrated with OwnCloud platform.
  - External storage: Owncloud will be integrated with external system in accordance with the elastic cloud requirement.

In the Use Case 4 prototype, access can be done through a web portal (agentless) or via a pre-installed agent (desktop or mobile application). In both cases, access is done via HTTP API allowing the communication interface with ESCUDO-CLOUD Middleware enabling session management (including registration, login by authentication, user session and logout) and file management (upload, download, share and delete).

4.3.2 ECS Integration

Complementary to the core functional blocks are the storage providers offering elastic Cloud storage capabilities that enable further security, geographic diversity, and data replication and redundancy. The explosion of data that is being experienced by IT organizations in recent years has led to increased adoption of public Cloud storage platforms such as AWS S3. While these platforms offer significant economic advantages over traditional storage solutions, they introduce new challenges in the areas of data residency and impact that has on compliance with local laws and regulations.

The integration of ECS into the Use Case 4 architecture enables the primary Cloud provider (hosting the ESCUDO-CLOUD middleware) free up their primary storage (therefore reducing costs) while offloading *inactive* data to lower cost storage offsite. This is a particularly useful configuration for smaller private CSPs that require the flexibility to burst into public Cloud to meet storage requirements. Designed to leverage industry standard (lower cost) hardware to support flexible deployment models (including a software only solution), the approach is also more cost-effective than public Cloud services, with up to 48% lower Total Cost of Ownership (TCO)[7].

The technical integration of ECS required the configuration of three ECS nodes (CSPs) to demonstrate how data can be distributed across a multi-provider network of CSPs. Control of the data flow is managed through Owncloud which issues commands to the ECS control plane (running on a Virtual Machine (VM)) via either a REST API or S3. Compatibility with these pre-existing communications protocols reduces the integration complexity and therefore lowers any barriers to adoption for private Cloud providers.

[6] https://owncloud.org

[7] Enterprise Strategy Group: Dell EMC Elastic Cloud Storage, Economic Benefit Analysis of On-premises Object Storage Versus Public Cloud, December 2016

4.3.3 User Experience

The current version of the web app presents the registration, login and confirmation user views. Once logged in users can view files, and create, navigate and modify folders. From the security perspective, users are also able to encrypt and upload, and decrypt and download files (See Figure 4.3 for details on these actions).

The *user view* shows user information, the list of devices the user has logged in to, and the list of configured external storage. This view allows the user to change basic information and security controls (e.g., password), manage devices in order enable or revoke device access. The management of external storage at this stage is not automatic, the administrator via back-end application manage the external storage (mount, associate to user). Use Case 4 prototype offers four options for the storage, one local and three external (Google drive, Dropbox and ECS from Dell-EMC). An interesting feature for future development would be to automate the selection of external storage taking into account different criteria as to economic aspect and SLA compliance. This work could also lead to the implementation the functionality to blacklist providers and give the user access to the selection of external storage.

As an additional security measure, only a limited number of failed attempts to the login process is allowed. If this limit is reached, the user account is locked and only an administrator can unlock it.

4.3.4 Data Structure

The data structure enabling the encrypted file storage for the Use Case 4 core solutions is described in Figure 4.4.

people and user are related to the end user; *people is representing the client and* user is only using for the back-end application and contains the useful data for the administrator user. *devicesr contains the information of all user devices, which could be laptop, smartphone, tablets, etc., used to access the front-end application mainly encompassing session management and files management.* efiles and *sources are related to the encrypted stored files and are mainly used for the index of files and their synchronization. Another entity* mounts (external storage) is used but at this stage only from the back-end by the administrator (not depicted in this figure); it is not accessible at this stage by the user.

4.3.5 REST API

The current version of the API implementation for Use Case 4 enables:

- User management (used to create, update users and retrieve user information),
- File management (used to operate with files, download/upload from/to File Storage),
- Secret management (used to store and retrieve encryption keys from Secret Storage),
- Device management (used to store and delete the devices from which user has logged in),
- Mount management (used to create and remove external files storage mount points, as Google Drive and Dropbox using OAuth). WT exploits the username (API: NAMESPACE of url) in order to perform the new user registration and the authentication by means of JWT based application.

