João Carlos Cristo Reis

Bachelor of Computer Science and Engineering

TREDIS – A Trusted Full-Fledged SGX-Enabled REDIS Solution

Dissertation submitted in partial fulfillment of the requirements for the degree of

Master of Science in
Computer Science and Engineering

Adviser: Henrique João Lopes Domingos,
DI-FCT-UNL, NOVA LINCS

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

December, 2020

TREDIS – A Trusted Full-Fledged SGX-Enabled REDIS Solution

This document was created using the (pdf)LaTeX processor, based on the "novathesis" template[1], developed at the Dep. Informática of FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

To my family and friends

Acknowledgements

I would like to thank my advisor, Professor Henrique Domingos, for all the help and time spent guiding me throughout this work. I would also like to thank the institution FCT-UNL for all the years I studied here, years of constantly growth which prepared me to take the next step with confidence. A big thank you to my friends, from back home, from school, and all the ones who followed me throughout this university journey, specially Miguel, Tiago and Ricardo, to whom I'm greatful for all the tireless help and support during this seven years. Finally, my biggest thank you goes to my family, who helped me through everything since I can remember, who were always there for me and will continue to be, pushing me to achieve more and better things. And sorry it took so long!

"It’s supposed to be hard!
If it wasn’t hard everyone would do it.
The hard... is what makes it great."

Abstract

Currently, offloading storage and processing capacity to cloud servers is a growing trend among web-enabled services managing big datasets. This happens because high storage capacity and powerful processors are expensive, whilst cloud services provide cheaper, ongoing, elastic, and reliable solutions. The problem with this cloud-based outsourced solutions are that they are highly accessible through the Internet, which is good, but therefore can be considerably exposed to attacks, out of users' control. By exploring subtle vulnerabilities present in cloud-enabled applications, management functions, operating systems and hypervisors, an attacker may compromise the supported systems, thus compromising the privacy of sensitive user data hosted and managed in it. These attacks can be motivated by malicious purposes such as espionage, blackmail, identity theft, or harassment. A solution to this problem is processing data without exposing it to untrusted components, such as vulnerable OS components, which might be compromised by an attacker.

In this thesis, we do a research on existent technologies capable of enabling applications to trusted environments, in order to adopt such approaches to our solution as a way to help deploy unmodified applications on top of Intel-SGX, with overheads comparable to applications designed to use this kind of technology, and also conducting an experimental evaluation to better understand how they impact our system. Thus, we present TREDIS - a Trusted Full-Fledged REDIS Key-Value Store solution, implemented as a full-fledged solution to be offered as a Trusted Cloud-enabled Platform as a Service, which includes the possibility to support a secure REDIS-cluster architecture supported by docker-virtualized services running in SGX-enabled instances, with operations running on always-encrypted in-memory datasets.

Keywords: Intel SGX, REDIS, Trusted Computing, Trusted Execution Environments, Data Protection, Privacy-Preservation, Dependable In-Memory Key-Value Stores

Resumo

A transição de suporte de aplicações com armazenamento e processamento em servidores *cloud* é uma tendência que tem vindo a aumentar, principalmente quando se precisam de gerir grandes conjuntos de dados. Comparativamente a soluções com licenciamento privado, as soluções de computação e armazenamento de dados em nuvens de serviços são capazes de oferecer opções mais baratas, de alta disponibilidade, elásticas e relativamente confiáveis. Estas soluções fornecidas por terceiros são facilmente acessíveis através da Internet, sendo operadas em regime de *outsourcing* da sua operação, o que é bom, mas que por isso ficam consideravelmente expostos a ataques e fora do controle dos utilizadores em relação às reais condições de confiabilidade, segurança e privacidade de dados. Ao explorar subtilmente vulnerabilidades presentes nas aplicações, funções de sistemas operativos (SOs), bibliotecas de virtualização de serviços de SOs ou hipervisores, um atacante pode comprometer os sistemas e quebrar a privacidade de dados sensíveis. Estes ataques podem ser motivados por fins maliciosos como espionagem, chantagem, roubo de identidade ou assédio e podem ser desencadeados por intrusões (a partir de atacantes externos) ou por ações maliciosas ou incorretas de atacantes internos (podendo estes atuar com privilégios de administradores de sistemas). Uma solução para este problema passa por armazenar e processar a informação sem que existam exposições face a componentes não confiáveis.

Nesta dissertação estudamos e avaliamos experimentalmente diversas tecnologias que permitem a execução de aplicações com isolamento em ambientes de execução confiável suportados em hardware Intel-SGX, de modo a perceber melhor como funcionam e como adaptá-las à nossa solução. Para isso, realizámos uma avaliação focada na utilização dessas tecnologias com virtualização em contentores isolados executando em hardware confiável, que usámos na concepção da nossa solução. Posto isto, apresentamos a nossa solução TREDIS - um sistema *Key-Value Store* confiável baseado em tecnologia REDIS, com garantias de integridade da execução e de privacidade de dados, concebida para ser usada como uma "Plataforma como Serviço"para gestão e armazenamento resiliente de dados na nuvem. Isto inclui a possibilidade de suportar uma arquitetura segura com garantias de resiliência semelhantes à arquitetura de replicação em *cluster* na solução original REDIS, mas em que os motores de execução de nós e a proteção de memória do *cluster* é baseado em contentores docker isolados e virtualizados em instâncias SGX, sendo os dados mantidos sempre cifrados em memória.

Palavras-chave: Intel SGX, REDIS, Computação Confiável, Ambientes de Execução Confiáveis, Proteção de Dados, Preservação de Privacidade, *Key-Value Stores* Confiáveis em Memória

Contents

List of Figures

List of Tables

Acronyms

CPU	Central Processing Unit
CRUD	Create Read Update Delete
DB	Database
DBMS	Database Management System
EK	Endorsment Key-pair
EPC	Enclave Page Cache
FDE	Full-Disk Encryption
FPGA	Field-Programmable Gate Array
HAP	High-Assurance Process
HSM	Hardware Security Modules
HTTP	HyperText Transfer Protocol
IaaS	Infrastructure as a Service
KVS	Key-Value Store
OS	Operating System
PCR	Platform Configuration Registers
RAID	Redundant Array of Independent Disks
SGX	Intel Software Security Guard Extensions
SSL	Secure Socket Layer

TCB	Trusted Computing Base	
TCE	Trusted Computing Environment	
TCPA	Trusted Computing Platform Alliance	
TEE	Trusted Execution Environment	
TLS	Transport Layer Security	
TPM	Trusted Platform Module	*
VM	Virtual Machine	
VMM	Virtual Machine Manager	
XOM	eXecute Only Memory	

Chapter 1

Introduction

In this chapter, we introduce the context and motivation of this dissertation, as well as its problem statement, followed by the goals and expected contributions. In the end, we present the structure of the document.

1.1 Context and Motivation

Data has more value than ever before. The adoption of mobile devices as an almost indispensable thing in people's lives led to an immense amount of information produced each second, by everyone, at a global scale [19, 20]. To deal with this, numerous fields of computer science were born, and new technologies more adapted to deal with vast amounts of data were designed. A major one was Cloud Computing, which appeared as a way to remotely provide computing and storage capabilities to systems like we never saw before, as well as fault tolerance and scalability (as well as many other properties) for a reduced cost to their users [59]. Because high storage capacity and powerful processors are expensive, the tendency to move data to these cloud providers emerged as a convenient way to provide just that, thus making the users free of worries regarding physical resources in their own machines. Both users and companies could now choose to rely on a cloud provider, which are usually hosted by huge corporations, to run their services. However, the fact that these cloud systems are highly accessible over the Internet made them major attack targets [38] and security problems, more specifically the ones regarding privacy and security of personal information, were found to be very concerning [39]. Since data is stored in the provider's system, the information depends on the provider's security, as well as the behavior of its staff, which have physical access to the system and can act maliciously. All this brings insecurity to the data and raises privacy concerns. Also, after the data reached the cloud provider, privacy is not assured during the execution phase, even if it stored safely (encrypted) on disk since to be executed it has to be fully decrypted in volatile memory. The data is then vulnerable during memory attacks, which lead to many efforts to be put into solving this particular problem.

A few solutions were thought to be effective, either by using some kind of virtualization or containerization, or even by relying on the hardware itself in a special way. Some [12, 18, 29, 31, 32, 60] offer the possibility to isolate the execution of data from the host OS, making system calls ineffective and blocking typical permissions this might have over the whole system, while others, particularly the more recent ones regarding Trusted Execution Environments [2, 11, 23, 43], focused on creating a trusted memory region where data could execute fully encrypted from the outside. However, while offering trust and integrity to the data, TEEs have been proven to have some performance issues, since they depend a lot on encryption and decryption (usually big performance droppers). Unfortunately, the performance loss really impacted their adoption in modern systems, which lead to figuring out a way to make this kind of technology more viable. More recently, a few different approaches to place on top of TEEs have proven to soften the performance issues, creating the impression that we are on the right path to take the most advantage of TEEs.

1.2 Problem Statement

In this dissertation, we study how unmodified applications can be deployed with ease on top of trusted hardware (TEE), and still offer comparable performance overheads relatively to other applications running without this extra layer of security. To achieve this, we took a deep look into existing approaches capable of enabling applications to run with isolation in trusted execution environments and conducted an evaluation by adopting one of them to be ported on top of trusted hardware (SGX).

1.3 Objective and Expected Contributions

Our main objective in this thesis is to implement a secure system based on Intel-SGX where it is possible to deploy unmodified applications without any major performance drop, by adopting the usage of an SGX-enabled framework to be ported on top of SGX. Keeping that in mind, our focus lays on analyzing the behavior of an unmodified application (REDIS KVS) executing on top of SGX, benefiting from privacy and isolation properties of its data.

Our work delivers the following contributions:

- The design of TREDIS, a full-fledged REDIS solution leveraging Intel-SGX and supported by a trusted SGX-enabled framework, running with isolation guarantees provided by hardware-shielded capabilities, to be supported as a service in the cloud;

- Implementation of the TREDIS prototype as a cloud-enabled platform as a service, using real Intel-SGX-enabled hardware on commodity servers of a Cloud-Provider (as a solution designed as a candidate for an OVH product offer);

- Experimental evaluation of the prototype, to study the overheads of TREDIS against an untrusted REDIS solution. For this purpose, we analyze (1) Performance conditions observed by latency and throughput measurements; (2) Adequacy to manage operations on privacy-enhanced big-datasets; (3) Scalability conditions under different client-workloads; (4) Analysis of required resources, including memory and CPU usage.

1.4 Report Organization

The remaining of this thesis is organized as follows:

- **Chapter** 2 gives an initial background essential to understand the objectives and expected contributions of this dissertation, while also covering related work references;

- **Chapter** 3 introduces our system approach, covering the model and architecture, along with the specification of the solution's adversary model;

- **Chapter** 4 presents a detailed description of the implementation of the prototype, describing the environments and technologies we used;

- **Chapter** 5 shows the evaluation and analysis we performed on the system, along with the discussion of the results;

- **Chapter** 6 is where we present our conclusions and also address some open issues and future work directions.

Chapter 2

Related Work

Cloud computing has emerged as an efficient way for modern systems to deal with modern problems, caused by the growth of internet users worldwide over the years [24], where scalability became a must and the cloud's ability to offer storage and computing power on demand made it so useful. With that said, users (including companies) started choosing cloud providers as a convenient way to store their data and services, trusting that data privacy would be assured. However, that is not always the case in modern cloud providers.

In this chapter, we address existing solutions able to grant a better level of privacy to data, by protecting applications from the OS/hypervisor regardless of the machine they're running on, increasing the level of trust of users in the remote execution of an application.

These existing solutions are organized in different sections in the following way: Section 2.1 covers protection against untrusted OSes; Section 2.2 covers TEEs and hardware-enabled approaches; In Section 2.3 we cover, in more detail, hardware-enabled TEE solutions used today; Section 2.4 covers shielded applications and frameworks compatible with Intel-SGX, which is the TEE technology we choose for our approach; Finally, in Section 2.5 we make a critical analysis of the topics previously discussed while covering their main advantages and disadvantages.

2.1 Protection in untrusted OSes

A lot of applications these days depend on sensitive data to operate. Therefore protecting this data must be taken into account while designing the application. One of the things we have to think about is the size of the TCB, and how to reduce it as much as possible without losing the operability of the system. Typically, the host OS is considered safe and trustworthy, although that is not always the case. A compromised OS can give complete access to sensitive data, if not isolated from the application. That's why this is a major security problem and must be tackled in today's systems.

