European Commission | Horizon 2020 European Union funding for Research & Innovation

RNP

GOVERNO FEDERAL BRASIL PÁTRIA EDUCADORA

SecureCloud

Joint EU-Brazil Research and Innovation Action
SECURE BIG DATA PROCESSING IN UNTRUSTED CLOUDS

https://www.securecloudproject.eu/

Specification and implementation of the micro-service framework and API
D3.1

Due date: 31 December 2016
Submission date: 24 January 2017

Start date of project: 1 January 2016

Document type: Deliverable
Work package: WP3

Editor: Florian Kelbert (IMP)

Reviewer: Marcell Feher (CC)
Keiko Veronica Ono Fonseca (UTFPR)

Dissemination Level

PU	Public	√
CO	Confidential, only for members of the consortium (including the Commission Services)	
CI	Classified, as referred to in Commission Decision 2001/844/EC	

SecureCloud has received funding from the European Union's Horizon 2020 research and innovation programme and was supported by the Swiss State Secretariat for Education, Research and Innovation (SERI) under grant agreement No 690111.

Tasks related to this deliverable:

Task No.	Task description	Partners involved°
T3.1	Framework for restartable secure micro-services	IMP*, TUD, UniNE
T3.2	Set of reusable secure micro-services	IMP*, TUD, UniNE
T3.3	Dependability management	TUD*, IMP, UniNE

°This task list may not be equivalent to the list of partners contributing as authors to the deliverable
*Task leader

Contents

1 Introduction **2**

2 Security-critical Requirements of the SecureCloud Framework **3**

3 SecureCloud Framework for Secure, Dependable, and Restartable Microservices **4**
3.1 SecureCloud Architecture 4
3.2 Scheduling and Orchestration 6
3.3 Attestation 7
3.3.1 Microservices as attestees 8
3.3.2 Verifiers attesting microservices 9
3.4 Auditing 11
3.4.1 Linking against libseal 12
3.4.2 Parsing the application layer protocol 12
3.4.3 Formalising invariants 13

4 Demonstrator **15**
4.1 The SecureCloud auditing service 15
4.1.1 Running the secure Git microservice 15
4.1.2 Using the secure Git microservice 16
4.1.3 Auditing the secure Git microservice 17
4.1.4 Auditing the Secure Git microservice after malfunctioning 18
4.1.5 Summary 18
4.2 The SecureCloud attestation facilities 18
4.2.1 Demonstration Requirements 21
4.2.2 Setting up Infrastructure 21
4.2.3 Starting the Example Program 23
4.2.4 Summary 24

5 Summary **25**

1 Introduction

The goal of the SecureCloud project is to enable the secure execution of big data applications within potentially malicious cloud environments. While the developed SecureCloud solutions are generic, the projects' use cases originate from the domain of the smart grid, thus providing a large range of security requirements that are also applicable in many other scenarios; they include: confidentiality, integrity, and availability of the data and application logic, as well as general dependability of the SecureCloud infrastructure. To address these challenges, the SecureCloud project leverages recent technological trends: the emergence of microservice-based applications, hardware security mechanisms, and microservice orchestration frameworks:

(1) Modern distributed big data applications are composed of microservices. Hereby, each microservice contributes a specialised functionality to the big data application. For a microservice-based big data application to be secure, it is essential that all individual microservices, their communication, and their composition are secure. The SecureCloud framework thus provides solutions to develop, deploy, execute, compose, and manage secure big data applications from individual microservices.

(2) SecureCloud achieves confidentiality and integrity guarantees by leveraging the novel Intel SGX CPU instruction set. A corresponding analysis of existing hardware security mechanisms is presented in Deliverable 2.1. By using Intel SGX, the SecureCloud infrastructure exposes only a minimal trusted computing base and is able to defend against powerful attackers that have physical access to the hardware or administrative privileges. In addition, the SecureCloud consortium develops original attestation and auditing technology on the basis of Intel SGX.

(3) SecureCloud achieves availability and dependability of big data applications by leveraging the state-of-the-art Docker Swarm container orchestration framework. By using Docker Swarm, the SecureCloud framework provides highly available and dependable big data applications that are resistant against hardware and operating system failures.

The mentioned smart grid use cases and early secure big data application development results have already been presented in Deliverables D1.1 and D4.1. Building upon these results, this deliverable serves the following purposes:

- It describes the overarching SecureCloud framework and API, thus integrating work packages WP2, W3, and WP4 (Section 3.1).
- It introduces the SecureCloud approach on how to achieve availability and dependability by using Container orchestration frameworks (Section 3.2);
- It describes the SecureCloud approach on how to attest remote microservices (Section 3.3).
- It describes the SecureCloud approach on how to audit microservices (Section 3.4).
- It provides a demonstrator showing an initial deployment of the above framework (Chapter 4).

We organize this deliverable as follows: Chapter 2 succinctly recaps security-critical use case requirements from D1.1. Chapter 3 presents details of the overall SecureCloud framework as detailed above. Chapter 4 describes our demonstrator, and Chapter 5 concludes.

2 Security-critical Requirements of the SecureCloud Framework

We develop the SecureCloud framework on the basis of the use case requirements described in D1.1. To understand and justify the design decisions behind the SecureCloud framework, we recap the most important security-critical requirements.

The use cases' functional requirements are covered by the big data applications developed in WP4 and described in D4.1. In this deliverable we thus focus on security-related non-functional requirements.

The use cases considered in the SecureCloud project are from the smart grid domain. They deal with dynamic electrical safety assessment (D1.1, Section 3.1), and smart meter data collection and processing for the purpose of fault analysis (D1.1, Section 3.2.1), fraud detection (D1.1, Section 3.2.2), and billing (D1.1, Section 3.3). In summary, their security requirements are as follows:

- Data confidentiality: Parts of the involved data, such as meter readings and bills, are personal and highly sensitive. The confidentiality of any such data, both at rest and in transit, must be assured.
- Data integrity: The integrity of data, such as safety assessments, fault analyses, meter readings, and bills must be assured both at rest and in transit.
- Application logic integrity: Applications performing safety assessments, fault analyses, and billing must operate correctly at all times. Any faulty behaviour must be prevented or detected in a timely manner.
- Application and data availability: Essential smart grid applications (safety analyses, safety assessments), must be available at all times. Similarly, smart grid data must be available and must not be lost.

3 SecureCloud Framework for Secure, Dependable, and Restartable Microservices

This chapter introduces the overall SecureCloud framework. The framework has been developed on the basis of the results reported in Deliverables D1.1 and D4.1: D1.1 described several use cases and their requirements, as well as a secure microservice runtime for single microservices; D4.1 described secure components for big data processing, storage, and communication. The SecureCloud framework, as depicted in Figure 3.1 and detailed in the following, serves the following purposes:

- Provide the ability to connect microservices to large-scale big-data applications
- Schedule and orchestrate microservices in an efficient and effective manner
- Provide means for the secure operation of microservices

3.1 SecureCloud Architecture

The security and dependability of SecureCloud microservices and microservice-based big data applications is based on two main pillars, the security features provided by Intel SGX and the SecureCloud framework API:

(1) SecureCloud microservices leverage Intel SGX to provide essential confidentiality and integrity guarantees for the sensitive data they process. To make use of these security features, SecureCloud microservices are compiled using the Intel SGX software development kit (SDK)[1]. The SecureCloud platform currently supports microservices written in C and C++. The consortium is further working on the support of additional programming languages, namely Rust[2] and Lua[3]. Deliverable D1.1 provided details on the Intel SGX technology and described how Docker[4] applications can be transparently protected by Intel SGX enclaves. In the following we thus assume microservices to be protected by SGX enclaves and focus on the description of the actual SecureCloud platform.

