University of Piraeus

School of Information and Communication Technologies

Department of Digital Systems

Postgraduate Program of Studies

MSc Digital Systems Security

Course: Network Security

ASYLO: An open-source Framework for Confidential Computing

Supervisor Professor: Xenakis Christos

Name - Surname	E-mail	Student ID
Theodoros Alexandros Chandrinos	alexandros.chandrinos@ssl-unipi.gr	MTE2131
Ioannis Chouchoulis	giannischouch@ssl-unipi.gr	MTE2132

Piraeus

28/02/2022

Abstract

This paper focuses on Google Asylo, which is an open-source framework, powerfull enough to build portable enclave applications and implement cryptographic functions running on a trusted environment. Furthermore, there is dedicated section regarding the Trusted Execution Environment and its functionality, as the Asylo is based on it. A short reference about Local and Remote Attestation is provided, because the Asylo framework currently supports mechanisms for Remote Attestation. Finally, concerning the implementation of the basic cryptographic functions, RSA is used for the key generation & storage, the hash functions (MD5, SHA1, SHA2), the AES encryption - decryption, the digital signing and the Diffie Hellman protocol key exchange.

Table of Content

Introduction

Data exists in three states: at rest, in use, and in transit. Data that is stored is "at rest", data that is being processed is "in use", and data that is traversing across the network is "in transit". Even if we encrypt data at rest and in transit across the network, the data they process are still vulnerable to unauthorized access and tampering at runtime. Protecting the data in use is critical to offer complete security across the data lifecycle. Cryptography or encryption is now commonly used by organizations to protect data confidentiality (preventing unauthorized viewing) and data integrity (preventing unauthorized changes). There are now advanced data security platforms that enable applications to run within secure enclaves or trusted execution environments that offer encryption for the data and applications. Confidential computing technology protects data during processing.

Confidential computing is a cloud computing technology that isolates sensitive data in a protected CPU enclave during processing. The contents of the enclave - the data being processed, and the techniques that are used to process it - are accessible only to authorized programming code, and are invisible and unknowable to anything or anyone else, including the cloud provider. The primary goal of confidential computing is to provide greater assurance that the data in the cloud is protected and confidential, and to encourage anyone to move more of their sensitive data and computing workloads to public cloud services.

For years, cloud providers have offered encryption services to help protect data at rest (in storage and databases) and data in transit (moving over a network connection). Confidential computing eliminates the remaining data security vulnerability by protecting data in use — that is, during processing or runtime. Before it can be processed by an application, data must be unencrypted in memory. This leaves the data vulnerable just before, during and just after processing to memory dumps, root user compromises and other malicious exploits. Confidential computing solves this problem by leveraging a hardware-based trusted execution environment, or TEE, which is a secure enclave within a CPU. The TEE is secured using embedded encryption keys;

embedded attestation mechanisms ensure that the keys are accessible to authorized application code only. If malware or other unauthorized code attempts to access the keys — or if the authorized code is hacked or altered in any way — the TEE denies access to the keys and cancels the computation.

In this way, sensitive data can remain protected in memory until the application tells the TEE to decrypt it for processing. While the data is decrypted and throughout the entire computation process, it is invisible to the operating system, to other compute stack resources, and to the cloud provider and its employees.

Trusted Execution Environments

A hardware-based trusted execution environment is a secure and isolated environment that prevents unauthorized access or modification of applications and data while they are in use.

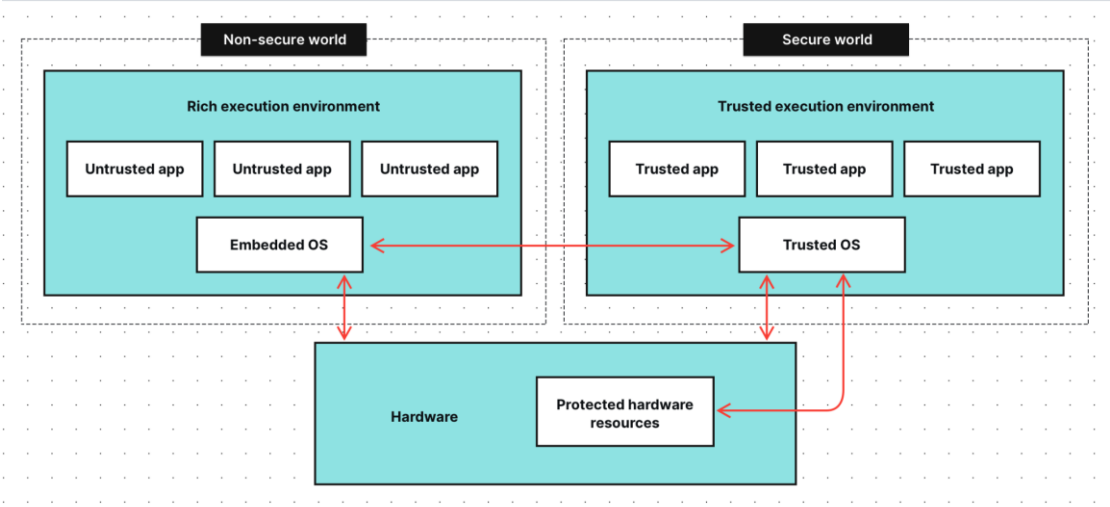

Figure 1

By that, it increases the security level of data in organizations that manage sensitive and regulated information.

A trusted execution environment (TEE) is a secure area of a main processor that guarantees optimal protection for highly sensitive data in all its states, with respect to confidentiality and integrity. TEE can be used on-premises, in the cloud or within embedded hardware platforms. It can include a built-in certified hardware crypto engine for key management and encryption. You can be sure that your data maintains its integrity and confidentiality, achieving compliance throughout its entire lifecycle.

TEEs were created to further secure previously trusted platforms. In the mid-2000s, the implementation of TEE began to become a standard-based approach for internet-connected devices. At this point, more organizations began developing TEEs, such as the Trusted Logic and Texas Instruments in 2004. In 2006, ARM developed a commercialized product for TEE called TrustZone. That same year, the Open Mobile

Terminal Platform wrote the first set of requirements for trusted environments. These requirements were revised again in 2008.

The 2010s saw a growth in the use of TEEs. In 2012, GlobalPlatform and the Trusted Computer Group founded began working together to create another set of specifications for TEE, used in conjunction with the Trusted Platform Module. Since this time, GlobalPlatform has been the driving force behind driving TEE standardization. TEEs are now used widely in complex devices, such as smartphones, tablets and set-top boxes. TEEs are also used by manufacturers of constrained chipsets and IoT devices in sectors such as industrial automation, automotive and healthcare, who are now recognizing its value in protecting connected things.

Running parallel to the operating system and using both hardware and software, a TEE is intended to be more secure than the traditional processing environment. This is sometimes referred to as a rich operating system execution environment, or REE, where the device OS and applications run.

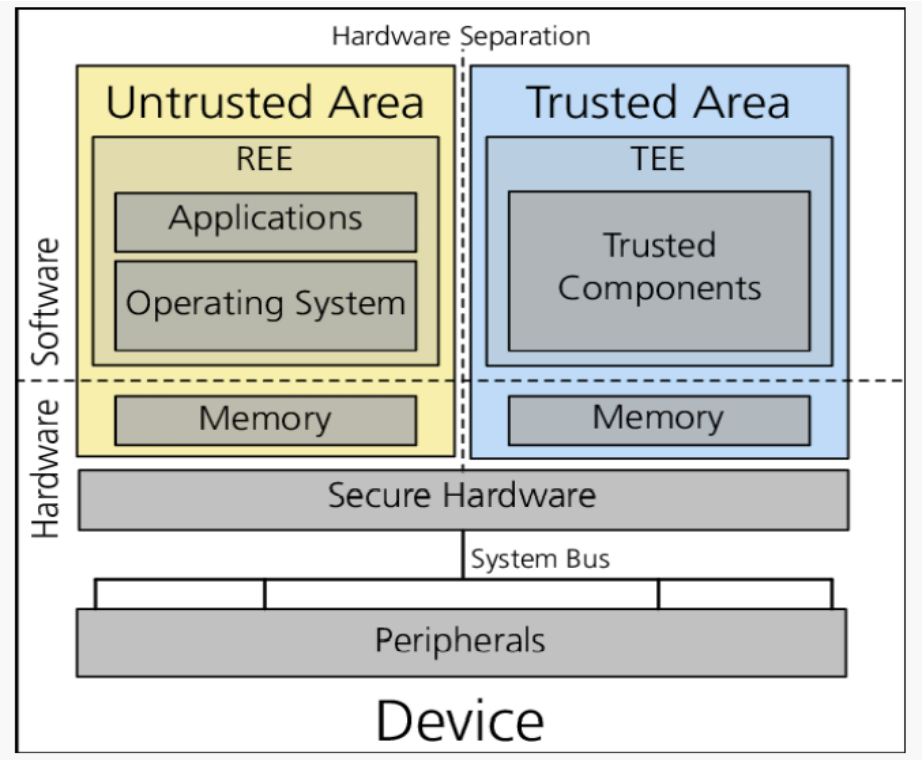

Figure 2

The benefits of a Trusted Execution Environment

Deploying an application inside a trusted execution environment protects data in use with confidential computing technology, without any changes in the application itself. In other words, it simplifies the process of adding a layer of security to an existing solution. Other benefits of deploying in a trusted execution environment include:

- **Rapid deployment:** Advanced TEE solutions are quick to deploy.
- **Ensured data confidentiality and integrity**: Ensuring the security of sensitive data in transit, in use, and at rest for critical applications and data.
- **Secured deployment and execution of applications**: By deploying inside a TEE, hostile applications are not able to exploit the operating system on which they run and cannot access the central processing unit (CPU) without explicit permission.
- **Trusted Collaboration:** A TEE is a safe environment for multiple parties to share and process their data. This opens up the possibility of using a TEE to securely collaborate with other organizations and individuals and get even more value out of your sensitive data.
- **Simplified compliance**: Achieve compliance with key management and encryption through an easy-to-use cryptographic API. Using a certified solution to encrypt or digitally sign sensitive data across the entire data encryption lifecycle: at rest, in transit and in use.
- **Immediate return on investment**: Inexpensive, rapid to deploy and easy to integrate and maintain.
- **Weakest link protected:** Access to data in use is blocked from unauthorized third-parties. [1]

How does TEE work

In a system with a TEE, we have untrusted applications running on a Rich Execution Environment (REE) and trusted applications (TAs) running on a Trusted Execution Environment (TEE). Only trusted applications running on a TEE (Secure World) have complete access to the main processor, peripherals and memory, while hardware isolation protects these from untrusted applications running on the main operating system (Non-Secure World).

