Comparison of TEE Integration on Applications

Jorge Pereira, Pedro Camponês, Rafael Sequeira
Departamento de Informática
FCT-UNL
Portugal
{jff.pereira,p.campones,rm.sequeira}@campus.fct.unl.pt

Abstract—When an entity who owns data that mustn't be disclosed is not in control of the infrastructure in which the data will be processed, there must be some guarantee that the data won't be tampered nor unduly disclosed.
With the advent of cloud services, the control of the infrastructure is migrating from its owners to the cloud providers, who in turn may modify the software being used by applications or use an invasive operating system.
The use of Trusted Execution Environments (TEE) allows blocks of code to run in protected regions, isolated from the remaining hardware. Data within the TEE is inaccessible from the outside, even to the operating system. This technology allows the code running in a protected environment to be attested, guaranteeing its authenticity. These characteristics protect the data in use in an application.
This report explores how to enable the use of a TEE in an application and how a careless integration of this technology affects the overall system.

1 Introduction

Confidentiality is a fundamental pillar of Computer Security, being one of the three elements that compose the *CIA Triad* [10]. For a system to provide Confidentiality, information must not be unduly disclosed.

In the competitive corporate world, the unwanted disclosure of data bears elevated costs, whether these emerge from a loss of advantage against competitors or fines resulting from a breach in users' privacy.

The *General Data Protection Regulation* (GDPR) is a set of regulations that protect the personal data of all citizens of the European Union; failure to comply with GDPR's demands leads to the issue of substantial fines to the entity at fault. Since GDPR's establishment in *2016*, Confidentiality has become a very relevant topic, and companies today strive to guarantee their users' data is kept protected [2].

There are three situations where data might be threatened: As it is in transit, stored, and as it is in use [11].

Cryptography is used to protect data both in transit and stored. When in transit, TLS provides an established communication protocol that uses encryption to provide data confidentiality [12]; and when stored, data can be protected using *Homomorphic Encryption*, a set of encryption algorithms that maintain some properties of the data being encrypted, and as such some processing can be done on data stored using this type of encryption.

Despite allowing some operations to be performed over encrypted data, *Homomorphic Encryption* does not provide the required freedom in the handling of data that most applications require. As such, at some point in these applications, data in use is not encrypted. Thus, *Homomorphic Encryption* is not an adequate solution to guarantee the protection of data in use.

The importance of protecting data as it is in use is diminished if the company involved has complete control over the infrastructure running its applications. Supposing a company has several servers being used in an application, for the data in use to be disclosed, its necessary that the software running in one or more of these servers be modified.

The necessity to tightly control the infrastructure in order to preclude the deployment of malicious code conflicts with the recent trends in the adoption of cloud solutions.

As computation is migrating towards the extremes, the edge and the cloud are gaining preponderance over traditional computing paradigms [1]. Cloud computing in particular is the subject of billion dollar investments by leading tech companies [14]; competition over who provides the best cloud solutions is fierce, leading to great benefits for the customers. The benefits provided by these solutions entail a competitive advantage for cloud services' customers.

Despite the advantages achieved by subscribing to these services, the lack of control over the infrastructure where applications' software is deployed is a deterrent to the further adoption of these services and the benefits entailed.

The field of *Confidential Computing* focuses on hardware solutions to guarantee software security, in particular, the confidentiality of data in use [11]. Following the standards set by this emergent field, critical software whose contents must be kept assured and confidential is run in a secure environment that cannot be compromised, the *Trusted Execution Environment* (TEE). There are several limitations to the use of this technology; as such, an application that desires to make use of TEE must be subject to architectural changes on its software. The modifications required to adapt an application to be able to make use of a TEE is the subject of this report, as well as the benchmark of the use of the application using a TEE in comparison with the previous, unmodified, version.

To accomplish the proposed objectives, a modified version of the application presented in WA3 [9] has been restructured to use the *Intel SGX* [5] TEE.