Approaches like Virtual Ghost, Flicker, MUSHI, SeCage, InkTag, Sego, all grant security by isolating the sensitive data from the untrusted OS either by monitoring the application while it runs or by enforcing memory isolation by using virtualization.

2.1.1 Virtual Ghost

Virtual Ghost [12] provides application security against untrusted OSes by implementing the idea of ghost memory, which is inaccessible for the OS to read or write, as well as providing trusted services like ghost memory management, key management, and encryption and signing services. It relies on sandboxing to protect the system from the OS, where a thin layer of abstraction is interposed between the kernel and the hardware. This layer works as a library of functions that the kernel can call directly, without needing higher privileges. Thus, Virtual Ghost protects the system against a direct threat from the OS, without losing significant levels of performance.

2.1.2 Flicker

Flicker [32] provides secured isolated execution of sensitive code by relying on commodity hardware, such as AMD and Intel processors, to run certain pieces of code in a confined environment while reducing the TCB to, as few as, 250 additional lines of code. When Flicker starts, none of the software already executing can monitor or interfere with its execution and all its traces can be eliminated before non-Flicker execution resumes. Thus, with a small TCB and a good level of isolation during the execution phase, where no data is leaked nor possible to access while inside the confined environment, the system can achieve reliability and security.

2.1.3 MUSHI

MUSHI [60] is designed to deal mainly with multi-level security systems and provides isolation to individual guest VMs executing in a cloud infrastructure. MUSHI ensures that VMs are instantiated securely and remain that way throughout their life cycle. It is capable of offering: (1) **Trusted Execution**, where both the kernel and user image, as well as MUSHI itself, are attested upon a VM start by using a TPM, thus defining a trusted initial state; (2) **Isolation**, where each VM executing on the same machine runs isolated, as a way to guarantee confidentiality and integrity; (3) **User Image Confidentiality**, by encrypting the user image with a cryptographic key provided by the user itself.

MUSHI guarantees confidentiality and integrity of a VM even during malicious attacks from both inside and outside the cloud environment. It trusts a relatively small TCB, that includes only the hardware, hardware virtualization, BIOS, and System Management Mode, and can be implemented with quite ease using modern commodity hardware containing SMM memory (SMRAM), necessary for the isolation between the host and VM.

2.1.4 SeCage

SeCage [31] uses hardware virtualization to protect user-defined secrets from potential threats and malicious OSes, by isolating sensitive code and critical secrets while denying the hypervisor any possibility of intervention during runtime. It divides the system into compartments, where secret compartments have all the permissions to access and manipulate the user-defined secrets, and a main compartment responsible for handling the rest of the code. SeCage is designed to assure the confidentiality of user-secrets, adding a small overhead while supporting large-scale software. To achieve this, it ensures:

1. **Hybrid analysis of secrets**, where static and dynamic analysis are combined to define secret compartments to execute secrets, preventing them from being disclosed during runtime;

2. **Hypervisor protection**, using hardware virtualization to isolate each compartment;

3. **Separating control and data plane**, where minimal hypervisor intervention is required to deal with communications between compartments. The hypervisor is limited to define policies on whether two compartments can communicate (control plane) for as long as they conform to those policies.

2.1.5 InkTag

InkTag [18] also uses a virtualization-based approach in order to grant applications protection from untrusted OSes. But unlike SeCage, InkTag admits trust in the hypervisor. The hypervisor is responsible to protect the application code, data, and control flow from the OS, allowing applications to execute in isolation, in high-assurance processes (HAP). Trusted applications can communicate directly with the InkTag hypervisor via hypercalls, as a way to detect OS misbehavior. It introduces a concept called paraverification, which simplifies the hypervisor by forcing the untrusted OS to participate in its own verification. As a result, the OS notifies the InkTag hypervisor upon any update to be made to the state, which the hypervisor can check for correctness. InkTag also isolates secure from unsafe data through hardware virtualization and allows each application to specify their own access control policies, managing their data privacy and integrity through encryption and hashing. Another important aspect of InkTag is recoverability. InkTag hypervisor can protect the integrity of files even if the system crashes, by ensuring consistency between file data and metadata upon a crash.

2.1.6 Sego

Similar to InkTag, Sego [29] is also a hypervisor-based system that gives strong privacy and integrity guarantees to trusted applications. To protect applications from untrusted OSes, Sego removes the trust from the OS, relying only on a trusted hypervisor which is assumed to always execute correctly. It also enforces paraverification, where the OS communicates its intentions to the hypervisor, thus keeping track of its behavior.

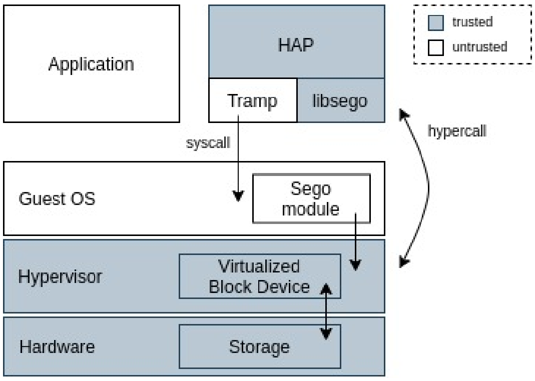


Figure 2.1: Sego Architecture Overview

Sego is designed to execute trusted code in a HAP. After booting the OS, the hypervisor starts the HAP, in a way that the HAP itself can verify its own initial code and data, similar to a TPM. Once running, the hypervisor ensures that the HAP's registers and trusted address space are isolated from the OS. Every time the HAP wants to perform a system call, it must inform the hypervisor of its intent so that the hypervisor can verify the OS's activity. HAPs use a small library called libsego as a way to handle system calls and get Sego services without having to change their code. Each HAP also contains an untrusted trampoline code, that uses to interact with the OS. This protects its control-flow, since it uses this trampoline as the issuer for system calls, therefore never compromising the HAP itself. Figure 2.1 shows well enough Sego's overall design without going into too much dept, indicating which components are included in the TCB and which are not. Context switches are handled by the hypervisor, thus hiding any information about the HAP from the OS. Sego does not guarantee OS availability. A compromised OS can simply shut down or refuse to schedule processes. However, this is easily detected.

In conclusion, although all these approaches are prepared to isolate an application from untrusted hosts, they emphasize the use of software to do it, leaving behind the importance of hardware trustworthiness to the whole system.

In the following sections, we will take a look at existing solutions that are able to tackle also hardware related security problems.

2.2 Hardware-Enabled TEE - Trusted Execution Environments

A TEE is an abstraction provided by both software and hardware that guarantees isolated execution of programs in a machine from the host OS, hypervisor, or even system administrators, preventing them from leveraging their privileges. A TEE also provides integrity of applications running inside it, along with the confidentiality of their assets. The first attempts to implement a TEE on a cloud system consisted of combining a hypervisor with isolation properties and a TPM.

A TPM [17] consists of a hardware chip, called microcontroller, that aims to create a trustable platform through encryption and authenticated boot, and make sure it remains trustworthy through remote attestation. It provides cryptographic functions that can't be modified, and a private key (Endorsement Key) that is unique to every TPM made, working as an identifier for the TPM itself. However, TPMs have several problems when applied to cloud systems, due to being designed with the intention to offer security to a single machine. It is not flexible enough to guarantee that anyone can get the encrypted data from a different node. Thus, a distributed environment would not be the best kind of environment for a TPM to work on.

The current best practice for protecting secrets in cloud systems uses HSMs. An HSM [6] is a physical hardware component that provides and stores cryptographic keys used to encrypt/decrypt data inside a system. HSMs also perform cryptographic operations (e.g. encryption, hashing, etc.) as well as authentication through verifying digital signatures and accelerating SSL connections [48], by relieving the servers from some of the workload caused by operations involving cryptography. Thus, the system can protect critical secrets (cryptographic keys) and support a range of cryptographic functions.

With that in mind, new hardware-enabled solutions were developed to be more flexible and cloud-friendly than the TPM or to incorporate the advantages of HSM to the system. We'll dive into technologies like ARM TrustZone, Intel SGX, AMD-SEV, and some others, in the following section.

2.3 Hardware-Enabled TEE Solutions

The idea of using hardware to provide trusted execution environments to run code appeared as a way to deal with piracy, with examples like TCPA [56] and Microsoft's Palladium [35] being the most popular at that time. By providing protection during execution through hardware, it became possible to encrypt data (e.g. DVD's) that could only be decrypted by specific hardware, making it almost impossible to pirate. Although these approaches were effective back in the day, both of them place their trust in the hardware, not trusting the OS entirely. Thus, since any application does not trust the OS, it does not trust the application to properly use its resources either. Therefore, some of the protection aspects of the OS should be moved into the hardware, while also changing the interface between the OS and the application so it supports hardware security features.

XOM, described in the next subsection, was one of the first approaches developed as a way to deal with these changes, and one of the stepping stones that lead us to the modern TEE technology we see nowadays, that we will also describe in this section.

2.3.1 XOM

XOM [30] is a processor architecture able to provide copy protection and tamper-resistance functions, useful for enabling code to run in untrusted platforms, deployment of trusted clients in distributed systems like banking transactions, online gaming, electronic voting, but also fundamental to deal with piracy back in the day it was published.

The main idea is to only trust the processor to protect the code and data, thus not trusting the main memory nor any software, including the host OS. However, this idea of only trusting hardware has some implications for OSes design. This happens due to the fact that sharing hardware resources between multiple users is a hard job, especially without trusting any software. It is usually easier to have these policies performed by the OS. Therefore, not trusting the software entirely can sometimes be a drawback. For XOM architecture to be used, it is required a specific OS (XOMOS). XOMOS runs on hardware that supports tamper-resistant software and is adapted to manage hardware resources for applications that do not trust it. XOM offers protection against attackers who may have physical access to the hardware itself, as well as main memory protection if compromised. For it, the XOM processor encrypts the values in memory and stores the hash of those values in memory as well. It then only accepts encrypted values from memory if followed by a valid respective hash.

2.3.2 ARM TrustZone

ARM TrustZone [43] is ARMs approach to offer a TEE where software can execute in a secured and trustable way, safe from the host machine, as well as its OS and/or hypervisor.

To create this abstraction, ARM processors implement two virtual processors backed by hardware access control, where the software stack can switch between two states: secure world (SW) and normal world (NW). The first one has higher privileges than the second one, therefore it can access NWs copies of registers, but not the other way around. SW is also responsible for protecting running processes in the CPU while providing secure access to peripherals. Each world acts as a runtime environment and has its own set of resources. These resources can be partitioned between the two worlds or just assigned to one of them, depending on the ARM chip specs. For the context switch between worlds, ARM processors implement a secured mode called Secure Monitor, where there is a special register responsible of determine if the processor runs code in SW or NW. Most ARM processors also offer memory curtaining. This consists of the Secure Monitor allocating physical addresses of memory specifically to the SW, making this region of memory inaccessible to the rest of the system. By default, the system boots always in SW so it can provision the runtime environment before any untrusted code start to run. It eventually transitions to NW where untrusted code can start to be executed.

2.3.3 AMD-SEV

AMD Secure Encrypted Virtualization (SEV) [2] is the AMD approach to provide a TEE, integrated with virtualization. It is a technology focused on cloud computing environments, specifically in public IaaS, as its main goal is to reduce trust from higher priviledged parties (VMMs or OS) so that they can not influence the execution on the other "smaller"parties (VMs). To achieve this, AMD grants encryption of memory through a technology called Secure Memory Encryption (SME), or through TransparentSME (TSME) if the system runs a legacy OS or hypervisor with no need for any software modifications. After the data is encrypted, SEV integrates it with AMD virtualization architecture to support encrypted VMs. By doing this, every VM is protected from his own hypervisor (VMM), disabling its access to the decrypted data. Although incapable of accessing the VM, the VMM is still responsible for controlling each VM's resources. Thus, AMD provides confidentiality of data by removing trust from the VMM, creating an isolated environment for the VM to run, where only the VM and the processor can be trusted.

However, ADM-SEV does not provide integrity of data, allowing replaying attacks to take place, and has a considerably large TCB, since the OS of each VM is trusted [36].

2.3.4 Sanctum

Sanctum [11] offers strong isolation of software modules, although following a different approach focused on avoiding unnecessary complexity. To make this possible, Sanctum, which typically runs in a RISC-V processor, combines minimal invasive hardware modifications with a trusted software security monitor that is receptive to analysis and does not perform cryptographic operations using keys. This minimality idea consists of reusing and slightly modifying existing well-understood mechanisms, while not modifiying CPU building blocks, only adding hardware to the interfaces between blocks. This causes Sanctum to be adaptable to many other processors besides RISC-V.