(2) The SecureCloud framework implements and provides further services for the secure and dependable execution of microservices and microservice-based big data applications, namely microservice orchestration and scheduling, microservice attestation, and microservice auditing: (i) orchestration and scheduling (Section 3.2) provide strategies to automatically deploy and scale microservices as well as complex microservice-based applications across multiple heterogeneous physical or virtual hosts; (ii) attestation (Section 3.3) provides functionalities to attest the trustworthiness of microservices—both to other microservices and to external parties; (iii) auditing (Section 3.4) provides the ability to ensure data and application logic integrity of microservices. Together, the API exposed by these services (as detailed in Sections 3.2 to 3.4) form the API of the SecureCloud framework.

Figure 3.2 shows an example instantiation of various secure microservices that, together, enable the composition big data applications. The depicted microservices, i.e., map/reduce, key/value store, SQL store, and Pub/Sub broker, are described in detail in Deliverable D4.1. In addition, Figure 3.2 highlights another dimension of the SecureCloud architecture: the secure communication between microservices. SecureCloud supports the secure communication between microservices on the basis of state-of-the-art

[1] https://software.intel.com/en-us/sgx-sdk
[2] https://www.rust-lang.org/
[3] https://www.lua.org/
[4] https://www.docker.com/

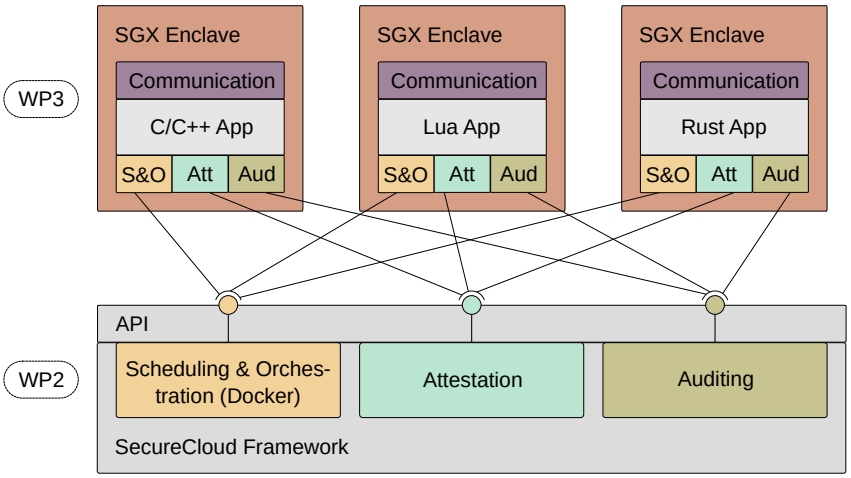

Figure 3.1: SecureCloud Architecture.

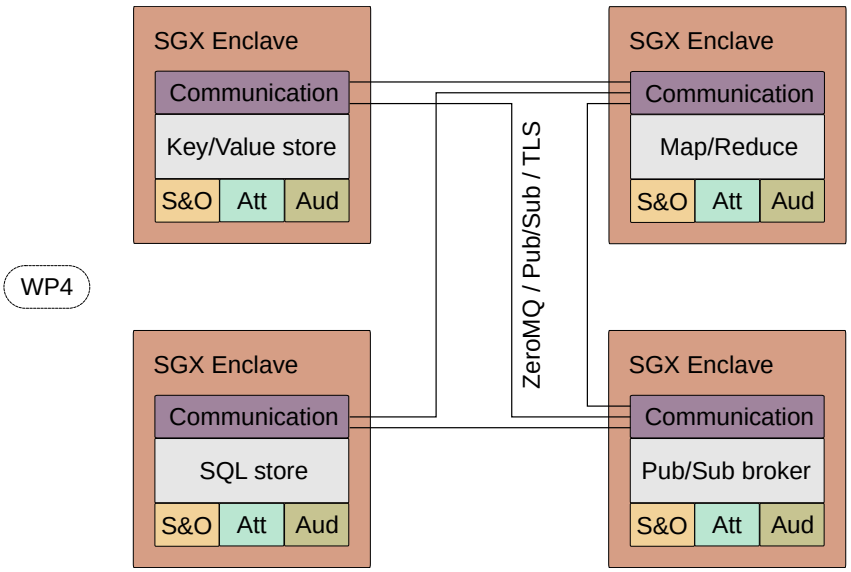

Figure 3.2: Instantiation of the SecureCloud architecture.

communication protocols. In particular, the SecureCloud architecture supports TLS, as well as secure versions of ZeroMQ and publish/subscribe as detailed in D4.1.

3.2 Scheduling and Orchestration

As described in Deliverable D1.1, the technology that will be used to facilitate the deployment of applications for secure big data processing in the cloud is based on containers. In addition, we will rely on OpenStack, a popular open-source platform for the implementation of public and private clouds. In this section, we describe the current state of scheduling and orchestration in OpenStack.

For an OpenStack cloud, we provision SGX-secured microservice containers (see Deliverables D1.1 and D4.1) on the basis of the OpenStack Nova Docker driver. Nova[5] is the compute service for OpenStack, managing how microservices are mapped onto physical nodes. The Nova service can be enhanced with drivers: (i) the KVM[6] (Kernel-based Virtual Machine) driver enables the creation of compute instances that are virtual machines hosted in a physical node with a KVM hypervisor; (ii) the Ironic driver enables the creation of compute instances that are physical nodes themselves; (iii) the Docker driver[7] enables the creation of compute instances that are Docker containers.

Because containers created through Nova-Docker are regular OpenStack instances, this model has a low adoption barrier for users as it inherits the widely-used OpenStack API for managing compute instances. This includes basic functions for the creation and deletion of instances, as well as for the interaction with other components such as the Nova scheduler and external orchestration components (in the case of OpenStack, the Heat component). Therefore, by using the Nova-Docker driver, the scheduling of secure containers requires only the following steps from the cloud operator:

- Hosts that are SGX capable must have SGX enabled in the BIOS and the SGX driver configured in the host operating system.
- The Nova service must be configured to group these physical hosts in a host aggregate[8]. A host aggregate is a operator-defined abstraction that typically represents a set of machines with some common features. This host aggregate should then have special metadata set, for example, `SUPPORT_SGX=TRUE`. If needed, a machine may belong to many host aggregates simultaneously.
- The operator must create one or more configurations of compute instances with the desired amount of resources (e.g., RAM, CPUs, disk space). This instance configuration is named a flavor. The flavor for an instance (be it a container or a VM) is a mandatory parameter when an instance is created. For our purposes, the flavor would have extra (optional) specifications, more specifically, the same metadata as in the host aggregate (in our case, `SUPPORT_SGX=TRUE`). It is also recommended that the flavor indicates the support in the name, for example, a non-secure version of the flavor could be "m1.small" (using the nomenclature common in OpenStack and in Amazon Web Services), and the secure version with the same amount of resources would be "secure.m1.small". This would help users to quickly identify the flavor they need.