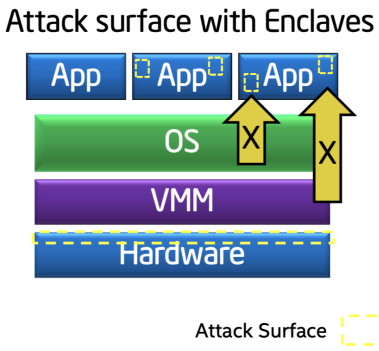

Fig. 1. Attack Surface when using SGX enclaves [retrived from [13]]

Section 2 presents a brief explanation of the SGX TEE. The auction application presented in WA3 as well as the increased security features achieved through the use of a TEE are the subject of sections 3 and 4, respectively.

Section 5 explains the process of integrating SGX in the application; both through a naive approach and by factorizing the code, minimizing the amount of code in the TEE. The performance of both approaches are compared in section 6.

Further improvements on the work performed are presented in section 7. In this section, flaws made when integrating a TEE are explored, as well as the use of other mechanisms, unrelated with TEE, to provide further security guarantees to the application.

Section 8 terminates the report presenting an overview of the work undertaken and extracting insight from it.

2 TEE Technology

In order to protect data as it is in use, it must run in an hardware isolated section, a Trusted Execution Environment (TEE). Using a TEE, the data used in the computations inside this protected environment is inaccessible even to the operating system, and its possible for the user to attest the code being run through cryptographic signatures; guaranteeing the code being used is that which the entity responsible for the data believes. These two properties guarantee that the data is protected even if it not encrypted, as it cannot be accessed directly through the addition of malicious code, for the code running is attested; and the data cannot be indirectly inferred as it is isolated inside the TEE.

The TEE *Intel Software Guard Extensions* (SGX) [5], is a popular solution to guarantee Confidential Computing in cloud services, having left the purely academic panorama and starting to get traction as a commercial venue. SGX guarantees the security of sensitive computations where all privileged software (kernel, hypervisor, etc.) is potentially malicious [5], thus guaranteeing the integrity and confidentiality for all data produced or used in the TEE.

The *x86* processor architecture provides a layered hierarchy of privileges. Each layer, designated *Ring* is responsible for a specific part of the operations executed by the processor; Ring 3 is used to run application specific code using instructions unable to harm the correct execution of the computer and Ring 0 can perform every instruction the

```
#
# A fatal error has been detected by the Java Runtime Environment:
#
#  SIGSEGV (0xb) at pc=0x0000001001be88e7, pid=6, tid=2
#
# JRE version: OpenJDK Runtime Environment (11.0.4+4) (build 11.0.4+4-alpine-r1)
# Java VM: OpenJDK 64-Bit Server VM (11.0.4+4-alpine-r1, mixed mode, tiered, compressed oops, g1 gc, linux-amd64)
# Problematic frame:
# j  org.springframework.util.ClassUtils.isAssignable(Ljava/lang/Class;Ljava/lang/Class;)Z+14
#
# Core dump will be written. Default location: core.6 (may not exist)
#
# An error report file with more information is saved as:
# /./hs_err_pid6.log
[SCONE|WARN] src/shielding/proc_fs.c:368:_proc_fs_open(): open: /proc/2/maps is not supported
^C[SCONE|FATAL] ./tools/starter-exec.c:97:exit_signal(): Aborted
^C^C
```

Fig. 2. SIGSEGV while executing a Python SmartContract

hardware offers, and is generally used for critical operations that could harm the system. Rings 1 and 2 have configurable permissions between those of Ring 3 and Ring 0.

The operations in Ring 0 are performed by the operating system and cannot be programmed by a user. Should an adversary run a trusted piece of software in a modified OS, the software might not perform as intended. The developer has no control or information about the code being run on an OS and cannot prevent an attack from this front.

SGX creates *Enclaves* in specific parts of code running in Ring 3. The code running in the Enclaves is inaccessible from code running in other Rings, thus protecting the software being used even when running it in an infrastructure not owned by the entity responsible for the data.

Fig. 1 represents the trust base of an application using Enclaves. The trust base is reduced from the application, OS, VMM, drivers, and hardware to only some parts of the applications running on the enclave and specific parts of the hardware. For these characteristics, SGX has been chosen as the TEE to guarantee the protection of data in use for the auction system application.

3 WA3: Auction Service

In WA3 [9] an application was presented which offers a secure auction service. The application uses BFT-SMaRt [3] to guarantee State Machine Replication in the face of Byzantine Faults. The application consists of three components: A client software, the application server, and a repository database that manages nonvolatile memory.