Sanctum is a practical approach that shows that strong software isolation is achievable with a small set of minimally invasive hardware changes, causing reasonably low overhead. This approach provides strong security guarantees dealing with side-channel attacks, such as cache timing and passive address translation attacks.

2.3.5 Intel-SGX

Intel Software Security Guard Extensions (SGX) [23] are a set of instructions built-in Intel CPUs, that allows programmers to create TEEs, by using enclaves. Enclaves are isolation containers that create a trusted environment where sensitive code can be stored and executed inside, ensuring integrity and confidentiality to it. By doing so, it reduces the TCB in a way that most of the system software, apart from the enclaves and the CPU, is considered not trusted.

A system that incorporates SGX under its architecture is divided into two: a trusted component being the enclave, and an untrusted component being the rest of the system. Enclaves are mapped into private regions of memory - called the enclave page cache (EPC) - where only the CPU has access to, where code running outside the enclave cannot access the enclave memory region, but code running inside can access untrusted memory. This is made possible by a list of functions incorporated. Enclaves are initiated by untrusted code, using the ECREATE function, which initializes the EPC in memory. One of the downsides of SGX consists of the fact that the size of EPCs depends on the hardware used in the system, and it's usually quite limited. This can imply that a larger application can't fit entirely into the EPC. To deal with this, SGX provides a mechanism for swapping pages between the EPC and the untrusted memory. But since this process involves a lot of encryption and decryption operations to keep the integrity of data, it can be a problem performance-wise.

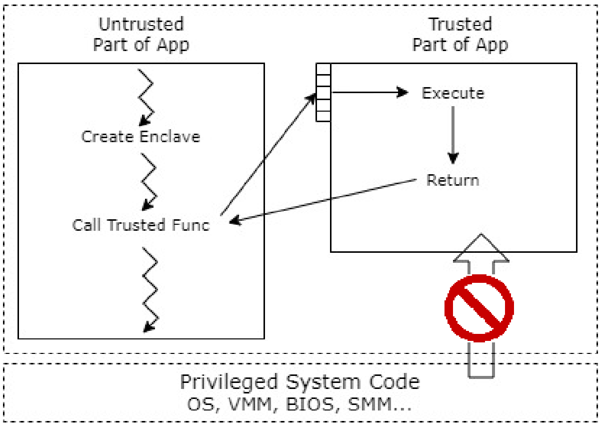


Figure 2.2: Intel-SGX Enclave execution flow

Although the main purpose of SGX is to isolate the application, even from the host OS, there are other instructions to interact with the flow of the enclave, that an application can use to deal with the lack of utility libraries, that are often offered by the OS itself. EENTER and EEXIT are examples used for a thread to enter and leave the enclave, by switching the CPU enclave mode. This allows a thread, for example, to make calls for privileged instructions, which isn't possible while inside the enclave, and re-enter it after, with safety measures assured by the SGX. The fact that when executing enclave code the system can't run privileged instructions, makes that the threads need to exit and re-enter the enclave to execute them. Such transitions come at a cost since a lot of security measures take place (checks and updates) to ensure that the integrity of the code running inside the enclave is kept intact. This may involve a lot of page swapping between the EPC and untrusted memory which, as we stated before, takes a lot of effort from the system. ECALLS and OCALLS are another examples of instructions, that are used as a way to securely communicate between trusted and untrusted parties.

2.4 SGX-Enabled Frameworks and Shielded Applications

The need for cloud computing is constantly growing in modern applications, based on the fact that it is a cost-effective and practical solution to run large distributed applications. However, the fact that it requires users to fully trust the cloud provider with their code and data creates some trust concerns for developers. Although the usage of TEEs like SGX aims to tackle this problem by running and storing sensitive data in an isolated environment, protecting that data from unauthorized access, SGX itself has some limitations and does offer this extra level of security at some costs for the systems.

Hence, to deal with the SGX limitations, some approaches were developed to be implemented on top of it, as a way to make systems more practical by the integration of trusted computation. We will discuss those approaches in the next subsections.

2.4.1 Shielded protected applications in untrusted Clouds

As we said previously, cloud computing is becoming more and more adopted in today's systems. By being such a popular technology, it is a must that their users' data remains confidential. However, most of today's cloud systems are build using a classical hierarchical security model more worried about the cloud providers' software itself then their users' code. Hereupon, for the users of a cloud platform to trust the provider software entirely, as well as the provider staff (i.e. system administrators or anyone with physical access to the hardware), some new measures need to be adapted.

Several approaches were developed as a way to give the user some sense of privacy, by creating the notion of shielded execution for applications running in the cloud. This concept consists of running server applications in the cloud inside of an isolated compartment. The cloud provider is limited to offer only raw resources (computing power, storage, and networking) to the compartment, without being able to access any of the data, except the one being transmitted over the network. Assuring a shielded execution of an application fundamentally means that both confidentiality and integrity are granted and that if the application executes, it behaves as it is expected. As for the provider, it retains control of the resources and may protect itself from a malicious guest [8].

2.4.2 SCONE

Container-based virtualization has become quite popular for offering better performance properties than the use of VMs, although it offers weaker isolation guarantees, and therefore less security. That's why we observe that containers usually execute network services (i.e. REDIS). These are systems that don't need as many system calls as others since they can do a lot via networking, thus keeping a small TCB for increased security.

SCONE [5] is a mechanism for Linux containers that increases the confidentiality and integrity of services running inside them by making use of Intel-SGX. SCONE increases the security of the system while keeping the performance levels reasonable. It does it by:

(1) reducing as much as possible the container's TCB, by linking a (small) library inside the enclave to a standard C library interface exposed to container processes. System calls are executed outside the enclave, and networking is protected by TLS;

(2) maximizes the time threads spend inside the enclave by supporting user-level threading and asynchronous system calls, thus allowing a thread outside the enclave to execute system calls without the need for enclave threads to exit. This increases the performance since major performance losses are caused by enclave threads entering/exiting, due to the costs of encrypting/decrypting the data.

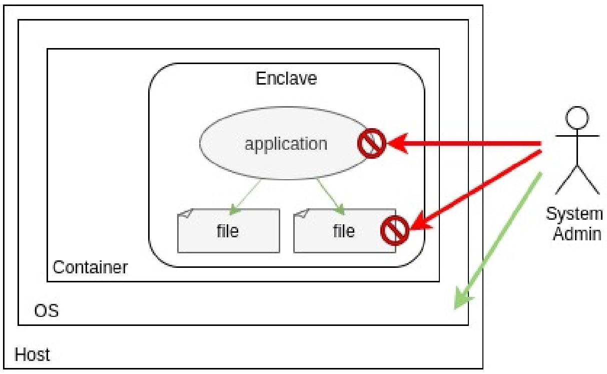


Figure 2.3: Scone Protection for SGX Enclaves

SCONE also has a mechanism for remote attestation, in which a remote component can attest applications running inside SCONE containers, as a way to check their integrity and to make sure they run as they were instructed to.

2.4.3 Haven

Haven is the first system to achieve shielded execution of unmodified legacy applications for a commodity OS (Windows) and hardware, achieving mutual distrust with the host software. It leverages Intel-SGX to protect against privileged code and physical attacks, but also against the challenge of executing unmodified legacy binaries while protecting them from an untrusted host. Instead of shielding only specific parts of applications and data by placing them inside enclaves, Haven aims to protect entire unmodified applications, written without any knowledge of SGX. However, executing entire chunks of legacy binary code inside an SGX enclave pushes the limits of the SGX itself, and while the code to be protected was written assuming that the OS executing the code would run it properly, this may not be the case since the OS can be malicious. For this latest problem, the so-called Iago attack [10], Haven uses a library OS adapted from Drawbridge [41] running inside an SGX enclave. By combining it with a remote attestation mechanism, Haven is able to guarantee to the user end-to-end security without the need of trusting the provider. Although this approach may need a substantial TCB size (LibOS quite large), all this code is inside the enclave, which makes it under users' control.

That's the main goal of Haven: give the user trust by granting confidentiality and integrity of their data when moving an application from a private area to a public cloud.

2.4.4 OpenSGX

OpenSGX [25] was developed as a way to help with access to TEE software technologies, since this type of technologies were only available for a selected group of researchers. It was made available as an open-source platform, and by providing TEE and OS emulation, it contributed a lot for expanding the possibility of research in this area, as well as promoting the development of SGX applications. OpenSGX emulates the hardware components of Intel-SGX and its ecosystem, including OS interfaces and user libraries, as a way to run enclave programs. To emulate Intel-SGX at instruction-level, OpenSGX extended an open-source emulator, QEMU.

Its practical properties result from six components working together: (1) **Hardware emulation module**: SGX emulation, by providing SGX instructions, data structures, EPC and its access protection, as well as SGX' processor key; (2) **OS emulation**: since some SGX instructions are privileged (should be executed by the kernel), OpenSGX defines new system calls to perform SGX operations, such as dynamic memory allocation and enclave provisioning; (3) **Enclave loader**: enclave must be properly loaded to EPC; (4) **User library**: provides a library (sgxlib) with a useful set of functions to be used inside and outside the enclave; (5) **Debugging support**; (6) **Performance monitoring**: allow users to collect performance statistics about enclave programs.

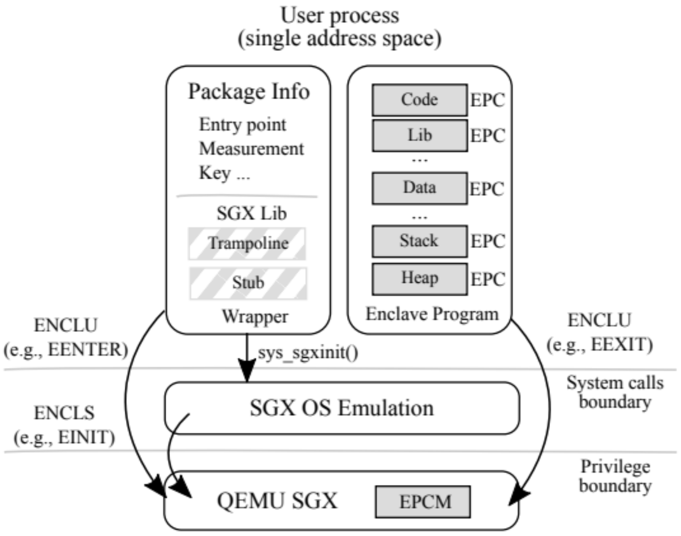


Figure 2.4: Overview design of OpenSGX framework and memory state. Source: [25]

In Figure 2.4 we see an illustration of this framework, where a regular program (Wrapper) and a secured program (Enclave Program) both run as a single process in the same virtual address space. Since Intel-SGX uses privilege instructions to setup enclaves, the requests from the Wrapper program are handled by the OpenSGX set of system calls. OpenSGX was proven capable of running non-trivial applications while promoting the implementation and evaluation of new ideas. By being the first open-source framework to emulate an SGX environment, it was fundamental to the growth of the TEE field.

2.4.5 Panoply

Panoply [52] is a system that works as a bridge between SGX-native abstractions and standard OS abstractions, required by most commodity Linux applications. It divides a system into multiple components, called micro-containers (or "micron"), and runs each one of them inside its own enclave. However, when microns communicate with each other, their communication goes through a channel under OS control. Thus, Panoply design goals are focused on supporting OS abstractions with a low TCB, while also securing inter-enclave communications.

Panoply's design consists of a set of runtime libraries (shim library included) and a build toolchain that helps developers to prepare microns. With this, a programmer can assign annotations to functions as a way to specify which micron should execute a specific function. If not assigned any annotation, a function will be designated to a default micron, who shall execute it. Inter-micron flow integrity is also provided during this stage. Each micron is given a micron-id when it starts, which will be used for all further interactions with other microns. It will use this id as a way to assure that only authorized microns can send/receive messages. To extend this inter-micron interaction security, Panoply also provides authenticated encryption of every message, makes use of unique sequence numbers, and acknowledgment messages are sent for every inter-enclave communication, thus protecting the containers from silent aborts, replaying, or tampering attacks.

Unlike many other SGX-based frameworks, Panoply supports unlimited multi-threading and forking. Multi-threading in a way that, if a micron reaches its maximum concurrent thread limit, a new micron is launched and all shared memory operations are safely performed. Forking is achieved by replicating a parent micron's data and sending it to the child, over a secure communication.