[5] `https://wiki.openstack.org/wiki/Nova`
[6] `http://www.linux-kvm.org/page/Main_Page`
[7] `https://wiki.openstack.org/wiki/Docker`
[8] `http://docs.openstack.org/admin-guide/dashboard-manage-host-aggregates.html`

- Finally, the operator must set the `AggregateInstanceExtraSpecsFilter` filter in the Nova scheduler. This forces the Nova compute service to only schedule instances from a flavor that has a special metadata to a node in a host aggregate that has the same metadata key.

The configuration steps above will then enable containers that require SGX to only be scheduled on nodes that have adequate hardware. If an operator misconfigures a node that does not have the required hardware, the scheduler will still allocate the machine, but the user software stack will not boot completely. This can be detected during the attestation process, which will be detailed later.

To instantiate a secure container, the user (i) creates a secure Docker image as described in D1.1, (ii) submits this image to the OpenStack Image service called Glance[9], (iii) requests an instance of the image and a flavor that supports SGX, e.g., via the Nova API or via the OpenStack GUI (Horizon).

If the user targets applications that needs multiple containers, orchestration becomes useful. The OpenStack Nova-Docker driver handles containers individually. If an application requires a set of containers to provide the required microservices, this could be done using a component outside Nova, such as OpenStack Heat. Recently, however, container orchestrators, such as Kubernetes[10] and Docker Swarm[11] have gained massive attention from developers. These orchestrators offer services that facilitate the deployment, as well as offline and runtime management of applications composed of multiple containers.

These container orchestrators can used both as standalone tools or integrated with OpenStack, through the OpenStack Magnum project[12]. Among the advantages of using Magnum or the higher-level container orchestrators to handle containers, there are the following: (i) Magnum enables the replacement of the container technology (and not be bound to Docker); (ii) Nova-Docker does not support layered images, which could delay the creation of the instances; (iii) Nova-Docker requires that images are previously stored in the OpenStack Glance service; (iv) Nova-Docker does not support rolling updates of application containers and also will not handle availability.

Currently, the SecureCloud consortium is working on the integration of the above orchestration technology with secure containers. In particular, orchestration must be performed through tools that do not handle security-related aspects, such as attestation, transparently. We are further investigating in WP2 how users can reuse unmodified container orchestration tools (such as Docker Swarm) to manage their secure containers and benefit from the short provisioning times and other features enabled by the higher-level orchestrators. After that, we will investigate how we can embed the SecureCloud security aspects into the OpenStack Magnum framework and whether changes to the scheduling procedure can optimize resource usage or improve overall performance.

3.3 Attestation

The SecureCloud framework supports the attestation of SecureCloud microservices. The attestation functionality allows users and microservices to build trust into remote, previously unknown, SecureCloud microservices before making confidential data available to them. Attestation generally involves two

[9] `http://docs.openstack.org/developer/glance/`
[10] `https://kubernetes.io/`
[11] `https://www.docker.com/products/docker-swarm`
[12] `https://wiki.openstack.org/wiki/Magnum`

entities: a verifier and an attestee. The verifier is trusted and its goal is to extend trust into the yet untrusted attestee.

The SecureCloud framework provides the functionality needed by the attestee through a C-API called *Enclave Attestation Interface (EAI)*. The functionality needed by the verifier is provided through a Rust library. We chose different implementation languages for the attestee API and the verifier API for two reasons: (i) we want to maximize compatibility of the attestee's role with existing microservices, since all SecureCloud components have to act as attestee at least once. Consequently, we chose a C API with which any microservice can easily interface; (ii) Because the verifier's role is comparatively rare, we chose a more secure Rust API for the verifier's API.

SecureCloud's attestation functionality is based on the attestation facilities provided by Intel SGX where the trusted CPU collects security-relevant information of an enclave in a data structure called report. To allow the recipient to verify the authenticity of the report it is signed with private key only known to the CPU producing the quote, i.e., a data structure containing the report as well as its signature.

We improve Intel's design in two ways: (i) we hides internal attestation details; (ii) we reduces attestation latency, thereby improving dependability of SecureCloud microservices. To this end, SecureCloud introduces the *Local Attestation Service (LAS)* that runs on each cloud platform. The LAS serves two purposes: (i) it offers access to Intel's attestation facilities; (ii) it contains an enclave acting as a software root of trust taking over the responsibility of signing reports with a known keypair. Since a signature of a known keypair can be easily verified locally this allows us to reduce network communication with Intel's attestation service decreasing attestation latency.

Figure 3.3 summarizes the resulting infrastructure. Attestation is orchestrated by the EAI C-API and the Verifier Rust library. Intel's attestation scheme is implemented with the platform local *Intel Platform Software (Intel PSW)*, which generates quotes, and the IAS, that verifies former quotes. Our optimization adds the software root of trust enclave (SRoT) whose quotes can be verified by the Rust library directly. Both quoting mechanism are exposed via the LAS to the enclave.

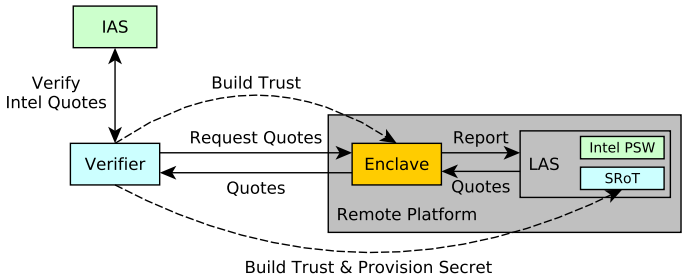

Figure 3.3: Architectural Overview of the Attestation Infrastructure

In the following first the microservice's integration with the EAI API (Section 3.3.1) is described, before verifier's integration with the Rust library is detailed (Section 3.3.2).

3.3.1 Microservices as attestees

The attestee functionality is exposed to microservices via a set of C functions explained in the following. Before using these, the cloud platform operator declares via environment variable `MUSL_LAS_ADDR` how

the LAS can be reached (e.g., `MUSL_LAS_ADDR=localhost:18766`). Notably, these settings might be overwritten by the user deploying the microservice, thereby superseding the LAS provided by the cloud platform provider.