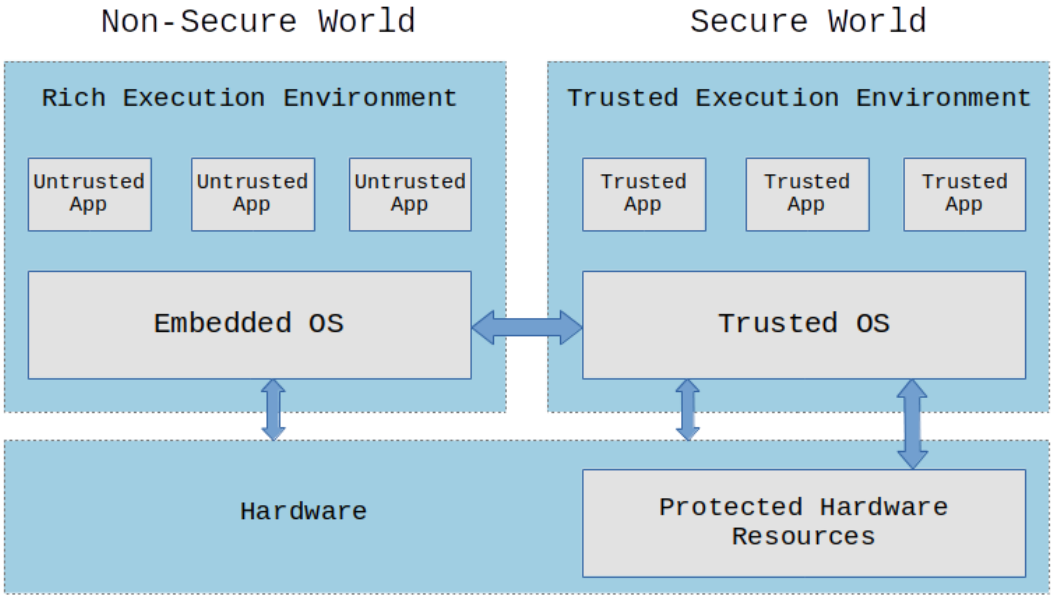

Figure 3

Although the diagram above exemplifies a TEE with an operating system (Trusted OS), we could just have a bare-metal firmware exposing an interface with exclusive access to certain hardware resources. In a TEE, all trusted applications (TAs) and associated data is completely isolated from the normal (untrusted) operating system and their applications. Also, trusted applications must run in isolation from other trusted applications and from the TEE itself.

Also, TEE only accepts code for execution that has been appropriately authorized and checked by other authorized code. So in TEE, we need a secure boot feature to check the integrity and authenticity of all operating system components

(bootloaders, kernel, filesystems, trusted applications, etc). This ensures that nobody has tampered with the operating system's code when the device was powered off. [2]

Current and future uses of TEE

The TEE is not an emerging technology. For example, apps such as Samsung Pay or WeChat Pay, and many of the leading Android device makers' flagship phones, all use a TEE. In this way, TEE has become a central concept when considering security of sensitive data in smartphones. The increase of the internet of things is also expanding the need for trusted identification to new connected devices. TEEs are one technology helping manufacturers, service providers and consumers to protect their devices, IP and sensitive data.

Google Asylo

Introduction

Asylo is an open framework for developing enclave applications. Asylo lets you take advantage of a range of emerging trusted execution environments, including both software and hardware isolation technologies.

Asylo provides:

- The ability to execute trusted workloads in an untrusted environment, inheriting the confidentiality and integrity guarantees from the security backend, i.e., the underlying enclave technology.
- Ready-to-use containers, an open source API, libraries, and tools so you can develop and run applications that use one or more enclaves.
- A choice of security backends.
- Portability of your application's source code across security backends.

Asylo enables you to take advantage of emerging hardware and software technologies that provide trusted execution environments, known as enclaves. Asylo provides an API, trusted and untrusted runtime libraries, and C/C++ toolchain so you can integrate your application with an enclave backend of your choice. The benefit of running your application using an enclave is that it provides confidentiality and integrity assurances for your sensitive workloads.

The main goals of Asylo are:

- **Ease of use**. Asylo provides ready-to-use containers, an open source API, libraries, and tools that you can use to run your application backed by an enclave technology of your choice. Ease of deployment is a focus.
- **Portability across enclave backends**. Once your application is adapted to the Asylo API and toolchain, you can switch to a different backend by simply re-compiling and re-packaging your application. There is no need to refactor your code because the Asylo API and toolchain are designed to work with various enclave technologies. [3]

Enclaves

On traditional systems, the Operating System (OS) kernel has unrestricted access to a machine's hardware resources. The kernel typically exposes most of its access permissions to a root user without any restrictions. Additionally, a root user can extend or modify the kernel on a running system. As a result, if an attacker can execute code with root privileges, they can compromise every secret and bypass every security policy on the machine. For instance, if an attacker obtains root access on a machine that manages TLS keys, those keys may be compromised.

Enclaves are an emerging technology paradigm that changes this equation. An enclave is a special execution context where code can run protected from even the OS kernel, with the guarantee that even a user running with root privileges cannot extract the enclave's secrets or compromise its integrity. Such protections are enabled through hardware isolation technologies such as Intel SGX or ARM TrustZone, or even through additional software layers such as a hypervisor. These technologies enable new forms of isolation beyond the usual kernel/user-space separation.

New security features are exciting for developers building secure applications, but in practice there is a big gap between having a raw capability and developing applications that leverage that capability. Building useful enclave applications requires tools to construct, load, and operate enclaves. Doing useful work in an enclave requires programming-language support and access to core platform libraries.

In Asylo, an enclave runs in the context of a user-space application. However, for security and portability reasons, Asylo does not support direct interactions between the enclave code and the OS. Instead, all enclave-to-OS interactions must be mediated through code that runs outside the enclave. We refer to the code running outside the enclave as the untrusted application and the code running inside the enclave as the trusted application, or simply the enclave.

Asylo takes an object-oriented approach to enclave application development. Conceptually, an enclave is a collection of private data and private logic, along with public methods to access it. To this end, Asylo models an enclave using

TrustedApplication, an abstract class that defines various enclave entry-points. To implement an enclave application, a developer creates a subclass of TrustedApplication and implements the appropriate methods.

The TrustedApplication class declares methods corresponding to each entry point defined by the Asylo API.

- **Initialize** initializes the enclave with configuration values that are bound to the enclave's identity. This should be used for security-sensitive configuration settings.
- **Run** takes input messages from the untrusted environment code, performs trusted execution, and returns an output message to the untrusted environment.
- **Finalize** takes finalization values from the untrusted environment and is called before the enclave is destroyed.

If any of these methods are not overridden, they simply return an OkStatus. [4]

Architecture

Asylo allows an untrusted execution environment to host a trusted environment. The benefit is to enable sensitive workloads in an untrusted execution environment with the inherited confidentiality and integrity guarantees of the chosen trusted execution environment.

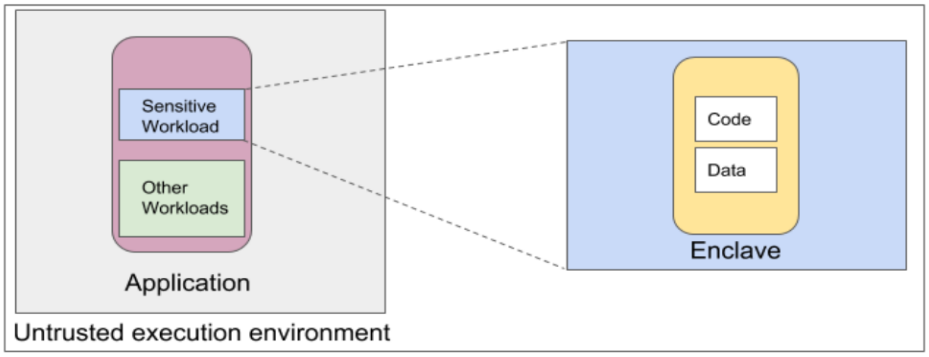

Figure 4

The untrusted execution environment consists of surrounding infrastructure—for instance the operating system and system libraries—and human operators that are less trusted than an enclave. The trusted execution environment consists of one or more enclaves, which protect code and data in a sensitive workload. A trusted execution environment allows an enclave to:

- Prevent vulnerabilities outside the enclave from compromising the workloads that run in the enclave. Depending on the enclave technology, this protection may shield application code and sensitive data from one or more of the following:
    - Guest virtual machine (VM) kernel or user mode vulnerabilities
    - A compromised hypervisor or host operating system
- Provide confidentiality and integrity assurances for sensitive workloads at the enclave communication boundary with its host in an untrusted environment. Confidentiality and integrity protection within an enclave is stronger than protection provided by OS-native applications.
- Provide integrity assurances for the enclave during execution. Depending on the enclave technology, users can either remotely or locally verify the integrity of the enclave that runs their sensitive workloads. [5]

Security Backends

Asylo provides a choice of security backends:

- Software backends, such as those based on isolation provided by hardware virtualization
- Hardware backends, such as those based on proprietary CPU manufacturer implementations

Developers choose a backend that meets the security requirements for their sensitive workloads. The Asylo team has long-term design plans to protect the confidentiality and integrity of a wider range of sensitive workloads and additional backends. Depending on the backend, Asylo is designed to integrate applications with enclaves that provide confidentiality and integrity guarantees against the following threats:

- Malicious or compromised administrator
- Malicious or compromised tenant of a hypervisor
- Malicious or compromised network
- Compromised operating system
- Compromised BIOS

The current release of Asylo supports a simulated enclave backend, which should not be used in production environments. Support for additional backends will be added over time to meet the varying needs and security requirements of Asylo users. [6]

Asylo API Overview

Asylo provides strong encapsulation around data and logic for developing and using an enclave. In the Asylo C++ API, an enclave application has trusted and untrusted components. The API has a central manager object for all hosted enclave applications.

The following are the main classes for developing and using an enclave.

- TrustedApplication is the trusted component of an enclave application that is responsible for sensitive computations.
- EnclaveClient is the untrusted component of an enclave application that is responsible for communicating messages with the trusted component.

EnclaveManager is a singleton object in the untrusted system that is responsible for managing enclave lifetimes and shared resources between enclaves.