2.4.6 VC3

Verifiable Confidential Cloud Computing (VC3) [49] is a framework that achieves confidentiality and integrity of data, as well as verifiability of code execution with good performance through MapReduce [14] techniques. It uses Intel SGX processors as a building block and runs on unmodified Hadoop [53]. In VC3 users implement MapReduce jobs, compile and encrypt them, thus obtaining a private enclave code, named **E-**. They

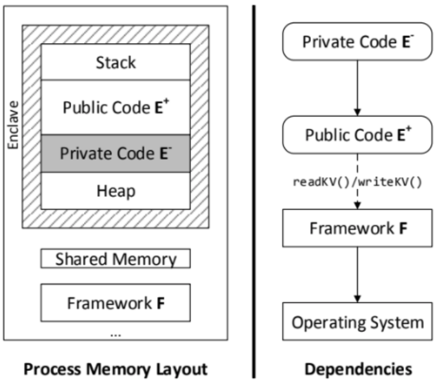


Figure 2.5: VC3 memory design model and component dependencies. Source: [49]

then join it with a small portion of public code called **E+**, which implements the protocols for key exchange and job execution. Users then upload the resulting binary code to the cloud, where enclaves containing both E- and E+ are initialized by an untrusted framework F.

A MapReduce begins with a key exchange between the user and the **E+** code running in the enclave. After this, **E+** can proceed to decrypt **E-** and process the encrypted data. VC3 isolates this processing from the OS by keeping an interface between the E+ layer and the outside of the enclave. This interface consists of basically two functions: **readKV()** and **writeKV()**, for reading or writing a key-value pair on Hadoop, respectively. Also, the data inside the enclave is passed to the outside, more specifically from **E+** to the untrusted F, by using a virtual address space shared by both. With VC3, both **E-** and the user data are always encrypted while in the cloud, except when processed by the trusted processor, while allowing Hadoop to manage the execution of VC3 jobs. Map and reduce nodes are seen as regular worker nodes to Hadoop, therefore Hadoop can keep providing its normal scheduling and fault-tolerance mechanisms, as well as load balancing. VC3 considers Hadoop, as well as both the OS and the hypervisor as untrusted, thus keeping the TCB size as small as possible.

2.4.7 Trusted ZooKeeper Approach

ZooKeeper [21] is a replicated synchronization service for distributed systems with eventual consistency that does not guarantee the privacy of stored data by default.

Trusted ZooKeeper was presented in [9] as an approach that eliminates these privacy concerns, by placing an additional layer between the client and the ZooKeeper, referred to as ZooKeeper Privacy Proxy (ZPP). ZPP is the layer responsible for the encryption of all sensitive information, during a communication between a client and the ZooKeeper. Clients communicate with the proxy via SSL, where the packets are encrypted by an individual session key. Here, ZPP acts like a normal ZooKeeper replica to the client. After receiving the packets from the client, ZPP extracts the sensitive data, encrypts it with a mechanism that allows the data to be decrypted by the proxy later on, and forwards the encrypted packet to a ZooKeeper replica where it can be stored with integrity ensured. ZPP runs inside a TEE, located in the cloud, allowing it to store encryption keys and process data safely. As a result, even if the threat comes from the cloud provider itself, the integrity of the data will still be granted since the attacker won't be able to access or alter anything running inside the TEE. ZPP also retains all original ZooKeeper functionality and does not affect ZooKeeper's internal behavior. Therefore adapting existing ZooKeeper applications to this concept can be done with quite ease.

This approach allows applications in the cloud to use ZooKeeper without privacy concerns at the cost of a small decrease in throughput.

2.4.8 Ryoan

Ryoan [22] consists of a distributed sandbox approach that allows users to protect the execution of their data. This is achieved with the help of Intel-SGX [23] [33] technology, by running NaCl (Google Native Client) sandbox instances inside enclaves, protecting the data from untrusted software while also preventing leaks of data, which is a weakness of enclaves caused by side-channel attacks. Ryoan does not include any privileged software (e.g. OS and hypervisor) in its TCB. It trusts only the hardware (SGX enclaves) to assure the secrecy and integrity of the data.

Its main goal is to prevent leakage of secret data. This is done by keeping the modules from sending sensitive data over communications outside the system boundaries, but also by eliminating the possibility to store data into unprotected memory regions, as well as crossing off the possibility to make most system calls, granted by using NaCl instances. Ryoan's approach consists of confining the untrusted application in a NaCl instance, responsible for controlling system calls, I/O channels, and data sizes. This NaCl sandbox is running inside enclaves' memory region and can communicate with other NaCl instances, forming a distributed sandbox between users and different service providers. Inside the sandbox, the untrusted application can execute safely on secret data. The NaCl sandbox uses a load time code to ensure that the module cannot do anything it shouldn't, thus preventing it from violating the sandbox. To handle faults, exceptions, or errors inside the NaCl sandbox, Ryoan uses an unprotected trampoline code, that can enter the enclave and read the information about the fault, so it can handle it.

2.4.9 Opaque

Opaque [61] is a distributed data analytics platform that guarantees encryption, secure computation, and integrity to a wide range of queries. Therefore, instead of being implemented in the application layer or the execution layer as this kind of security approaches usually are, Opaque is implemented in the query optimization layer.

It is implemented with minimal modifications on Apache Spark [4], a framework for data processing and analytics, and leverages Intel-SGX technology as a way to grant confidentiality and integrity of the data. However, the use of enclaves can still be threatened by access pattern leakage that can occur at memory-level, when a malicious OS infers information about encrypted data just by monitoring memory page accesses, and also at network-level when network traffic reveals information about encrypted data. Opaque hides access patterns in the system by using distributed oblivious relational operators and optimizes these by implementing new query planning techniques. It can be executed in three modes: (1) **Encryption mode** - provides data encryption and authentication, while granting correct execution; (2) **Oblivious mode** - provides oblivious execution, protecting against access pattern leakage; (3) **Oblivious pad mode** - extends the oblivious mode by adding prevention of size leakage.

2.4.10 Graphene-SGX

The usage of Intel-SGX and similar technologies have proven to add a great sense of privacy to the storage and execution of data. However, these technologies impose restrictions (e.g., disallowing system calls inside the enclave) that require the applications to be adapted to this technology, so they can benefit from their properties.

Graphene-SGX [57] came to help circumvent these restrictions, while still assuring security to the data. It is a libraryOS that aims to reproduce system calls so that unmodified applications can use them to keep executing normally without interacting directly with the OS or hypervisor. By using a libraryOS, the system is expected to lose performance and, since a new layer of software was added, increase the size of the TCB. Although these assumptions are true, they are quite often exaggerated. Graphene-SGX's performance goes from matching a Linux process to less than 2x, in most executions of single-processes. Graphene-SGX has also shown some great results comparing it to other similar approaches that use shim layers, such as SCONE [5] and Panoply [52], where it shows to be performance-wise similar to SCONE and faster 5-10 percent than Panoply, while adding 54k lines of code to the TCB comparing to SCONE's 97k and Panoply 20k.

Graphene's main goal is to run unmodified applications on SGX quickly. Thus, whilst the size of the TCB is not the smallest comparing to the other approaches, developers can reduce the TCB as needed, as a way to reach a more optimal solution. Graphene-SGX also supports application partitioning, enabling it to run small pieces of one application in multiple enclaves. This can be useful, for instance, to applications with different privilege levels, while still increasing the security of the application.

2.4.11 Other approaches

Other approaches appeared as a way to deal with more specific problems, by making use of trusted computing. We'll go into topics like SGX-Enabled Networking, Administration of Cloud Systems, Virtualization, Containers, searchable Encryption as well as encrypted Databases, and how to take the best advantages of SGX in these specific areas.

SGX-Enabled Network Protocols and Services. The increased need for security seen nowadays caused network-related technologies to become popular, from security protocols (TLS) to anonymous browsers (Tor), leading to a lot of effort by the community to make viable approaches to tackle network security problems. Hardware approaches capable of providing TEEs (e.g. Intel-SGX) are some of those contributions, which deal with modern network security concerns as a way to, for example, solve policy privacy issues in inter-domain routing, thus protecting ISPs policies. In [27] it is shown that leveraging hardware protection of TEEs can grant benefits, such as simplify the overall design of the application, as well as securely introduce in-network functionality into TLS sessions. The same paper also presents a possible approach to reach security and privacy on a network level, by building a prototype on top of OpenSGX, that shows that SGX-enabled applications have modest performance losses compared to one with no SGX support, while significantly improving its security and privacy.

Also at the networking security level, the adoption of Network Function Virtualization (NFV) architecture by applications nowadays implies the creation of an internal state as a way to allow complex cross-packet and cross-flow analysis. These states contain sensitive information, like IP addresses, user details, and cached content (e.g. profile pictures), creating the necessity to ensure their protection from potential threats. S-NFV [51] has proven to be a valid approach, by providing a secure framework for NFV applications, securing NFV states by using Intel-SGX. S-NFV divides the NFV application into two: S-NFV enclave and S-NFV host. The enclave is responsible to store the states and state processing code, while the host deals with the rest. In [51] by implementing the S-NFV approach with Snort [46] on top of OpenSGX was concluded that this SGX-enabled approach results in bigger overheads (approx. 11x for gets and 9x for sets) than an SGX-disabled Snort application, at the cost of extra security.

Trusted Cloud-Based System Administration. The usage of cloud platforms leads to a significant increase in security and privacy risks. Thus, cloud services are now highly dependent on trusting their administrators, as well as their good behavior. Since this is not always the case, it has become a must to protect the users from potential cloud system administration threats. To tackle this problem, some solutions have been proposed with the help of trusted computing technology. However, their focus has been conducted on Infrastructure-as-a-Service (IaaS) environments, which are simpler to maintain than in PaaS and SaaS approaches.

[45] proposes a solution that addresses trustworthiness and security in PaaS and SaaS environments, while preserving essential system administration functions by leveraging Intel-SGX. Thus, this solution provides an environment for cloud customers to review the security conditions of cloud nodes, more specifically those that run their applications and handle their data. The main idea behind this approach is to allow the administrative staff necessary privileges according to escalation policies enforced for different roles, instead

of never granting full administrative privileges on the computing nodes. The administrative roles for cloud nodes are divided into four or five independent roles, depending if it is a SaaS or PaaS environment, each with their own permissions. Each role works under the supervision of internal or external auditors. The internal auditors being the ones hired by the provider, while the external are hired by customers, as a way to execute protocols decided by both in a trustable way. The solution enables control of cloud trusted nodes operational states, designated to run customer's computations, as well as remote attestation of the boot sequence of PaaS or SaaS stacks. Finally, by including logging of changes in node's states, this solution is able to offer trustworthy execution of functions and protocols. The approach was shown in [45] to have a minimal performance impact, as well as low storage overheads while proving to be a compelling approach that can be used to increase the detail of management protocols and tools in cloud environments and data-centers.

SGX-enabled Virtualization. As we already pointed out previously, Intel-SGX has drawn much attention from the community in the last few years, which also made cloud providers start adopting SGX into their cloud systems (Microsoft's Azure confidential computing or IBM Cloud are examples of that). This lead to an increase of interest in developing new cloud programming frameworks capable of supporting SGX. However, while most of the research on Intel-SGX has been concentrated on its security and programmability properties, there are a lot of questions to answer about how the usage of SGX affects the performance of a virtualized system, which are considered the main building block of cloud computing.

In [37] an exhaustive evaluation about the performance of SGX on a virtualized system made some interesting conclusions:

1) Hypervisors don't need to intercept every SGX instruction to enable SGX to virtualization. It was concluded that there is only one indispensable SGX function, ECREATE, which is responsible to virtualize SGX launch control. As a result, glssgx on VMs is considered to have an acceptable overhead.

2) SGX overhead on VMs when running memory-heavy benchmarks consists mainly of address translation when using nested paging. If it uses shadow paging instead, the overhead becomes insignificant. [37] shows that this can be optimized by using shadow paging for EPC to reduce translation overhead and nested paging for general usage.

3) On the contrary, when running benchmarks involving many context switches (e.g., HTTP benchmarks), shadow paging performs worse than nested paging.

4) SGX causes a heavy drop in performance switching between application and enclave, whether it is using virtualization or not. This drop causes server applications using SGX to be affected. [37] specifies that this can be addressed by using mechanisms (e.g. HotCalls [58]) that work as a fast call interface between the application and enclave code, reducing the overhead of ecalls and ocalls, helping the porting of applications to SGX.

5) Swapping EPC pages is really expensive, and this also applies to all systems using

SGX. Upon the start, SGX measures the contents of the enclave, thus triggering enclave swapping if the enclave's memory size is larger than the available EPC size. This was shown in [37] to be optimizable by minimizing the enclave's size, thus reducing swapping and consequently increasing enclave performance. Virtualization causes an additional overhead, which increases based on the number of threads running inside the enclave.