This application has been extended, having the objectives proposed as future work been completed. The application now provides security when a client performs a request to a Byzantine application server, by making the client responsible for verifying the signed answers of each application server replica. Another modification to the original WA3 design is the introduction of an access control mechanism based on *Smart Contracts*, a dynamic code injection mechanism.

The application components were programmed using the Spring Java Framework, thus inducing some overheads which are addressed in section 6.1. The database system being used in WA3 is MySQL, this system has been modified to MongoDB. All experimental results presented in the report account for the use of the MongoDB database system.

The data in use that is to be protected is in the software of the application servers. As such it's this part that will be modified to be placed in the TEE.

```
***************************
APPLICATION FAILED TO START
***************************

Description:

Failed to bind properties under 'server.ssl' to org.springframework.boot.web.server.Ssl:

    Reason: Failed to bind properties under 'server.ssl' to org.springframework.boot.web.server.Ssl

Action:

Update your application's configuration
```

Fig. 3. SSL Property could not be binded error

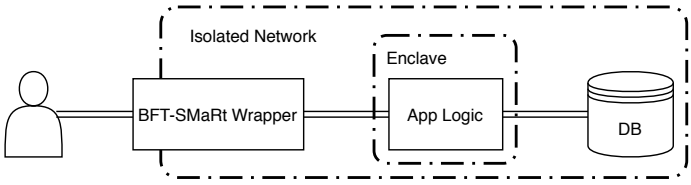

Fig. 4. Micro-service Architecture

4 Threat Model

All the threads presented in WA3's [9] Threat model are supported by this version of the application. Furthermore, the application now supports having the client propose requests to a Bizantine Replica and guarantees confidentiality to the data in use in the application server component.

To provide the confidentiality, it's assumed that the adversary may have access to superuser permissions and to the physical hardware. The adversary can control the entire stack, including privileged code. The threat model does not cover a physical attack that can pass through the SGX.

5 Enabling SGX

The use of SGX is dictated through the use of special instructions provided by CPUs supporting this technology. Prior to enabling code running in the SGX, it's necessary to reserve memory and load content into the enclave. After these steps are performed, the enclave can be initialized and the code placed inside can start to run. It is at this last stage that the code placed in the enclave generates the cryptographic proves required in order to be attested.

The added instructions to handle SGX demand that an application be modified in order to support this TEE. However, the handling of the SGX instructions can be abstracted through the use of frameworks.

Scone [4] is a secure container mechanism that protects the confidentiality and integrity of the memory, code and external and network I/O of a Linux process from unauthorized and potentially privileged adversaries, thus implementing an overlay over SGX to reduce the complexity of using its API. The use of Scone has the added advantage of improving performance by creating threads specialized in doing system calls that must leave the enclave.

With Scone, it's possible to run an application using SGX without requiring modifications on its software. However, despite the ease afforded by Scone, the naive use of this technology by blindly containerizing an application will result in several inefficiencies that may hinder the correct execution of the system.

To test the previous statement, two versions of the application server were containerized with Scone. One version consists of the original project with minimal modifications, and the second consists of an heavily modified version of the original, optimized for use in SGX.

In both versions the use of *Smart Contracts* mentioned in section 3 is incompatible with Scone. The dynamic injection of code that requires instructions that are not part of the Scone's SGX stack, resulting in the error presented in fig. 2.

5.1 Naive Version

Excluding small or very specialized devices, the size of the code being used to perform the desired computations is not an issue. However, with SGX this problem is a real deterrent, especially when using frameworks and libraries.

The memory that can be reserved for SGX is limited, since the *Processor Reserved Memory* (PRM) consists of merely *128 MB*, capping the amount of code that can be placed in an enclave. The application server in WA3 [9] was built with the aid of the Spring Boot framework for Java. Java's JVM and the Spring Boot Framework together comprise the vast majority of the code in the application. In addition the Scone framework, required to use enclaves without modifying the original software contributes to the bloated size of the code.

The containerization of the application server using these tools proved unstable. As such, an effort was made to reduce the applications' code dependencies to the minimum possible. Through this process, the original container size of *628 MB*, was skimmed to *413 MB*.