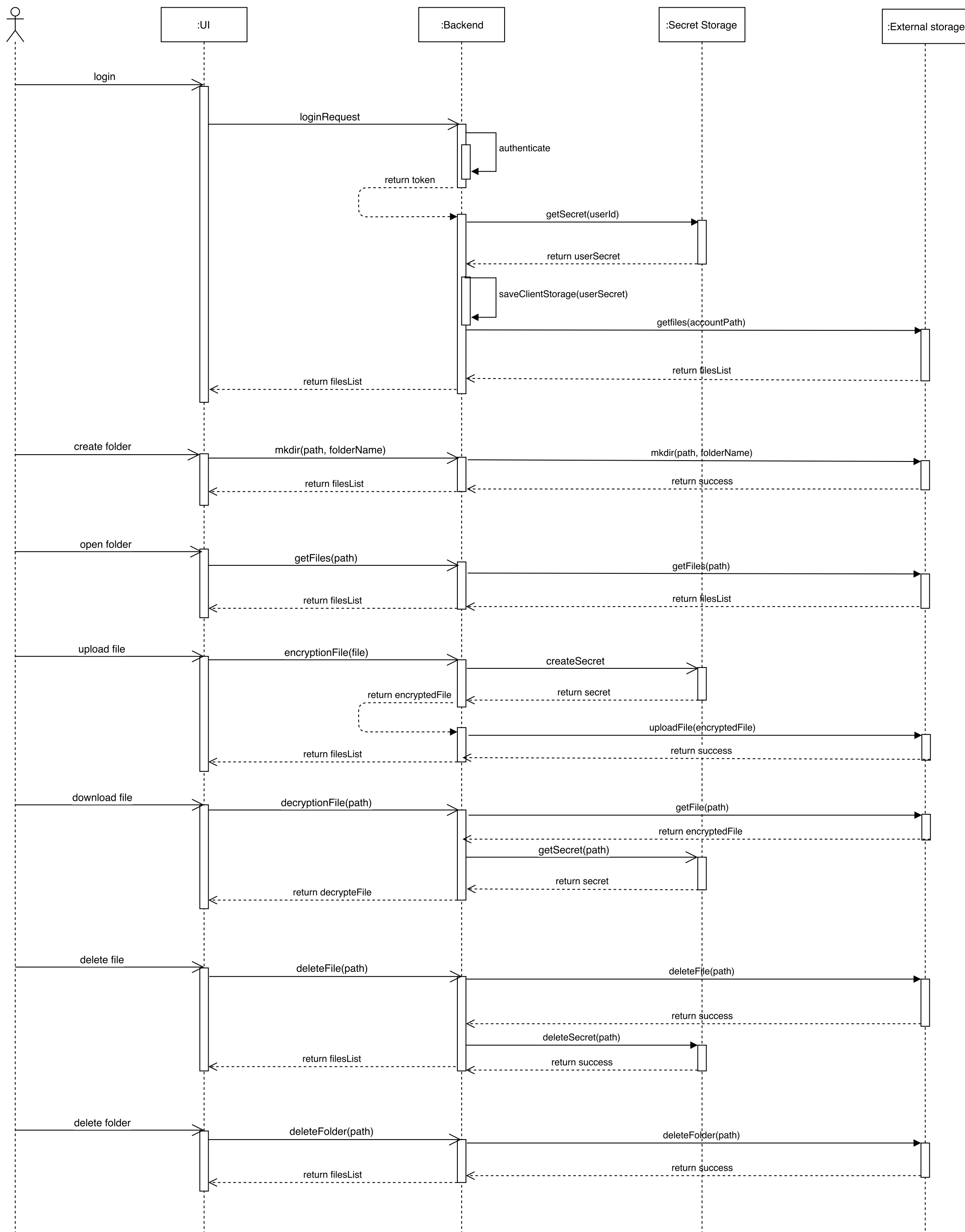


Figure 4.3: Use Case 4 user actions sequence

WT has also implemented a web interface (back-end application) for the administration of clients on the ESCUDO-CLOUD middleware. This is accessible only to users with administrative privileges. From this interface it is possible to perform operations on client profiles, such as confirm registered clients and lock or unlock clients. For file management, the ESCUDO-CLOUD middleware maintains a synchronized index of every file tree for each user has in file storage. This index also speeds-up file operations from the client in order to reach a good response time of the

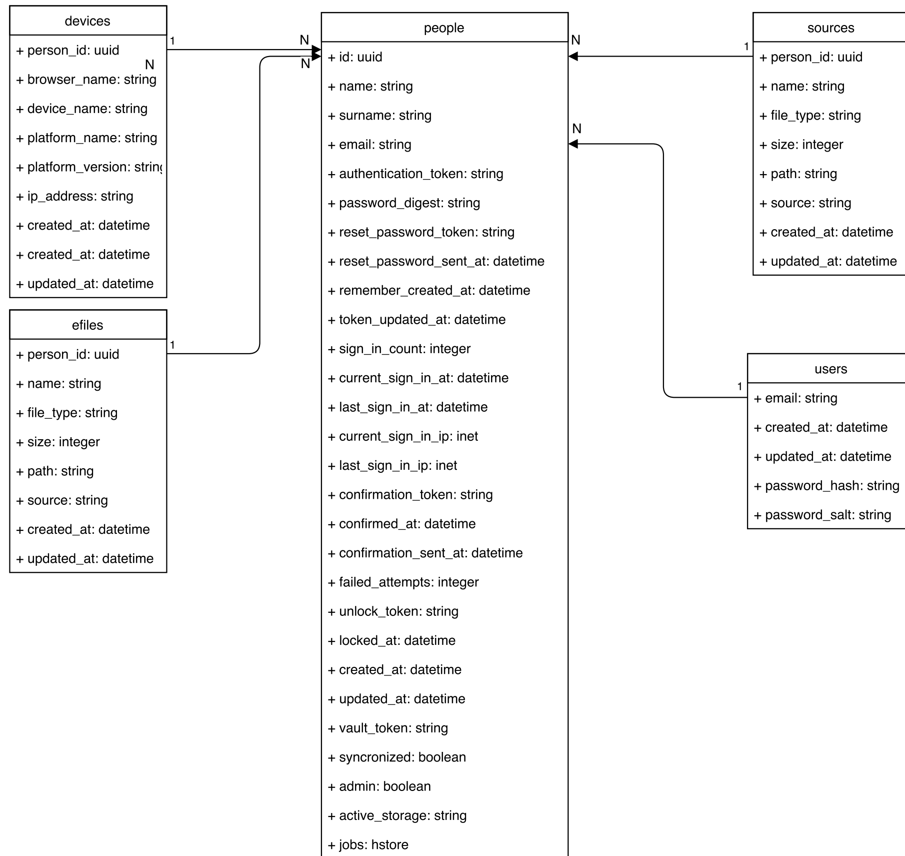


Figure 4.4: Use Case 4 data structure

application, a technical challenge for encryption process.

4.3.6 Architecture Summary

By implementing Use Case 4 elastic Cloud architecture, a CSP can offer cost-effective file storage configurations to suit the requirements of the end users. Data are stored across local and external storage resources in a dynamic way that enables the primary provider to extend the storage capacity on demand. This is ideally suited to private CSP that wish to avail of Cloud bursting mechanisms in periods of high load, or as a cost-effective approach for storing *inactive* data. Through client-side encryption and third party key management (accessed only via the ESCUDO-CLOUD middleware) the data remains secure as it moves within and across provider boundaries. Additional security mechanisms (i.e. encryption) implemented in ECS provides an additional layer of security, applying over-encryption to the data.

The prototype of the architecture is compatible with multiple providers such as Google Drive, Dropbox and EMC-ECS, therefore achieving the objective of implementing an elastic service. The technical objectives of the file storage are to store physically encrypted files and to provide an API to configure externally files store and an API to manage a back-up. The related implementation mainly encompasses the configuration and the deployment of Owncloud server (https://owncloud.org/), and the API endpoint developed to create new external storage. Another core objective is to store keys and sensitive user data in a secure way, i.e. in a secret storage. This was achieved through the configuration and deployment of Vault (https://vaultproject.io) and Consul (https://www.consul.io/) already shortly introduced in D1.4.

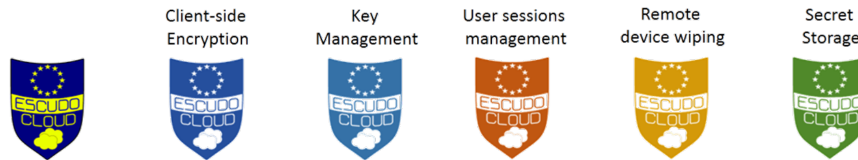


Figure 4.5: Security Features

This section has provided a technical overview of the core functional components of the Use Case 4 architecture with some discussion of the security mechanisms. The following section will explore the tools used for data protection in more detail.

4.4 Tools for Data Protection

While the adoption of the Cloud places a wealth of cost effective resources and services within reach of users and organizations, one also needs to assess the impact of the Cloud on trust on security. This requires the characterization of the security, privacy and dependability levels available to the user. The diversity of Cloud providers and the range of measurable dimensions (mainly security, privacy, and dependability) describing the *trust score* offered by each provider complicates the users' task in selecting the provider that can best fulfill their security requirements. Figure 4.5 highlights just some of the core security features available through ESCUDO-CLOUD within the context of Use Case 4. The following sections will provide further detail on how these features have been implemented in the Use Case 4 architecture.

