Rest Assured

Secure Data Processing in the Cloud

Deliverable D4.1
Conceptual Foundation of the RestAssured Secured Enclave
Release 1.0

The research leading to these results has received funding from the European Community’s H2020 research and innovation programme under grant agreement n° 731678.
<table>
<thead>
<tr>
<th>Project Number</th>
<th>731678</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project Title</td>
<td>RestAssured - Secure Data Processing in the Cloud</td>
</tr>
<tr>
<td>Deliverable Type</td>
<td>External Report</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Title of Deliverable</th>
<th>Conceptual Foundation of the RestAssured Secured Enclave</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nature of Deliverable</td>
<td>External Report</td>
</tr>
<tr>
<td>Dissemination Level</td>
<td>Public</td>
</tr>
<tr>
<td>Contractual Delivery Date</td>
<td>30 April 2018</td>
</tr>
<tr>
<td>Actual Delivery Date</td>
<td>30 April 2018</td>
</tr>
<tr>
<td>Contributing WPs</td>
<td>All</td>
</tr>
<tr>
<td>Editor(s)</td>
<td>Oshrit Feder, Eliot Salant</td>
</tr>
</tbody>
</table>
## Contributors

<table>
<thead>
<tr>
<th>Name</th>
<th>Organization</th>
<th>Sections</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oshrit Feder</td>
<td>IBM</td>
<td>Overall</td>
</tr>
<tr>
<td>Gidon Gershinsky</td>
<td>IBM</td>
<td>Overall</td>
</tr>
<tr>
<td>Ronen Kat</td>
<td>IBM</td>
<td>Various fixes</td>
</tr>
</tbody>
</table>
## Document History

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1</td>
<td>March 2018</td>
<td>Initial version</td>
</tr>
<tr>
<td>V1</td>
<td>April 2018</td>
<td>Peer review</td>
</tr>
</tbody>
</table>
Contents

List of Figures ........................................................................ 7
List of Tables ......................................................................... 8
1 Introduction ...................................................................... 9
  1.1 Motivation . ................................................................. 10
  1.2 Problem Definition ..................................................... 10
2 Conceptual Foundation of the RestAssured Secured Enclave ............ 12
  2.1 Introduction . ................................................................. 12
  2.2 Trust Management Framework Design ............................... 12
  2.3 The TruCE Framework Implementation ......................... 16
    2.3.1 TruCE Framework API and Configuration ............... 16
    2.3.2 Untrusted application API ....................................... 16
    2.3.3 Client API .............................................................. 18
3 Spark SQL with hardware enclaves .......................................... 20
  3.1 Introduction . ................................................................. 20
  3.2 RestAssured infrastructure with Opaque ............................. 20
    3.2.1 Implementation ...................................................... 20
  3.3 Query Gateway ............................................................. 22
    3.3.1 Implementation ...................................................... 22
4 Performance evaluation ........................................................ 24
  4.1 Introduction . ................................................................. 24
  4.2 Evaluation .................................................................... 24
    4.2.1 Testing a simple function ....................................... 24
    4.2.2 Testing SHA256 ..................................................... 25
    4.2.3 Testing AES Encryption ........................................... 26
    4.2.4 Opaque vs native Spark ......................................... 28
5 Data Access Protection in Usecases ........................................... 30
  5.1 Environment Settings ..................................................... 30
  5.2 Query Gateway ............................................................. 30
  5.3 Secured Data Processing ............................................... 31
  5.4 Scant Prototype ............................................................. 32
5.5 Gatekeeper Integration ................................................................. 33
5.6 Audit Log .................................................................................. 33
List of Figures

Figure 1.1: Secured data processing in RestAssured architecture .................................................. 9

Figure 2.1: Application attack surface ......................................................................................... 13

Figure 2.2: The Truce platform and SDK for the different parts of the application (trusted and untrusted parts of the code as well as the client application) .................................................. 14

Figure 2.3: TruCE Framework deployment and libraries interactions ............................................. 15

Figure 3.1: Secured processing infrastructure components and API ............................................. 22

Figure 4.1: Find Max Throughput ............................................................................................... 25

Figure 4.2: SHA256 Throughput ................................................................................................. 26

Figure 4.3: AES128-GCM Encryption Throughput ........................................................................ 27

Figure 4.4: AES128-GCM Encryption Throughput in Enclave Memory ........................................ 27

Figure 4.5: AES128-GCM Encryption Throughput ......................................................................... 28

Figure 4.6: AES128-GCM Encryption Throughput in Enclave Memory ........................................ 29

Figure 5.1: Secured data processing query using Opaque and Truce .............................................. 31

Figure 5.2: Memory dump of secured data processing ................................................................. 31

Figure 5.3: Benchmark - non secured query processing ............................................................... 32

Figure 5.4: Memory dump leaking sensitive information ............................................................. 32

Figure 5.5: Prototype RestAssured infrastructure ......................................................................... 33
List of Tables

Table 2.1: Example truce.config file ................................................................. 16
Table 3.1: Query Gateway API ................................................................. 23
Table 3.2: Query Gateway configuration .................................................. 23
1 Introduction

In this work package report we focus on the design, implementation and performance aspects of the secured processing in the RestAssured platform which aims to significantly reduce the ability of rogue employees or hackers to extract sensitive data from RestAssured applications. Our approach is to provide not only data-at-rest protection, but also data-in-processing (with memory) protection. The report is organized as follows: First we discuss the overall architecture overview, motivation and define the problem we faced. Later we break down each of the tasks and methods we have worked on to provide a secured processing framework solution in more details including technical information on the implementation.

Figure 1.1: Secured data processing in RestAssured architecture

Work package 4 is the data processing entity in RestAssured platform, as indicated in report 3.2 and marked in red in Figure 1.1 ensuring secure processing of sensitive personal information.

The protection of sensitive data is a fundamental security concern for any organization retaining and utilizing such data. Numerous mechanisms to protect sensitive data exist, including encryption of data, access control mechanisms and virtual or physical data isolation.

A key weakness of such mechanisms is their reliance on a centralized entity that is entrusted with the management of the IT system, its health, scalability and availability. Typically, IT administrators will have nearly unlimited access to data or have the privileges to bypass or change the software that manages the data. In addition, the existence of powerful privileged users in a system leaves an opening for foul play, whether by a rogue employee or by hackers.