Despite the reduction on the code size in the enclave, it could be observed that the application produces a high number of page faults during execution, resulting in the execution of an excess number of cryptographic operations to evict and load memory pages from and to the enclave.

In the process of initializing the application server replicas, oftentimes the error presented in fig. 3 occurs. This problem has not been fixed and it's believed to result from an incompatibility between Spring Boot and Scone.

5.2 Microservices Version

The amount of code required by the application server is a deterrent to the use of JVM and Spring Boot. Given BFT-SMaRt's [3] implementation in Java, to avoid putting JVM code in SGX the application server must be decomposed into more than one monolithic program, such that one program not containing the BFT-SMaRt can be containerized with Scone. This program should be as light as possible, minimizing the dependencies used. These dependencies would increase the number of page faults, which result in very expensive cryptographic operation when using SGX.

The new application server architecture, presented in fig. 4, decomposes the monolithic program into two parts. One micro server, designated *App Logic*, which runs in the protected hardware (SGX), contains all the business logic and works with plaintext information; and *BFT-SMaRt Wrapper*, which contains the BFT-SMaRt logic.

The *BFT-SMaRt Wrapper* is a service that receives client requests, orders the requests using BFT-SMaRt, issues the received requests to the *App Logic* component, and aggregates the signed responses from the *App Logic*. Because *BFT.SMaRt Wrapper* does not run in SGX, the information contained in

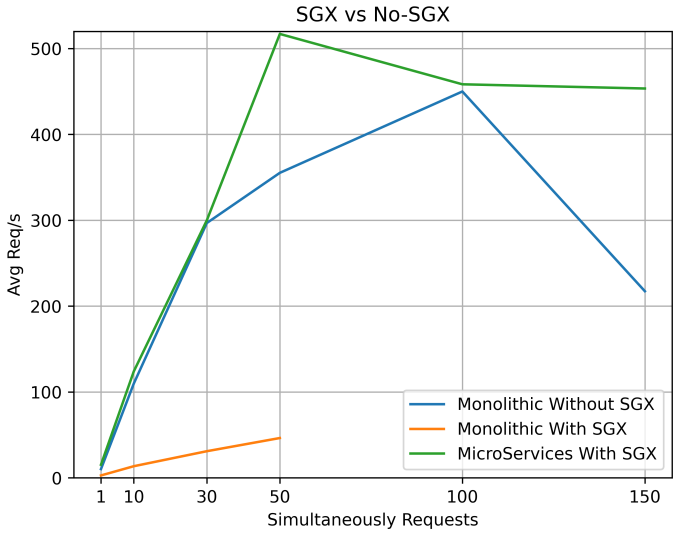

Fig. 5. Average Requests per second achieve using Monolithic app with and without SGX. There is also a comparation in a microservice with a Go component running on the SGX

this program can be unduly accessed and disclosed, and there is no attestation of the code being used.

To protect the data in the face of these limitations, the client side program would have to be modified to encrypt and sign the information transmitted to the *BFT-SMaRt Wrapper*. Due to time constraints, the encryption and signage of data has not been preformed, however, for the purposes of demonstrating the architecture, it's assumed the data is secure.

Because *BFT-SMaRt Wrapper* merely orders the requests and issues their execution to the *App Logic*, it has no need to decipher the data sent by the client, and thus has no means to do it. By being signed, the information sent by the client is irrevocably its; guaranteeing rogue code from a Bizantine *BFT-SMaRt Wrapper* would not propose false requests to a *App Logic* service. As such, in *BFT-SMaRt Wrapper* the data in use is protected, despite a TEE not being used.

The only data *BFT-SMaRt Wrapper* has access in a payload received from a client is a flag indicating whether or not the operation to be performed should be ordered.

The entire business logic from WA3 [9] has been remade in Go. This language is lightweight and does not incur the JVM overhead. Using this language, it was possible to reduce the memory used by the program, such that running four replicas of the *App Logic* requires only *97 MB*, instead of the *1 GB* necessary when using the monolithic version.

The *App Logic* receives encrypted and signed information from the *BFT-SMaRt Wrapper*. The *App Logic* shares secrets with the clients in order to be able to decipher the received data, and execute the issued client instructions with it.