The attestee initializes an attestation context with the two following functions:

```
eai_t* eai_create();
bool eai_init(eai_t* ctx, int fd);
```

Once the context was initialized, the attestee initiates remote attestation by calling function

```
bool eai_attest(eai_t* ctx);
```

This function invoces the main functionality of EAI: (i) it establishes a secure channel protected with TLS through the file descriptor provided in the `eai_init` call; (ii) it receives a quote request from the verifier; (iii) enriches the request with the report and forwards it to the LAS; (iv) the LAS returns the requested quotes, which are subsequently forwarded by EAI to the verifier. On the basis of these quotes, the verifier decides whether to trust or distrust the attestee as detailed in Section 3.3.2. In case the verifier decides to trust the attestee, the above function returns a positive return value.

Only if attestation succeeded, the established secure communication channel can subsequently be used to transmit arbitrary data using the functions

```
bool eai_send(eai_t* ctx, const void* buf, int len);
bool eai_recv(eai_t* ctx, void* buf, int len);
```

Confidentiality and integrity of the transmitted data is guaranteed by TLS.

Upon end of the communication, resources of the context are released by calling function

```
void eai_free(eai_t* ctx);
```

If any of the aforementioned function calls fails, an error string can be obtained via function

```
const char* eai_get_error(eai_t* ctx);
```

The verifier uses the verifier API described in Section 3.3.2.

3.3.2 Verifiers attesting microservices

The entity verifying a SecureCloud microservice (i.e., the verifier) must implement the verifier role of the procedure. The necessary functionality is provided through a Rust library detailed in the following.

Since Intel's remote attestation is the foundation of SecureCloud's functionality necessary code is included in the Rust library. In Intel's remote attestation scheme the verifier queries the IAS to make sure that a received quote was signed by a genuine Intel CPU. IAS is only available to registered user or service providers, in IAS terms. In the registration procedure the service provider has to produce a X509 certificate and decide whether he would like to have linkable or unlinkable quotes[13]. Subsequently, he is assigned a service provider ID. To make use of Intel's service the user has to produce his service provider ID, linkability setting, certificate and proof his possession of the certificate's private key.

In our Rust library interaction with the RESTful *Intel Attestation Service (IAS)* is encapsulated in the `Ias::IasVerifier` structure. In its constructing `new()` method all the required data to interact with IAS has to be provided:

[13]Linkability of quotes is a detail of the underlying EPID group signature scheme. With linkable quotes the verifier is able to associate two quotes produced by the same platform, which is not possible with unlinkable quotes

```
Ias::IasVerifier::new(
        intel_spid: [u8; 16],
        intel_linkable: bool,
        cert: &openssl::x509::X509,
        priv_key: &openssl::crypto::pkey::PKey)
    -> Result<Ias::IasVerifier, Ias::IasError>;
```

Upon successful creation of the `Ias::IasVerifier` structure, the verifier is able to verify enclave quotes using Intel's remote attestation procedure. This functionality is encapsulated within function

```
Ias::IasVerifier::verify_quote(
        self: &Ias::IasVerifier,
        quote: &[u8])
    -> Result<Ias::IasVerificationResponse, Ias::IasError>;
```

Thus, `Ias::IasVerifier` encapsulates a service providers access to the IAS, which is the basic building block to use Intel's remote attestation.

The majority of SecureCloud's attestation facilities are implemented in the `Eai::EaiVerifier` structure and its descendent `Eai::EaiVerifierConn` structure. They take care of communication with the enclaves, orchestrate the retrieval and verification of quotes, as well as, building trust into SecureCloud's software roots of trust in the LAS instances.

A `Eai::EaiVerifier` structure presents one verifier instance of a user wanting to build trust into SecureCloud microservices. This user is registered with Intel's IAS and might make use of SecureCloud's improved attestation scheme. If he wishes to use our optimized scheme, he has a list of LAS software roots of trust, i.e., enclaves offering the necessary signing capabilities, he is confident of and a public ed25519[14] key used to identify him. Furthermore, the user assigned all trusted software roots of trust a ed25519 keypair and a signature of the keypair generated with the user ed25519 keypair. This data will be used to build trust into previously unknown LAS instances and provision them signing keys afterwards. The `Eai::EaiVerifier` structure is created by calling the `new()` method that takes the previously described data:

```
Eai::EaiVerifier::new(
        ias_verifier: &Ias::IasVerifier,
        user_pubkey: Option<[u8; 32]>, // ed25519 public key of the user
        kkdb: Option<&Eai::kkqe_db>)
    -> Result<Eai::EaiVerifier, Eai::EaiVerifierError>;
```

Since `Eai::EaiVerifier` can always fall back to Intel's remote attestation procedure, the user's public key as well as the database containing trusted software roots of trust and their keys may be omitted within the above call.

The `Eai::EaiVerifier` structure can be used for multiple attestations. Each of which consists of three steps: (i) a TLS channel is established between the verifier and the attestee; (ii) the attestation is conducted; and (iii) after a successful attestation the secure channel can be used for additional communication. The context of an individual attestation is encapsulated in a `Eai::EaiVerifierConn` structure. Its constructing `new()` method takes a plain unprotected communication channel and a previously created `Eai::EaiVerifier` instance

```
Eai::EaiVerifierConn::new(
```

[14] `https://ed25519.cr.yp.to/`

```
        stream: Read+Write,
        eai_verifier: Eai::EaiVerifier)
    -> Result<Eai::EaiVerifierConn, Eai::EaiVerifierError>;
```

The attestation procedure is triggered by calling the structure's `attest()` method:

```
Eai::EaiVerifierConn::attest()
    -> Result<Ias::SgxReport, Eai::EaiVerifierError>;
```

Upon success, this function returns a `Ias::SgxReport` structure containing all data of the enclave report. Note that the calling instance is now responsible to decide whether the attested enclave is trustworthy or not. This can, for instance, be done by comparing the report's data to a list of trusted enclaves.

To facilitate post attestation communication, the structure implements Rust's `std::io::Read` and `std::io::Write` traits allowing straight forward usage of the communication channel in preexisting Rust code.

Section 4.2 describes a demonstrator that shows how the SecureCloud attestation facilities can be used to securely provision confidential configuration data to previously unknown enclaves.

3.4 Auditing

Microservices might misbehave in a multitude of ways and for a multitude of reasons. The consequences of such misbehavior include data loss and data damage—consquences that are particularly harmful in the smart grid use cases considered in the SecureCloud context. Reasons for misbehaving microservices include: software and hardware bugs, software misconfiguration, malware, misbehaving administrators, power outages, etc. The SecureCloud framework provides means to reliably identify such misbehaving microservices, which is an essential requirement to ensure the correct functioning of dependable microservice-based big-data appliactions.

Concretely, the SecureCloud framework provides a secure auditing service called *libseal* that enables the detection of incompliant microservice behavior. To detect such incompliant microservices, libseal records all of the microservice's communication, thus creating a *log* of all observed communication. libseal further regularly checks this log against microservice-specific invariants, i.e., properties that must hold at all times. As soon as one of the invariants is violated, libseal can take further action, e.g., shutdown the service, or notify the SecureCloud framework operator or the microservice operator. This aspect of libseal is flexibly configurable.

Technically, libseal is based on Intel SGX technology and runs within an SGX enclave along the actual microservice. The main idea behind libseal is to (i) terminate the encrypted communication between microservices inside the SGX enclave, and to (ii) securely log and check against invariants within the enclave. libseal is completely transparent to the actual microservice, i.e., the implementation of the microservice must not be adopted for use with libseal.