Finally, [37] proposes an automatic selection of an appropriate memory virtualization technique, by dynamically detecting the characteristics of a given workload to identify whether it is suitable with nested or shadow paging.

SGX-Enabled Linux Containers. Lately, container solutions such as Linux Containers (LXC) and Docker have proved to be compelling alternatives to virtualization in cloud computing systems, due to the fact that they need less computing resources, allowing more deployments per physical machine to take place, as well as reducing infrastructure costs. However, some concerns have been raised since containers share a common OS kernel, causing any vulnerability of the kernel to be a danger to all the other containers on the system. While various solutions like Haven [2.4.3], Graphene-SGX [2.4.10], SCONE [2.4.2], Panoply [2.4.5] have been proposed to protect applications and containers in cloud environments by leveraging Intel-SGX, these approaches still generate some concerns, since they lead to a growth of the TCB and enclave size, offer limited support for key features (e.g. remote attestation) and ignore hardware constraints on EPC size (instead of relying on EPC page swapping which, on the other hand, leads to serious performance losses). These issues are the result of a still incomplete infrastructure, from the OS all the way to the application layer.

In [55] these exact concerns are addressed by introducing a platform for Linux Containers (LXC) that leverage Intel-SGX in the cloud environment, called **lxcsgx**. This **lxcsgx** platform offers an infrastructure that supports: (1) Remote attestation; (2) EPC memory control for containers to prevent malicious overuse of resources; (3) Software TPM that can easily allow legacy applications to use SGX; (4) GCC plugin to assist with the partitioning of applications, thus reducing the TCB. In the same paper, **Lxcsgx** was proven in [55] to offer the lowest overhead to the overall system when compared to the previously spoken solutions, while also addressing potential issues these could have.

Databases with Encrypted Query Processing and SGX-Enabled Searchable Encryption. Processing and storing data in cloud environments is still not considered trustworthy enough. Thus, systems started to look at TEEs as a way to change this popular prespective, providing privacy to their users' data. However, working with encrypted data is not always as easy as it sounds, since software-based approaches made specifically for searching encrypted information still lack some properties, good performance being one of the main ones. Well known approaches like Fully Homomorphic Encryption (FHE), although offering extra security properties, are not practical in large distributed systems. As for hardware-based existing approaches, they proved not to scale well due to hardware limitations, as well as depending on a large TCB, becoming more exposed to threats.

CryptDB [40] is a database system that, although it does not offer isolation to the system (does not support any TEE or isolation technique), we think it is important to mention due to the privacy properties it offers, and also to the contribution it had in this particular field, dealing with security and privacy in data storage. It is capable of processing SQL queries over encrypted data, while supporting order-preserving encryption for efficient search, leading to low-performance overheads due to the use of index structures. It also allows range queries on ciphertexts to happen the same way as in plaintext. However, some research [26] was made about this particular type of encryption, proving that it is possible to recover the original plaintexts, which proved to be a big vulnerability, making systems like this incapable of providing the security needed.

As a result, new approaches like Cipherbase [3], and years later HardIDX [16], started to gain relevance for providing security properties to the storage by supporting trusted computing techniques. Cipherbase appeared as one of the first approaches capable of offering security properties to stored data by leveraging secure hardware and commodity Microsoft servers. It extends Microsoft's SQL Server for supporting efficient execution of queries in a safe way, due to the use of FPGAs [15], which is where we start to see some level of isolation. HardIDX came after, as a hardware approach that provides the possibility of searching over encrypted data, leveraging Intel-SGX. It implements only a small core of operations, in particular, searches (on a single value or value ranges), in the TEE. This approach uses B+-tree as a structure to organize all data, which is found in many DBMSs. Unlike previous hardware-based approaches, HardIDX implements a small size TCB and memory footprint in the TEE, exposing a small attack surface as well as granting good performance results while executing complex searches on large chunks of data. It also offers scalability properties, since it can scale the system list of indexes [16].

SGX-Protection for Key-Value Store Solutions. The increasing research on trusted computation technologies, in particular TEEs like SGX, made it possible for cloud users to run their applications safely in potentially malicious systems. One of the most used types of applications in these cloud systems are key-value stores, like REDIS [44] and memcached [34]. This type of data stores is used in many systems nowadays due to their architecture offering fast access to data by maintaining data in main memory, as well as granting durability by writing the data to persistent storage. Due to the importance of these types of systems in today's systems stack, it is important to figure out how to protect data inside of these systems by leveraging trusted technology (in our case, Intel-SGX). However, one of the critical limitations of SGX is the size of the EPC, which represents its protected memory region.

To circumvent this memory restriction, an in-memory key-value store was designed, ShieldStore [28]. It provides fast execution of queries over large data, by maintaining the majority of the data structures outside the enclave memory region, hence contributing to overcome SGX memory limitations.

ShieldStore runs inside the enclave to protect encryption keys, remote attestation, and to perform all the logic necessary to the Key-Value Store execution.

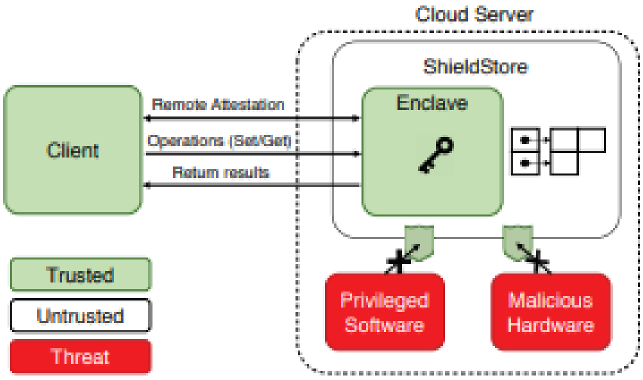


Figure 2.6: Design of ShieldStore

Its design starts by remote attesting the server-side, verifying SGX support of the processor, the code, and other critical memory states of an enclave. By using Intel-SGX libraries, the client and the server exchange keys so that a secure channel between both parties is created. The client then sends a request, to which the server deciphers and verifies, accessing the Key-Value Store for the desired data. The server decrypts the stored data and encrypts it again with the session key previously decided when establishing the secure channel with the client. Finally, a reply is sent to the client. This design shown in Figure 2.6 minimizes the unnecessary memory encryption overhead of paging, and also eliminates EPC page faults, which are the main factors impacting the performance SGX-enabled systems. Adding to the optimization of the index structures, that enabled them for fast access and protection of keys and values, ShieldStore proved to be an efficient and reliable Key-Value store capable of taking the best advantages of SGX.

EnclaveDB [42] is also an SGX-enabled approach designed to deal also with the protection of data, both when stored and queried. It offers confidentiality and integrity by working alongside Intel-SGX in order to handle all the sensitive data (queries, tables, indexes) inside enclave memory, thus keeping the data safe in cases where the database administrator is malicious, when the OS or hypervisor is compromised, or even when running the database in an untrusted host.

EnclaveDB is divided into two modules (Figure 2.7): trusted, running inside the enclave, and untrusted, outside the enclave. The trusted compartment hosts a query processing engine, a transaction manager, pre-compiled stored procedures, and a trusted kernel responsible for sealing and remote attestation. As for the untrusted module, it is responsible to run all the other components of the database system. This approach fundamentally provides a database system with a SQL interface capable of ensuring security guarantees, while dealing with low overheads. Also, by depending on a smaller TCB

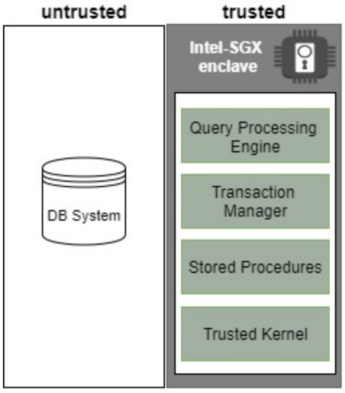


Figure 2.7: Overview of EnclaveDB compartments

than any other conventional database server, the security provided increases considerably, making EnclaveDB a valid approach to work as a trusted database system.

VeritasDB [54] is a KVS that guarantees integrity to the client in the presence of exploits or implementation bugs in the database server. In this approach, the protection enabled by SGX is not focused on the KVS service itself, but in the protection of the intermediation between clients and KVS operations. VeritasDB is implemented as a network proxy that mediates communication between the unmodified clients and the unmodified database server, which can be any off-the-shelf database engine (e.g. Redis, RocksDB, or other solutions). Since the proxy is trusted, the solution addresses security primitives supported in Intel-SGX enclaves, to protect the proxy's code and state, thus completely eliminating trust in the cloud provider. To perform integrity checks in the proxy, the VeritasDB includes an authenticated Merkle B-tree that leverages features of SGX (protected memory, direct access to unprotected memory from enclave code, and CPU parallelism) to implement several novel optimizations based on caching, concurrency, and compression.

SPEICHER [7] was another approach designed as a secure storage system that not only provides strong confidentiality and integrity properties but also ensures data freshness to protect against rollback/forking attacks. SPEICHER exports a Key-Value (KV) interface backed by Log-Structured Merge Tree (LSM). The solution provides secure data storage and trustable query operations. SPEICHER enforces the security properties on an untrusted host by leveraging shielded execution based on a hardware-assisted TEE — more specifically, Intel-SGX. The design of SPEICHER extends the trust in shielded execution beyond the secure enclave memory region to ensure that the security properties are also preserved in the stateful setting of an untrusted storage medium. To achieve these security properties while overcoming the architectural limitations of Intel-SGX, the authors designed a direct I/O library for shielded execution, a trusted monotonic counter, a secure LSM data structure, and associated algorithms for storage operations. The SPEICHER prototype is based on the base RocksDB [1] KVS and evaluated using a RocksDB benchmark suite called db_bench [13]

Finally, we thought it was important to mention again HardIDX since it is an approach to search for ranges and values over encrypted data using hardware support, making it deployable as a secure index in an encrypted database. The approach joins a security proof explicitly including side channels and the protection of the secure index by leveraging Intel-SGX. In a more focused vision, HardIDX is deployable as a highly performant encrypted database index optimized to require only a few milliseconds for complex searches on large data and scale to almost arbitrarily large indices. The authors argue that the solution only leaks access patterns with the trusted code protected by SGX hardware being very small.

2.5 Summary and Discussion

After going through all those approaches, we can look at the technologies presented in this chapter and pick the ones that we think to suit better our objective for this thesis.

Starting with OS isolation systems discussed in 2.1, although capable of assuring a good sense of privacy from the OS/hypervisor, they do not take into account the potential hardware vulnerabilities that a system can have while making use of sensitive data. With that in mind, opting for a TEE approach made more sense for offering a more complete sense of security. While TPMs have proven to have problems adapting to the Cloud, modern TEE technology, on the other hand, has proven to be the way to go, by combining hardware solutions with specific software.

Going deeper into these modern technologies in section 2.3, we knew beforehand that our focus would be on Intel-SGX. However, we picked it from the two most used (along with ARM TrustZone) mainly due to the fact that ARM TZ supports only one secured zone, which led it to be more adopted by the phone industry, and it is not what we are focusing in this dissertation. On the other hand, Intel-SGX is able to support multiple secured zones (enclaves) for each processor, which is really relevant in cloud systems due to the need to manage data from multiple users at once. Also by not requiring additional code to grant hardware attestation, since its design supports it, Intel-SGX is able to keep a smaller TCB than ARM TrustZone.

Since the adoption of TEE technologies was found to drop significantly the overall performance of systems when used by itself, in 2.4 we go into detail on how different existing technologies can positively impact the work of SGX. Since our focus resides on running applications in this kind of security environment, we found SCONE [2.4.2] to be a good fit. By running containers on top of SGX with access to a small library of restricted and secured system calls, this solution has proven to improve significantly the performance of some applications running on top of SGX, while still allowing to obtain integrity and confidentiality for the data, as well as other valuable characteristics such as scalability and ease of deployment, which can come in handy when testing our prototype.

Chapter 3

System Model And Design

In this chapter, we present an overview of the system model for our solution. We introduce an architecture model that is able to assure confidentiality and integrity to the execution of unmodified applications that run sensitive data on cloud computing servers, by leveraging trusted computing techniques provided by both software and hardware, all according to our adversary model which focus fundamentally on dealing with attackers that can access and read the data during runtime, thus taking value from it. All this without crippling too much the performance levels of the overall system.