The execution of the operations on *App Logic* is equal to what was done in the monolithic version, with the exception that the responses ought to be encrypted with the secrets shared between the clients and the server. By encrypting the answers, the *BFT-SMaRt Wrapper* does not have access to the information sent from the *App Logic* to the client.

The *App Logic* cannot be accessed directly by the client programs because it is isolated in the network, being accessed only by a *BFT-SMaRt Wrapper*. This isolation protects from attacks attempting to circumvent the ordering of operations. It is however possible to create an inconsistency in a replica's database if the *BFT-SMaRt Wrapper* code is Byzantine, given that it might not order operations, instead issuing operations only to the *App Logic* service with which it is associated.

```
java.lang.OutOfMemoryError: unable to create native thread: possibly out of memory or process/resource limits reached
        at java.base/java.lang.Thread.start0(Native Method) ~[na:na]
        at java.base/java.lang.Thread.start(Thread.java:803) ~[na:na]
        at io.netty.util.concurrent.ThreadPerTaskExecutor.execute(ThreadPerTaskExecutor.java:32) ~[netty-all-4.1.48.Final.jar!/:4.1.48.Final]
        at io.netty.util.internal.ThreadExecutorMap$1.execute(ThreadExecutorMap.java:67) ~[netty-all-4.1.48.Final.jar!/:4.1.48.Final]
        at io.netty.util.concurrent.SingleThreadEventExecutor.doStartThread(SingleThreadEventExecutor.java:978) ~[netty-all-4.1.48.Final.jar!/
:4.1.48.Final]
        at io.netty.util.concurrent.SingleThreadEventExecutor.startThread(SingleThreadEventExecutor.java:947) ~[netty-all-4.1.48.Final.jar!/:4
.1.48.Final]
        at io.netty.util.concurrent.SingleThreadEventExecutor.execute(SingleThreadEventExecutor.java:830) ~[netty-all-4.1.48.Final.jar!/:4.1.4
8.Final]
        at io.netty.util.concurrent.SingleThreadEventExecutor.execute(SingleThreadEventExecutor.java:818) ~[netty-all-4.1.48.Final.jar!/:4.1.4
8.Final]
        at io.netty.channel.AbstractChannel$AbstractUnsafe.register(AbstractChannel.java:471) ~[netty-all-4.1.48.Final.jar!/:4.1.48.Final]
        at io.netty.channel.SingleThreadEventLoop.register(SingleThreadEventLoop.java:87) ~[netty-all-4.1.48.Final.jar!/:4.1.48.Final]
        at io.netty.channel.SingleThreadEventLoop.register(SingleThreadEventLoop.java:81) ~[netty-all-4.1.48.Final.jar!/:4.1.48.Final]
        at io.netty.channel.MultithreadEventLoopGroup.register(MultithreadEventLoopGroup.java:86) ~[netty-all-4.1.48.Final.jar!/:4.1.48.Final]
```

Fig. 6. JVM unable to allocate more memory to handle connection

The presented vulnerability is not present in the naive monolithic version, as the code which orders the operations can be attested. Because *BFT-SMaRt* is being used, unless over one third of the replicas suffer this attack, the system as a whole is not compromised.

6 Evaluation

In order to evaluate the integration of SGX in both architectures, a benchmark has been conducted in which the request rate per concurrent connection has been measured. To assist the justification of the results achieved, the page faults incurred during the tests have been measured.

All experiences use an Intel Xeon -E 2288G CPU with 8 cores at *3.7GHz* with 16 hardware threads (2 per core) and *8MB* L3 cache. The server has 128 GB of DDR4 memory and runs on Ubuntu *18.04.4* LTS with Linux kernel version *4.15*. The server has access to a *1GB* public Ethernet connection and has an average RTT with the client of *66.161ms*

The workload generators run on a machine with 6-core Intel Core i7 at *2.6GHz* with 16 DDR4 RAM and MacOs Catalina. The workload was generated using the *bombardier* benchmarking tool on a *500MB* public connection.

In every test made, docker containerization was used.