History has shown that such trust can be breached, or compromised, whether maliciously or not. Incidents of data theft by authorized persons have been abundant, with some high profile data breaches such as the Snowden leaks. Data leaks continue to plague various organizations, be it by a disgruntled employee, inadvertent error, or by hacking of accounts of individuals that have the credentials to override the security obstacles instated for data protection. Finally, many of the defense mechanisms are implemented in software and can be bypassed by administrators with direct access to the underlying hardware.
1.1 Motivation

Protecting sensitive business and personal information is a central requirement when organizations move their data storage and processing to the cloud. Many aspects of this requirement are already handled at various levels. Data-at-rest can be secured in cloud stores by encrypting it before storage, while data-in-flight is transmitted on protected channels such as TLS and HTTPS. Data-in-use, processed in cloud compute nodes, is the most vulnerable link in the end-to-end information flow chain, since process memory can be accessed by malicious, privileged software or system administrators. In this Work Package we aim to provide trusted cloud enclaves that enable secure data processing in the cloud, protecting the sensitive data not only from other cloud tenants, but also from cloud infrastructure and administration. The result is an end-to-end protection of information flow across public and private domains, with data-at-rest, data-in-flight and data-in-use never exposed.

1.2 Problem Definition

- **Secure analytics on encrypted data** - Sensitive data must be encrypted before delivery to public clouds for storage and subsequent processing. The encryption can be performed at the data source, such as sensors and end-user clients, or in a trusted private environment that collects the data from the sources on secure channels. Sometimes, the sensitive data is produced locally inside private domains, such as company private clouds or other on-premise data centers. In either case, the data is protected with a strong encryption before being sent to the public cloud. This poses a problem for analytic engines such as Apache Spark, that cannot work directly with encrypted data. These engines rely on storage systems to decrypt the data before sending it for processing. This means that the data encryption keys and the data itself are visible to the storage infrastructure, making the sensitive information vulnerable to attacks from malicious software or administration. Moreover, the analytic engine itself is not sufficiently protected in a public cloud, as described in the previous section. While isolation technologies such as VMs, containers or VPNs, can be used to protect the engine against access by other cloud tenants, the engine process memory is not encrypted, and therefore accessible by privileged software, host hypervisors and system administrators.

- **Trust establishment in hardware enclaves** - Intel SGX (Software Guard Extension) technology allows to create memory regions (enclaves) protected with hardware encryption in the SoC (system on chip). The data is opened only inside the processor. It is encrypted in SoC before leaving to the main memory, and decrypted in SoC upon fetching from the main memory. Paging is done on encrypted data. This is a powerful security tool, but it is highly challenging for usage in practice. One of the main challenges is that a complex trust establishment mechanism is required to verify the CPU and the application binary running in the enclave. Without it, the user can not send secrets (such as data encryption keys) to the enclave, because the CPU can be substituted by a fake processor that leaks the secrets to a malicious party, or the application binary can be replaced by malicious code that leaks the data even in a genuine SGX CPU. Intel specifies a procedure for such verification, called *remote attestation*, but unfortunately, does not provide an end-to-end mechanism that implements this procedure. Instead, the SGX SDK contains a collection of API calls that can be utilized by an expert developer to create such mechanism. In practice, it is a highly challenging task even for experienced hands-on developers who find it hard to understand the concepts and security model behind the remote attestation, making the implementation not only complex, but also prone to trust breach flaws. Besides implementation, the procedure requires registration with Intel for use of its attestation service (IAS).

- **Interaction with policy and adaptation components** - Another challenge in protecting sensitive data in public clouds is efficient access control. Even if the data is encrypted, an unauthorized user who managed to get access by acquiring the key or by gaining trust from analytic engines that have the
key, is breaking the security boundaries. The interaction between secure analytic systems and access control policy management must be carefully designed and implemented. The analytic systems also interact with adaptation components that manage runtime changes in the infrastructure and in data flows, with a focus on performance/availability and security aspects.

- **Platform interface: applications and data ingestion** - A cloud data processing platform needs a well-defined and secure interface for interaction with off-cloud components, such as use case applications and data sources. The intermediate gateways, running in a trusted environment, should be able to receive a definition of the incoming data structure, encrypt the ingested data for subsequent storage in the public cloud, and manage the encryption keys. The gateways should create secure endpoints that can accept data queries from applications, verify the queries by interaction with a policy engine, execute the queries and send the results back to application on a secure channel.
2 Conceptual Foundation of the RestAssured Secured Enclave

2.1 Introduction

Intel SGX in Linux initially became made available to the general public in September 2016 and has steady evolved over the last year (eight new releases since the initial SGX launch). RestAssured had access to SGX hardware from the start of the project, and started by conducting several investigations and explorations of the technology and its applicability to various scenarios, in particular those inspired by the project use cases. This has been followed by the designing and developing of SGX framework building blocks to enable hardware security for RestAssured computation infrastructure.

Intel SGX overview

Intel Software Guard Extensions ®(SGX) is a new set of instructions on Intel 6th Generation Intel® Core processor (X86 Skylake or later) which aims to allow application developers to run parts of their code in a protected, isolated fashion. These instructions create execution environments, called Enclaves, which run in secluded memory areas and use hardware based encryption to hide all memory used by the enclave process, effectively shielding data stored in the memory from external observers and minimize the attack surface Figure 2.1. No other process on the same processor, not even the operating system, hypervisor, administrator or system management module, can access the memory that is associated with the enclave.

Enclaves are signed at build time. Signing an enclave is a process that involves producing a signature structure that contains enclave properties such as the enclave measurement and signer identity. Enclaves are verified using attestation, a process that verifies the platform (i.e., genuine Intel CPU, and not a simulated platform) and the enclave code. If the enclave’s content and layout do not match the signed enclave measurement, enclave attestation will fail and it will not be trusted with application secrets.

Enclave sealing provides secure persistence for the secret or secrets needed by the enclave in order to save and restore its state in the event of a power transition, an application upgrade, or between application sessions. Efficient sealing can save the overhead of remote attestation starting with the second execution of the enclave code.