To enable the libseal auditing service for a microservice, the developer must perform three additional steps before deployment of the microservice: (i) link the microservice against the libseal library (Section 3.4.1); (ii) provide code to parse and interpret the microservice's application layer protocol (Section 3.4.2); (iii) formalise the invariants to be enforced (Section 3.4.3).

3.4.1 Linking against libseal

libseal is developed on the basis of the LibreSSL library[15]. LibreSSL is a simpler, supposedly more secure, and API-compatible fork of the more popular OpenSSL library[16]. By retaining LibreSSL's API, libseal is compatible with any application that uses LibreSSL or OpenSSL for secure SSL/TLS-based communication. To enable the facilities provided by libseal, the microservice developer links the microservice against libseal instead of LibreSSL or OpenSSL. While the API of libseal is equivalent to the API of LibreSSL, we mention the most important interfaces and explain their role within libseal:

```
int SSL_accept(SSL *s);
int SSL_do_handshake(SSL *s);
```

Any two microservices may use methods `SSL_accept()` and `SSL_do_handshake()` to set up a bidirectional, private, and secure communication channel. Upon these calls, libseal allocates any data structures that are needed during the remainder of the communication.

Once the secure communication channel is established, the two microservices exchange data by reading to and writing from the corresponding socket endpoints using API calls `SSL_read()` and `SSL_write()`, respectively:

```
int SSL_read(SSL *ssl, void *buf, int num);
int SSL_write(SSL *ssl, const void *buf,int num);
```

While the original LibreSSL implementation is in charge of decrypting and encrypting any data that comes from / goes to the communication channel, libseal causes the further processing and logging of the plaintext data as described in Section 3.4.2.

3.4.2 Parsing the application layer protocol

Logging all raw communication data would result in very large log files that are not particularly useful for invariant checking. The microservice developer must thus provide code that parses and prepares the communication data for logging and invariant checking. The corresponding API provided by libseal consists of a single function that must be implemented by the microservice developer:

```
char *libseal_log(const char *req, const char *rsp, const unsigned int req_len,
const unsigned int rsp_len, int *result_len);
```

Upon implementation of this function, the developer is provided with the following input parameters:

- `req`: the request being sent to the microservice
- `rsp`: the response being sent from the microservice
- `req_len`: the length of the request
- `rsp_len`: the length of the response

[15] `https://www.libressl.org/`
[16] `https://www.openssl.org/`

When implementing function `libseal_log()`, the developer must (i) transform these input parameters into a more structured representation of the communication and (ii) return the result as the function's return parameter, i.e., a character string. In addition, the developer must return its length within return parameter `result_len`. The returned character string must represent all essential communication data as a structured tuple as further detailed and motivated in Section 3.4.3.

3.4.3 Formalising invariants

libseal formalises and enforces invariants on the basis of a relational query language. Concretely, the implementation uses an in-memory sqlite[17]. Invariants are thus formalised as relational SQL database queries over the logged data that has been inserted into an SQL database.

The developer must thus follow three steps:

1. Define a database schema that suits the data being exchanged by the application layer protocol. For this, the developer must implement function

   ```
   char *libseal_init_relations();
   ```

   This function must return a character string that corresponds to an SQL query that creates SQL database tables[18]:

   ```
   CREATE TABLE ... (col_name col_type, ...); [CREATE TABLE ...]
   ```

   Notably, the developer may choose to define multiple database tables for a single microservice. In addition, auxiliary SQL VIEWs and FUNCTIONs may be defined to ease later data retrieval:

   ```
   CREATE VIEW ... AS SELECT ...;
   CREATE FUNCTION (...) RETURNS ... AS ...;
   ```

2. Preprocess the payload communication data such that it can be inserted into the above database tables. For this, the developer must implement function `libseal_log()` (as defined in Section 3.4.2) to return a character string that defines one or more SQL-INSERT[19] statements that are compatible with the SQL tables defined above. `libseal_log()` is thus expected to return a string along the following lines:

   ```
   INSERT INTO ... VALUES( ... ); [INSERT INTO ...]
   ```

3. Specify the desired invariants. Invariants are formalised as SQL-SELECT[20] queries over the above database tables. Since developers are likely interested in invariant violations rather than satisfactions, it might be useful to specify the negation of the actual invariant. For this, the developer implements the following function to return the desired SELECT statement as character string:

   ```
   char *libseal_check();
   ```

[17] https://sqlite.org/
[18] http://www.w3schools.com/sql/sql_create_table.asp
[19] http://www.w3schools.com/sql/sql_insert.asp
[20] http://www.w3schools.com/sql/sql_select.asp

In summary, the SecureCloud auditing service, called libseal, enables the detection of incompliant microservice behavior. Since libseal adheres to well-known APIs (OpenSSL and SQL), its integration with existing technologies and services can be easily achieved.

Section 4.1 describes a demonstrator that shows how the SecureCloud libseal auditing service is able to detect misbehaving SecureCloud microservices.

4 Demonstrator

We developed a demonstrator that showcases early results of the SecureCloud approach to dependability. The demonstrator runs the Git version control system[1] as a secure microservice using the SecureCloud framework. Git is commonly used by software developers for the management, versioning, and synchronisation of software source code; GitHub[2] and GitLab[3] are two famous centralized instantiations of Git.

Using Git as a secure microservice, the demonstrator shows how the SecureCloud libseal auditing service provides additional integrity protection, thereby allowing to detect misbehaving microservices. integrity guarantees

The remainder of this chapter provides a demonstrator walkthrough on the basis of a Vagrant[4] virtual machine image. Alternatively, if a Docker installation is already available on the reader's machine, the SecureCloud Docker images may be downloaded directly and used within the existing Docker installation.

Section 4.1 describes the SecureCloud auditing demonstrator.

4.1 The SecureCloud auditing service

This demonstrator presents how the SecureCloud auditing service, libseal, detects misbehaving service providers. The following description is divided into the following parts: Section 4.1.1 sets up the secure Git microservice; ...

This demonstrator involves three machines:

1. the host machine named `host`,
2. the vagrant virtual machine named `vagrant`, which is running on `host`, and
3. the Git microservice named `docker-git`, which is running within `vagrant`.

In the following, command prompts (#) will be prepended by the scheme `[username]@[machine]:[dir]`, indicating which user is issuing the command on which machine within which directory.