In Section 3.1 we describe a general overview of the system as a whole, introducing the components that make the system. In Section 3.2 we talk about the assumptions kept in mind while creating our solution, as well as defining what is out of our scope. After that, in Section 3.3 we go into more details about the components that make part of the whole system, explaining in a fine-grained view how each component works, and what is their purpose in the solution. Section 3.4 mentions the tradeoffs that our system model faces. We present negative impacts that we expect the technologies we included in our model might add to the system. In Section 3.5 we take a look into potential problems that we acknowledged our solution's design might induce, and lastly, we end with a summary in Section 3.6.

3.1 System Model Overview

Our solution can be seen in figure 3.1, which is a macro overview of the system. The solution we implemented can be divided into two parts - client-side and server-side - in which the clients interact with the server, which is protected inside a TEE.

Although the **Client** and its purpose are easy to understand - make requests to the server via network - the server its components are more complex. Hereupon, the server is composed by:

- **Proxy:** Works as an entry point to the whole server-side. It's the component responsible of handling the communication with the Client;
- **Authentication Server:** Authenticates the clients so they can only access the system if they are authorized;
- **Attestation:** Allows the components of the system to prove their identity, thus allowing the rest of the components to consider them trustworthy;
- **Key-Value Store:** Stores data in-memory. This is the main component we intend to protect, since it is where the information running is held.

All the server components itemized above run inside a third-party cloud server on top of trusted hardware (TEE), with the exception of the Authentication Server.

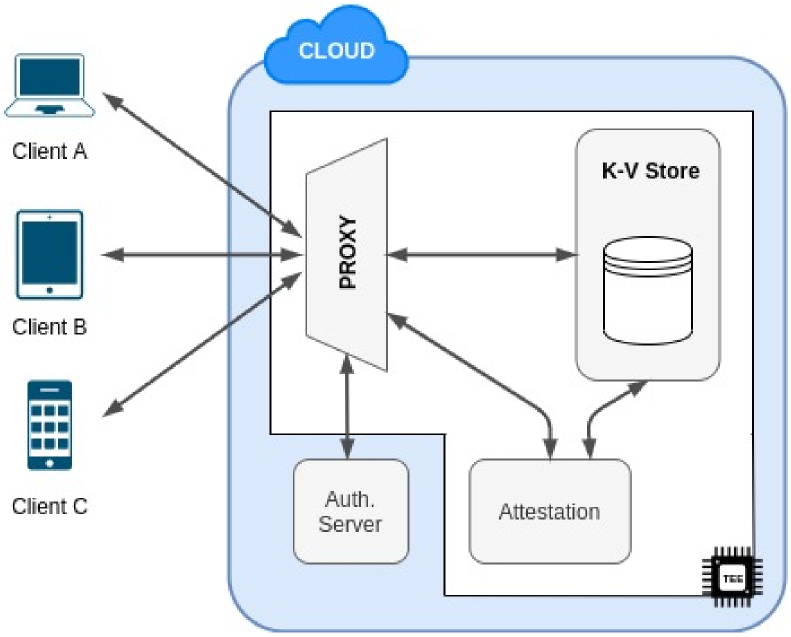


Figure 3.1: General overview of the System Model

3.2 Threat Model And Security Properties

Our solution is designed to offer privacy-enhanced guarantees in protecting data confidentiality, while also ensuring integrity and completeness of results returned to the clients, which we do by ensuring that sensitive data never runs in plaintext. Thus, our system model protects from attackers with intentions of accessing sensitive data and taking advantage and value from it, regardless if the attack is coming from the inside or outside the host system.

3.2.1 Adversarial Model Definition

Since the main objective of the solution is to protect data privacy during its execution, we specifically focused on two types of potential threats:

1- Users that attack the system and find a way of getting access to high privileges. This is a type of user that can control the system as he pleases, with superuser access, meaning that he will be capable of manipulating the host OS and other low level privileged components, through which he will manage to access the data running in memory;

2- Honest-but-curious users, which are users that already have higher privileges, and may or may not have direct access to the hardware. They can snoop easily on private data, since they are considered trusted, so they can read it, learn it, and take advantage of it.

We consider being out of scope denial-of-service attacks, side-channel attacks that exploit timing and page faults. It is important to refer that with this solution, since we are focusing on using in-memory KVSs, our target will not include ensuring confidentiality and integrity to data stored in disk since we do not resort on persisting the data.

3.2.2 Countermeasures For Privacy-Preservation

Since our objectives are pointed towards an isolated system capable of offering security and privacy properties, we depend a lot on isolation techniques to make this possible, provided by both hardware and software (containers).

We looked at TEE technologies capable of assuring computation and storage security to our system's data during runtime, although implying a performance tradeoff.

In addition, to grant an extra layer of isolation and to ensure privacy to the data in each element of our model, we opted to use containerization as a way to keep them independent and the system modular and scalable, enabling ease in the deployment of software running inside the containers, whether it is an OS, a library OS, or even entire applications. Running our system inside containers allows it to be deployed in a very similar way, whether running locally or in the cloud, which can be very helpful in the implementation process.

Since our intention is to expose our solution as a service, we added access control policies, verified at the entry point every time a client performs a request, while also assuring that all the communications were done via TLS, whether internally in the system or to external entities.

3.3 System Architecture

In this section, we dive into more detail about the components that make part of the system and what purpose do they have in the solution.

Our system, which is based on the system model introduced in Section 3.1 can be split into two parts, one part being the Client-side, responsible for making requests to the system, and the other being the server, which deals with the execution and storage of the data.

For the Client-side, we only considered them to be benchmark applications and not entire web applications, with only the intent to evaluate the system for our experimental analysis shown later in this dissertation. Thus, we assume that the client is trusted, as long as he can authenticate himself in the system.

As for the server, our goal is to provide privacy to sensitive data running on top of trusted technology - TEEs - without crippling too much the performance levels of the application running. Hereupon, our server-side, which we run inside a Cloud server, can itself be divided into multiple components: a Proxy, an Authentication Server, an Attestation component and a KVS component. All these, apart of the Authentication Server, are running inside containers on top of a TEE, more specifically Intel-SGX.

According to what we have already studied, to run applications on top a TEE usually causes the system to take a performance penalty. To mitigate those penalties, we included additional layers of technology, frameworks specifically designed to work with these trusted technologies and to soften the performance impact TEEs have on applications that leverage their properties.

3.3.1 Client-Side Operations

For the client-side, as mentioned at the beginning of this section, we only considered benchmark clients. We use these benchmarks to evaluate the system by making simple requests to the Proxy, while calculating metrics that we found essential to use in our practical evaluation of the solution, in order to validate the full-fledged conditions in supporting the KVS REDIS. And as long as the user is registered in the authentication server, it is considered trusted.

To start a communication with the server, a client:

1. Reaches the authentication server to authenticate itself;

2. Interacts with the proxy after being granted authorization to do so, as long as the authorization is valid, as we can observe in Figure 3.2.

All this communication process is secured through TLS over HTTP.

3.3.2 SGX-Enabled REDIS Solution

All the components that make up the server, with the exception of the authentication server, run on top of a TEE, that being Intel-SGX. As we studied, applications can't simply be placed on top of a TEE and be expected to perform as efficiently as they do without this extra layer of security. Since SGX completely isolates what is running inside its enclaves from the rest, even from the OS, it cripples the performance of the system for multiple reasons. Every time a system call needs to be performed by running code, the thread of execution needs to leave the enclave to execute it. Only after it has completed executing the system call, the thread can come back inside the enclave. This process takes

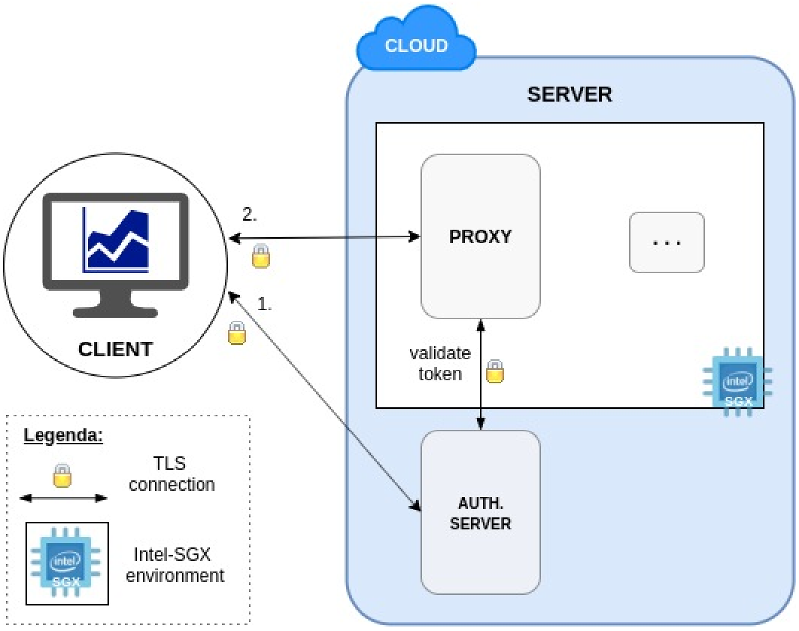


Figure 3.2: Communication between client and server

a lot of effort for the system since it involves a lot of encryption functions to take place. Also, since enclaves are small in-memory regions (size depends on the hardware used), when running a normal or large-sized application on top of SGX it usually means that the code does not fit all at once inside the EPC. Thus, parts of the application need to leave the enclave in order to fit the other parts that are needed at that time, resulting in a lot of encryption and decryption taking place due to page swapping between the EPC and the rest of the memory, in order to keep the integrity of the data.

Hence, we opted to use SCONE, which we covered in Section 2.4.2, as a way to better leverage SGX properties. It allows us to run the server components mentioned before on containers capable of running applications inside SGX enclaves with effectiveness. This happens mainly because SCONE containers include a small library of system calls that can be accessed statically inside the enclave. Also, it supports asynchronous system calls, meaning that when dealing with the need to execute a system call outside the enclave, SCONE switches the execution thread to one running outside the EPC, avoiding the performance penalty caused by the need for a thread to exit the enclave.

Thus, we mitigate some of the downsides of the usage of SGX, so that the following components that are part of the system can leverage SGXs security properties with ease:

1) Proxy - Although it is an extra layer of overhead, we thought the addition of a Proxy component to be a worthy investment because it is designed to serve multiple purposes. First of all, we use it as a gateway for the system to communicate with the outside. It acts as a single point of access, enabling the rest of the system to scale, adding no extra complexity to the client-side. By having a Proxy, everything the client has to do is to reach the Proxy itself, while the logic regarding the redirection to the correct server instance that will manage the client request will be done by the Proxy. Adding to that, it allows the system to only need a single firewall, instead of configuring one for each KVS instance.

The proxy is also responsible for only allowing authenticated clients to access the system, through interactions with the authentication server.

2) **Authentication Server** - We added an authentication component responsible for authenticating every client that wants to interact with the system. Upon validating a client authentication request, the server assigns an access token (JSON Web Token) which the client will then use to make every request to the Proxy. The Proxy then validates the token, and if it proves to be valid, the request is forwarded to the system. However, the communication between Proxy and Authentication Server to make this validation does not have to happen every time, since the Proxy can store the token's public keys in cache, thus being able to verify signatures and expiration times without contacting the Authentication Server.

Although some KVSs (i.e., Redis) have the possibility to configure authentication for each replica, we believe that option would add an extreme layer of complexity that we do not want, due to the fact that each replica needs to be configured individually. Therefore, if we use a cluster of KVS instances and begin to scale their number, the complexity of that task every time a new user is being granted permission to access the system will be huge. Thus, by including a server designed only to deal with the authentication process, the configuration only needs to happen once for the system to know which users are allowed.

In figure 3.3 we see that in the first interaction with the system, the Client reaches the Authentication Server in order to get an access token to use as proof of authentication in its future requests. This token is valid for only a certain period of time. After it has expired, the Client needs to reach the Authentication Server again, in order to get a new valid token. While the token is valid, the Client can make requests to the server-side by interacting with the Proxy. The Proxy then validates the token with the Authentication Server and only upon passing this validation will the Proxy process the Client request.

We do not run our Authentication Server inside SGX, thus making it the only server component to not leverage SGX security properties. Ensuring security to the external authentication of clients is not the main goal of this thesis, therefore we considered it to be out of the scope.

3) **Attestation** - For the component accountable to deal with attestation for the system, we follow a remote attestation policy where a remote system is the one who holds the defined secrets and provides them to the system components when they successfully attest themselves to this remote server. Those secrets are essential for the boot process of the application, thus without attesting themselves, applications can not start.