Intel provides SDK for developing SGX enclaves in C/C++ language. Recently a new cryptographic library was published, called Intel SGX SSL, that provides OpenSSL 1.1.0 functionality built to run inside an enclave.

An application that aims to use the Intel Software Guard Extensions needs to be divided into two logical components:

- A Trusted component, where the code that accesses protected data resides. This component is also called an enclave. More than one enclave can exist in an application.
- Untrusted component. The rest of the application including all modules that are not expected to perform direct computation on the protected data. The application developer should make the trusted part as small as possible. It is suggested that enclave functionality should be limited to that which performs direct operations on the protected data to minimize the attack surface.

2.2 Trust Management Framework Design

To address the Trust establishment in hardware enclaves challenge described in the problem definition section above, we designed and developed a framework for trust management in SGX enclaves, called TruCE (Trust in Cloud Enclaves) for trust and easy-to-use verification of computation enclaves. TruCE framework
Figure 2.1: Application attack surface

provides services and APIs to use secure enclaves, and to enable RestAssured application developers to focus on the application logic. The framework includes:

- Platform services and toolkit to simplify the use of enclaves for sensitive parts of the application by handling the remote attestation and the secret delivery process to Intel SGX enclaves with simple API calls.
- A reference implementation for RestAssured enclave development
- Performance focused implementation with benchmarks experiments

As described in Figure 2.2, our trust management framework contains several building blocks to construct an effective infrastructure with sensitive application code executing within enclaves.

- Attestation service - The first challenge encountered with enclave development is the need to guarantee that the code running in the enclave has not been tampered with and to verify the platform it is launching on is a genuine SGX CPU. This mechanism ensures that an application does not send its secrets to a malicious enclave pretending to be a trusted one. The remote attestation is a critical component that employs a complex protocol using Intel SGX SDK. Currently, Intel only provides a reference implementation example of how to attest enclaves upon initialization. TruCE enriches the secured processing in RestAssured infrastructure with a full functioning attestation framework that handles the enclave verification and the required interaction with the Intel Attestation Service (IAS). Each RestAssured enclave is attested upon creation, assuring the enclave can be trusted by the application, using the TruCE server (also referred as service provider) and the toolkit API. Initially, we followed Intel’s reference implementation, [https://software.intel.com/en-us/articles/intel-software-guard-extensions-remote-attestation-end-to-end-example#sample-code](https://software.intel.com/en-us/articles/intel-software-guard-extensions-remote-attestation-end-to-end-example#sample-code), which uses sigma protocol [https://en.wikipedia.org/wiki/Proof_of_knowledge#Sigma_protocols](https://en.wikipedia.org/wiki/Proof_of_knowledge#Sigma_protocols) with several messages exchanged until a signed enclave report is generated. Later we introduced a novel attestation mechanism to replace Intel’s sample service provider. In our new attestation mechanism the enclave keys are generated inside the trusted enclave code using Intel SGX SSL library. The generation of keys inside the enclave, enables us to include the public key in the enclave quote which is signed by Intel and therefore the resulting...
attestation report can be kept in an untrusted storage, reducing the trust requirements placed on TruCE server.

Once an enclave has been remotely attested, we use the SGX sealing mechanism to seal its private key locally on the host. Each time an enclave is created, the TruCE attestation toolkit first checks if a seal file exists for the enclave, and if so, the enclave key is restored from the sealed file. This eliminates the need to perform remote attestation again for the same enclave on the same hardware and results in better enclave creation performance.

- Key passing toolkit to securely send data keys to an enclave.

TruCE framework ([https://github.com/occsysadmin/RestAssured-Code.git](https://github.com/occsysadmin/RestAssured-Code.git)) contains several components:

- TruCE server: A standalone process that registers with Intel Attestation Service and assists in remote attestation of RestAssured platform enclaves. Can also be seen as a cloud service, an entity that communicate with the Intel Attestation Service (IAS) and stores all IAS reports of the applications’ SGX enclaves. In order to run the code in real IAS mode, we followed Intel’s procedure (see [https://software.intel.com/en-us/articles/intel-software-guard-extensions-remote-attestation-end-to-end-example#remote-attestation-flow](https://software.intel.com/en-us/articles/intel-software-guard-extensions-remote-attestation-end-to-end-example#remote-attestation-flow) to acquire a service provider ID (SPID). After the registration with a certificate (currently we work with self-signed certificates for development purposes), Intel responded with an SPID. All those values are configurable at defs.h (SPID, certificate and the quote signing type).

- TruCE SDK: A toolkit for application development, containing API and libraries for trusted (enclave) part of the cloud application, untrusted part of the cloud application, and the off-cloud client code that interacts with the cloud application.

  - Applications - entities that contain SGX enclaves
    - Untrusted API: app/truce_u.h
    - Untrusted Library: libtruce_u.so module
    - Reference implementation: truce_app/app.cpp
Figure 2.3: TruCE Framework deployment and libraries interactions

- Enclaves secured application entities
  - Trusted API: enclave/truce_t.h
  - Trusted Library: libtruce_t.a

- Clients - entities that need to attest an application’s SGX enclave and trust it with secret data
  - Client API: client/truce_client.h
  - Client Library: client/libtruce_client.so
  - Reference implementation: client/client.cpp

Using TruCE services and SDK: Applications register with TruCE server for enclave attestation by creating a truce_session, that sends enclave "attestation quote" to the server. The TruCE server uses the quote to prepare an attestation report and sends the report to Intel Attestation Service, which verifies the report, signs it with the Intel key and returns it the TruCE server. The verification will

1. Attest that the report quote was indeed created by a genuine Intel SGX CPU.
2. Attest the code being executed by the enclave.

The report also contains a hash of the enclave’s unique public key, later used as a means for connecting and communicating with the enclave. Given an attestation report, one can verify it off-line, and use the public key to form a secure channel with the corresponding enclave. This step can be repeated by many clients without a need to reconnect to the IAS. Any client that wants to attest an SGX enclave and communicate with it, queries the TruCE server for the enclave’s record using the truce_client_recv_enclave_record API, and verifies the attestation report using the truce_client_verify_enclave_record call which makes sure the report is signed by the Intel attestation public key. The client also has to compare the enclave and signer measurements, embedded in the verified report, with the expected values. Once the enclave is attested, the client can retrieve the enclave public key and use it to encrypt secrets (such as data keys) with the truce_client_encrypt_secret call. Then the secrets are sent to the enclave for subsequent decryption and processing of sensitive data.