6.1 Architectures Benchmark

To attest the efficiency of using SGX in the naive monolithic version and the microservices version, a series of requests were performed to the remote server for each of the versions. The latency of the answers were measured and the rate of request completion was calculated based on how many parallel connections were open. To contextualize the performance overheads of using SGX, the WA3 [9] version was also tested. Fig. 5 presents the results of the evaluation. As can be observed, using SGX with the modified architecture performs better than a naive SGX integration.

Besides the underwhelming performance, the monolithic version is unable to answer simultaneous requests to more than 50 concurrent clients. For each client request, a thread is created; these threads consume part of the limited memory afforded by Scone. Fig. 6 presents the error unleashed when opening more than 50 parallel connections, and from it, is can be observed that the JVM is unable to create more threads to serve the required concurrent connections.

	Microservice	Monolithic
SGX	181950	26091160
No SGX	3167	379241

TABLE 1
Average Page Faults

An attempt was made to allocate more memory to Scone's heap. This effort was however unsuccessful.

Surprisingly, the average rate of requests was superior using the SGX with the modified architecture than using the original version of WA3 without SGX. It can be inferred that the overheads caused by the JVM and Spring Boot are superior to those caused by SGX on an application optimized for its use.

6.2 Page Faults

Whenever a page fault occurs in SGX, information that was in the enclave must be evicted. In order to keep the security guarantees of the TEE, this information must be encrypted. Information previously evicted must be deciphered when loaded into the enclave. For the presented reasons page faults are an extreme source of inefficiency in the SGX.

Table 6.2 presents the average page faults incurred when running the tests in the two architecture versions and using or not an SGX enclave. Observing the table, it's gathered that the use of SGX increases the number of page faults incurred in both architectures, and that the microservices architecture has fewer page faults than the monolithic version.

It is relevant to consider that while the use of SGX in the microservices version increased the number of page faults by a factor of *5745%*, this value increases to *6880%* in the monolithic version. The considerable increase in the number of page faults, the high number of page faults incurred without using SGX, and the high overhead caused by each page fault are solid contributors to the low performance the monolithic version with SGX achieved in fig. 5.

It is interesting to consider that the number of page faults using SGX in the microservices is half the number of page faults incurred in the monolithic version without SGX. The low number of page faults helps justify the fact that the microservices version performed better than the monolithic version without SGX. Despite page faults being an expensive overhead in SGX, they are compensated by the sheer number of faults incurred by the monolithic version.

7 Future Work

The presented report is inserted in the CSD course's practical component. The overall objective of this component is the study of several security properties that can be implemented on a system to improve its trustability.

7.1 Confidentiality of Data in use

An important aspect of a system's trustability, in particular its security, is the ability to maintain the confidentiality of its data. Through the use of TLS [12], the data in transit is secured. This report studies the use of a TEE to provide confidentiality of the data in use.

In the report, it's been stated that the naive integration of a TEE in an application, without factorizing the program to minimize the amount of critical blocks that must be protected, leads to an inefficient and crash prone application.

The factorization process on its own is not sufficient to ensure an application guarantees confidentiality for data in use. The data that is loaded into the SGX enclaves must be secure; as such, it's been deemed necessary to encrypt this data on an application level from the client program and decipher it only in the TEE protected region.

Due to time constraints, data is not encrypted at application level. Thus confidentiality is not guaranteed for data in use. However, the system is prepared to receive encrypted data; both the client program as well as the *BFT-SMaRt Wrapper* have been redesigned accounting for *BFT-SMaRt Wrapper*'s inability to extract information from the data received from the client.

The encryption of data requires that a secret is shared between each client and each *App Logic* service. Supposing that each *App Logic* service had a pair of asymmetric cryptographic keys, and supposing the clients had access to this service's public key through secure means, then the clients could perform a Diffie-Helman [6] exchange and agree on a session key. An example of a secure means by which the clients could obtain *App Logic*s' public keys is if there was a mechanism allowing the clients to communicate directly with the service to enquire the value of the public key.

7.2 Attestation Endpoint

SGX allows the attestation of code running in an enclave. Through attestation, it's possible to assert the code running in an enclave is that which is thought to be.

Currently, the code running in the enclaves is not attested and a Byzantine adversary could replace *Logic App*'s code unbeknownst to the clients. The process of attesting code is complex [8], and would require the integration of C or C++ code with the system developed; both in the client program and in the server program.