4.1.1 Running the secure Git microservice

1. Download both the Vagrant virtual machine configuration file (`Vagrantfile`) and the Docker image for the secure Git microservice (`libseal.tar`) into the same directory:

```
you@host:~# mkdir ~/newdir && cd ~/newdir
you@host:newdir# wget https://www.securecloudproject.eu/wp-content/uploads/
Vagrantfile
you@host:newdir# wget https://www.securecloudproject.eu/wp-content/uploads/
libseal.tar
```

2. Boot and initialise the downloaded virtual machine:

```
you@host:newdir# vagrant up
```

[1] `https://git-scm.com/`
[2] `https://github.com/`
[3] `https://about.gitlab.com/`
[4] `https://www.vagrantup.com/`

3. Log into the virtual machine via the SSH protocol. We enable X11 forwarding (flag `-X`) for a better looking user interface later on. As a result, you will end up with a command prompt inside the `vagrant` machine:

```
you@host:newdir# vagrant ssh -- -X
vagrant@vagrant:~#
```

4. Start the Git microservice. This microservice has already been provisioned within the virtual machine as a Docker image:

```
vagrant@vagrant:~# docker run -dt -p 7777:7777 -p 7778:7778 -p 2277:22
        libseal:flat /usr/sbin/sshd -D
```

   Upon issuing the above command, Docker will initiate a new running Docker container. This Docker container is configured to provide the following capabilities:

   - run an Httpd Apache Webserver[5] on ports 7777 (HTTTP) and 7778 (HTTPS),
   - run a Git-server behind the above Apache webserver, and
   - run an SSH server for further maintenance of the container.

   Concretely, the above `docker run` command achieves the following:

   - it runs an SSH daemon within the Docker container (`/usr/sbin/sshd -D`),
   - it maps the Docker container's port 7777 to the host's port 7777 (`-p 7777:7777`),
   - it maps the Docker container's port 7778 to the host's port 7778 (`-p 7778:7778`), and
   - it maps the Docker container's port 22 to the host's port 2277 (`-p 2277:22`).

5. Log into the Git microservice Docker container via SSH, enabling X11 forwarding; the root password is 'root':

```
vagrant@vagrant:~# ssh -p 2277 -X root@localhost
root@localhost's password:  root
root@docker-git:~#
```

6. Within the Git microservice Docker container, start the Apache webserver as follows:

```
root@docker-git:~# /libseal/start-demo.sh
```

   With this, the Git microservice is running. The SecureCloud framework takes care of auditing the service.

4.1.2 Using the secure Git microservice

Once the Git microservice was started, clients may use the service according to its provided interface. For Git, this interface consists of command line interface invocations that either push data from the client to the server, or that pull data from the server to the client. Please refer to the Git documentation[6] for further details.

[5] `https://httpd.apache.org/`
[6] `https://git-scm.com/documentation`

1. For the purpose of this demonstrator, the set of commands below will achieve the following:

   (i) Clone an empty repository from the server, thereby creating an exact copy at the client (`git clone ...`)

   (ii) Create a new text file at the client (`echo ...`)

   (iii) Add this new text file to the client's local repository (`git add ...`)

   (iv) Commit this new text file to the client's local repository (`git commit ...`)

   (v) Push the aforementioned commit, and hence the new text file, from the client to the server repository (`git push ...`)

   To achieve this, execute the following commands from within the `vagrant` machine. For this, we first need to get another command line prompt within this machine:

```
you@host:~# cd ~/newdir
you@host:newdir# vagrant ssh -- -X
vagrant@vagrant:~# git clone https://user:user@localhost:7778/git/git1.git
        copyA
vagrant@vagrant:~# cd copyA
vagrant@vagrant:~/copyA# echo "Demonstrating libseal" > new.txt
vagrant@vagrant:~/copyA# git add new.txt
vagrant@vagrant:~/copyA# git commit -m "Adding new.txt"
vagrant@vagrant:~/copyA# git push
```

2. After adding a first file to the server repository, we clone the server repository once more. This simulates the existence of another client and shows that other clients would actually receive the new text file that we added to the server repository just before. For this, the below commands will

   - Clone the same repository to another local folder (`git clone ...`)
   - Inspect that the new file exists and has the same content (`cat ...`)

```
vagrant@vagrant:~/copyA# cd ~
vagrant@vagrant:~# git clone https://user:user@localhost:7778/git/git1.git
        copyB
vagrant@vagrant:~# cd copyB
vagrant@vagrant:~/copyB# cat new.txt
Demonstrating libseal
vagrant@vagrant:~/copyB#
```

4.1.3 Auditing the secure Git microservice

While issuing the above commands, the SecureCloud libseal auditing service was recording the messages exchanged between the client and the server. We can inspect this log by getting another command line prompt into the `docker-git` machine and inspecting the sqlite database using the tool `sqlitebrowser`:

```
vagrant@vagrant:~/copyB# cd ~
vagrant@vagrant:~# ssh -p 2277 -X root@localhost
root@localhost's password:  root
root@docker-git:~# sqlitebrowser /libseal/git.sqlite
```

`sqlitebrowser` allows to inspect which updates were sent from the client to the server (table `updates`, Figure 4.1), as well as which updates were sent from the server to the client (table `advertisements`, Figure 4.2). Figures 4.1 and 4.2 show for this example: (i) a commit with id 'ac99c...' was made to repository '/git/git1.git/' on branch 'refs/heads/master' at time 25 (reflecting the initial `git push` to the previously empty repository; step 1 in Section 4.1.2); (ii) the same commit id of the same repository and the same branch was advertised to a client at time 27 (reflecting the second `git clone` of the non-empty repository; step 2 in Section 4.1.2).

`sqlitebrowser` allows to issue a manual audit of the above database tables: Figure 4.3 shows the execution of the SQL query `SELECT * FROM unsound`, which uses auxiliary table `unsound` to report unsound advertisements from the server to the client. As the query result in Figure 4.3, shows, there are currently no unsound advertisements to report. Note that this check is performed regularly, frequently, and automatically by the libseal auditing service.

4.1.4 Auditing the Secure Git microservice after malfunctioning

To simulate malfunctioing of the Git microservice, we use `sqlitebrowser` to modify the commit id of an entry within the `advertisements` table. For the purpose of this demonstrator, we change the commit id of branch 'refs/heads/master' of repository '/git/git1.git/' at time 27 from 'ac99c...' to 'bc99c...' (see Figure 4.4). In the real world, this would mean that the server advertised a wrong commit id, and consequently wrong data, to one of the clients.

Once libseal performs another audit of the database by executing SQL query `SELECT * FROM unsound`, it will detect the advertisement of the wrong commit id. Figure 4.5 shows the execution of the SQL query. The query result shows that the modification of was indeed detected, i.e., that there was a wrong advertisement at time 27.

4.1.5 Summary

This demonstrator has shown how the SecureCloud libseal auditing service is able to detect misbehaving SecureCloud microservices. While in the demonstrator the auditing was performed manually for demonstration purposes, libseal performs these checks automatically at regular and configurable time intervals.