TruCE is implemented with Intel SGX SDK version 2.0 and Intel SGXSSL 2.0
2.3 The TruCE Framework Implementation

2.3.1 TruCE Framework API and Configuration

The TruCE server uses a configuration file truce.config for settings such as service ports, Intel Attestation Service URL, certificate and SPID. The following table is an example of the configuration used in our first demo implementation:

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP.CERT</td>
<td>&quot;cert_and_key.pem&quot;</td>
</tr>
<tr>
<td>SPID</td>
<td>0x00,0x86,0x48,0x6F,0x9A,0xB2,0x18,0xC8,0xA9,0x8E,0x8F,0x3C,0x91,0x7A,0xF1,0xF5</td>
</tr>
<tr>
<td>QUOTE_SIGN_TYPE</td>
<td>SGX_LINKABLE_SIGNATURE</td>
</tr>
<tr>
<td>SP_AS_PORT</td>
<td>4000</td>
</tr>
<tr>
<td>SP_CS_PORT</td>
<td>5000</td>
</tr>
<tr>
<td>APP_PORT</td>
<td>6000</td>
</tr>
</tbody>
</table>

2.3.2 Untrusted application API

Type: truce_session_t
Description: TruCE session identifier
Members:

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sgx_enclave_id_t</td>
<td>Enclave identification</td>
</tr>
<tr>
<td>enclave_id</td>
<td>TruCE identification and computation of SHA256 on enclave public key</td>
</tr>
<tr>
<td>uint8_t truce_id[SHA256_DIGEST_LENGTH]</td>
<td></td>
</tr>
</tbody>
</table>

Type: truce_config_t
Description: Configuration settings for TruCE setup
Members:

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>char *truce_server_address</td>
<td>IP address of TruCE attestation server</td>
</tr>
<tr>
<td>char *seal_path</td>
<td>Sealing file name, Enclave’s keys will be stored and restored to this file using sgx sdk. If the Seal file is not specified new keys will be generated</td>
</tr>
</tbody>
</table>

Methods:

```c
bool truce_session(sgx_enclave_id_t enclave_id,
                        const truce_config_t &config,
                        truce_session_t &truce_session)
```

Description: Create TruCE session: Upon first invocation, generate private/public keypair in the enclave, register with TruCE server, perform remote attestation, and optionally seal the enclave private key. Upon later invocations, retrieve the private key from seal file if available and or repeat the full session creation.
procedure. To be invoked following an Intel SGX SDK `sgx_create_enclave` call.

Members:

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>enclave_id</td>
<td>IN</td>
<td>Enclave identification to attest</td>
</tr>
<tr>
<td>Config</td>
<td>IN</td>
<td>TruCE server to use for attestation, Path of seal file</td>
</tr>
<tr>
<td>Truce_session</td>
<td>OUT</td>
<td>TruCE identification in case of successful enclave attestation</td>
</tr>
</tbody>
</table>

Results: Returns whether the enclave was successfully attested, and its report stored in TruCE records. In case of unsuccessful result, applications should exit and not pass secrets to the enclave as it might be compromised.

```cpp
bool truce_add_secret(const truce_session_t &t_session, const uint8_t* secret_buf, uint32_t secret_buf_size)
```

Description: Store and Process encrypted buffer inside a given enclave

Members:

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>t_session</td>
<td>IN</td>
<td>TruCE session identifier</td>
</tr>
<tr>
<td>Secret_buf</td>
<td>IN</td>
<td>Encrypted buffer to be passed to the session enclave</td>
</tr>
<tr>
<td>Secret_buf_size</td>
<td>IN</td>
<td>Size in bytes of secret_buf value</td>
</tr>
</tbody>
</table>

Results: Returns whether the encrypted buffer was passed to the enclave

2.3.2.1 Trusted application API

Type: `truce_secret_t`

Description: List of sensitive data store

Members:

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned char *secret</td>
<td>Plain text buffer data</td>
</tr>
<tr>
<td>int secret_len</td>
<td>Size in bytes of secret store buffer</td>
</tr>
<tr>
<td>_truce_secret *next</td>
<td>Next secret store buffer</td>
</tr>
</tbody>
</table>

Methods:

```cpp
truce_secret_t *truce_get_secrets();
```

Description: Retrieve list of secrets, stored in the enclave
2.3.3 Client API

bool truce_client_recv_enclave_record(const char* truce_server_address, const truce_id_t&t_id, truce_record_t &t_rec)

Description: Connects to a given TruCE server to get an enclave record containing: Enclave’s report Enclave’s signature Enclave’s public key

TruCE server keeps all attested enclave records. Members:

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>truce_server_address</td>
<td>IN</td>
<td>TruCE server address to use</td>
</tr>
<tr>
<td>t_id</td>
<td>IN</td>
<td>TruCE identification of the enclave record</td>
</tr>
<tr>
<td>t_rec</td>
<td>OUT</td>
<td>TruCE enclave record</td>
</tr>
</tbody>
</table>

Results: Returns whether TruCE record was found in TruCE server

bool truce_client_extract_quote_from_record( const truce_record_t&t_rec, sgx_quote_t &quote)

Description: Returns enclave’s report for a given enclave

Members:

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>t_rec</td>
<td>IN</td>
<td>TruCE enclave record</td>
</tr>
<tr>
<td>quote</td>
<td>OUT</td>
<td>Enclave’s quote</td>
</tr>
</tbody>
</table>

Results: Returns whether the quote was successfully extracted from the given enclave TruCE record.

bool truce_client_verify_enclave_record(
const truce_id_t&t_id,
const truce_record_t&t_rec,
const sgx_measurement_t &expected_mrenclave,
const sgx_measurement_t &expected_mrsigner)

Description: Compare TruCE enclave record with values recorded during enclave build time

Members:

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>t_id</td>
<td>IN</td>
<td>TruCE identification of the enclave record</td>
</tr>
<tr>
<td>t_rec</td>
<td>IN</td>
<td>TruCEs enclave record</td>
</tr>
<tr>
<td>expected_mrenclave</td>
<td>IN</td>
<td>Actual enclave code measurement</td>
</tr>
<tr>
<td>expected_mrsigner</td>
<td>IN</td>
<td>Actual enclave signer measurement</td>
</tr>
</tbody>
</table>

Results: Returns whether the t_id matches the Truce enclave record and the enclave report was successfully verified to match the given mrenclave and mrsigner

bool truce_client_encrypt_secret(
const truce_record_t&t_rec,
const uint8_t *secret,
uint32_t secret_len,
uint8_t *output,
        uint32_t &output_size)
  Description: Encrypts the secret using the enclave’s public key
  Members:

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>t_rec</td>
<td>IN</td>
<td>TruCe’s enclave record</td>
</tr>
<tr>
<td>secret</td>
<td>IN</td>
<td>Buffer to encrypt</td>
</tr>
<tr>
<td>secret_len</td>
<td>IN</td>
<td>Secret buffer length</td>
</tr>
<tr>
<td>output</td>
<td>OUT</td>
<td>Encrypted buffer</td>
</tr>
<tr>
<td>Output_size</td>
<td>OUT</td>
<td>Encrypted buffer length</td>
</tr>
</tbody>
</table>

Results: Returns whether the buffer was successfully encrypted
3 Spark SQL with hardware enclaves

3.1 Introduction

We started with the goal of integrating RestAssured use cases with Intel SGX using the TruCE framework, but it quickly became apparent that integrating the code of the RestAssured use cases with SGX enclaves is not trivial. While the TruCE framework made creating enclaves and verifying them easy thanks to TruCE API and the reference implementation - the use cases applications are web based and refactoring them to fit into enclaves required a skilled C++ developer, following by cross technology efforts to interface and communicate with the enclaves from the original application (non C++ code) as currently only C/C++ is supported within SGX enclaves. In addition, education and experience with SGX technology is a must for complying with SGX limitations and framework as well as to write enclaves that perform well - as our performance evaluation indicated. Last, security understanding to design the enclaves API in a way that does not leak any sensitive data, i.e. that sensitive data resides always inside the enclave and is not returned from it in any of the exposed enclave APIs.

Hence, we directed our efforts to providing a high-level interface for data handling that can be used with RestAssured use cases. We opted for a data access layer that can handle sensitive data using enclaves, while exposing standard APIs such as an SQL query interface. By moving to a standard SQL interface, we reduce the overhead for use cases developers, eliminate the need of re-writing sensitive parts of the application into code that run in an enclave. Thus, we allow the application developers to focus on their core technology, while we support code running as a service in enclaves rather than application-specific ones.

3.2 RestAssured infrastructure with Opaque

The UC Berkeley Riselab presented Opaque in Usenix2017(https://www.usenix.org/system/files/conference/nsdi17/nsdi17-zheng.pdf). Opaque is a research project providing distributed data analytics platform based on Apache Spark 2.0.1 that supports a wide range of queries while providing strong security guarantees.

Opaque, an Apache 2.0 licensed open source, (https://github.com/ucbrise/opaque) is a package for Apache Spark SQL that utilizes Intel SGX to enable very strong security for SQL queries. With SGX technology, memory-level data encryption and authentication are active, so that even an attacker who has root access never sees decrypted data.

3.2.1 Implementation

Our aim was to evaluate the use of Opaque for the use cases in RestAssured. As part of the evaluation, we have assisted in raising the maturity level of Opaque code and integrating it with our tools for SGX.

We integrated TruCE framework with Opaque to fit RestAssured secured computation framework design as follows:

To enable deployment of Spark master in the cloud, Opaque enclave creation and attestation was replaced with the TruCE SDK and service. Using the TruCE server’s enhanced attestation guarantees computation integrity by verifying the enclave binary code, making sure no tampering have been performed, and that the Opaque runs on a genuine Intel SGX hardware.

To simplify integration of applications and services in the cloud with data processing executing inside SGX enclaves, we added a mechanism to pass data encryption keys to the software running in the Opaque enclaves. A key store service using TruCE toolkit was implemented to support several data encryption keys and enable a mechanism to pass those data encryption keys to the enclaves. We compiled TruCE with Opaque into ibm_opaque_truce jar and loaded it in the RestAssured Apache Spark cluster.
We collaborated with Riselab to enhance Opaque and adapt it to support the use cases that are part of RestAssured. This includes identifying issues (bugs), and providing fixes and patches for some issues in Opaque's GitHub repository. We have shared RestAssured use cases with the Opaque team and are working with them to address open issues. To date, we provided code to Opaque addressing issues such as supporting min, max, and count queries on additional data types, and adding support for SGX version 2.0. In addition, we are in the process of contributing TruCE integration. The Riselab team investigated the issues we identified while working on the RestAssured use cases and provided solutions as well, for example, they fixed errors for join queries. Through this work, we expect Opaque code to continue and mature.

RestAssured uses Opaque to encrypt the use case data on a private cloud installation and store the encrypted data in the cloud. All queries are running on the encrypted data files in the cloud and processed securely in SGX enclaves.
3.3 Query Gateway

To allow better and simpler integration of the analytic framework with the use cases and perform query manipulation, we added a query gateway service on top of Opaque. The query gateway performs RestAssured pre and post processing and exposes REST API for SQL queries. During those stages the query fields are screened by the Data GateKeeper (RestAssured data access control component) that enforces data policies per data subject. The filtered results are serialized into JSON objects and returned over a secured channel to the querying service.

3.3.1 Implementation

The query gateway is implemented using a web framework for Scala, Play framework (https://www.playframework.com/) version 2.5. Environment parameters are set in a configuration file. On start up, the Query Gateway creates a SparkSession and initiates TruCE-Opaque environment to enable secure data processing. All subsequent requests are routed to the active SparkSession and performed on the encrypted data stored in the cloud, decrypted only inside SGX enclaves.

Switching between different clusters is possible through the Query Gateway API. All activities concerning data access are logged to the configurable audit trail, and several log formats are currently supported including stdout, files and UDP Gelf formats.