4.4.1 Client-side Encryption

In the Use Case 4 architecture, the primary CSP has the capability to select and use different Cloud storage services. This is done in a transparent way so that it does not impact the user experience of the service. Client-side encryption provides a uniform level of security that is independent of the where the data ultimately will reside, whether it remains local to the primary provider or if it is offloaded to an external provider. While the other ESCUDO-CLOUD use cases have defined their own trust boundaries that dictate their file encryption/decryption approaches, Use Case 4 has followed the client-side approach where it is the client's responsibility to encrypt/decrypt files, and to generate the keys used for encryption/decryption. Figure 4.6 and Figure 4.7 describe the sequence diagrams for the encryption and decryption processes respectively. Further details on this mechanism were reported in D2.1. The technologies used for data encryption in Use Case 4 are PBKDF2-SHA512, AES-256-GCM, crypto JS and the SJCL Stanford Javascript crypto Library [8] (more details in the section 5.4.2. Vault with Consul backend are used for the storage of encrypted keys.

4.4.2 Key Management

In Use Case 4 prototype, user keys (i.e. master key, derived private key and public key) are generated and encrypted, and then sent to the server and stored in secret storage (i.e. the Vault

[8] http://bitwiseshiftleft.github.io/sjcl/

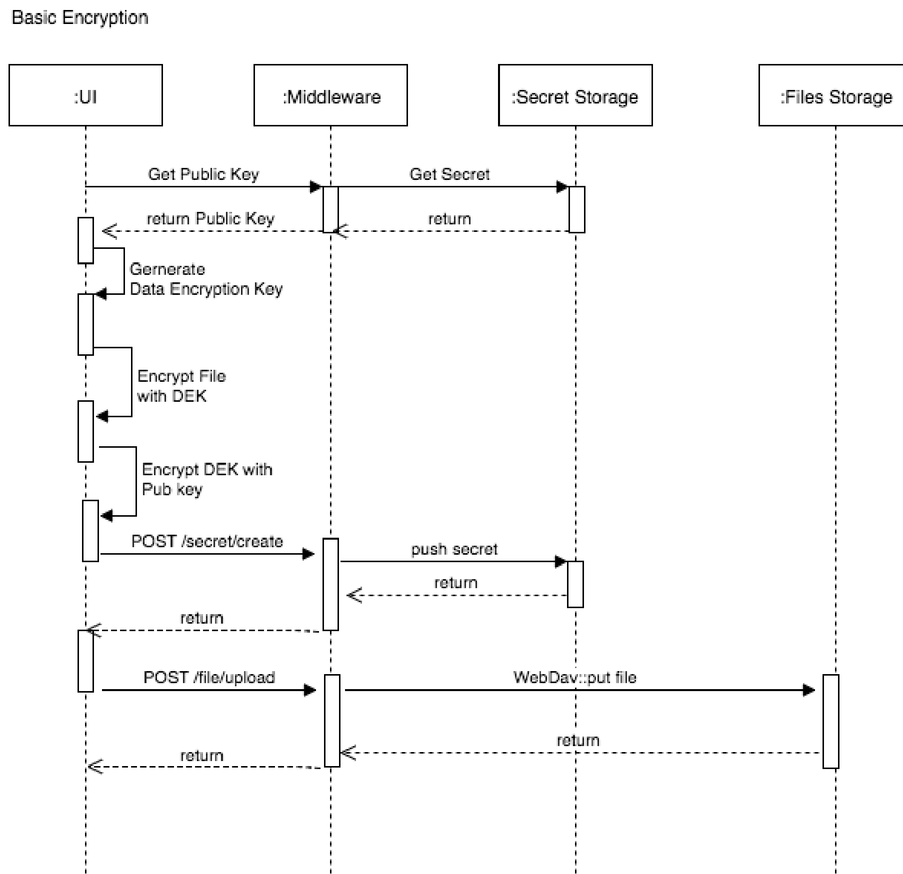


Figure 4.6: Use Case 4 encryption sequence

component). Key management mainly encompasses the generation of a master key and an asymmetric keys pair (i.e the public and private key components of a key pair) for each user and from the private key a derived private key that is an encrypted version of the private key. When a user logs into the client, a new master key to decrypt the private key is generated from the user password and the stored metadata (i.e. metadata of the password: hash, salt and iteration). A Data Encryption Key (DEK) is used to encrypt and decrypt the file contents and when a user shares a file, the client re-encrypts DEK with target user's public key.

The user's master key is created during her first login and it is tied to her password. At technical level, the master key is generated by using Crypto JS on a validated password of the first login and their metadata. The keys pair are generated by using elGamal combined with a random key of 256bits. And the derived private key is generated by using SJCL (Stanford Javascript Crypto Library) on the private key of the pair combined with the master key. The user keys is stored in the secret storage (a strategic architectural component of Use Case 4 enabling storing secrets).

4.4.3 Shuffle Index

The problem of protecting data confidentiality has been widely addressed by the research community (e.g., [DFS12]). However, protecting data confidentiality may not be sufficient. Indeed, by observing accesses, a Cloud provider could infer sensitive information about the user performing the access and the possibly sensitive content of the outsourced dataset. The distributed shuffle index [DFP+15b] [BDF+17] strengthens the guarantees of access confidentiality provided by the