4.2 The SecureCloud attestation facilities

This demonstration shows our attestation mechanism in action. The demonstration involves three Docker containers: i) a container running a *configuration and attestation service (CAS)*, ii) a container hosting the platform local LAS, and iii) a container running the example program. The example program will get its configuration, i.e., program arguments, environment variables and its working directory, from CAS after it has attested itself successfully and is allowed to read the requested configuration. This demo

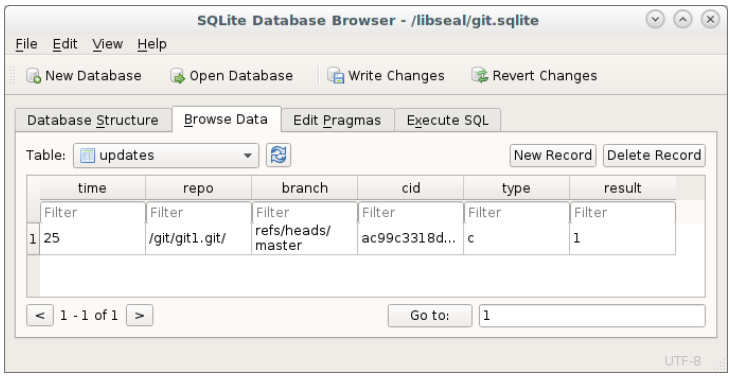

Figure 4.1: libseal record of updates to the Git server.

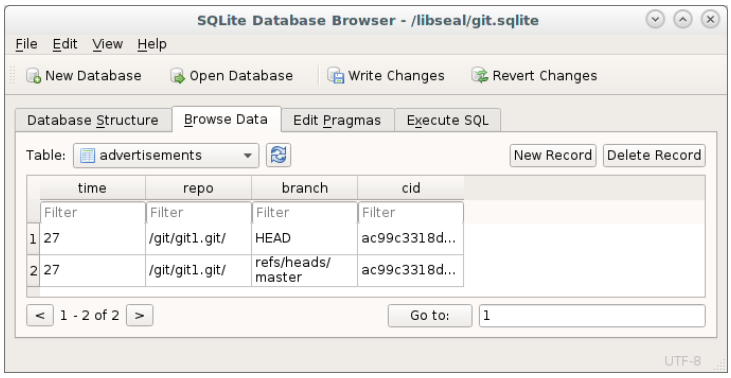

Figure 4.2: libseal record of updates to the clients.

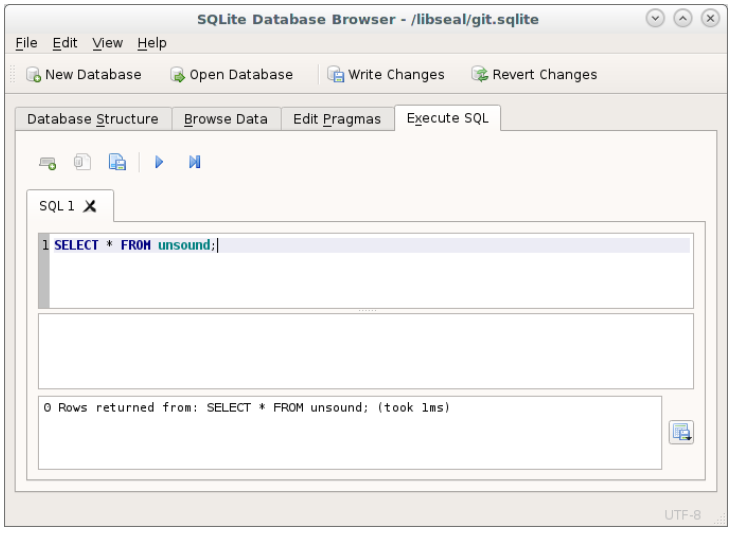

Figure 4.3: libseal manual audit.

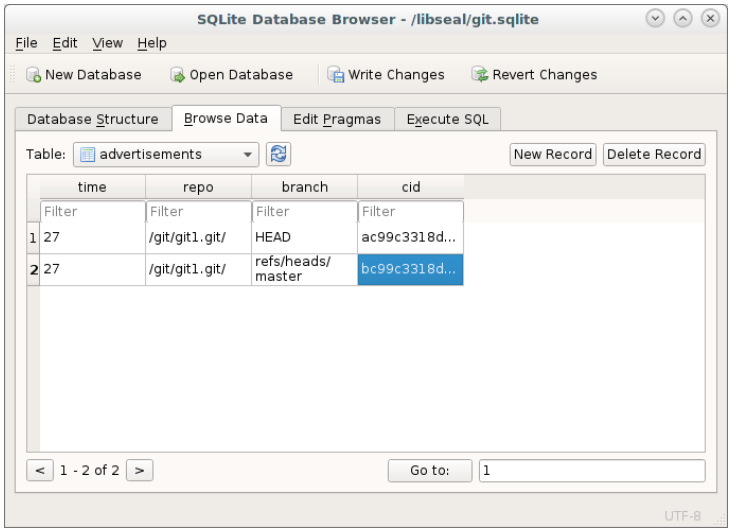

Figure 4.4: Modifying the database to advertise a wrong commit id.

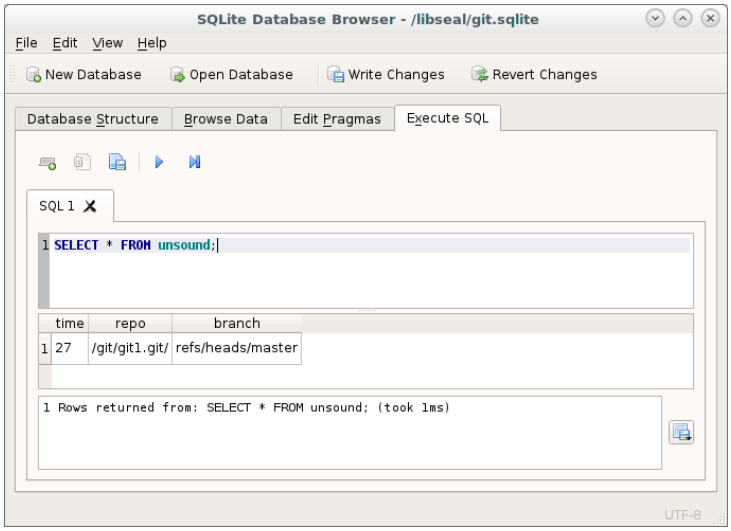

Figure 4.5: libseal reporting a wrong advertisement.

makes use of a prototype of the previously described SecureCloud attestation facilities which is currently limited to using Intel's remote attestation scheme to build trust. CAS utilizes the Rust verifier library to attest the actual enclave before sending the configuration through the secure communication channel that remains usable after a successful attestation. For the sake of clarity, the example program just produces the received configuration on the screen. We compiled it with a modified version of the SCONE compiler that we fitted with the EAI implementation. A program created with this compiler will execute a special initialization procedure tries to connect to a CAS instance, conduct remote attestation and obtain and apply the configuration before the programs main function is execute. Note that, since with the SCONE compiler the whole program is executed inside of an enclave, the terms enclave and program become interchangeable.

4.2.1 Demonstration Requirements

To run the demonstration the following requirements have to be meet:

1. the user has to be registered with Intel's Attestation Service [7] having access to his certificate and private key,
2. a machine with SGX support with a Linux operating system, equipped with Intel's SGX driver version 1.7 [8] and a docker installation, and
3. access to the SecureCloud docker repository.

4.2.2 Setting up Infrastructure

All containers are automatically downloaded by Docker from the SecureCloud repository. This demo makes use of the SGX functionality of the CPU. Since SGX virtualization is currently not commonly available in off-the-shelf VM hypervisors, the demo is executed on a bare-metal host using Docker to isolate the components.