![Diagram showing secured processing infrastructure components and API](image)

Figure 3.1: Secured processing infrastructure components and API
### Table 3.1: Query Gateway API

<table>
<thead>
<tr>
<th>URL</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET query/v1/sql select statement</td>
<td>Test method - Executes given query, returns JSON. Parameter: query string. Example: [<a href="http://132.252.68.52:9000/query/v1/sql">http://132.252.68.52:9000/query/v1/sql</a> select statement](<a href="http://132.252.68.52:9000/query/v1/sql">http://132.252.68.52:9000/query/v1/sql</a> select statement), Gender from escant groupby AddressId, Gender</td>
</tr>
<tr>
<td>GET query/v2/serviceId/usageIndex/SQL query</td>
<td>Executes given query while calling Data Gatekeeper to enforce sticky policies. Returns query records results in JSON format. Parameters: serviceID, usageIndex, query string. Example: [<a href="http://132.252.68.52:9000/query/v2/scant/gov_report/selectcount(">http://132.252.68.52:9000/query/v2/scant/gov_report/selectcount(</a><em>) from escant groupby AddressId, Gender](<a href="http://132.252.68.52:9000/query/v2/scant/gov_report/selectcount(">http://132.252.68.52:9000/query/v2/scant/gov_report/selectcount(</a></em>) from escant groupby AddressId, Gender)</td>
</tr>
</tbody>
</table>

### Table 3.2: Query Gateway configuration

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>environment.cloudCluster</td>
<td>The master URL for the cloud cluster secured analytic platform</td>
</tr>
<tr>
<td>environment.premiseCluster</td>
<td>The master URL for the private cluster secured analytic platform</td>
</tr>
<tr>
<td>environment.dataFolder</td>
<td>Cloud location of encrypted application data</td>
</tr>
<tr>
<td>environment.TruceLib</td>
<td>Parameters for the secured platform analytic application, including Truce application libraries and Intel sgxsdk libraries</td>
</tr>
<tr>
<td>environment.opaqueJar</td>
<td>Path to the secured platform analytic application</td>
</tr>
<tr>
<td>environment.ef.endpoint</td>
<td>GateKeeper Rest API endpoint</td>
</tr>
</tbody>
</table>
4 Performance evaluation

4.1 Introduction

There is a known challenging tradeoff between performance, security and functionality. Therefore, an interesting aspect of securing computation code using enclaves is the impact on performance. We set out to explore some base aspects of using enclaves using micro-benchmarks and how they perform.

One would expect some overhead due to the overhead of executing CPU instruction and accessing the encrypted memory in an enclave, memory copies and context switch of CPU modes when entering and exiting an enclave. In addition, extra security measures such as integrity tests and memory usage limitations can also affect performance.

Several third party software offerings for encryption libraries are compatible for use in SGX. For the “trusted” variants (inside an enclave), we evaluated three libraries: Intel sgxsdk, Intel sgxssl and an encryption library used in the open source project Opaque (based on AES-NI+PCLMULQDQ implementation in Mozilla NSS, will be referenced as Opaque in this section). In the untrusted area we tested two libraries: openssl and Opaque.

We ran several micro-benchmarks and settings to measure performance aspects of different encryption libraries and the viability of using enclaves vs. running outside enclaves. We expect to provide in the next deliverable details of end-to-end performance.

4.2 Evaluation

Our benchmarks were tested in the following setup: We used P50 series Lenovo laptops with 8-core Intel @Core(TM) i76820HQ CPU with 2.70GHz, 32GB memory running on an Ubuntu 16.04 OS. The libraries that we used were the Intel SGX SDK Linux 1.8 and SGXSSL (taken from the intel-sgx-ssl git repository on Nov 29th 2017). To gain an understanding of the expected overhead of invoking code inside sgx enclave (ECALL context-switches, MEE operations overhead and copying data using the in (or out) declarations in the edl overhead) we tested a simple function in four implementation variations

4.2.1 Testing a simple function

We tested the overhead of finding the maximum 4-byte integer of a given byte array (i.e., we treat an array of N bytes as an array of N/4 integers) with a single thread.

1. A regular function running in the untrusted area as one would run it without enclaves.

2. Copy and compute version - the array is copied into the enclave’s encrypted memory and then iterated over. This is implemented using an ECALL, in which the input array is declared with the in option in the edl file.

3. Compute on encrypted memory variant in this option we find the maximum on an array that resides in the enclaves encrypted memory. The array is prepared before the ECALL (by a previous ECALL). This is similar to option 2, but does not include the initial buffer copying operation in the measurements.

4. Compute on cleartext memory - in this option the ECALL finds the maximum of a given external input array without copying it into the enclave memory (namely, accessing only clear-text memory). This is achieved by declaring the array with the "user_check" option in the edl file.
Of the three enclave variants, option 4 should be the fastest, since it does not require the array to be decrypted by the SGX Memory Encryption Engine (MEE). Options 2 and 3 do require the MEE decryption in order to perform the computations (while option 2 also involves MEE encryption as well as decryption).

**Figure 4.1: Find Max Throughput**

For evaluating the performance of each option, we compared the throughput of the calls using various array sizes. The results in Figure 1 show the throughput (MBs processed per second) as a function of the array size. The first observation is that for small arrays, there is a huge overhead in running a function in an enclave which is likely caused by the context switches overhead of entering and exiting the enclave. This overhead becomes negligible for larger buffers and the gap between the untrusted and the two faster trusted versions is mostly closed with arrays of size larger than 256KB.

Next, we tested the overhead of using enclaves for a number of computational tasks, with a focus on cryptographic functions that are carried out in TruCE enclaves, namely: Key generation, hashing keys and signatures and encryption/decryption operations.

### 4.2.2 Testing SHA256

We start by examining the overhead of computing SHA256. In the untrusted area, we tested the openssl implementation of SHA256. In the trusted area (i.e., inside the enclave), we tested the intel-sgxssl implementation of SHA256 which in turn calls openssl. We also tested the function `sgx_sha256_msg`, provided by intel-sgx-sdk API.