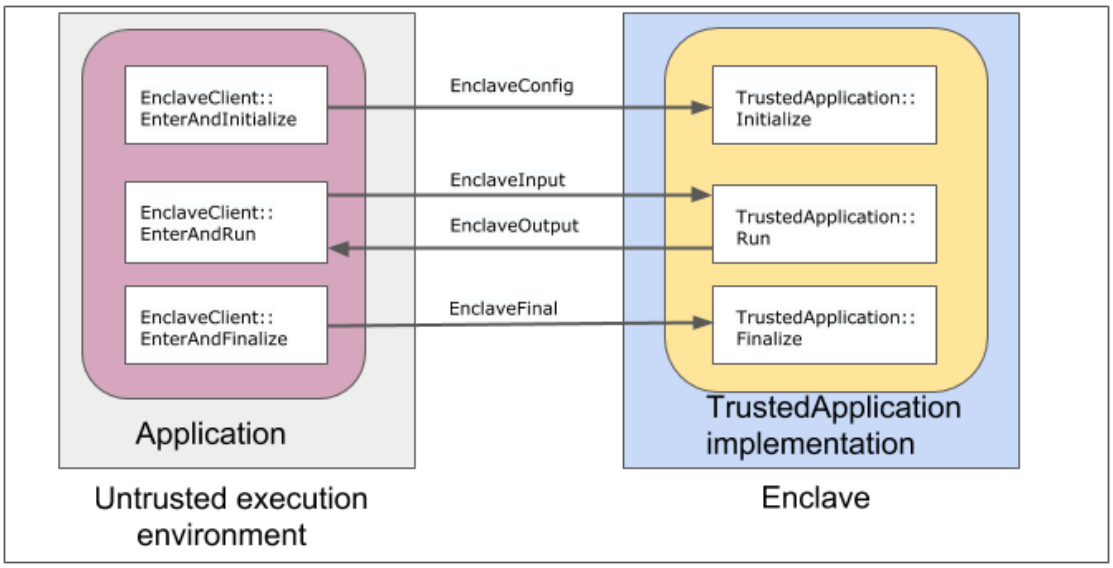

Figure 5

We refer to the process of switching from an untrusted execution environment to a trusted environment as entering an enclave. Every enclave provides a limited number of entry points where trusted execution may begin or resume. [4]

Attestation

Sometimes enclaves need to collaborate with other enclaves on the same platform due to different reasons such as data exchange if the enclave is too small to hold all the information, or communication with Intel reserved enclaves to conduct specific Intel services.

Therefore, the two exchanging enclaves have to prove to each other that they can be trusted. In other scenarios when an SGX enabled ISV client requests secrets from its ISV client, password management service for example, the client has to prove to the server that the client application is running on a trusted platform that can process the secrets securely. Both of those two conditions require a proof of secured execution environment, and Intel SGX refers to this proving process as attestation.

There are two types of attestation with respect to the two above mentioned scenarios: Local Attestation and Remote Attestation. The successful result of local attestation provides an authenticated assertion between two enclaves running on the same platform that they can trust each other and exchange information safely, while remote attestation provides this kind of verification for the ISV client to the server so that ISV server can confidently provide the client with the secrets it requested.

Local Attestation

Before multiple enclaves collaborate with each other on the same platform, one enclave will have to authenticate the other locally using Intel SGX Report mechanism to verify that the counterpart is running on the same TCB platform by applying the REPORT based Diffie-Hellman Key Exchange. This procedure is referred to as local attestation by Intel. The successful result of local attestation will offer a protected channel between two local enclaves with guarantee of confidentiality, integrity, and replay protection.

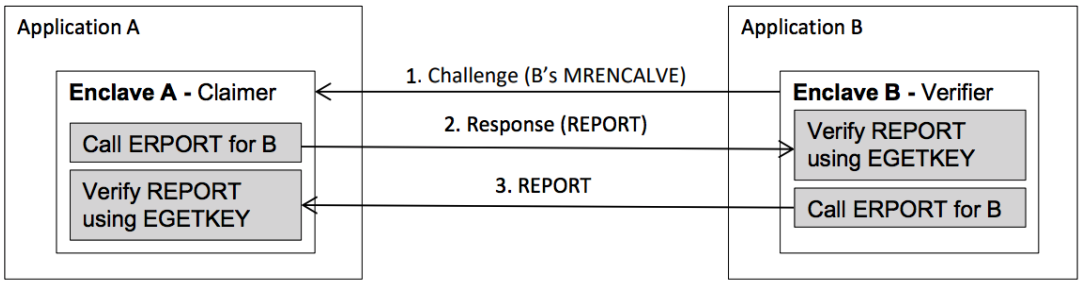

Figure 6

The process is the following:

1. There are two enclaves on the same platform, referred to as Enclave A and Enclave B. We assume they have established a communication path between each other, and the path doesn't need to be trusted. W.l.o.g we assume B is asking A to prove it's running on the same platform as B.
2. First, B retrieves its MRENCLAVE value and sends it to A via the untrusted channel.
3. A uses EREPORT instruction to produce a report for B using B's MRENCLAVE. Then A sends this report back to B. A can also include Diffie-Hellman Key Exchange data in the REPORT as user data for trusted channel creation in the future.
4. After B receives the REPORT from A, B calls EGETKEY instruction to get REPORT KEY to verify the REPORT. If the REPORT can be verified with the REPORT KEY, then B assures that A is on the same platform as B because the REPORT KEY is specific to the platform.
5. Then B use the MRENCLAVE received from A's REPORT to create another REPORT for A and sends the REPORT to A.
6. A then also can do the same as step 4 to verity B is on the same platform as A.
7. By utilizing the user data field of the REPORT, A and B can create a secure channel using Diffie-Hellman Key Exchange. Information exchange can be encrypted by the shared symmetric key. [7]

Remote Attestation

Remote attestation enables an enclave to remotely attest its identity to another entity, which can be another enclave or a non-enclave. Remote attestation mechanisms are strictly stronger than local attestation mechanisms because they use an external authority (usually a certificate authority) as the root of trust rather than a machine-local root of trust, such as a symmetric secret.

Asylo currently supports two mechanisms for remote attestation for Intel SGX. With Intel SGX, Remote Attestation software includes the application's enclave, and the Intel-provided Quoting Enclave (QE) and Provisioning Enclave (PvE). The attestation Hardware is the Intel SGX enabled CPU. Remote attestation provides verification for three things: the application's identity, its intactness (that it has not been tampered with), and that it is running securely within an enclave on an Intel SGX enabled platform.

Asylo supports two different quoting enclaves: the Intel ECDSA Quoting Enclave and the Asylo Assertion Generator Enclave.

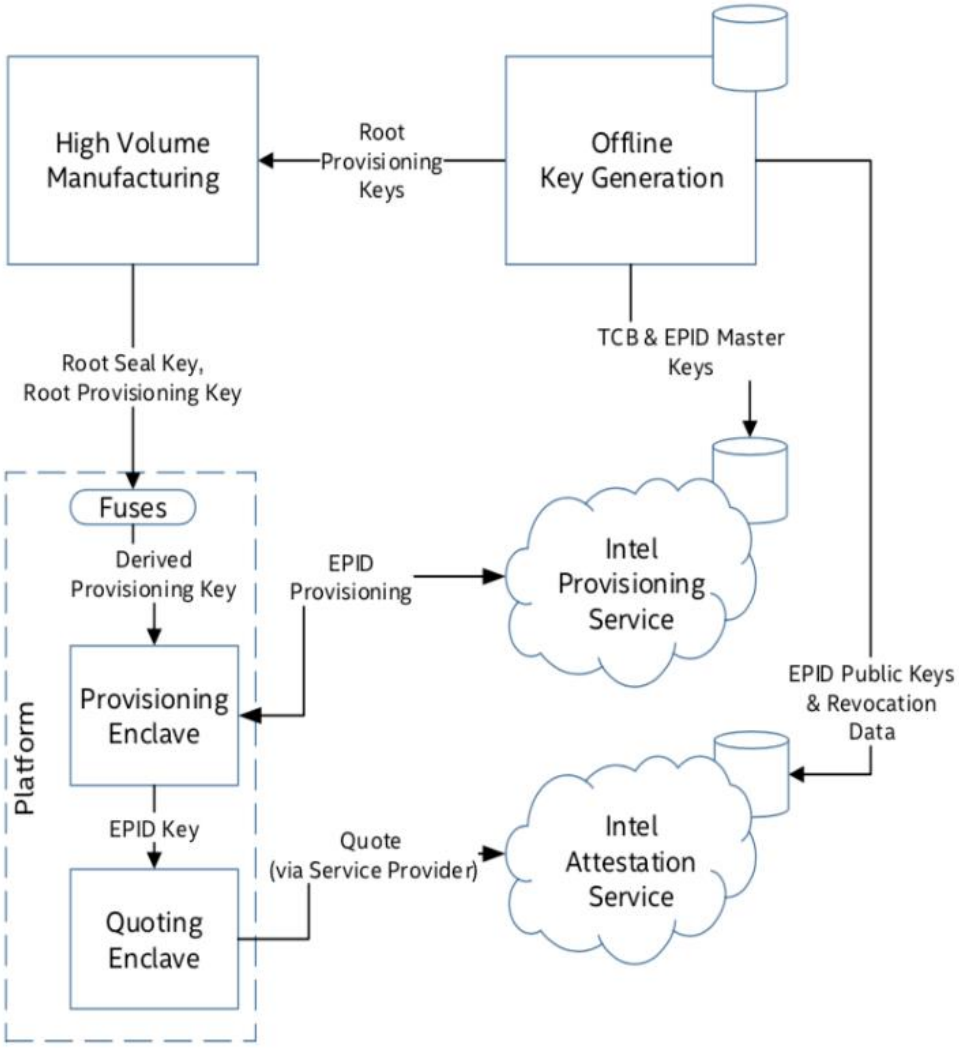

Figure 7

SGX remote attestation is enabled by a quoting enclave. A quoting enclave is an enclave that produces quotes, or remote attestations, on behalf of enclaves running on the same machine. A quoting enclave is certified by one or more certificate authorities. This enables the quoting enclave to produce quotes that chain back to a trusted authority.

A quoting enclave can be certified by Intel by having its hardware REPORT signed by the Provisioning Certification Key (PCK). The PCK is a signing key wielded by the Provisioning Certification Enclave (PCE), a special enclave written, signed, and distributed by Intel. Intel issues PCK certificates chaining back to the Intel SGX Root CA, the root of the Intel SGX PKI.

Remote attestation is built on top of local attestation: a quoting enclave uses SGX local attestation to verify another enclave's identity before signing a statement about that enclave's identity. This same mechanism is used by the PCE to certify an quoting enclave.

For remote attestation, both symmetric and asymmetric key systems are used. The symmetric key system is used in local attestation with only the quoting enclave and the EREPORT instruction having access to the authentication key. Asymmetric key system is used for creating an attestation that can be verified from other platforms. The attestation key itself is asymmetric (EPID keys).