As we mentioned earlier, we resort on SCONE as a way to run our components inside

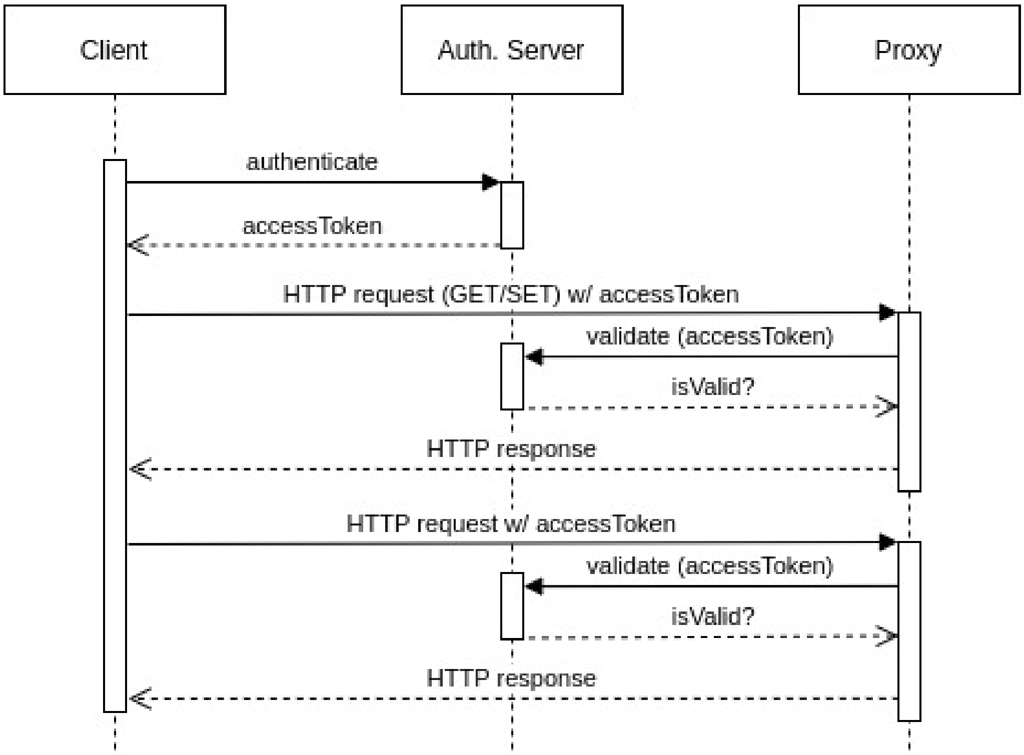


Figure 3.3: Authentication Process

SGX. SCONE also offers a mechanism to attest the enclaves where SCONE containers are running the applications, which we use to add the attestation property to our system and to assure that the server components run inside SGX enclaves.

SCONE's attestation mechanism, called SCONE Configuration and Attestation Service (or CAS) [50], consists of exposing a remote component provided by SCONE itself that manages the secrets (i.e., keys) of an application, to whom enclaves will try to prove their identity in order to access those secrets, so they can execute their designated application.

In CAS we define each application access policy, reflecting which enclave have permission to execute them. The confidentiality and integrity of these policies and their secrets are ensured by CAS itself. To modify and read a policy, the client (in this case, us) needs to have knowledge of the private key that pairs with a public key, which is stated in the policy itself upon creation. Thus, no admin managing CAS can read or modify the policies defined remotely.

For an application to prove their identity to CAS, and thus get access to the secrets, it needs to be attested locally so it gets the attestation key involved - a key pair that CAS shares with a local component so they can validate each other's involvement. CAS then recognizes the key and the attestation is considered valid (although no access to the secrets is given yet). With that in mind, SCONE's attestation mechanism adds a second component, which runs locally inside the system environment, the Local Attestation Service (or LAS). LAS receives attestation requests by the enclaves that intend to attest themselves to CAS, and signs those requests with the attestation key, creating a quote that can be verified by CAS. After getting this quote, any enclave can reach the CAS to try and access the secrets to run the application, only reaching them if they are allowed by its defined policy. This completes the attestation process of an application running in a SGX enclave.

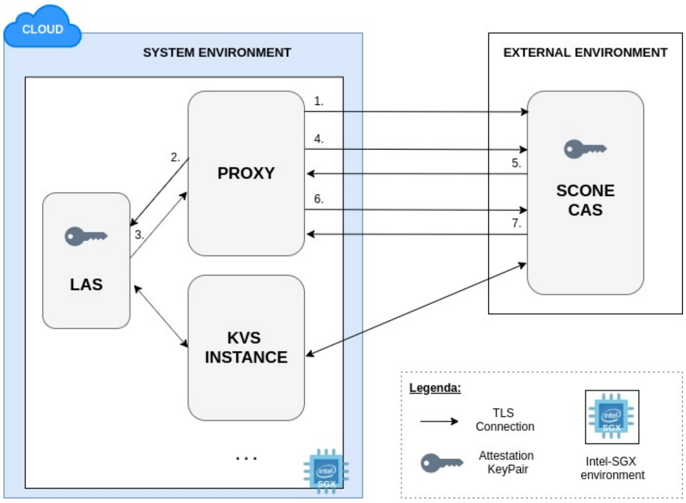


Figure 3.4: Attestation Model

In Figure 3.4 we present a visual display of the attestation process described above, taking place inside our system model.

The attestation starts with the enclave of an application (i.e., the Proxy component of our system) establishing an HTTPS connection with the remote component CAS endpoint. Afterward, the enclave requests the attestation from LAS, which will then sign a message with the attestation secret key, resulting in a Quote. The Quote is then used by the enclave to validate its own attestation with the help of CAS. After validation, the enclave requests access to the secrets defined in the policies that CAS holds. If the enclave is indeed specified to run the referred application, CAS shares the secrets, thus completing the attestation process with success. Note that the secrets injected from CAS into containers are always stored in protected memory and are not accessible by anyone but the code running on the same enclave. In Figure 3.5 we present a diagram with the steps just described.

4) **Key-Value Store -** For the KVS component, we use the Redis KVS that can be used with different configurations, offering multiple strengths to the execution and storage of data, especially if run in Cluster mode, offering scalability, fault-tolerance, and eventual consistency of data, in some cases without even resulting in any overhead. If each replica of the cluster is set to run in independent machines, the majority of the complexity is handled by the network, allowing the performance levels to match the values obtained by a single instance Redis while still offering all the properties mentioned above. Redis supports different configurations:

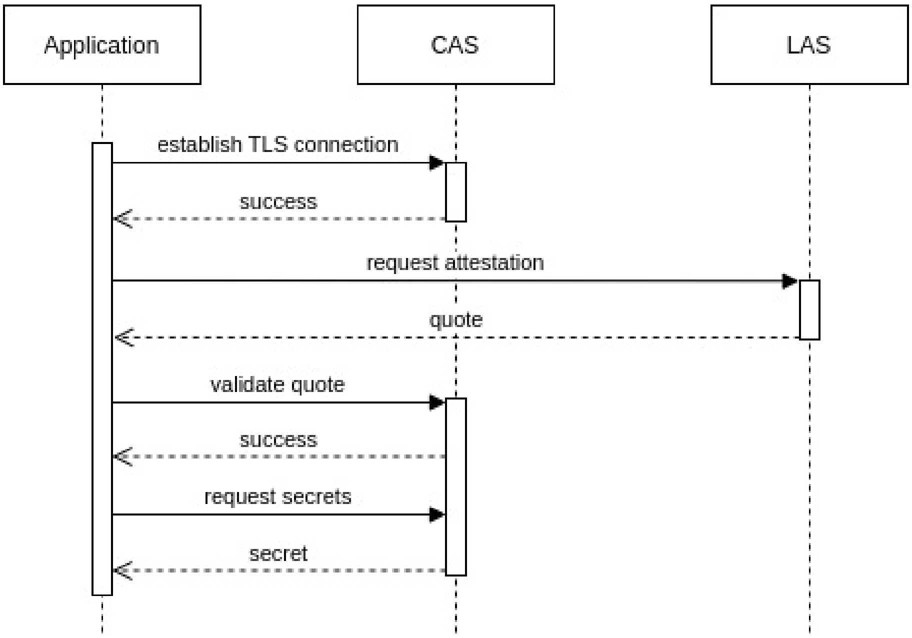


Figure 3.5: Attestation Process

Standalone. The simplest configuration that a Redis instance can run in. It offers the properties of a single Redis database. It is very simple, very stable, and easy to maintain.

Master-Slave. Offers replication and eventual consistency of data, with writes only possible in the Master node, and read-only Slave nodes.

Cluster. Although it is the most complex configuration, beyond replication and consistency, it also adds huge scalability possibilities to the system. It also considers Slave nodes to be read-only.

In our solution, we deploy Redis instances capable of running in all the configurations mentioned - Standalone mode, Master-Slave, and Cluster - as a way to test the behavior of the system in each of those configurations. It is also important to note that each Redis replica runs inside its own container, regardless of the configuration it was set to run in.

All the communications between the components that make up the server-side are secured by TLS over HTTP, as a way to keep confidentiality of the data during communications all over the system. Access to these TLS libraries inside SGX is possible due to the inclusion of openSSL on the static library that SCONE containers offer. By having openSSL statically inside the enclave, the overhead from switching to a thread outside to handle the system call is mitigated, thus the code can establish an HTTPS connection directly from inside the enclave. However, the addition of extra libraries to the TCB induces a tradeoff, since it increases the memory needed for the application to execute, thus consuming more memory space inside the EPC, which can eventually lead to more page swappings.

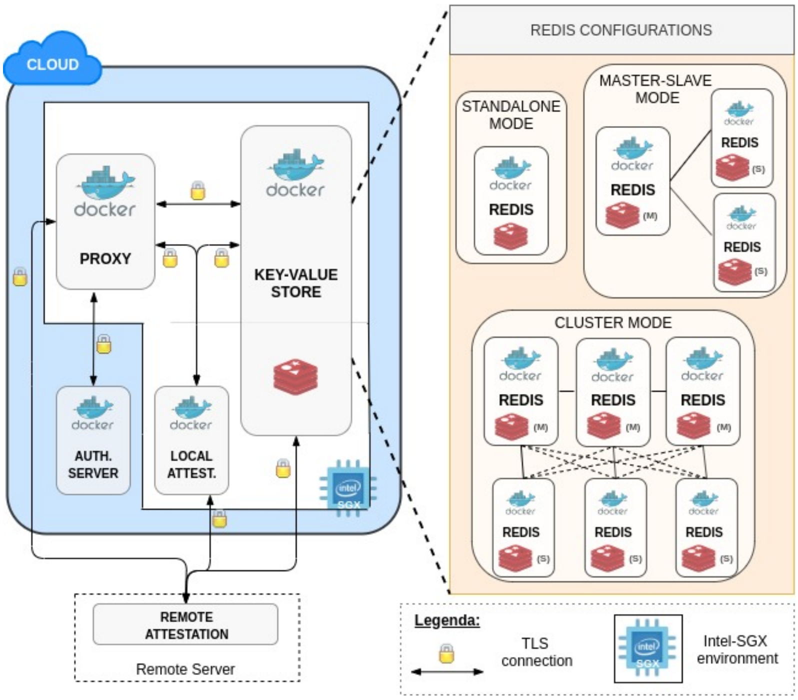


Figure 3.6: Server-side overview of the solution

3.4 System Model Design Tradeoffs

Although our system model focus on assuring confidentiality and integrity to application data running inside a third party system, by leveraging trusted computation techniques, this extra security layer comes at a cost. Tradeoffs have to take place, particularly between security and performance, where more security usually means less performance. Thus, the usage of these trusted techniques are not always considered worthy, alongside the fact that the levels of isolation they assure can also mean less fluidity for the system in general.

TEEs give the system extra levels of security during the execution of data by isolating the code from the rest of the system, even from the high privileged components. This isolation is either given by encrypting a VM in which the code is executing, or encrypting only the region of memory dedicated to run the code.

In our system model, SGX works alongside the second option, encrypting the enclave memory region dedicated to the application, while also trying to keep the TCB as small as possible, limiting the functions supported inside the enclave as a way to increase the level of security. This leads to a tradeoff since less supported functions inside the enclave mean that the thread of execution will more likely need to leave the enclave in order to execute system calls, or another kind of essential operations, resulting in major overheads due to all the encryption involved. This has a big performance impact when dealing, i.e., with network-heavy services, which usually have a high system call frequency.

Also, by only relying on the encryption of the enclave memory region, SGX encounters problems when confronted with applications that are bigger than the enclave itself (keep in mind that an enclave size is usually really small, only around 64MB and 128MB [5]). When the memory space needed for the application is bigger than the enclave size, page swaps between the enclaves EPC and the untrusted memory take place, through a mechanism provided by SGX itself (although not yet supported on machines running Windows OS). Since this scenario involves a lot of encryption and decryption to happen, it induces the system into huge performance overheads.