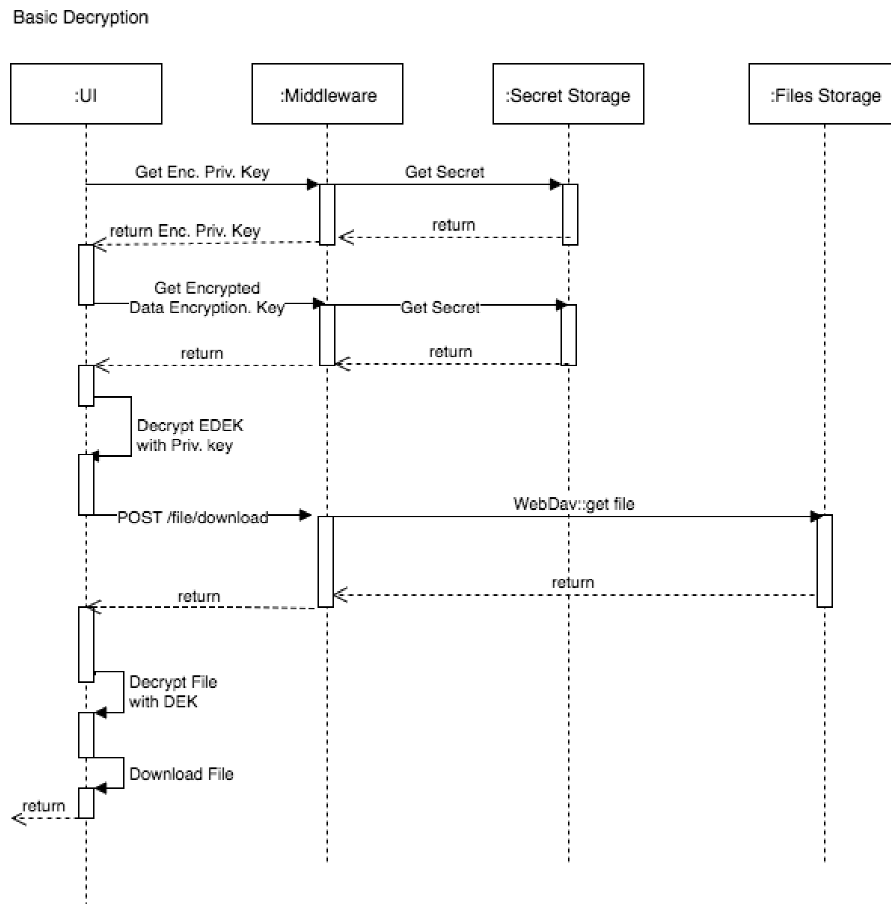


Figure 4.7: Use Case 4 decryption sequence

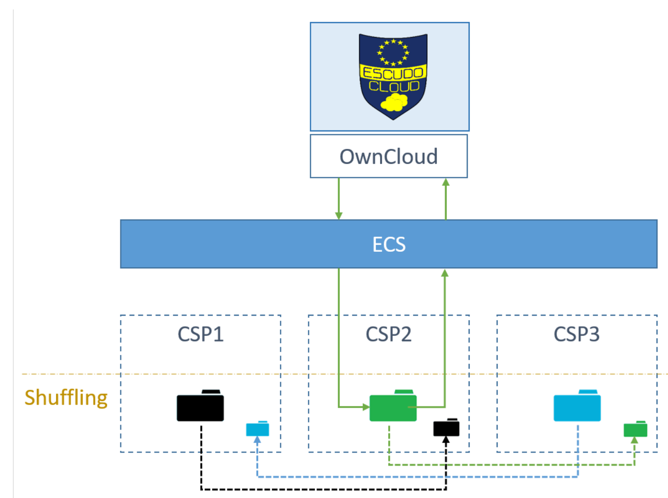


Figure 4.8: Use Case 4 Shuffle Index implementation

shuffle index through the distribution of data among three Cloud providers. Figure 4.8 illustrates the effect of the implementation on file access made to ECS (via Owncloud and the ESCUDO-CLOUD middleware). Accessing a file results in the retrieval of that file from ECS while also triggering an interation of the shuffle index, moving data across the three providers.

As part of the performance analysis of its implementation in D4.3, experimental results were

obtained from the Use Case 4 architecture - specifically the ECS Cloud storage component that enabled access to the shuffle index via multiple APIs (SWIFT, S3). Furthermore the solution demonstrated that it offered performance comparable to public Cloud platforms that were previously evaluated against [BBB+17].

4.4.4 Over-Encryption

One of the prominent challenges facing the approach of client-side encryption adopted by Use Case 4 is managing the overhead of user access to the data. Use Case 4 has already implemented access control mechanisms that require users to authenticate themselves, therefore enabling the service to verify their identity and determine if they are authorised to access the data. However, if this were the only required security mechanism to protect data, then encryption would not be an essential component of any data protection solution. This is essentially the scenario facing data owners when users access to data is revoked. If the data is not re-encrypted, from the perspective of the revoked users, it is plaintext data. Therefore, it is a recommended security practice to re-encrypt data sets when updates to the Access Control List (ACL) are made.

This process of encryption implies a modification and redistribution of the encryption keys. Use Case 4 places the most responsibility for encryption and the key management on the client. Therefore, should a user have their access to a particular data set revoked, in order to maintain the isolation of the encryption process from the CSP, the data set would need to be downloaded to the client and decrypted with the old key, re-encrypted with a new key, and uploaded to the provider. The key would also need to be uploaded back to the secret store on the provider so that it can be requested by authorized clients. The overhead of this process is potentially costly depending on the size of the dataset and the expected frequency of access policy modifications.

An alternative solution is over encryption. D2.2 evaluated this mechanism in three modes: *on the fly*, *on resource*, and *end to end* [BDF+16a] [BDF+16b]. The results of this evaluation determined that *on the fly* mode was the slowest and most computationally intense mode, but offered faster policy updates as it is applied as data is requested. The *on resource* and *end to end* modes on the other hand, provide faster response since they apply the Surface Encryption Layer (SEL) only during the policy change requests. This also results in slower policy updates as the update must be implemented across the entire data set. Assessing the applicability of the latter two modes for ECS, *on resource* maps to the existing architecture and offers the desired protection against revoked users. The Base Encryption Layer (BEL) applied at the client ensures that the data remains protected from the Cloud provider. Figure 4.9 illustrates how over encryption *on resource* mode is implemented.

4.5 Tools for Compliance Checking

When considering the empowerment of end users, the implementation of mechanisms for compliance checking is critical for the establishment of a basic contract between the CSP and data owner. The ability for a data owner to assess services against their requirements prior to acquiring that service enables them to make an informed decision and compels the provider make a commitment to the terms of the service they are offering.

The integration of a policy checking mechanism requires revisiting the use case architecture, identifying the points of integration, proposing the deployment model of the compliance checking tool, and specifying the relationships between it and the architectural components relevant to the

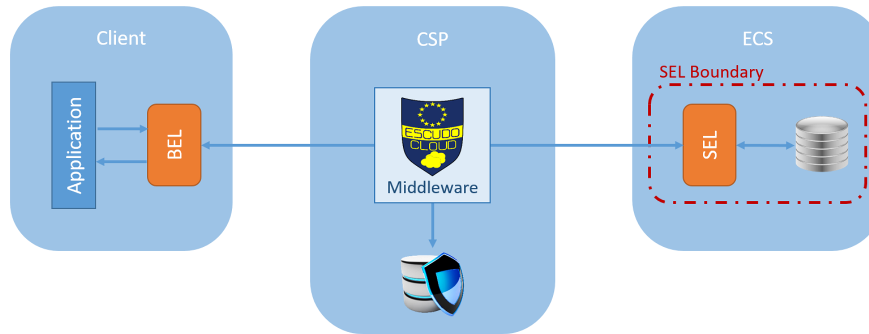


Figure 4.9: Use Case 4 Over Encryption (On Resource Mode)

security metrics. In this way, a reference approach will be established making use of the security metrics defined in D1.1 to evaluate policy compliance. This approach can be easily applied to the other use cases through the development of policy templates that address their respective security metrics.