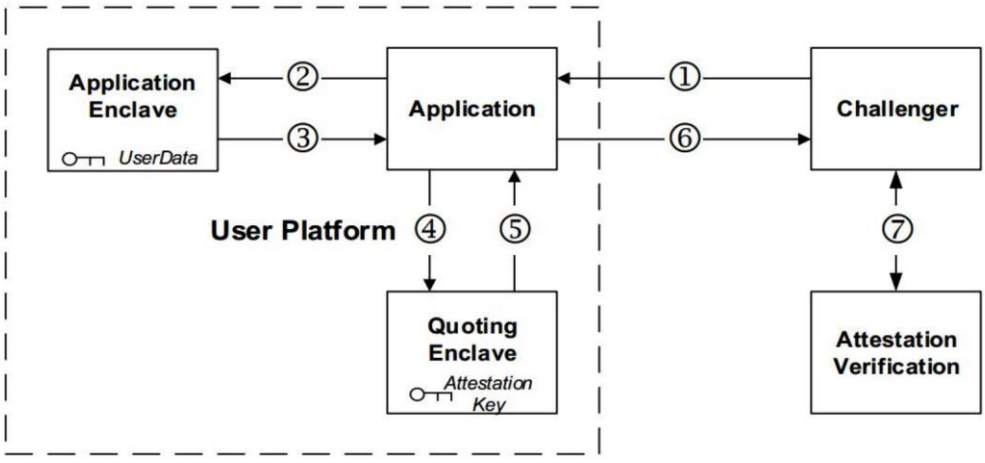

Figure 8

There are mainly three platforms involved in Remote Attestation:

- The service provide (challenger)
- The application with its enclave and its QE
- Intel Attestation Service (IAS) that verifies the enclave Stages

And it can be used only when the requirement of running on an Intel CPU, that supports SGX with Flexible Launch Control (FLC), is fulfilled.

Regarding the Figure 8, these are the stages for the remote attestation:

1. At first, the ISV enclave sends out an initial request to the remote service provider, which includes the EPID group the platform claims to currently be a member of.
2. If the service provider wishes to serve members of the claimed group, it may proceed by requesting an updated SigRL from the IAS.
3. The service provider then constructs a challenge message that consists its SPID, a liveliness random nonce, the updated SigRL and an optional basename parameter (if a pseudonym signature is required).
4. If the enclave supports the requested signature mode, it invokes the EREPORT instruction to create a locally verifiable report addressed to platform's QE. In order to establish an authenticated secure channel between the enclave and the service provider, a freshly generated ephemeral public key may be added to the report's user data field. The report and SP's challenge are sent to QE.
5. (Local Attestation happens here) The QE calls EGETKEY to obtain the REPORT KEY and verifies the report. If successful, QE calls EGETKEY again to receive platform's Provisioning Seal Key to decrypt platform's remote attestation key (EPID private key). The attestation key is first used to produce an identity signature by either signing the challenged basename or a random value, according to the attestation mode requested.
6. The attestation key is then used to compute two signatures of knowledge over the platform's identity signature MRENCLAVE. The first proves the identity signature was signed with a key certified by Intel. The second is a non-revoked proof that proves the key used for the identity signature does not create any of the identity signatures listed in the challenged SigRL. A final QUOTE is then

generated and encrypted using IAS's public key, which is hardcoded in QE, and the result is sent back to the attesting enclave. The QUOTE holds the identity of the attesting enclave, execution mode details (e.g. SVN level) and additional data.

7. The enclave then forwards the QUOTE to the SP for verification.
8. Since the QUOTE is encrypted, it is verifiable exclusively by Intel. Hence, the service provider simply forwards the QUOTE to the IAS for verification.
9. The IAS examines the QUOTE by first validating its EPID proofs against its identity signature. It then verifies the platform is not listed on the group Priv-RL by computing an identity signature on the QUOTE basename for each private key in the list and verifying that none of them equals the QUOTE's identity signature. This finalizes the validity check of the platform, and the IAS then creates a new attestation verification report as a response to the SP. The Attestation Verification Report includes the QUOTE structure generated by the platform for the attesting enclave.
10. A positive Attestation Verification Report confirms the enclave as running a particular piece of code on a genuine intel SGX processor. It is then the responsibility of the SP to validate the ISV enclave identity and serve an appropriate response back to the platform. [8]

Asylo Assertion Generator Enclave

The Assertion Generator Enclave (AGE) is a quoting enclave that is part of the Asylo framework. It is written using the Asylo framework and, consequently, has a larger Trusted Computing Base (TCB) than the Intel QE. The AGE TCB notably includes:

- Protobuf (like all Asylo enclaves)
- Abseil (like all Asylo enclaves)
- gRPC
- BoringSSL

The AGE runs a long-lived gRPC attestation server. An enclave can make an RPC to this service to obtain an attestation. Due to running an RPC server, the AGE must be run as a long-lived process (like a system daemon). See instructions below for how to run it.

The AGE defines various entry-points for key and certificate management using the Asylo API. This includes support for co-certification: Asylo users can extend the AGE's logic to allow for custom certification logic.

The AGE is recommended for any Asylo applications that make use of Asylo's gRPC authentication mechanism (and therefore already take a dependency on gRPC), as well as for users that want to manage their own SGX PKI. [9]

Implementations

It is aforementioned that Google provides an open-source API for the features of Asylo. Based on the extended documentation [10] and the guidelines provided through the respective Github repository [11], we implemented our demo. This demo includes all the necessary files, libraries and interdependencies in order to ensure the appropriate compilation of the project. The main files of our demo are the CryptoMain and the CryptoFunctions files, which are accompanied by the imperative files of WORKSPACE, BUILD and CryptoSelection.

To be more specific, following the guidelines of the Github repository and making the necessary file-matching, CryptoMain file is our driver and CryptoFunctions file is the participating enclave. In order to complete the building process, we altered the files of WORKSPACE and BUILD to fulfil our case and finally CryptoSelection is used for message parsing.

In the following sections, we provided the execution command and the whole project code, which can be also found in our personal Github repository [12], with the respective results depicted in the screenshots. The context of our project is the implementation of the cryptographic functions of MD5, SHA1, SHA512 AES encryption & decryption with the utilities of ECB & CBC, RSA encryption & decryption and the Diffie Hellman Key Exchange.

Regarding the Remote Attestation, we followed the documentation provided by the Google Asylo [13] and provided the results in the respective subsection via screenshots.

Code Review

Supposing having installed all the necessary requirements of the open-source API, run the following execution command:

```
docker run -it --rm     -v bazel-cache:/root/.cache/bazel     -v
"/home/${USER}/our_code/asylo":/opt/my-project     -w /opt/my-project     gcr.io/asylo-
framework/asylo     bazel run
//network_security_semester_project:network_security_semester_project_sgx_sim -- --
md5="STRING1" --sha1="STRING2" --sha512="STRING3" --aes="STRING4" --rsa="STRING5" --
dh="STRING6"
```

In order to run the demo, simply change the STRINGx at each parameter. The parameters are related to the cryptographic functions we abovementioned.

Below, we present the code of each file in order to compile and run our demo.

BUILD

```
load("@linux_sgx//:sgx_sdk.bzl", "sgx")
load("@rules_cc//cc:defs.bzl", "cc_proto_library")
load("@rules_proto//proto:defs.bzl", "proto_library")
load("@com_google_asylo//asylo/bazel:asylo.bzl", "cc_unsigned_enclave",
"debug_sign_enclave", "enclave_loader")


licenses(["notice"])

package(
    default_visibility = [
        "@com_google_asylo//asylo:implementation",
    ],
)

# Example and exercise for using Asylo toolkits.
proto_library(
    name = "CryptoSelection_proto",
    srcs = ["CryptoSelection.proto"],
    deps = ["@com_google_asylo//asylo:enclave_proto"],
)

cc_proto_library(
    name = "CryptoSelection_cc_proto",
    deps = [":CryptoSelection_proto"],
)
```

```
cc_unsigned_enclave(
    name = "CryptoFunctions_unsigned.so",
    srcs = ["CryptoFunctions.cc"],
    deps = [
        ":CryptoSelection_cc_proto",
        "@com_google_absl//absl/base:core_headers",
        "@com_google_absl//absl/strings",
        "@com_google_asylo//asylo:enclave_runtime",
        "@com_google_asylo//asylo/crypto:aead_cryptor",
        "@com_google_asylo//asylo/util:cleansing_types",
        "@com_google_asylo//asylo/util:status",
    ],
)

debug_sign_enclave(
    name = "CryptoFunctions.so",
    unsigned = "CryptoFunctions_unsigned.so",
)

enclave_loader(
    name = "network_security_semester_project",
    srcs = ["CryptoMain.cc"],
    backends = sgx.backend_labels,  # Has SGX loader dependencies
    enclaves = {"enclave": ":CryptoFunctions.so"},
    loader_args = ["--enclave_path='{enclave}'"],
    deps = [
        ":CryptoSelection_cc_proto",
        "@com_google_absl//absl/flags:flag",
        "@com_google_absl//absl/flags:parse",
        "@com_google_asylo//asylo:enclave_cc_proto",
        "@com_google_asylo//asylo:enclave_client",
        "@com_google_asylo//asylo/util:logging",
    ] + select(
        {
            "@linux_sgx//:sgx_hw":
["@com_google_asylo//asylo/platform/primitives/sgx:loader_cc_proto"],
            "@linux_sgx//:sgx_sim":
["@com_google_asylo//asylo/platform/primitives/sgx:loader_cc_proto"],
        },
        no_match_error = "network_security_semester_project",
    ),
)
```

WORKSPACE

```
 workspace(name = "network_security_semester_project")

load("@bazel_tools//tools/build_defs/repo:http.bzl", "http_archive")

# Download and use the Asylo SDK.
http_archive(
    name = "com_google_asylo",
    sha256 = "21e2798a345d4d508c4f09c4b92c7cdc06a445d8fca4094e3751cde48d1d27c9",
    strip_prefix = "asylo-0.6.3",
    urls = ["https://github.com/google/asylo/archive/v0.6.3.tar.gz"],
)

load(
    "@com_google_asylo//asylo/bazel:asylo_deps.bzl",
    "asylo_deps",
    "asylo_testonly_deps",
)
```

```
asylo_deps()

asylo_testonly_deps()

# sgx_deps is only needed if @linux_sgx is used.
load("@com_google_asylo//asylo/bazel:sgx_deps.bzl", "sgx_deps")

sgx_deps()

# remote_deps is only needed if remote backend is used.
load("@com_google_asylo//asylo/bazel:remote_deps.bzl", "remote_deps")

remote_deps()

# grpc_deps is only needed if gRPC is used. Projects using gRPC as an external
# dependency must call both grpc_deps() and grpc_extra_deps().
load("@com_github_grpc_grpc//bazel:grpc_deps.bzl", "grpc_deps")

grpc_deps()

load("@com_github_grpc_grpc//bazel:grpc_extra_deps.bzl", "grpc_extra_deps")

grpc_extra_deps()
```