The necessary infrastructure consist of the platform local LAS and the CAS, which might also be located on a different machine.

1. Start the LAS by executing the following command on the host machine:

   ```
   $ docker run --name las -it --device /dev/isgx
       docker.securecloud.works/d3.1_attestation_demo:las
   ```

   We have to mount the SGX-device file inside of the container because LAS needs access to SGX functionality to create quotes. A successful start of the container will generate an output like this:

   ```
   /dev/isgx available! Starting Intel AESM service!
   aesm_service[9]: The server sock is 0x12e5360
   LAS is listening on 0.0.0.0:18766
   ```

   The LAS is now running and waiting for requests from the enclave.

[7] https://software.intel.com/formfill/sgx-onboarding
[8] https://github.com/01org/linux-sgx-driver/

2. In a second terminal, start CAS. To allow it to authenticate itself with IAS, you have to bind the IAS client certificate you are registered with and its private key into the container. Assuming these reside in the current working directory with the names `ias_client_cert.pem` respectively `ias_client_key.pem`, you have to execute this statement:

```
$ docker run --name cas -it
    -v ias_client_cert.pem:/root/ias_client_cert.pem
    -v ias_client_key.pem:/root/ias_client_key.pem
    docker.securecloud.works/d3.1_attestation_demo:cas
```

   Before CAS can be invoked, your IAS service provider ID and quote linkability setting have to be set in the configuration file `/root/config.json`. After that we start CAS by switching into `root`'s home directory and executing the `rust_cas` executable:

```
root@c4b80f333a87:/# vim /root/config.json
root@c4b80f333a87:/# cat /root/config.json
{
"spid" : [ 46, 197, 190, 100, 226,  66, 188, 201,
          228, 103,  13,  73, 229, 199,   9,  30],
"linkable" : true,
"configurations" : [
        {
            "id" : 12345678,
            "mrenclave" : [],
            "args" : ["postres", "-d", "--password", "secret"],
            "environ" : [
                    ["PGHOST", "postgres.securecloud.works"],
                    ["PGPASSWORD", "01234567"]
                ],
            "pwd" : "/home/enclave"
        }
    ]
}
root@c4b80f333a87:/# cd /root/
root@c4b80f333a87:~# ./rust_cas
```

   Note that there is a existing example configuration with the ID 1234567 and an empty list of enclaves allowed to access it. The CAS is now waiting for enclave requests.

3. Finish the infrastructure setup by obtaining the IP addresses of the started LAS and CAS instance. To do so, open a new terminal and execute the following commands:

```
$ docker inspect --format '{{ .NetworkSettings.IPAddress }}' cas
172.17.0.9
$ docker inspect --format '{{ .NetworkSettings.IPAddress }}' las
172.17.0.8
```

   In this case CAS is located at `172.17.0.9` and LAS at `172.17.0.8`. This concludes the infrastructure setup.

4.2.3 Starting the Example Program

The program is deployed via a Docker image as well. Therefore, you have to start by invoking the container before you can execute the program.

1. Start the enclave's container by executing:

   ```
   $ docker run --name enclave -it --device /dev/isgx
       docker.securecloud.works/d3.1_attestation_demo:enclave
   ```

   The SGX-device file has to be available in the container to allow SCONE's starter code to create the enclave program in which it will execute.

2. The addresses of LAS and CAS have to be known by the initialization procedure for it to contact them. Provide addresses via environment variables before invoking the executable located at `/root/enclave`. Furthermore, the procedure has to be informed which configuration it should request, which is also done with an environment variable:

   ```
   root@1c425db67505:/# export MUSL_LAS_ADDR=172.17.0.8
   root@1c425db67505:/# export MUSL_CAS_ADDR=172.17.0.9
   root@1c425db67505:/# MUSL_CONFIG_ID=12345678 /root/enclave
   ```

   Now the actual attestation and configuration procedure takes place unseen. After the invocation the enclave program's initialization routine contacts CAS to obtain the indicated configuration. CAS answers this request with a quote request to the enclave. To serve this request the enclave communicates with LAS and send the obtained quote back to CAS. Verifying the quote CAS queries IAS for its authenticity. Only after a successful outcome CAS will ensure that the enclave is allowed to obtain the requested configuration, provide it if that is the case and terminate the connection.

   Remember that previously configuration 12345678 did not list any enclave allowed to obtain it. Therefore, CAS will reject the enclave's request halting its initialization routine. In CAS's terminal window you will see a log message comparable to this one:

   ```
   Requesting enclave (MRENCLAVE: [136, 198, [...] 140, 246]) is
   not permitted to access requested configuration (ID: 12345678)!
   ```

3. Fix this issue by adding the enclave hash given in the previous message into the configuration database. Terminate the running CAS instance with `Ctrl + C`, edit the configuration file and restarting CAS afterwards:

   ```
   root@c4b80f333a87:~# vim config.json
   root@c4b80f333a87:~# cat config.json
   {
   "spid" : [ 46, 197, 190, 100, 226,  66, 188, 201,
              228, 103,  13,  73, 229, 199,   9,  30],
   "linkable" : true,
   "configurations" : [
           {
               "id" : 12345678,
               "mrenclave" : [[136, 198, 253, 159, 211, 157, 232, 170,
                               39, 253, 124,  90,  76, 143, 150,  65,
   ```

```
                                201,  73, 177,  48,  97,  68, 230, 155,
                                46,  27, 174, 133,  29, 112, 140, 246]],
            "args" : ["postres", "-d", "--password", "secret"],
            "environ" : [
                    ["PGHOST", "postgres.securecloud.works"],
                    ["PGPASSWORD", "01234567"]
                ],
            "pwd" : "/home/enclave"
        }
    ]
}
root@c4b80f333a87:~# ./rust_cas
```

4. After the enclave's hash has been added to the list of allowed enclaves CAS will provision the configuration. The enclave program is able to start producing the configuration previously known from the configuration database:

```
root@1c425db67505:/# MUSL_CONFIG_ID=12345678 /root/enclave
4 program arguments:
0: postres
1: -d
2: --password
3: secret
environ: PGHOST=postgres.securecloud.works
environ: PGPASSWORD=01234567
pwd: /home/enclave
```

4.2.4 Summary

This concludes the attestation demonstration. This demonstration showed how the SecureCloud attestation facilities can be used to securely provision confidential configuration data to previously unknown enclaves.

5 Summary

The SecureCloud framework provides mechanisms for the secure operation of microservice-based big data applications. In this Demonstrator-Deliverable we described the specification and a first implementation of the SecureCloud microservice framework and the corresponding API.

The framework is comprised of (i) a container-based scheduling and orchestration framework that integrates with secure microservices as developed within SecureCloud; (ii) a SecureCloud attestation service that allows verifiers to build trust into previously unknown microservices; (iii) a SecureCloud auditing service that allows to detect misbehaving microservices. We described the API of the aforementioned services. First prototypical implementations of these services show the feasibility of the proposed architecture and API.