4.5.1 RSA Archer

The Archer GRC platform is a commercial product produced by RSA, which is part of Dell Technologies. Governance, Risk and Compliance (GRC) are critical business functions within any organization, and are particularly challenging for medium-size and enterprise companies, where corporate information is spread between different departments and divisions. The Archer platform enables organizations to build an efficient, collaborative enterprise governance, risk and compliance program across IT, finance, operations and legal domains. It does this by providing a common repository for GRC data and applying consistent policies across the organization.

Archer solutions can be used within an organization to: *a)* manage the lifecycle of an organizations policies and procedures; *b)* optimize the compliance with industry regulations; *c)* visualize and effectively communicate risks at all business levels; *d)* investigate and resolve cyber incidents; *e)* business continuity, high availability and disaster recovery planning across the organization; and *f)* manage internal risk audits.

Archer has a number of mechanisms to ingest data into its GRC repository. It provides: *a)* REST interfaces that enable data to be pushed into Archer; *b)* SQL statements that enable Archer to query external databases; *c)* surveys that allow third-party users to interface with Archer in a controlled manner; *d)* bulk data uploads directly into the Archer repository; and *e)* custom integration with other RSA security products such as Netwitness for cyber security.

Archer has a large number of predefined corporate policies and procedures in domains such as IT, finance, legal and operations. In addition to the predefined policies and procedures, Archer provides tools that allow administrators to extend existing policies and procedures, or define new ones. The policies and procedures are grouped into solutions that cover a general corporate domain; a solution is composed of applications that focus on specific aspects of a corporate domain. Archer also provides a rich user interface, with many reporting and workflow features pre-built. It is also possible for the administrator to extend the user interface with custom screens and workflows. For example, there is an Archer solution that covers IT policies and procedures, which has an application that covers password security policy. An Archer administrator can alter the default security policy to conform with the organizations security requirements, e.g., passwords must be

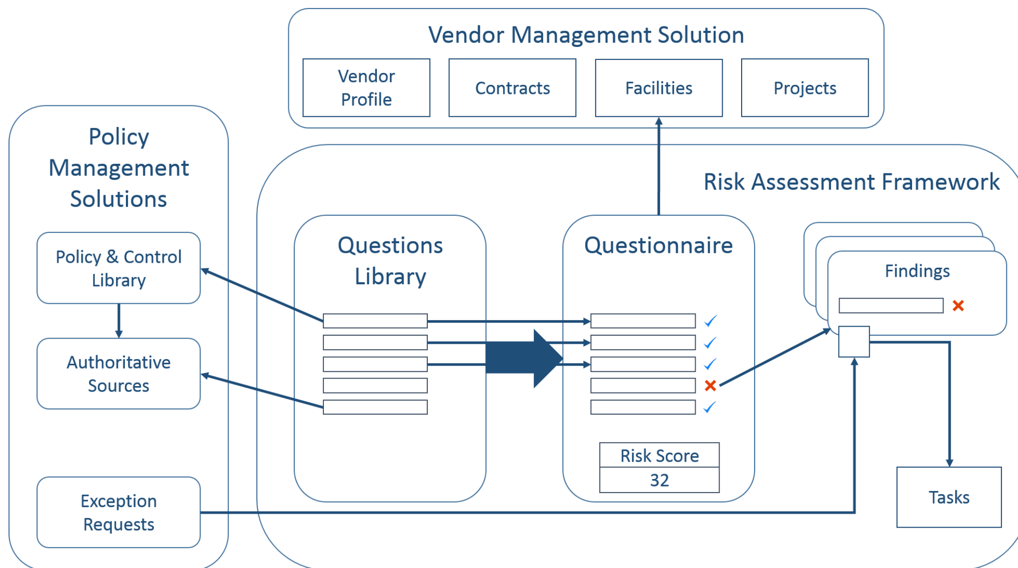


Figure 4.10: Archer Risk Assessment Framework

8 characters or longer, and have at least a number, a lower case, and an upper case letter. New IT systems can then be interrogated, using one of the ingest mechanisms listed above, to ensure that they are compliant with the organization's IT policies, and if not, a red flag is raised with the relevant manager(s).

4.5.2 RSA Archer Integration

Figure 4.11 describes the high level component view of the Use Case 4 reference architecture that has been prototyped. Here, RSA Archer [9] is introduced to the architecture, providing the tools for policy checking against the virtual CSP and third party services. Note that the solution does not aim to provide coverage of the client agent and web browser - only server side components are checked for compliance with policy. As described in Section 4.5.1, Archer tackles the challenge of policy compliance through the implementation of a self-assessment portal for evaluation of service against requirements.

4.5.3 Use Case 4 Requirements and Policy Management

End users wishing to implement the reference architecture illustrated in Figure 4.11 face many challenges when managing risks and compliance with the policies they set out. The pertinent data that enables checks against defined policies are often stored in a decentralised manner, typically across several documents that only represent the data at a specific point in time. This presents a challenge in obtaining real-time data that feed into autonomous processes managing the service, and triggering compliance alerts to the data owners.

Table 4.1 maps the requirements defined in D1.1 to the components identified in the architecture in Figure 4.11. This is a complete set of the relevant requirements for Use Case 4.