CryptoSelection.proto

```
 syntax = "proto2";

package guide.asylo;

import "asylo/enclave.proto";

option java_package = "com.example";
option java_multiple_files = true;
option java_outer_classname = "EnclaveDemoExtension";

// A custom message to pass in and out of our enclave.
message Demo {
  // This string value is used for both user input and enclave output.
  optional string value = 1;
  enum Action {
    UNIDENTIFIED = 0;
    MD5 = 3;
    SHA1 = 4;
    SHA512 = 5;
    RSA = 6;
    AES = 7;
    DH = 8;
  }

  optional Action action = 2;
}

// The EnclaveInput message that is passed to the enclave can be extended with
// a Demo message to communicate a value our enclave knows to expect.
extend .asylo.EnclaveInput {
  optional Demo quickstart_input = 9001;
}

// The EnclaveOutput message that is passed out of the enclave can be extended
// with a Demo message to communicate a value our driver knows to expect.
extend .asylo.EnclaveOutput {
  optional Demo quickstart_output = 9001;
}
```

CryptoMain.cc

```
 #include <iostream>
#include <string>

#include "absl/flags/flag.h"
#include "absl/flags/parse.h"
#include "asylo/client.h"
#include "asylo/platform/primitives/sgx/loader.pb.h"
#include "asylo/util/logging.h"

#include "network_security_semester_project/CryptoSelection.pb.h"

ABSL_FLAG(std::string, enclave_path, "", "Path to enclave binary image to load");

ABSL_FLAG(std::string, sha1, "", "Message to encrypt with sha1");
ABSL_FLAG(std::string, sha512, "", "Message to encrypt with sha512");
ABSL_FLAG(std::string, md5, "", "Message to encrypt with md5");
ABSL_FLAG(std::string, rsa, "", "Message to encrypt-decrypt with rsa");
ABSL_FLAG(std::string, aes, "", "Message to encypt-decrypt with aes");
ABSL_FLAG(std::string, dh, "", "Keys exchanged with DH");


// Populates |enclave_input|->value() with |user_message|.
void SetEnclaveUserMessage(asylo::EnclaveInput *enclave_input,
  const std::string &user_message,
  guide::asylo::Demo::Action action) {
    guide::asylo::Demo *user_input = enclave_input-
>MutableExtension(guide::asylo::quickstart_input);
    user_input->set_value(user_message);
    user_input->set_action(action);
}


// Retrieves encrypted message from |output|. Intended to be used by the reader
// for completing the exercise.
const std::string GetEnclaveOutputMessage(const asylo::EnclaveOutput &output) {
  return output.GetExtension(guide::asylo::quickstart_output).value();
}


int main(int argc, char *argv[]) {
  absl::ParseCommandLine(argc, argv);

  constexpr char kEnclaveName[] = "trusted_enclave";

  const std::string enclave_path = absl::GetFlag(FLAGS_enclave_path);
  LOG_IF(QFATAL, absl::GetFlag(FLAGS_md5).empty() &&
                 absl::GetFlag(FLAGS_sha1).empty() &&
                 absl::GetFlag(FLAGS_sha512).empty() &&
                 absl::GetFlag(FLAGS_rsa).empty() &&
                 absl::GetFlag(FLAGS_aes).empty() &&
                 absl::GetFlag(FLAGS_dh).empty())
      << "At least one of the following flags should be specified: --sha1, --sha512, --
md5, --rsa, --rsa, --aes, --dh. ";


  // Part 1: Initialization

  // Prepare |EnclaveManager| with default |EnclaveManagerOptions|
  asylo::EnclaveManager::Configure(asylo::EnclaveManagerOptions());
  auto manager_result = asylo::EnclaveManager::Instance();
  LOG_IF(QFATAL, !manager_result.ok()) << "Could not obtain EnclaveManager";
```

```
  // Prepare |load_config| message.
  asylo::EnclaveLoadConfig load_config;
  load_config.set_name(kEnclaveName);


  // Prepare |sgx_config| message.
  auto sgx_config = load_config.MutableExtension(asylo::sgx_load_config);
  sgx_config->set_debug(true);
  auto file_enclave_config = sgx_config->mutable_file_enclave_config();
  file_enclave_config->set_enclave_path(enclave_path);


  // Load Enclave with prepared |EnclaveManager| and |load_config| message.
  asylo::EnclaveManager *manager = manager_result.ValueOrDie();
  auto status = manager->LoadEnclave(load_config);
  LOG_IF(QFATAL, !status.ok()) << "LoadEnclave failed with: " << status;


  // Part 2: Secure execution

  // Prepare |input| with |message| and create |output| to retrieve response
  // from enclave.
  asylo::EnclaveInput input;
  asylo::EnclaveOutput output;


  // Get |EnclaveClient| for loaded enclave and execute |EnterAndRun|.
  asylo::EnclaveClient *const client = manager->GetClient(kEnclaveName);
  status = client->EnterAndRun(input, &output);

  if (!absl::GetFlag(FLAGS_md5).empty()) {
    SetEnclaveUserMessage(&input, absl::GetFlag(FLAGS_md5), guide::asylo::Demo::MD5);
    status = client->EnterAndRun(input, &output);
    LOG_IF(QFATAL, !status.ok()) << "EnterAndRun failed with: " << status;
    std::cout << "Encrypt with md5:" << std::endl
              << GetEnclaveOutputMessage(output) << std::endl;
  }

  if (!absl::GetFlag(FLAGS_sha1).empty()) {
    SetEnclaveUserMessage(&input, absl::GetFlag(FLAGS_sha1), guide::asylo::Demo::SHA1);
    status = client->EnterAndRun(input, &output);
    LOG_IF(QFATAL, !status.ok()) << "EnterAndRun failed with: " << status;
    std::cout << "Encrypt with sha1:" << std::endl
              << GetEnclaveOutputMessage(output) << std::endl;
  }

  if (!absl::GetFlag(FLAGS_sha512).empty()) {
    SetEnclaveUserMessage(&input, absl::GetFlag(FLAGS_sha512),
guide::asylo::Demo::SHA512);
    status = client->EnterAndRun(input, &output);
    LOG_IF(QFATAL, !status.ok()) << "EnterAndRun failed with: " << status;
    std::cout << "Encrypt with sha512:" << std::endl
              << GetEnclaveOutputMessage(output) << std::endl;
  }

  if (!absl::GetFlag(FLAGS_rsa).empty()) {
    SetEnclaveUserMessage(&input, absl::GetFlag(FLAGS_rsa), guide::asylo::Demo::RSA);
    status = client->EnterAndRun(input, &output);
    LOG_IF(QFATAL, !status.ok()) << "EnterAndRun failed with: " << status;
    std::cout << "RSA encryption - decryption:" << std::endl
              << GetEnclaveOutputMessage(output) << std::endl;
  }

  if (!absl::GetFlag(FLAGS_aes).empty()) {
    SetEnclaveUserMessage(&input, absl::GetFlag(FLAGS_aes), guide::asylo::Demo::AES);
    status = client->EnterAndRun(input, &output);
    LOG_IF(QFATAL, !status.ok()) << "EnterAndRun failed with: " << status;
```

```
    std::cout << "AES encryption - decryption:" << std::endl
              << GetEnclaveOutputMessage(output) << std::endl;
  }
  if (!absl::GetFlag(FLAGS_dh).empty()) {
    SetEnclaveUserMessage(&input, absl::GetFlag(FLAGS_dh), guide::asylo::Demo::DH);
    status = client->EnterAndRun(input, &output);
    LOG_IF(QFATAL, !status.ok()) << "EnterAndRun failed with: " << status;
    std::cout << "Diffie Hellman:" << std::endl
              << GetEnclaveOutputMessage(output) << std::endl;
  }


  // Part 3: Finalization

  // |DestroyEnclave| before exiting program.
  asylo::EnclaveFinal empty_final_input;
  status = manager->DestroyEnclave(client, empty_final_input);
  LOG_IF(QFATAL, !status.ok()) << "DestroyEnclave failed with: " << status;

  return 0;
}
```