The use of libraries that facilitate the attestation process reduce the amount of code required [7], however, it's still necessary to develop a component to bridge the information in the attestation program with the remaining system.

In order to attest if the code is official and signed by a trusted authority, as well as proving the code is run in a SGX enclave, it would be necessary to use a separate piece of software that validates and launches the code in SGX. The launcher would allow code previously defined and signed by a competent authority (i.e. Finance Department, Medical Association, etc.) in SGX and would bridge the attestation from SGX to a requester. Fig. 7 presents the communication protocol to perform the attestation.

In the protocol, a client starts sending a nonce to the launcher endpoint. The launcher redirects it to the SGX running in a replica of the application server, making a local attestation. SGX returns a signature on the nonce and the enclave hash. With this, the launcher would construct another signature over the SGX signature plus nonce + 1 and the signature of the code he put running in the enclave.

Any signature operation performed by the launcher would be performed in SGX, making it possible to make

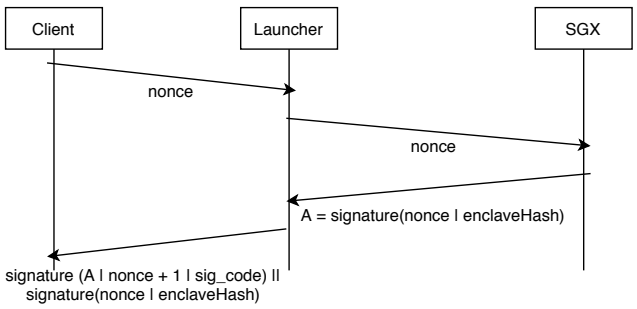

Fig. 7. Code and Enclave Attestation via launcher Message Passing Diagram

a statement about the code running in the SGX and about the enclave, proving that the code is running in the enclave.

7.3 Activate SmartContracts

The use SGX and Scone produced incompatibilities with a previously implemented feature of the application, the access control managed by *Smart Contracts*.

Since *Smart Contracts* run in the *App Logic* program, it's necessary to find a framework that is compatible with the implemented Scone instructions and allows Python to run.

To improve security, *Smart Contracts* would run in other microsevices. In SGX these only run in one port so that they can answer requesters; thus enabling a secure environment where *Smart Contracts* can run without damaging the server. Another possibility is to run a new container in the SGX every time a *Smart Contract* runs, although this solution incurs large overheads.

7.4 Database Homomorphic Encription

The focus of the work presented in the report is the use of TEE environments to guarantee confidentiality for data in use. Data in transit is already guaranteed through the use of TLS. However, there is no implemented mechanism to protect the data in storage.

Given that the application has availability through fault tolerance with the use of *BFT-SMaRt* [3], and given that data in use and in transit are secured, the protection of data in storage is the only aspect that is not trustworthy in the application. The implementation of mechanisms to guarantee the security of this data would fulfil the course objective of achieving a fully trustable system.

To protect data stored in non-volatile memory *Homomorphic Encryption* should be used, as it guarantees data is encrypted and allows operations to be performed directly on the encrypted data, obviating the necessity to transform it into plaintext.

The developed application merely requires the access of data for its logical operations and to list information. The simple handling of data required is within the scope of operations possible with *Homomorphic Encryption*.

Using this encryption system, the data would be encrypted in the *App Logic* program, as it is the only part of the server programs where data is in plaintext.

8 Conclusion

There are several requirements for a system to present trustable properties. One such requirement is the confidentiality of the data in the system.

Using encryption it's possible to guarantee the confidentiality of data in transit and in storage. However, to perform the business logic of an application, the data often has to be in plaintext. It was argued that in order to protect the data in these critical blocks of code, a TEE should be used.

The protection of data whilst in use is an important requirement when there is no assurance on the veracity of the code or operating system running on a machine.

When using cloud services, both the code running and the underlying OS can be tampered unbeknownst to the owner of the data, leading to an increasing importance of the use of TEE as the use of cloud services increases [14].

A modified version of the application developed in WA3 [9] has been presented and it's been discussed how to adapt the system to better take advantage of a TEE and increased security guarantees provided.