[9]https://www.rsa.com/en-us/products/governance-risk-and-compliance

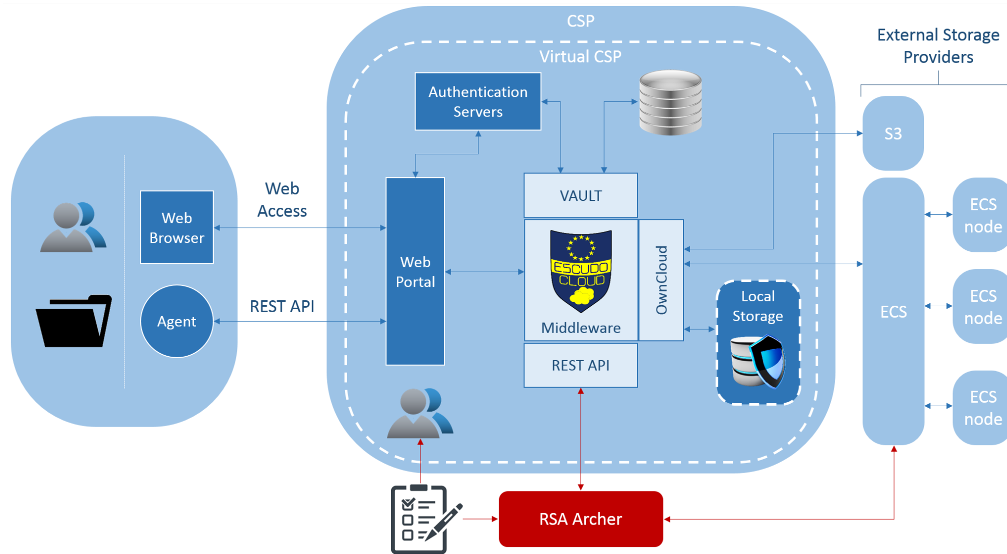


Figure 4.11: Elastic cloud service provider architecture with policy checking

Requirement Reference	Requirement Description	Covered by Component
REQ-UC4-AC-1	Access Control to the web portal	Middleware (authentication framework)
REQ-UC4-AC-2	Save credentials	Client; Vault;
REQ-UC4-AC-3	Access control to middleware	Owncloud (through Middleware)
REQ-UC4-AC-4	Access to shared files	Owncloud; ECS;
REQ-UC4-AC-5	Access grant by administrator for locked users	Middleware; ECS;
REQ-UC4-AC-6	Limit failed access attempts	Middleware
REQ-UC4-SS-1	Manage Cloud storage usage/capacity	Owncloud; ECS;
REQ-UC4-SS-2	Access control to storage	Middleware; Owncloud; ECS
REQ-UC4-SS-3	Responsive elastic Cloud storage	Owncloud; ECS;
REQ-UC4-SS-4	Comply with data protection directive	Middleware
REQ-UC4-SS-5	Data recovery control	Owncloud (through middleware); ECS;
REQ-UC4-DE-1	User encrypts/decrypts data control	Client; ECS;
REQ-UC4-DE-2	Encrypted data cloud storage	Owncloud; ECS;

REQ-UC4-DE-3	Ensure server synchronization	Middleware
REQ-UC4-DE-4	Secure file download	Client

Table 4.1: Use Case 4 requirements and their coverage by architectural components

The next step in implementing actual policy checking is to develop the policy templates for Archer against which these requirements must be checked. These policies will then be used for self-assessment data gather on the service. The following section will look at how those policies are used in a self-assessment conducted by a provider so that data owners can compare offerings from multiple CSPs.

4.5.4 Service Provider Assessment

Use Case 4 is particularly challenging from a security risk assessment perspective, as by definition Use Case 4 implements an elastic cloud service involving multiple parties. The user connects directly to the primary CSP, who, in turn, may subcontract the storage of the user's data to one, or more, secondary CSPs. Therefore, two entities may need to perform a security risk assessment, the primary CSP and the user wishing to store their data. Even if the security assessment information is held by an organization, it is often distributed between different departments and held in different formats, such as databases, spreadsheets, paper forms and contracts. CSPs may change their implementations from time to time, which may in turn change the security assessment. All these factors mean that it can prove almost impossible to obtain a concise security assessment that is maintained, and is up-to-date. For example, a secondary CSP may change its policy on the physical location, and store the user's data outside the EU, inadvertently breaking the user's security policy. Thus, it is important that an organization can collect the relevant CSP information on an ongoing basis, ingest it into a common canonical form, and store it in a central repository as security configuration metadata. Ideally, the process would be as automated as practical, to simplify the periodic ingestion of the CSPs information, and provide a suite of tools that apply predefined security policies, and raise alerts if the policies are breached.

As discussed in the previous section, Archer has a number of existing features that allow it to address the challenges of risk assessments, compliance, and external audits. In addition, Archer is highly extensible, which allows it to build CSP security assessments on top of its existing functionality. Archer will be extended to support a specific set of REST APIs that will automatically receive configuration updates from the CSP. However, not all CSP policies and procedures lend themselves to automated interfaces, such as personnel vetting procedures. Thus, the REST APIs will be supplemented with on-line questionnaires to be filled in by the CSPs. If the CSP makes an update to their terms and conditions, they will be requested to review, and if applicable update the questionnaires. In this way, Archer will be able to maintain an up-to-date picture of the CSP security assessment, stored in the central repository in a canonical form so that records are easily traceable. Archer stores the canonical data in the form of records (note, Archer's records are similar to database records, but are more flexible in their implementation), and records of a certain class are associated with a given application. Different applications, and records types, can be combined to form more complex interacting solutions. A set of applications and record types were defined for Archer to support the CSP Security Assessment of the ESCUDO-CLOUD use case architectures. The solution defines a number of standard policies and prerequisites that the

	BRONZE	SILVER	GOLD
Web Access Portal	✓	✓	✓
Data Access Control	✓	✓	✓
Data Recovery	✓	✓	✓
Data Encryption		✓	✓
3rd Party Key Management		✓	✓
Responsive Storage Capacity		✓	✓
User Key Management		✓	✓
Multi-User File Sharing			✓
Client Side Encryption			✓
Over Encryption			✓

Figure 4.12: Sample mapping between UC requirements and Gold, Silver, Bronze services

CSP must meet to be security compliant. Data owners are able to modify these policies as their security requirements change over time, in order to ensure they meet the specific requirements of their organization.

The organization, may wish to store different types of data that require different levels of security requirements. For example, HR personnel data is often of the highest security, as there is a regulatory requirement to keep the data from falling into unauthorized hands; additionally, there may be restrictions on which jurisdictions the data can pass through or be stored in. At the other end of the scale are might be product manuals that are freely distributable; the only requirement for this type of data is that it cannot be modified by an unauthorized party, and that their storage is resilient. These varying levels of security are often classified into bands, with increasing levels of security features. Figure 4.12 illustrates an example of such a classification, offering different sets of security features in each band.

For ease of use, the bands are typically given a notional value label, such as Bronze, Silver and Gold. Gold, being the highest level, has the most restrictive security requirements. Whereas Silver has lower/fewer security requirements, and Bronze is the lowest level of security that is acceptable to the organization. The new CSP Security Assessment solution will define the increasing policies and constraints that apply to the Bronze, Silver and Gold levels. There is likely to be an increasing level of cost associated with increasing levels of security. Therefore, the organization can associate a cost per GB/month of data stored to each of the security levels. The individual organization's departments can then make an informed decision on the price they are prepared to pay, for storing their data on CSPs, compared to the level of risk they are prepared to accept. In some cases, organizations may wish to offer a "Platinum" level of storage, where the user's data is stored within the organization and not outsourced to a CSP.