CryptoFunctions.cc

```
#include <string>
#include <vector>

#include "absl/base/macros.h"
#include "absl/strings/escaping.h"
#include "absl/strings/str_cat.h"
#include "asylo/crypto/aead_cryptor.h"
#include "asylo/crypto/util/byte_container_view.h"
#include "asylo/trusted_application.h"
#include "asylo/util/cleansing_types.h"
#include "asylo/util/status_macros.h"
#include "asylo/util/statusor.h"
#include "network_security_semester_project/CryptoSelection.pb.h"

#include <openssl/rand.h>
#include <openssl/rsa.h>
#include <openssl/pem.h>
#include <openssl/err.h>
#include <sys/stat.h>
#include <unistd.h>
#include <string>
#include <iostream>
#include <fstream>
#include <openssl/md5.h>
#include "openssl/sha.h"
#include "openssl/aes.h"
#include <openssl/dh.h>
#include <openssl/engine.h>


#define KEY_LENGTH 2048 // RSA Key length
#define PUB_KEY_FILE "pubkey.pem" // RSA public key path
#define PRI_KEY_FILE "prikey.pem" // RSA private key path

namespace asylo {
          namespace {
            // Dummy 128-bit AES key.
            constexpr uint8_t kAesKey128[] = {0x00, 0x01, 0x02, 0x03, 0x04, 0x05,
                                              0x06, 0x07, 0x08, 0x09, 0x10, 0x11,
                                              0x12, 0x13, 0x14, 0x15};
```

```
static void print_secret(unsigned char *sec, size_t len)
{
    size_t i;

    for (i = 0; i < len; ++i)
    printf("%x", sec[i]);

    printf("\n");
}
static int CreateDiffieHellman()
{
    DH *dh1, *dh2;
    //BIGNUM *p, *g;
    unsigned char *sec1, *sec2;
    size_t size;
    int ret;


    dh1 = DH_new();
    dh2 = DH_new();
    DH_generate_parameters_ex(dh1, 64, DH_GENERATOR_2, NULL);


    /* 1. set keys */
    ret = DH_generate_key(dh1);
    /* 2. set shared parameter p & g*/
    dh2->p = BN_dup(dh1->p);
    dh2->g = BN_dup(dh1->g);
    if (ret == 0) {
    fprintf(stderr, "dh1 DH_generate_key\n");
    exit(EXIT_FAILURE);
    }
    ret = DH_generate_key(dh2);
    if (ret == 0) {
    fprintf(stderr, "dh2 DH_generate_key\n");
    exit(EXIT_FAILURE);
    }

    /* 3. compute shared secret */
    size = DH_size(dh1);
    if (size != DH_size(dh2)) {
    fprintf(stderr, "size does not match!\n");
    exit(EXIT_FAILURE);
    }
    sec1 = (unsigned char *)malloc(size);
    sec2 = (unsigned char *)malloc(size);
    if (!sec1 || !sec2) {
    perror("malloc");
    exit(EXIT_FAILURE);
    }
    ret = DH_compute_key(sec1, dh2->pub_key, dh1);
    if (ret == -1) {
    fprintf(stderr, "DH_compute_key");
    exit(EXIT_FAILURE);
    }
    ret = DH_compute_key(sec2, dh1->pub_key, dh2);
    if (ret == -1) {
    fprintf(stderr, "DH_compute_key");
    exit(EXIT_FAILURE);
    }

    /* 4. compare shared secret */
    printf("shared secret 1\n");
    print_secret(sec1, size);
    printf("shared secret 2\n");
    print_secret(sec2, size);

    if (memcmp(sec1, sec2, size) == 0)
    ret = 1;
```

```
                else
                ret = 0;

                free(sec2);
                free(sec1);
                DH_free(dh2);
                DH_free(dh1);

                return ret;
            }
            void AesAll(std::string txt){

                uint8_t Key[32];
                uint8_t IV[AES_BLOCK_SIZE]; // Generate an AES Key
                RAND_bytes(Key, sizeof(Key));   // and Initialization Vector
                RAND_bytes(IV, sizeof(IV)); //

                // Make a copy of the IV to IVd as it seems to get destroyed when used
                uint8_t IVd[AES_BLOCK_SIZE];
                for(int i=0; i < AES_BLOCK_SIZE; i++){
                    IVd[i] = IV[i];
                }

                /** Setup the AES Key structure required for use in the OpenSSL APIs **/
                AES_KEY* AesKey = new AES_KEY();
                AES_set_encrypt_key(Key, 256, AesKey);

                /** take an input string and pad it so it fits into 16 bytes (AES Block
Size) **/
                const int UserDataSize = (const int)txt.length();   // Get the length
pre-padding
                int RequiredPadding = (AES_BLOCK_SIZE - (txt.length() %
AES_BLOCK_SIZE));   // Calculate required padding
                std::vector<unsigned char> PaddedTxt(txt.begin(), txt.end());   //
Easier to Pad as a vector
                for(int i=0; i < RequiredPadding; i++){
                    PaddedTxt.push_back(0); //  Increase the size of the string by
                }                           //  how much padding is necessary

                unsigned char * UserData = &PaddedTxt[0];// Get the padded text as an
unsigned char array
                const int UserDataSizePadded = (const int)PaddedTxt.size();// and the
length (OpenSSl is a C-API)

                /** Peform the encryption **/
                unsigned char EncryptedData[512] = {0};
                AES_cbc_encrypt(UserData, EncryptedData, UserDataSizePadded, (const
AES_KEY*)AesKey, IV, AES_ENCRYPT);
                std::cout<< "aes cbc enc -> " ;
                printf("hashedChars: ");
                int data_length = strlen((char*)EncryptedData);
                for (int i = 0; i < data_length; i++) {
                  printf("%x", EncryptedData[i]);
                }
                printf("\n");
                //AES_cbc_encrypt(const unsigned char *in, unsigned char *out,size_t
length, const AES_KEY *key,unsigned char *ivec, const int enc);

                /** Setup an AES Key structure for the decrypt operation **/
                AES_KEY* AesDecryptKey = new AES_KEY(); // AES Key to be used for
Decryption
                AES_set_decrypt_key(Key, 256, AesDecryptKey);   // We Initialize this so
we can use the OpenSSL Encryption API

                /** Decrypt the data. Note that we use the same function call. Only
change is the last parameter **/
                unsigned char DecryptedData[512] = {0};
                AES_cbc_encrypt(EncryptedData, DecryptedData, UserDataSizePadded, (const
AES_KEY*)AesDecryptKey, IVd, AES_DECRYPT);
```

```
                std::cout<< "aes cbc dec -> " << DecryptedData << std::endl;

                /* encrypt ecb */
                //AES_ecb_encrypt(const unsigned char *in, unsigned char *out, const
AES_KEY *key, const int enc);
                unsigned char EncryptedDataecb[512] = {0};
                AES_ecb_encrypt(UserData, EncryptedDataecb, (const AES_KEY *) AesKey,
AES_ENCRYPT);
                std::cout<< "aes ecb enc -> " ;
                printf("hashedChars: ");
                data_length = strlen((char*)EncryptedDataecb);
                for (int i = 0; i < data_length; i++) {
                  printf("%x", EncryptedDataecb[i]);
                }
                printf("\n");

                /* decrypt ecb*/
                unsigned char DecryptedDataecb[512] = {0};
                AES_ecb_encrypt(EncryptedDataecb, DecryptedDataecb, (const
AES_KEY*)AesDecryptKey, AES_DECRYPT);
                std::cout<< "aes ecb dec -> " << DecryptedDataecb << std::endl;

            }
            /*
             @brief: signs and verifies a givens string
             @para: data, the string to be signed and verified
            **/
            void doSign(std::string data)
            {
                RSA * pubkey=NULL;
                RSA * prikey=NULL;
                FILE *pbkey_from_file = fopen(PUB_KEY_FILE,"r");
                FILE *pkey_from_file = fopen(PRI_KEY_FILE,"r");
                unsigned char digest[SHA512_DIGEST_LENGTH];
                unsigned char sign[512];
                unsigned int signLen;
                int     ret;
                // read public key from file for verifying signature
                pubkey = PEM_read_RSAPublicKey(pbkey_from_file,&pubkey,NULL,NULL);
                // read private key from file for signing
                prikey = PEM_read_RSAPrivateKey(pkey_from_file,&prikey,NULL,NULL);

                // caclculate the hash of the messsage to be signed
                SHA512_CTX ctx;
                SHA512_Init(&ctx);
                SHA512_Update(&ctx,(const char *) data.c_str(), strlen(data.c_str()));
                SHA512_Final(digest, &ctx);

                unsigned char mdString[SHA512_DIGEST_LENGTH*2+1];
                for (int i = 0; i < SHA512_DIGEST_LENGTH; i++)
                    sprintf((char *) &mdString[i*2], "%02x", (unsigned int)digest[i]);

                /* Sign the hash*/
                ret = RSA_sign(NID_sha512 , mdString, SHA512_DIGEST_LENGTH, sign,
                               &signLen, prikey);
                //printf("Signature length = %d\n", signLen);
                printf("RSA_sign: %s\n", (ret == 1) ? "OK" : "NG");

                /* Verify signature*/
                ret = RSA_verify(NID_sha512, mdString, SHA512_DIGEST_LENGTH, sign,
                                 signLen, pubkey);
                printf("RSA_Verify: %s\n", (ret == 1) ? "true" : "false");

                fclose(pbkey_from_file);
                fclose(pkey_from_file);
            }

            /*
             @brief: private key encryption
```

```
             @para: clear_text -[i] The clear text that needs to be encrypted
                               pri_key -[i] private key
             @return: Encrypted data
            **/
            std::string RsaPriEncrypt(const std::string &clear_text, std::string
&pri_key)
            {
                std::string encrypt_text;
                BIO *keybio = BIO_new_mem_buf((unsigned char *)pri_key.c_str(), -1);
                RSA* rsa = RSA_new();
                rsa = PEM_read_bio_RSAPrivateKey(keybio, &rsa, NULL, NULL);
                if (!rsa)
                {
                    BIO_free_all(keybio);
                    return std::string("");
                }

                 // Get the maximum length of data that RSA can process at a time
                int len = RSA_size(rsa);

                 // Apply for memory: store encrypted ciphertext data
                char *text = new char[len + 1];
                memset(text, 0, len + 1);

                 // Encrypt the data with a private key (the return value is the length
of the encrypted data)
                int ret = RSA_private_encrypt(clear_text.length(), (const unsigned
char*)clear_text.c_str(), (unsigned char*)text, rsa, RSA_PKCS1_PADDING);
                if (ret >= 0) {
                    encrypt_text = std::string(text, ret);
                }

                 // release memory
                free(text);
                BIO_free_all(keybio);
                RSA_free(rsa);

                return encrypt_text;
            }
            /*
             @brief: public key decryption
             @para: cipher_text -[i] encrypted ciphertext
                               pub_key -[i] public key
             @return: decrypted data
            **/
            std::string RsaPubDecrypt(const std::string & cipher_text, const std::string
& pub_key)
            {
                std::string decrypt_text;
                BIO *keybio = BIO_new_mem_buf((unsigned char *)pub_key.c_str(), -1);
                RSA *rsa = RSA_new();

                 // Note--------Use the public key in the first format for decryption
                rsa = PEM_read_bio_RSAPublicKey(keybio, &rsa, NULL, NULL);
                 // Note--------Use the public key in the second format for decryption
(we use this format as an example)
                //rsa = PEM_read_bio_RSA_PUBKEY(keybio, &rsa, NULL, NULL);
                if (!rsa)
                {
                     unsigned long err = ERR_get_error(); //Get the error number
                    char err_msg[1024] = { 0 };
                            ERR_error_string(err, err_msg); // Format: error:errId:
library: function: reason
                    printf("err msg: err:%ld, msg:%s\n", err, err_msg);
                    BIO_free_all(keybio);