It's been concluded that in order to avoid damaging losses of performance, it's necessary to refactor the application software, rather than blindly using tools that enable the use of a TEE. In the refactor process it's been observed that further modifications are required, given that otherwise there would be several blocks of code containing unsecured plaintext data. The refactoring process to which the presented version of WA3 was subject has been demonstrated.

To prove the inefficiencies of a naive approach to integrating a TEE, a benchmark was conducted. In it, it's observed that a careful refactor of an application vastly improves the responsiveness of the application.

The integration of a TEE on WA3 did not fully complete the objective of protecting the data in use nor did it leave the application unscathed. It's been explored how the application could be further improved by providing application level encryption, eliminating the passage of unprotected plaintext in the server.

The use of *Smart Contracts* was affected by the integration with a TEE, and it is an issue that has not been solved.

The full range of possibilities enabled by using a TEE has not been fully experimented, as its integration with WA3 does not make use of the code attestation feature. However, an exploration on how this feature could be used to audit the code running in the TEE was presented.

Finally, it's been explored how to provide complete confidentiality of data to WA3 by using *Homomorphic Encryiption* on the databases. Using this encryption methods, data stored in non-volatile memory would be secured.

The study of TEE is still an emergent topic; a roadblock to the use of the guarantees it provides is the lack of tools to perform some operations as well as the lack of information on the topic. By performing this assignment, it's been observed the challenges faced by applications using this technology, as well as the compromises it entails. In contrast with the benefits provided by TEEs, it's clear that these are fundamental to systems that require a high level of security even though the technology is still improving.

REFERENCES

[1] Cisco annual internet report (2018–2023) white paper. https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html. Accessed 04/06/2020.

[2] The History of the General Data Protection Regulation. https://edps.europa.eu/data-protection/data-protection/legislation/history-general-data-protection-regulation_en. Accessed 11/06/2020.

[3] ALYSSON BESSANI, J. S., AND ALCHIERI, E. E. P. State machine replication for the masses with bft-smart, 2014.

[4] ARNAUTOV, S., TRACH, B., GREGOR, F., KNAUTH, T., MARTIN, A., PRIEBE, C., LIND, J., MUTHUKUMARAN, D., O'KEEFFE, D., STILLWELL, M. L., GOLTZSCHE, D., EYERS, D., KAPITZA, R., PIETZUCH, P., AND FETZER, C. SCONE: Secure linux containers with Intel SGX. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016* (2016).

[5] COSTAN, V., AND DEVADAS, S. Intel SGX Explained. *IACR Cryptology ePrint Archive* (2016), 1–118.

[6] DIFFIE, W., AND HELLMAN, M. New directions in cryptography. *IEEE Trans. Inf. Theor. 22*, 6 (Sept. 2006), 644–654.

[7] GGERSHINSKY. Linux sgx trust management framework. https://github.com/IBM/sgx-trust-management. Accessed 13/06/2020.

[8] GIDON GERSHINSKY, ELIAD TSFADIA, D. H. Trust management in intel sgx enclaves. https://medium.com/@gidon_16942/trust-management-in-intel-sgx-enclaves-fda7d1fe6cb5. Accessed 13/06/2020.

[9] JORGE PEREIRA, PEDRO CAMPONÊS, R. S. Wa3: Auditoria a sistema de leilões com bft-smart integrado, April 2020.

[10] PERRIN, C. https://www.techrepublic.com/blog/it-security/the-cia-triad/. Accessed 11/06/2020.

[11] RASHID, F. Y. What is confidential computing? https://spectrum.ieee.org/computing/hardware/what-is-confidential-computing. Accessed 11/06/2020.

[12] RESCORLA, E., AND DIERKS, T. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, Aug. 2008.

[13] TSAI, C. C., PORTER, D. E., AND VIJ, M. Graphene-SGX: A practical library OS for unmodified applications on SGX. *Proceedings of the 2017 USENIX Annual Technical Conference, USENIX ATC 2017* (2019), 645–658.

[14] ÁNGEL GONZÁLES, AND DAY, M. Amazon, microsoft invest billions as computing shifts to cloud. The Seattle Times https://www.seattletimes.com/business/technology/amazon-microsoft-invest-billions-as-computing-shifts-to-cloud/. Accessed 04/06/2020.