An additional useful feature of Archer, in the context of the modular architecture for Use Case 4, is a risk assessment tool that delivers a risk assessment questionnaire to third parties. Increasingly, organizations are using third parties to support their operations and services that are delivered to clients (as is the case in Use Case 4). This dependency on external resources introduces significant risks, most notably to regulatory compliance, data privacy, security breaches,

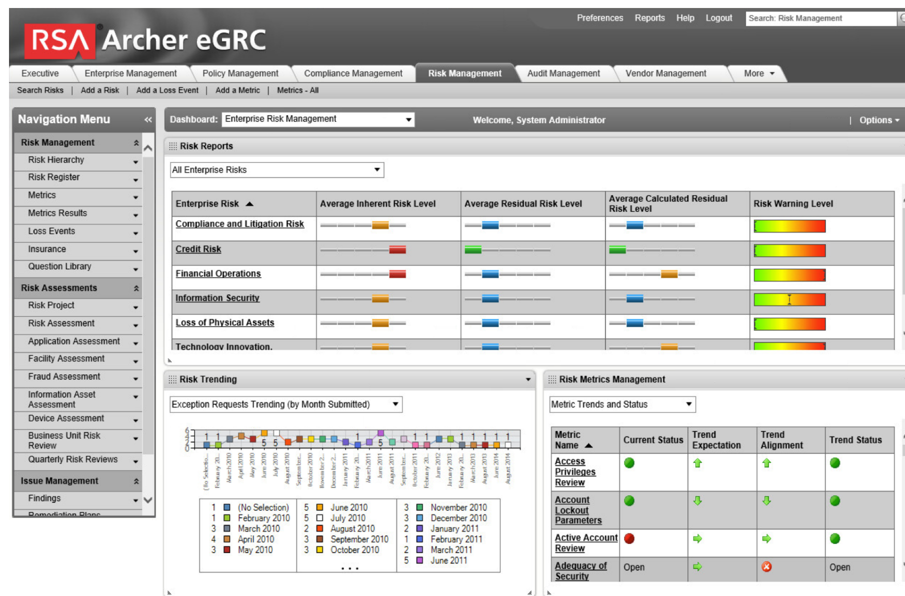


Figure 4.13: Archer Risk Assessment Dashboard

errors in supply chain, and damage to the reputation of the service. Figure 4.13 shows a view of the risk assessment dashboard that is used to manage and monitor risk reports and compare services against policy.

Complementary to the ability to verify the status of the service with respect to requirements defined in the service policy, this risk assessment enables the service provider to gain better insights into the risks associated with their third party providers, and the controls that they have in place to mitigate those risks. The results of the questionnaires conducted via Archer are used to assess the overall risk to the service across several risk categories (compliance, financial, security, reputation, resiliency, sustainability). These results are captured and managed so that action plans can be established, where required, to resolve any significant risks.

4.6 Summary

The core objective of UC4 was to develop an architecture for an elastic Cloud storage service that provides client-side encryption, and is supported by complementary mechanisms to manage cryptographic keys, and offer further security guarantees. One example of such a mechanism is the application of over-encryption [DFP$^+$15a] [DFLS16] (introduced in D3.1, with an initial study in D2.2), using EMC's ECS platform, protecting data stored in ECS nodes from revoked users with access to keys used for the BEL. WT performed a market comparison against different solutions (e.g. pCloud, Tresorit and BoxCryptor) and examined features such as file sharing, multi-cloud, device and session management, centralized management interface, virtual drives, data recovery, data synchronization and secret storage containers. With this baseline set of feature requirements, the architecture aimed to consolidate these into one solution, with particular emphasis on *device and session management*, *centralized management interface* and *secret storage containers*.

In the demonstrator, a data owner has access to their files stored in the Cloud using an access point available on the client. Three types of access points are available: a web portal access, a mobile application, and a standalone executable installed on the client. The access point will open a secure communication channel between the data owner and the CSP, allowing them to manage

their account and their stored data in the Cloud. In addition, the user can access the services by different devices share data and remove one in case of its theft (remote device wiping).

By implementing client-side encryption the trust boundary is moved right up to the user layer, meaning the data owner does not require the intervention of third-party providers to implement the first line of defense against attackers attempting to observe or manipulate their data. Often, the weakest point in any encryption scheme is the protection of the encryption/decryption keys. In this architecture, keys are first encrypted and then securely stored in the secret storage. As a result, the data owner has total control over how their data is protected in the Cloud, how it is accessed and who may access it. Access control mechanisms are in place to prevent unauthorized users from accessing the data (and keys). To further enhance the confidentiality guarantees of the solution, mechanism for over-encryption and index shuffling have been tested, and offer protection against revoked users and curious (or malicious) providers respectively.

Finally, a tool for policy compliance checking through self-assessment has also been implemented, with templates defined that apply across all the ESCUDO-CLOUD use cases. Self-assessment questionnaires completed by service providers ensure accountability for the services that they provide to end-users. This enshrines the services in record, enabling end-users to make decisions on service adoption based on trusted policy data from the provider matched against their own requirements. Archers core strength, as it relates to the ESCUDO-CLOUD use cases, is its ability to build policy templates that describe a Cloud services metrics (performance and security). The security metrics defined in D1.1, D1.2, and D4.1 feed directly into the definition of that policy template and allow Cloud consumers to not only ensure that the service they are acquiring is compliant with their requirements but also allow them to compare a number of providers offering similar features.

5. Conclusion

In this document, we presented the activities and results of the work carried out on the ESCUDO-CLOUD use cases under Work Package 1 between M1 and M36. WP1 Use Case 1 to 4 assemble a line of complementary tools that are empowering Cloud users with comprehensive means for security. ESCUDO-CLOUD industrial partners and SMEs in WP1 deployed use cases in the form of self contained demonstrators. The previous chapters outlined how the objectives per use case were achieved by transfer and integration of research results from the individual WP2, WP3, and WP4.

Use Case 1 addressed the scenario of guaranteeing to a data owner the protection of her data as well as the ability to efficiently access and operate on them when relying on the cloud for their storage. UC1 mainly fed from the WP2 research on protection techniques for oursourced data. The UC1 tools extend cloud storage offerings such as OpenStack swift with a solution for encryption of data-at-rest and key-management. By extending the OpenStack swift Cloud-service framework a wide audience in the Cloud industry is addressed. Use Case 2 addressed the scenario of providing the data owner with the ability not only to protect and access data, but also to selectively share them with other users and owners. UC2 mainly transferred research results from WP3 on selective information sharing in the Cloud. The tools of UC2 permit transition to the cloud to manufacturing companies. Given critical customer information and requirement for full data ownership the tools solve accountability problems on the on-premise as well as cloud operators side by processing over encrypted data in the Cloud. Use Case 3 and Use Case 4 addressed the scenario of enabling data owners to reason and assess trust in multi-clouds and federated clouds. To this end both use cases picked up the research results Data-Protection-as-a-Service provided by WP4. The tools of UC3 realize data protection for Cloud block, object and Big Data storage services. This helps in moving towards a modular design for a secure data protection solution in a Federated Cloud environment. In addition, UC4 tools specifically support SMEs in their competition with large Cloud providers. The introduced elastic Cloud storage service provides client-side encryption and policy compliance checking.

The line of ESCUDO-CLOUD Use Case 1 to 4 brought the trust boundary closer to the Cloud users in their respective scenarios.