                    return decrypt_text;
                }

                int len = RSA_size(rsa);
```

```
                char *text = new char[len + 1];
                memset(text, 0, len + 1);
                 // Decrypt the ciphertext
                int ret = RSA_public_decrypt(cipher_text.length(), (const unsigned
char*)cipher_text.c_str(), (unsigned char*)text, rsa, RSA_PKCS1_PADDING);
                if (ret >= 0) {
                    decrypt_text.append(std::string(text, ret));
                }

                 // release memory
                delete text;
                BIO_free_all(keybio);
                RSA_free(rsa);

                return decrypt_text;
            }

            /*
             Manufacturing key pair: private key and public key
            **/
            void GenerateRSAKey(std::string & out_pub_key, std::string & out_pri_key)
            {
                size_t pri_len = 0; // Private key length
                size_t pub_len = 0; // public key length
                char *pri_key = nullptr; // private key
                char *pub_key = nullptr; // public key
                RSA *keypair = NULL;
                BIGNUM *bne = NULL;
                int ret = 0;
                unsigned long e = RSA_F4;


                 // Generate key pair
                //RSA *keypair = RSA_generate_key(KEY_LENGTH, RSA_3, NULL, NULL);
                bne = BN_new();
                ret = BN_set_word(bne, e);
                keypair = RSA_new();
                ret = RSA_generate_key_ex(keypair, KEY_LENGTH, bne, NULL);

                BIO *pri = BIO_new(BIO_s_mem());
                BIO *pub = BIO_new(BIO_s_mem());

                     // Generate private key
                PEM_write_bio_RSAPrivateKey(pri, keypair, NULL, NULL, 0, NULL, NULL);
                     // Note------Generate the public key in the first format
                PEM_write_bio_RSAPublicKey(pub, keypair);
                     // Note------Generate the public key in the second format (this is
used in the code here)
                //PEM_write_bio_RSA_PUBKEY(pub, keypair);

                 // Get the length
                pri_len = BIO_pending(pri);
                pub_len = BIO_pending(pub);

                 // The key pair reads the string
                pri_key = (char *)malloc(pri_len + 1);
                pub_key = (char *)malloc(pub_len + 1);

                BIO_read(pri, pri_key, pri_len);
                BIO_read(pub, pub_key, pub_len);

                pri_key[pri_len] = '\0';
                pub_key[pub_len] = '\0';

                out_pub_key = pub_key;
                out_pri_key = pri_key;

                 // Write the public key to the file
                std::ofstream pub_file(PUB_KEY_FILE, std::ios::out);
```

```
    if (!pub_file.is_open())
    {
        perror("pub key file open fail:");
        return;
    }
    pub_file << pub_key;
    pub_file.close();

     // write private key to file
    std::ofstream pri_file(PRI_KEY_FILE, std::ios::out);
    if (!pri_file.is_open())
    {
        perror("pri key file open fail:");
        return;
    }
    pri_file << pri_key;
    pri_file.close();

     // release memory
    RSA_free(keypair);
    BIO_free_all(pub);
    BIO_free_all(pri);

    free(pri_key);
    free(pub_key);
}

std::string Md5(std::string input) {
    unsigned char digest[MD5_DIGEST_LENGTH];
    char string[] = "";

    strcpy( string, input.c_str() );
    MD5((unsigned char*)&string, strlen(string), (unsigned char*)&digest);

    char mdString[33];

    for(int i = 0; i < 16; i++)
         sprintf(&mdString[i*2], "%02x", (unsigned int)digest[i]);
    std::string md_string = std::string(mdString);
    return md_string;

}

std::string Sha1(std::string input) {

    unsigned char digest[SHA_DIGEST_LENGTH];

    SHA_CTX ctx;
    SHA1_Init(&ctx);
    SHA1_Update(&ctx,(const char *) input.c_str(), strlen(input.c_str()));
    SHA1_Final(digest, &ctx);

    char mdString[SHA_DIGEST_LENGTH*2+1];
    for (int i = 0; i < SHA_DIGEST_LENGTH; i++)
        sprintf(&mdString[i*2], "%02x", (unsigned int)digest[i]);

    std::string sha_string = std::string(mdString);
    return sha_string;

}

std::string Sha512(std::string input) {
    unsigned char digest[SHA512_DIGEST_LENGTH];

    SHA512_CTX ctx;
    SHA512_Init(&ctx);
    SHA512_Update(&ctx,(const char *) input.c_str(), strlen(input.c_str()));
    SHA512_Final(digest, &ctx);
```

```
                char mdString[SHA512_DIGEST_LENGTH*2+1];
                for (int i = 0; i < SHA512_DIGEST_LENGTH; i++)
                    sprintf(&mdString[i*2], "%02x", (unsigned int)digest[i]);

                std::string sha512_string = std::string(mdString);
                return sha512_string;
            //    printf("SHA512 digest: %s\n", mdString);

            }

            // Helper function that adapts absl::BytesToHexString, allowing it to be
used
            // with ByteContainerView.
            std::string BytesToHexString(ByteContainerView bytes) {
              return absl::BytesToHexString(absl::string_view(
                  reinterpret_cast<const char *>(bytes.data()), bytes.size()));
            }

            // Encrypts a message against `kAesKey128` and returns a 12-byte nonce
followed
            // by authenticated ciphertext, encoded as a hex string.
            const StatusOr<std::string> EncryptMessage(const std::string &message) {
              std::unique_ptr<AeadCryptor> cryptor;
              ASYLO_ASSIGN_OR_RETURN(cryptor,
                                     AeadCryptor::CreateAesGcmSivCryptor(kAesKey128));

              std::vector<uint8_t> additional_authenticated_data;
              std::vector<uint8_t> nonce(cryptor->NonceSize());
              std::vector<uint8_t> ciphertext(message.size() + cryptor-
>MaxSealOverhead());
              size_t ciphertext_size;

              ASYLO_RETURN_IF_ERROR(cryptor->Seal(
                  message, additional_authenticated_data, absl::MakeSpan(nonce),
                  absl::MakeSpan(ciphertext), &ciphertext_size));

              return absl::StrCat(BytesToHexString(nonce),
BytesToHexString(ciphertext));
            }

            // Decrypts a message using `kAesKey128`. Expects `nonce_and_ciphertext` to
be
            // encoded as a hex string, and lead with a 12-byte nonce. Intended to be
            // used by the reader for completing the exercise.
            const StatusOr<CleansingString> DecryptMessage(
                const std::string &nonce_and_ciphertext) {
              std::string input_bytes = absl::HexStringToBytes(nonce_and_ciphertext);

              std::unique_ptr<AeadCryptor> cryptor;
              ASYLO_ASSIGN_OR_RETURN(cryptor,
                                     AeadCryptor::CreateAesGcmSivCryptor(kAesKey128));

              if (input_bytes.size() < cryptor->NonceSize()) {
                return Status(
                    error::GoogleError::INVALID_ARGUMENT,
                    absl::StrCat("Input too short: expected at least ",
                                 cryptor->NonceSize(), " bytes, got ",
input_bytes.size()));
              }

              std::vector<uint8_t> additional_authenticated_data;
              std::vector<uint8_t> nonce = {input_bytes.begin(),
                                            input_bytes.begin() + cryptor->NonceSize()};
              std::vector<uint8_t> ciphertext = {input_bytes.begin() + cryptor-
>NonceSize(),
                                                 input_bytes.end()};

              // The plaintext is always smaller than the ciphertext, so use
              // `ciphertext.size()` as an upper bound on the plaintext buffer size.
```

```
            CleansingVector<uint8_t> plaintext(ciphertext.size());
            size_t plaintext_size;

            ASYLO_RETURN_IF_ERROR(cryptor->Open(ciphertext,
additional_authenticated_data,
                                                nonce, absl::MakeSpan(plaintext),
                                                &plaintext_size));

            return CleansingString(plaintext.begin(), plaintext.end());

          }
    } // namespace

    class EnclaveDemo : public TrustedApplication {
          public:
                    EnclaveDemo() = default;

                    Status Run(const EnclaveInput &input, EnclaveOutput *output) {
                              std::string user_message = GetEnclaveUserMessage(input);
                              std::string pubkey,prikey;
                              std::string encrypt_text;
                              std::string decrypt_text;


                              switch (GetEnclaveUserAction(input)) {
                              case guide::asylo::Demo::SHA1: {
                                        SetEnclaveOutputMessage(output,
Sha1(user_message));
                                        break;
                              }
                              case guide::asylo::Demo::SHA512: {
                                        SetEnclaveOutputMessage(output,
Sha512(user_message));
                              break;
                              }
                              case guide::asylo::Demo::MD5: {
                                        SetEnclaveOutputMessage(output,
Md5(user_message));
                              break;
                              }
                              case guide::asylo::Demo::RSA: {
                                        GenerateRSAKey(pubkey,prikey);
                                        encrypt_text = RsaPriEncrypt((user_message),
prikey);
                                        doSign(encrypt_text);
                                        decrypt_text = RsaPubDecrypt(encrypt_text,
pubkey);
                                        SetEnclaveOutputMessage(output,
decrypt_text);
                              break;
                              }
                              case guide::asylo::Demo::AES: {
                                        AesAll(user_message);
                                        SetEnclaveOutputMessage(output, "AES
Done");
                              break;
                              }
                              case guide::asylo::Demo::DH: {
                                        CreateDiffieHellman();
                                        SetEnclaveOutputMessage(output, "Diffie Hellman
Done");
                              break;
                              }
                              default:
                                        SetEnclaveOutputMessage(output, "ERROR - ON
SWITCH");
                    }
                              return absl::OkStatus();
                    }
```

```
      // Retrieves user message from |input|.
                    const std::string GetEnclaveUserMessage(const EnclaveInput &input) {
                              return
input.GetExtension(guide::asylo::quickstart_input).value();
                    }
      // Retrieves user action from |input|.
                    guide::asylo::Demo::Action GetEnclaveUserAction(const EnclaveInput
&input) {
                              return
input.GetExtension(guide::asylo::quickstart_input).action();
      }

      // Populates |enclave_output|->value() with |output_message|. Intended to be
      // used by the reader for completing the exercise.
                    void SetEnclaveOutputMessage(EnclaveOutput *enclave_output,
                                        const std::string &output_message) {
                    guide::asylo::Demo *output = enclave_output-
>MutableExtension(guide::asylo::quickstart_output);
                    output->set_value(output_message);
                    }
    };

    TrustedApplication *BuildTrustedApplication() { return new EnclaveDemo; }

}  // namespace asylo
```