The results in Figure 4.2 indicate that, as expected, we still have a huge gap in the throughput of computing sha256(msg) for small size messages. However, as the message sizes increases, the gap does not completely close even for very large size messages. We ran this test with two different implementations, first using user-check declaration in the edl file for both input and output buffers (i.e. clear text), and the second using an input message which was stored in the enclave’s local memory. The results were the same.
Testing AES Encryption

Testing the overhead of encrypting and decrypting messages, we focused on testing AES128-GCM encryption. The basic methodology used was creating a buffer in memory and encrypting it a large number of times using either a regular function or an ECALL to keep the buffer in cache in the CPU (single threaded execution). Each test is run 30 times and the results presented here are averages. The results for decryption (rather than encryption) were very similar, so we only present the encryption numbers here. In the trusted area, we tested three libraries: Intel sgxsdk, Intel sgxssl and an encryption library used in the open source project Opaque. In the untrusted area we tested two libraries: openssl and Opaque. We first focused on the fastest ECALL implementation in which both "in_buf" and "out_buf" are declared with the "user_check" option.

The results, shown in Figure 4.3, are somewhat surprising and more complex than one would hope for:

- The sgxsdk version achieves a maximal throughput of 2150MB/sec (about 43% of the untrusted throughputs). This is due to the fact that by default it runs a non-optimized version of Intel’s IPP Crypto library which does not use Intel’s AES-NI hardware optimizations. We were told that by manually compiling and linking the SDK with an optimized binary of the IPP Crypto for SGX, one might achieve the desired acceleration, but we have not tried this.

- The sgxssl implementation presumably runs the same code as the untrusted openssl version. However, its performance completely crashes and achieves only a 110MB/sec throughput (about 2% of the untrusted throughputs).

We then tested the encryption in the end-to-end use-case, in which the input messages reside in the enclave’s encrypted memory rather than clear-text input buffers from the untrusted memory. The results, in Figure 4.3, are very close to those of the figure 4.2 variant, except for a significant difference for the large buffers. This reduces the maximal throughput of the enclave variant on large buffers to about 70% of the throughput of untrusted version. Growing the buffer further (above the EPC size threshold of 92GB) results in a massive degradation in trusted code throughput.
Our posting the results in Medium blog [https://medium.com/@danny_harnik/impressions-of-intel-sgx-performance-22442093595a](https://medium.com/@danny_harnik/impressions-of-intel-sgx-performance-22442093595a) attracted Intel to look into this issue and found that due to an error in CPUID invocation, it used a non-hardware optimized version of openssl and uploaded a patch to fix it (see 3. Library initialization and .init section in D4.1 – March 1, 2018 27 of 33
Retracting the tests with this fix gets the sgxssl behavior very close to that of the Opaque library, and exhibits the same overall behaviors - both run reasonably well both inside and outside an enclave. On short messages, we have a significant gap between the trusted and untrusted throughputs, which is caused by the context switches overhead. However, for large messages, the trusted version of Opaque essentially closes the gap with the untrusted libraries, achieving around 96% of the throughput of untrusted versions (at a rate of about 5GB/sec). Figures [4.5] and [4.6] depict these new tests (just for the OpenSSL trusted and untrusted code bases).

![Figure 4.5: AES128-GCM Encryption Throughput](image)

We conclude the encryption library inside SGX enclaves should be selected with care to minimize performance degradation. Following those findings, and with the latest fix to Intel sgxssl, RestAssured framework uses Intel sgxssl and Opaque encryption libraries.

4.2.4 Opaque vs native Spark

Opaque enclaves use the AES-NI+PCLMULQDQ implementation in Mozilla NSS encryption library which was tested above and showed good performance compared to other SGX encryption libraries. Since with the use of Opaque in RestAssured, each data access is processed securely in SGX enclaves, we completed the check of Opaque compliance to RestAssured by also looking at Opaque processing throughput performance compared to the same Spark installation without data encryption. For the test we used the BDB benchmark with Spark 2.0.1 on local.

We compared Opaque versus native SparkSQL running the Spark worker on the same hardware and storing the data locally on disk to eliminate access time to data and focus on the processing aspects. We worked with the 1.2GB rankings dataset from BDB. The native Spark SQL ran the filter query (df.filter(pageRank > 3000).count) in 5 seconds, indicating a processing throughput of ~200 MB/sec.

We encrypted the 1.2GB rankings dataset using Opaque. This resulted in a 2.9GB dataset, almost 3 times
larger than the original. Apparently, this is the result of the Opaque serialization mechanism, since encryption does not impact size dramatically. Running the same filter query (df.filter(pageRank > 3000).count) in Opaque with SGX enclaves using the encrypted data, takes around 10 seconds, corresponding to a higher processing throughput of \( \sim 300 \) MB/sec, but still 2 times longer than the standard Spark SQL.

<table>
<thead>
<tr>
<th></th>
<th>Spark</th>
<th>Spark with Opaque</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data set (rows)</td>
<td>18M</td>
<td>18M</td>
</tr>
<tr>
<td>Data set (disk size)</td>
<td>1.2 GB</td>
<td>2.9 GB</td>
</tr>
<tr>
<td>Query runtime</td>
<td>5 sec</td>
<td>10 sec</td>
</tr>
<tr>
<td>Throughput (rows/sec)</td>
<td>3.6 M/sec</td>
<td>1.8 M/sec</td>
</tr>
<tr>
<td>Throughput (MB/sec)</td>
<td>200 MB/sec</td>
<td>300 MB/sec</td>
</tr>
</tbody>
</table>

Figure 4.6 shows that raw AES decryption using SGX enclaves runs at a rate of about 4 GB/sec on a single thread, an order of magnitude faster than Spark SQL throughput with either standard Spark or with Opaque. This implies that SGX decryption may not be the bottleneck here.
5 Data Access Protection in Usecases

We conclude with a description of the integration of the above framework with RestAssured use-cases on UDE testbed.