Bibliography

[AMB17] AMBARI. Apache Ambari. *https://ambari.apache.org/*, 2017.

[Bar16] Elaine Barker. Recommendation for key management — Part 1: General. NIST special publication 800-57 part 1 revision 4, National Institute of Standards and Technology (NIST), 2016. Available from `http://csrc.nist.gov/publications/PubsSPs.html`.

[BBB+05] Elaine Barker, William Barker, William Burr, William Polk, and Miles Smid. Recommendation for key management — Part 2: Best practices for key management organization. NIST special publication 800-57, National Institute of Standards and Technology (NIST), 2005. Available from `http://csrc.nist.gov/publications/PubsSPs.html`.

[BBB+17] Enrico Bacis, Alan Barnett, Andrew Byrne, Sabrina De Capitani di Vimercati, Sara Foresti, Stefano Paraboschi, Marco Rosa, and Pierangela Samarati. Distributed shuffle index: Analysis and implementation in an industrial testbed. In *Proc. of the 5th IEEE Conference on Communications and Network Security (CNS 2017)*, Las Vegas, NV, USA, October 2017. poster.

[BCH+10] Mathias Björkqvist, Christian Cachin, Robert Haas, Xiao-Yu Hu, Anil Kurmus, René Pawlitzek, and Marko Vukolić. Design and implementation of a key-lifecycle management system. In Radu Sion, editor, *Proc. Financial Cryptography and Data Security (FC 2010)*, volume 6052 of *LNCS*, pages 160–174. Springer, 2010.

[BD15] Elaine Barker and Quynh Dang. Recommendation for key management — Part 3: Application-specific key management guidance. NIST special publication 800-57 part 3 revision 1, National Institute of Standards and Technology (NIST), 2015. Available from `http://csrc.nist.gov/publications/PubsSPs.html`.

[BDF+16a] Enrico Bacis, Sabrina De Capitani di Vimercati, Sara Foresti, D. Guttadoro, Stefano Paraboschi, Marco Rosa, Pierangela Samarati, and A. Saullo. Managing data sharing in openstack swift with over-encryption. In *Proc. of the 3rd ACM Workshop on Information Sharing and Collaborative Security (WISCS 2016)*, Vienna, Austria, October 2016.

[BDF+16b] Enrico Bacis, Sabrina De Capitani di Vimercati, Sara Foresti, Stefano Paraboschi, Marco Rosa, and Pierangela Samarati. Access control management for secure cloud storage. In *Proc. of the 12th EAI International Conference on Security and Privacy in Communication Networks (SecureComm 2016)*, Guangzhou, China, October 2016.

[BDF$^+$17] Enrico Bacis, Sabrina De Capitani di Vimercati, Sara Foresti, Stefano Paraboschi, Marco Rosa, and Pierangela Samarati. Distributed shuffle index in the cloud: Implementation and evaluation. In *Proc. of the 4th IEEE International Conference on Cyber Security and Cloud Computing (IEEE CSCloud 2017)*, New York, USA, June 2017.

[Blu10] David Blumenthal. Launching hitech. *N Engl J Med*, 2010(362):382–385, 2010.

[BPTG14] Raphael Bost, Raluca A. Popa, Stephen Tu, and Shafi Goldwasser. Machine learning classification over encrypted data. *IACR Cryptology ePrint Archive*, 2014:331, 2014.

[CDL08] Daniel A Cohen, Aiyesha Dey, and Thomas Z Lys. Real and accrual-based earnings management in the pre-and post-sarbanes-oxley periods. *The accounting review*, 83(3):757–787, 2008.

[CHHS13] Christian Cachin, Kristiyan Haralambiev, Hsu-Chun Hsiao, and Alessandro Sorniotti. Policy-based secure deletion. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 259–270, New York, NY, USA, 2013. ACM.

[DFLS16] Sabrina De Capitani di Vimercati, Sara Foresti, Giovanni Livraga, and Pierangela Samarati. Practical techniques building on encryption for protecting and managing data in the cloud. In P. Ryan, D. Naccache, and J.-J. Quisquater, editors, *Festschrift for David Kahn*. Springer, 2016.

[DFP$^+$15a] Sabrina De Capitani di Vimercati, Sara Foresti, Stefano Paraboschi, Gerardo Pelosi, and Pierangela Samarati. Shuffle index: Efficient and private access to outsourced data. *ACM Transactions on Storage (TOS)*, 11(4):1–55, October 2015. Article 19.

[DFP$^+$15b] Sabrina De Capitani di Vimercati, Sara Foresti, Stefano Paraboschi, Gerardo Pelosi, and Pierangela Samarati. Three-server swapping for access confidentiality. *IEEE Transactions on Cloud Computing (TCC)*, 2015.

[DFS12] Sabrina De Capitani di Vimercati, Sara Foresti, and Pierangela Samarati. Managing and accessing data in the cloud: Privacy risks and approaches. In *Proc. of CRiSIS*, Cork, Ireland, October 2012.

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

[HIP03] HIPAA privacy rule and public health – guidance from CDC and the US Department of Health and Human Services. Morbidity and Mortality Weekly Report, April 2003.

[KS08] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free XOR gates and applications. In *Proceedings of the 2008 International Colloquium on Automata, Languages and Programming*, ICALP, pages 486–498, Reykjavik, Iceland, July 2008.

[KS14] Florian Kerschbaum and Axel Schröpfer. Optimal average-complexity ideal-security order-preserving encryption. In *Proceedings of the 2014 ACM SIGSAC Conference*

on Computer and Communications Security, CCS, pages 275–286, Scottsdale, AZ, USA, November 2014.

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

[LP07] Yehuda Lindell and Benny Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In *Proceedings of the 26th Annual International Conference on Advances in Cryptology*, EUROCRYPT, pages 52–78, Berlin, Heidelberg, 2007. Springer-Verlag.

[LP09] Yehuda Lindell and Benny Pinkas. A proof of security of yao's protocol for two-party computation. *J. Cryptol.*, 22(2):161–188, April 2009.

[Man13] Alessandro Mantelero. The eu proposal for a general data protection regulation and the roots of the 'right to be forgotten'. *Computer Law & Security Review*, 29(3):229–235, 2013.

[MR08] Edward A Morse and Vasant Raval. Pci dss: Payment card industry data security standards in context. *Computer Law & Security Review*, 24(6):540–554, 2008.

[NIS01] NIST. Fips pub 140-2. *Security Requirements for Cryptographic Modules*, 25, 2001.

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

[PRZB11] Raluca A. Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. Cryptdb: Protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP, pages 85–100, New York, NY, USA, 2011. ACM.

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

[TZS17] Ahmed Taha, Heng Zhang, and Neeraj Suri. Report on multi cloud and federated cloud. Deliverable D4.4, ESCUDO-CLOUD, October 2017.