Cryptographic Functions

The results of the previous presented code are shown in the following examples.

```
root@ubuntu:/home/alex# docker run -it --rm     -v bazel-cache:/root/.cache/bazel     -v "/home/alex/our_code/asylo":/opt/my-project     -w /opt/my-project     gcr.io/asylo-framework/asylo     bazel run
//network_security_semester_project:network_security_semester_project_sgx_sim -- --md5="alex"
Starting local Bazel server and connecting to it...
INFO: Analyzed target //network_security_semester_project:network_security_semester_project_sgx_sim (81 packages loaded, 4850 targets configured).
INFO: Found 1 target...
Target //network_security_semester_project:network_security_semester_project_sgx_sim up-to-date:
  bazel-bin/network_security_semester_project/network_security_semester_project_sgx_sim
INFO: Elapsed time: 9.596s, Critical Path: 0.16s
INFO: 1 process: 1 internal.
INFO: Build completed successfully, 1 total action
INFO: Build completed successfully, 1 total action
Encrypt with md5:
534b44a19bf18d20b71ecc4eb77c572f
```

Figure 9 - MD5

```
root@ubuntu:/home/alex# docker run -it --rm     -v bazel-cache:/root/.cache/bazel     -v "/home/alex/our_code/asylo":/opt/my-project     -w /opt/my-project     gcr.io/asylo-framework/asylo     bazel run
//network_security_semester_project:network_security_semester_project_sgx_sim -- --sha1="alex"
Starting local Bazel server and connecting to it...
INFO: Analyzed target //network_security_semester_project:network_security_semester_project_sgx_sim (81 packages loaded, 4850 targets configured).
INFO: Found 1 target...
Target //network_security_semester_project:network_security_semester_project_sgx_sim up-to-date:
  bazel-bin/network_security_semester_project/network_security_semester_project_sgx_sim
INFO: Elapsed time: 7.574s, Critical Path: 0.12s
INFO: 1 process: 1 internal.
INFO: Build completed successfully, 1 total action
INFO: Build completed successfully, 1 total action
Encrypt with sha1:
60c6d277a8bd81de7fdde19201bf9c58a3df08f4
```

Figure 10 - SHA1

```
root@ubuntu:/home/alex# docker run -it --rm     -v bazel-cache:/root/.cache/bazel     -v "/home/alex/our_code/asylo":/opt/my-project     -w /opt/my-project     gcr.io/asylo-framework/asylo     bazel run
//network_security_semester_project:network_security_semester_project_sgx_sim -- --sha512="alex"
Starting local Bazel server and connecting to it...
INFO: Analyzed target //network_security_semester_project:network_security_semester_project_sgx_sim (81 packages loaded, 4850 targets configured).
INFO: Found 1 target...
Target //network_security_semester_project:network_security_semester_project_sgx_sim up-to-date:
  bazel-bin/network_security_semester_project/network_security_semester_project_sgx_sim
INFO: Elapsed time: 7.463s, Critical Path: 0.11s
INFO: 1 process: 1 internal.
INFO: Build completed successfully, 1 total action
INFO: Build completed successfully, 1 total action
Encrypt with sha512:
35f319ca1dfc9689f5a33631c8f93ed7c3120ee7afa05b1672c7df7b71f63a6753def5fd3ac9db2eaf90ccab6bff31a486b51c7095ff958d228102b84efd7736
```

Figure 11 - SHA512

```
root@ubuntu:/home/alex# docker run -it --rm     -v bazel-cache:/root/.cache/bazel     -v "/home/alex/our_code/asylo":/opt/my-project     -w /opt/my-project     gcr.io/asylo-framework/asylo     bazel run
//network_security_semester_project:network_security_semester_project_sgx_sim -- --aes="alex"
Starting local Bazel server and connecting to it...
INFO: Analyzed target //network_security_semester_project:network_security_semester_project_sgx_sim (81 packages loaded, 4850 targets configured).
INFO: Found 1 target...
Target //network_security_semester_project:network_security_semester_project_sgx_sim up-to-date:
  bazel-bin/network_security_semester_project/network_security_semester_project_sgx_sim
INFO: Elapsed time: 7.966s, Critical Path: 0.16s
INFO: 1 process: 1 internal.
INFO: Build completed successfully, 1 total action
INFO: Build completed successfully, 1 total action
aes cbc enc -> hashedChars: ca11d87c12daaa222f1a74a8568c2f6
aes cbc dec -> alex
aes ecb enc -> hashedChars: 2cb318ad5dfbb7cb15517321903643ff
aes ecb dec -> alex
AES encryption - decryption:
AES Done
```

Figure 12 – AES

```
root@ubuntu:/home/alex# docker run -it --rm     -v bazel-cache:/root/.cache/bazel     -v "/home/alex/our_code/asylo":/opt/my-project     -w /opt/my-project     gcr.io/asylo-framework/asylo     bazel run
//network_security_semester_project:network_security_semester_project_sgx_sim -- --rsa="alex"
Starting local Bazel server and connecting to it...
INFO: Analyzed target //network_security_semester_project:network_security_semester_project_sgx_sim (81 packages loaded, 4850 targets configured).
INFO: Found 1 target...
Target //network_security_semester_project:network_security_semester_project_sgx_sim up-to-date:
  bazel-bin/network_security_semester_project/network_security_semester_project_sgx_sim
INFO: Elapsed time: 7.804s, Critical Path: 0.17s
INFO: 1 process: 1 internal.
INFO: Build completed successfully, 1 total action
INFO: Build completed successfully, 1 total action
RSA_sign: OK
RSA_Verify: true
RSA encryption - decryption:
alex
```

Figure 13 - RSA

```
root@ubuntu:/home/alex# docker run -it --rm     -v bazel-cache:/root/.cache/bazel     -v "/home/alex/our_code/asylo":/opt/my-project     -w /opt/my-project     gcr.io/asylo-framework/asylo     bazel run
//network_security_semester_project:network_security_semester_project_sgx_sim -- --dh="alex"
Starting local Bazel server and connecting to it...
INFO: Analyzed target //network_security_semester_project:network_security_semester_project_sgx_sim (81 packages loaded, 4850 targets configured).
INFO: Found 1 target...
Target //network_security_semester_project:network_security_semester_project_sgx_sim up-to-date:
  bazel-bin/network_security_semester_project/network_security_semester_project_sgx_sim
INFO: Elapsed time: 8.069s, Critical Path: 0.15s
INFO: 1 process: 1 internal.
INFO: Build completed successfully, 1 total action
INFO: Build completed successfully, 1 total action
shared secret 1
81f9e987a761074
shared secret 2
81f9e987a761074
Diffie Hellman:
Diffie Hellman Done
```

Figure 14 - Diffie Hellman

Remote Attestation

Since we initiated the Server-side and the Client-side for testing the Remote Attestation procedure using the technology of the Enclaves, the communication among the two sides is shown in the two figures below.

Server-side:

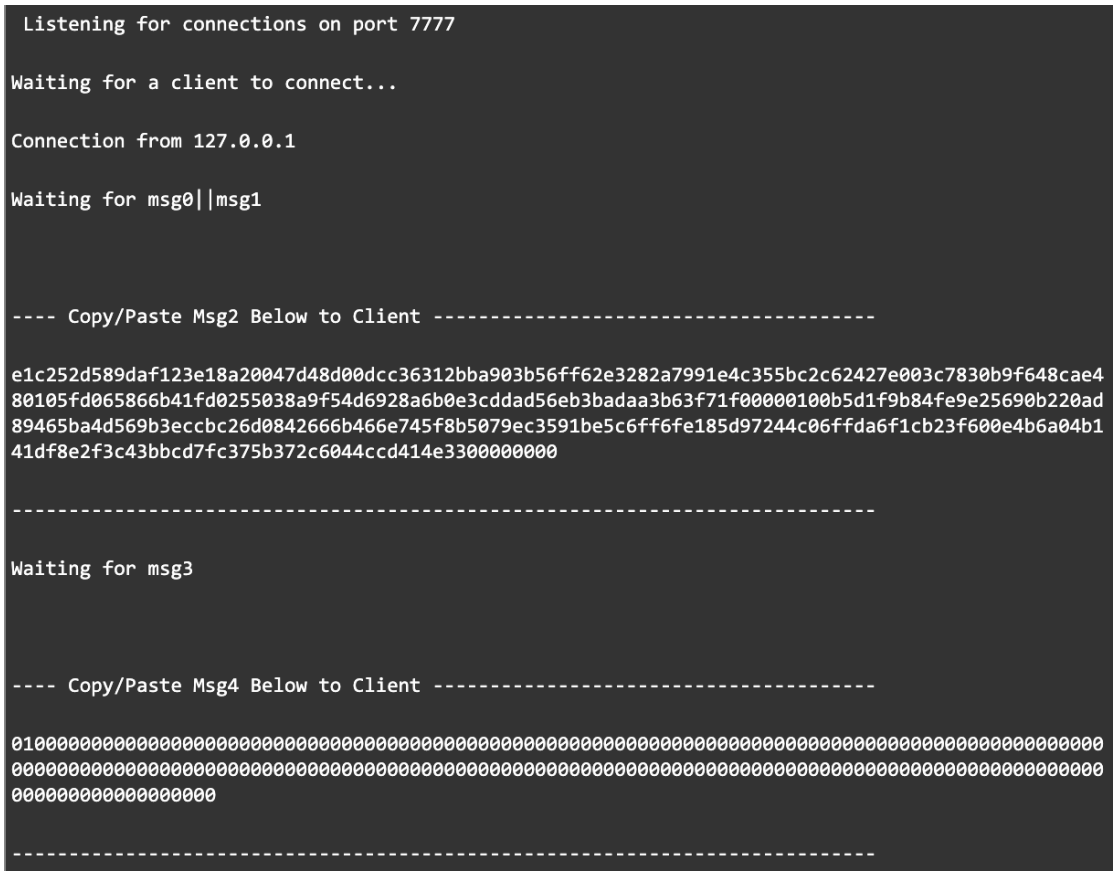

Figure 15

Client-side:

Figure 16

Conclusion

The Google Asylo is a powerful framework, which sets a strong basis for the development and extensive use of the technology of the Trusted Execution Environments. It is an emerging technology, which has great potential for the business-production community in the cyber-world in the decades to come. The provided features, such as the implementation of Enclaves and the Remote Attestation, are capable of ensuring the secure use and communication of many IoT devices and enhancing the cybersecurity of complex ecosystems. From our side, the experience during our demo implementations is more than exiting, as our engagement with the scientific area of the trusted execution environments and the frameworks that support them was superficial.

References

[1] Wiedmann, Florian, “What is a trusted execution environment (TEE) and how can it improve the safety of your data?”, 1 July 2021, https://piwik.pro/blog/what-is-a-trusted-execution-environment/ .

[2] Prado, Sergio, “Introduction to Trusted Execution Environment and ARM's TrustZone”, 16 March 2020, https://embeddedbits.org/introduction-to-trusted-execution-environment-tee-arm-trustzone/ .

[3] “About Asylo”, https://asylo.dev/about/ .

[4] “Asylo API Overview”, https://asylo.dev/docs/concepts/api-overview.html .

[5] “Asylo Overview”, https://asylo.dev/about/overview.html .

[6] “Security backends”, https://asylo.dev/about/overview.html#security-backends.

[7] “Local Attestation”, http://www.sgx101.com/portfolio/local_attestation/ .

[8] “Attestation”, https://sgx101.gitbook.io/sgx101/sgx-bootstrap/attestation / .

[9] “Asylo Assertion Generator Enclave”, https://asylo.dev/docs/concepts/remote_attestation.html#asylo-assertion-generator-enclave .

[10] “Asylo Docs”, https://asylo.dev/docs/ .

[11] “Asylo Github”, https://github.com/google/asylo .

[12] “Our Demo on Github”, https://github.com/giannischou/nssp .

[13] “Remote Attestation”, https://asylo.dev/docs/concepts/remote_attestation.html.