5.1 Environment Settings

UDE Testbed cloud and private clusters setup settings:
  Cloud cluster:
  Fujitsu Celsius w550 workstation
  16.04.1-Ubuntu linux, 4.10.0-42 kernel version
  8-core Intel(R) Xeon(R) CPU E3-1275 v5 @ 3.60GHz, 16 GB DDR4 memory
  Intel SGX enabled in BIOS and running in HW mode.
  Linux* Intel(R) SGX software stack V2.0: Intel SGX driver, Intel SGX SDK, and Intel SGX Platform Software (PSW), Intel SGX SSL 2.0.
  Spark 2.0.1 master with 1 Spark executor, 8 cores, 10G memory
  The private (located on premise) cluster is identical in settings to the cluster deployed in the cloud, but with Intel SGX running in emulated (SIM) mode instead of hardware mode.
  Intel SGX driver for Linux is a kernel module, therefore dependent on the Linux kernel version. TruCE and Opaque were developed on earlier versions, and had to be modified and adapted for the testbed hardware.

5.2 Query Gateway

The Query gateway component serves as a gateway between use case applications and Opaque analytic framework. The gateway opens a REST endpoint that accepts string SQL queries from the applications, sends the queries for verification and modification by the GateKeeper component engine, and executes the modified query in an appropriate Opaque cluster. The results are serialized into a JSON object and sent back to the application on a secure REST channel. The Query gateway runs in a trusted (off-cloud) part of RestAssured platform, since it works with plaintext data, query results and encryption keys. The gateway also handles the query transformation as required by Data Gatekeeper (access control) rules.

RestAssured currently uses two Opaque installations one in the trusted part of the platform, and another in the untrusted (public cloud) part of RestAssured platform. The main workload is performed in the public cloud Opaque cluster. The private Opaque installation in the trusted environment is running on an SGX emulation and serves as a fallback cluster when the cloud cluster becomes unavailable. This is done as a proof of concept; the failover can also be implemented by going to another SGX cluster in the cloud, running in a separate availability zone. The switch between main and backup clusters is triggered by the RestAssured Adaptation component, which sends a switch message to the Query gateway. The query gateway logs to a third party audit trail all activities concerning data access.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>TestBed environment configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>environment.cloudCluster</td>
<td>&quot;spark://132.252.68.52:7077&quot;</td>
</tr>
<tr>
<td>environment.premiseCluster</td>
<td>&quot;spark://132.252.68.53:7077&quot;</td>
</tr>
<tr>
<td>environment.dataFolder</td>
<td>&quot;/var/RestAssured/data&quot;</td>
</tr>
<tr>
<td>environment.TruceLib</td>
<td>&quot;/opt/RestAssured-Code/ibm/truce/application:&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;/opt/intel/sgxsdk/lib64:/opt/intel/sgxsdk&quot;</td>
</tr>
<tr>
<td>environment.opaqueJar</td>
<td>&quot;lib/ibm_opaque_truce_2.11-0.3.jar&quot;</td>
</tr>
<tr>
<td>environment.ef.endpoint</td>
<td><a href="http://213.137.178.176:80/">http://213.137.178.176:80/</a></td>
</tr>
</tbody>
</table>
5.3 Secured Data Processing

To demonstrate that no personal data is leaked in RestAssured, we use the Pay as You Drive (PAYD) use-case car dataset with a simple query that retrieves the number of cars in the system. For the demonstration, we treat the car model field as a sensitive personal information and our aim is to show that data is never in the clear - as it is stored encrypted and processed securely using SGX technology in enclaves.

To demonstrate, we use Spark with Opaque, under the same configuration and methods used in RestAssured Query Gateway component and execute a subset of Query Gateway commands serially Figure 5.1:

- Load data to Spark data set (load PAYD car dataset encrypted using Opaque in the private (secured) cluster on the secured test, and the plain original PAYD JSON on the trivial test).
- To enable SQL queries (Query Gateway API), create a temporary view on the data set.
- Execute select all data from the car dataset.

Following by the same query on the original JSON PAYD car data set Figure 5.3. We then look at the process dump, taken by gcore, and look for the sensitive information (e.g. car model = Sedan). The dumps shows clearly that the secured processing implementation memory Figure 5.2 leaks no sensitive information, while anyone with access to the machine memory could retrieve the sensitive information from the not protected analytic query implementation Figure 5.4.

```
scale> var df2 = spark.sql("select * from example")
df2: org.apache.spark.sql.DataFrame = [autoID: bigint, DriveTrain: string ... 9 more fields]
scale> df2.count
Call spx_get_extended_epid_group_id success.
Sending msg0 to remote attestation service provider.
Call spx_ra_get_msg0 success.
Send MSG1 to remote attestation service provider. Received MSG2.
Call spx_ra_proc_msg2 success.
msg: 
16:0153b1de77c7985f6352ab3ca4e657f9c648b612b49ff51aa149e

Remote attestation success!
Generating new enclave keys
MKCS #1 v1.5 encryption/decryption ok
Truce ID len: 526
JNI: Successfully created truce_session.
JNI: Getting the data key.
Connected to Key store
Received message, length: 528
Message type: 2, size: 512
Enclave: got secret
Starting an enclave
Got data key
rest: Long: 2500
```

Figure 5.1: Secured data processing query using Opaque and Truce

```
df7 = df7.select("*")
df7.count() / 2.0
0.0

remote_attestation_success()
new_enclave_keys
MKCS #1 v1.5 encryption/decryption ok
Truce ID len: 526
JNI: Successfully created truce_session.
JNI: Getting the data key.
Connected to Key store
Received message, length: 528
Message type: 2, size: 512
Enclave: got secret
Starting an enclave
Got data key
rest: Long: 2500
```

Figure 5.2: Memory dump of secured data processing
5.4 Scant Prototype

The current RestAssured prototype runs the Scant use-case with hundreds of records encrypted using Opaque and stored in the public cloud cluster. Currently we use an NFS storage to store the encrypted application data; in next version of the deliverable, we plan to use object storage.

5.5 Gatekeeper Integration

The Query Gateway joins the query results with the Gatekeeper to comply with the defined data protection policies. This is done through the Gatekeeper policy enforcement REST API. The Query Gateway API was designed to include parameters required for Gatekeeper integration while some other parameters are calculated and extracted from the query. The Gatekeeper returns a JSON object containing records which comply to the contract, or error otherwise. The JSON object also contains schema metadata required to perform the join.
5.6 Audit Log

The Query Gateway logs to RestAssured’s audit trail all activities concerning data access to the RestAssured Graylog component. This is done by a logstash-to-Gelf conversion using a plugin.