SCONE, along with some other recent technologies, has been referred to as a possibility to help mitigate some of those performance losses. It does it by: 1) trying to level the scale between the two factors, security and performance, and 2) maximizing the time threads spend inside the enclave. The first factor due to keeping the TCB as small as possible, but with enough functionalities to allow the system to perform with efficiency. As mentioned before, SCONE containers include a small (and trusted) library with system functionalities that can be used inside the enclave. It increases the TCB in a controlled way while improving the performance levels and fluidity of the system. And the second factor by allowing asynchronous system calls. This enables the possibility to swap the execution from inside the enclave to a thread outside, whenever a system call left out of the TCB is needed, thus avoiding threads to exit the enclave more often.

However, despite helping to soften some of the tradeoffs related to the usage of trusted technology to run our system model, using SCONE doesn't make our solution perfect. Thus, we still expect to observe security and performance tradeoffs, however we are confident those to be way less expressive with the help of SCONE.

3.5 Open Design Issues

Looking back at our model, we can think that the whole availability of the system depends highly on the single proxy instance working as entry-point for the whole system, and so all can be compromised if it fails to work. Replication of the proxy instance can be seen as a measure to mitigate this problem, and although this is doable, we considered it to be out of scope for what we choose to evaluate in this thesis.

3.6 Summary

We designed our model with the objective to grant integrity and confidentiality to applications with data running inside a cloud host. With that in mind, we focus on finding a solution that enables the use of a TEE, in this particular case Intel-SGX, while still assuring good performance to the system. For that, we adopted SCONE as a mediator between the application and the trusted hardware. SCONE provides secured Linux containers that can be used to deploy entire applications with ease on top of SGX, and allows better performance levels when running applications inside SGX enclaves, due to the inclusion of a static small library of system calls directly inside the EPC, thus minimizing the number of times enclave exits need to happen. This is a huge factor since it is considered a major performance dropper on applications running with SGX. With SCONE we are able to deploy various configurations of Redis KVS on top of SGX with relative ease, allowing us to assure our initial objectives: grant integrity and confidentiality to the data running and being stored in memory. As for the communication process, to better protect the system, we implemented a Proxy server. It serves as a gateway to the entire system, and it's responsible to validate the authentication of the clients, which is granted by an Authentication Server. Lastly, we use an attestation mechanism, also provided by SCONE, which is responsible for attesting all the components running in the system, through a remote attestation mechanism. To note that all the referred components run on top of SGX with the help of SCONE.

Also, by protecting each communication link with TLS, either between client and server or between the server components themselves, we can assume that our system complies with the adversary model defined.

In the next chapter, we will show how we implemented the prototype for the system model discussed here.

Chapter 4

Implementation

In this chapter, we describe the implementation of our prototype TREDIS, or Trusted REDIS, implemented on top of a TEE instantiated through Intel-SGX. We present how we implemented the system components that make up our system model defined in 3. Our implementation is deployed in an online repository in Github [1].

We start by presenting the environments in which we implemented our solution in Section 4.1, along with some general technologies we used to implement our system components. Then in Section 4.2 we describe in more detail the implementation of the components that make up the system, explaining the implementation process along with the technology stack we used to implement each one of them. Lastly, we finish with a summary in 4.3.

4.1 Implementation Architecture

Our prototype complies with the system model described in the previous chapter in 3.1 and, as we mentioned there, can be divided into two distinct parts, the client-side and the server-side. The first one consists of benchmark applications, that measure simple requests to the server-side of the system. We run the client on a local machine running Ubuntu 18.04.3 LTS OS on top of commodity hardware:

```
CPU:Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz - 4-core
RAM: 16Gb DDR4 2400MHz
Storage: Intel SSDPEKKF512G8L - 512Gb SSD M.2 NVMe
Network: Ethernet Connection (4) I219-V 1Gb/s
```

[1] https://github.com/jcreis/thesis_implementation

As for the server part, which runs the TREDIS solution itself, we implemented it in an environment hosted on OVH [2] Cloud, on a machine running Ubuntu 18.04.4 LTS 64 bits with the following hardware specifications:

```
CPU: Intel(R) Xeon(R) E-2288G CPU @ 3.70GHz - 8-core
RAM: 4x32Gb DDR4 2666MHz
Storage: Cannon Lake PCH SATA AHCI Controller - 4Tb HDD
Network: Ethernet Controller 10G X550T 10Gb/s
```

Note that Intel(R) Xeon(R) E-2288G is SGX-enabled, which allows us to run our TREDIS solution on top of a TEE as we intended, with 128Mb size enclaves.

In order to increase the privacy levels of our system, we opted to run each component of our TREDIS solution inside containers, as a way to increase isolation, while keeping the system modular and scalable. For that, we use Docker's version 19.03.6, which also allows us to integrate SCONE technology as a way to mitigate SGX performance issues.

Thus, we run SCONE[3] images inside each component container, allowing the application there deployed to execute inside SGX enclaves with more efficiency. To note that the SCONE curated images we used run with Alpine Linux version 3.8.5 with kernel 4.15.0-101-generic.

Communications between server components are secured via TLS 1.2. Since we do not have a signing service, we generated our own Certificate Authority which we used to sign the certificates for all the components.

4.2 Implementation Components And Options

In this section, we go deeper into the implementation details of our solution, specifying the technologies we used to implement each piece of it.

4.2.1 TREDIS solution

TREDIS, as we detailed in 3.3.2, is running in the OVH cloud environment and can be broken into four major components, that together make up the solution we intend to evaluate:

Proxy. Our Proxy component consists of an API implemented in Java 1.8.0_201 with the help of Spring Boot v2.3.1.RELEASE. It works as an entry point to our entire solution, facilitating client access to the system while also enabling it to scale.

We implemented the proxy to accept requests over a defined endpoint in order to manipulate data inside in-memory Redis instances, through Jedis v3.3.0, which is an open-source Redis Java client provided by Redis itself. Jedis enables our API to perform operations over the Redis KVS in any configuration it is set to run, whether it is running in

[2] https://www.ovhcloud.com/

[3] https://hub.docker.com/u/sconecuratedimages

Standalone mode or one of the more complex ones, Master-Slave or Cluster. It also allows us to set TLS connectivity to the KVS instances with two-way authentication, where each of the endpoints (Proxy and Redis instance) trusts each other's certificates, thus securing communications between the Proxy and the Key-Value Store components.

Our proxy also holds the Authentication server certificate. It establishes a TLS link with the Authentication server in order to validate access tokens upon Client requests.

Authentication Server. For our Authentication Server, we used an open-source identity and access management server called Keycloak v11.0.2. Keycloak grants access tokens to clients that are configurated inside its own database. It works well with Spring boot framework which we used to implement the Proxy, since it can be easily configured in order to check the validity of tokens received by the Spring API.

To note that this is the only component that is part of the system and doesn't run on SGX enclaves. Instead, it runs in a Docker container with a non-SCONE image.

Attestation Mechanism. In order to attest our system components that run on top of SGX, and also to reassure that they indeed run inside enclaves, we followed the SCONE's attestation mechanism consisting of a remote attestation policy that we described in the previous chapter.

In SCONE's approach, the remote entity CAS manages all the defined secrets for the applications in the system. This entity is provided by SCONE itself, and since the service that provides images of CAS requires a subscription, we implemented the attestation mechanism using their public CAS instance, *scone-cas.cf*[4]. We posted to the remote CAS instance the sessions where enclave's hash values are specified, along with the secrets for each application. We followed SCONE's advice on how to approach the use of secrets, by using implicit attestation with the help of TLS - *"The idea is that a service can only provide the correct TLS certificate if it runs inside an enclave. To do so, one would give the enclave an encrypted TLS private key in the file system (can be generated by Scone CAS if this is requested) and the enclave gets only access to the encryption key after a successful attestation. The decryption of the TLS private key is done transparently by SCONE."*. Thus, the secrets (TLS encryption keys) can only be obtained if the hash value of the enclave that is trying to get them matches with the one specified in the secret. This allows us to enforce the applications running with SCONE to run indeed inside an enclave.

Although, before CAS can verify the enclave's hash value, the enclave first needs a quote from a quoting enclave - LAS. LAS is running in our application inside a SCONE container, and is the component that holds the attestation key used to create a Quote, which is a message signed by LAS with that specific key that CAS will be able to verify. Thus, CAS will know that LAS has locally attested the enclave, and can then proceed to check if the enclave is entitled to the secrets or not.

After having LAS running locally in a container and by using the public CAS instance

[4] https://sconedocs.github.io/public-CAS/

provided by SCONE, we were able to add this attestation mechanism to the rest of our system components quite easily. After having the secrets defined and posted in CAS, we only needed to include a list of environment variables upon creation of the applications that we intend to attest in our system, where we specified both the CAS and LAS addresses as a way to perform this attestation upon each component start. A better demonstration on how to do this configuration is present in [47].

All the communications are secured through TLS, either between CAS and the containers running applications, or between those containers and LAS instance.

Key-Value Store. As we already mentioned before, we implemented our Key-Value Store component by using Redis KVS. To run Redis with SGX security properties, we had to run a Redis image incorporated with SCONE, to run in a secured container.

However, Redis images are mounted according to a configuration file, which is set upon their build. And since one of our objectives is to enable the KVS component to run in various configurations in order to evaluate its behavior on top of SGX, we build Redis images with customized configuration files for the Redis instances to run in Standalone, Master-Slave and Cluster modes. All the Redis images we build use Redis version 6.0-rc1 and have SCONE's[5] as the base image. To emulate the Master-Slave configuration, we went with only three Redis replicas, one Master node, and two Slave nodes. As for the Cluster configuration, we included six replicas, three Master nodes and three Slave nodes. Note that writes can only be executed by Master nodes since the Slave nodes are read-only.

Adding to that, building our own Redis image allowed us to specify the keys and certificates we intend to use in every instance, needed for establishing secured TLS connections with other components over the network. To note that in Redis configurations set to run with multiple Redis instances, like Master-Slave or Cluster, we use the same keys and certificates in every instance in order to facilitate the implementation of such configurations. We consider this to be unpractical and unsafe in a real-world application, however, we opted to do so in order to prevent an extra layer of complexity for the implementation phase of our solution.

Each component that runs inside a container with a SCONE-based image in our TREDIS solution, and so on top of SGX enclaves, is deployed as Figure 4.1 shows:

A container runs a Scone-based application image on top of an enclave. Inside the enclave, besides the application code, it is present a small static library provided by SCONE which allows the application to make some system calls (the ones that are included in this small library) to the OS running in the host machine. OpenSSL, which provides TLS and SSL protocols, is one of the static libraries included by SCONE, but represented in the figure above as independent from the static library component due to its importance. By having OpenSSL statically inside the enclave, components can communicate with

[5] sconecuratedimages/apps:redis-6-alpine

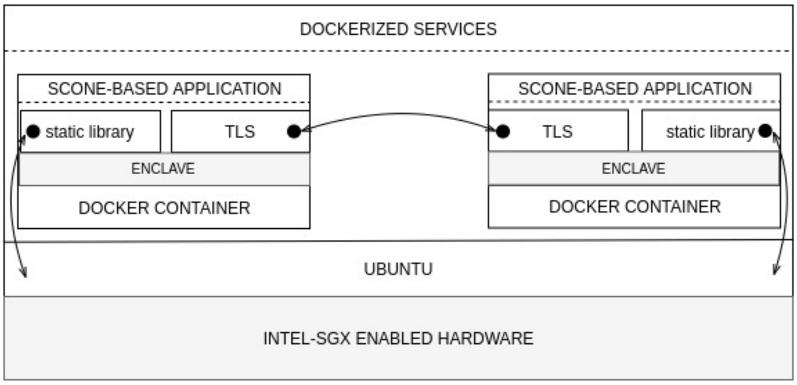


Figure 4.1: Server Component Technology Stack

each other securely over the network without directly inducing any tangible overhead. However, this can indirectly result in more page swapping events because of the increase of the TCB size, which leads to more memory occupied in the EPC.

4.2.2 Client-based benchmarks

To implement the Client, first we tested our solution directly against the KVS instances themselves, in order to get base values for the metrics we intend to study. For that we used redis-benchmark, which comes directly with the installation of Redis itself, therefore its version is induced by the Redis version present on the machine.

However, since our solution was designed with a customized entry point API (our Proxy component) which redirects client requests to the Redis KVS instances, we were unable to find a way of setting redis-benchmark to make requests to the Proxy. Thus, in order to perform an experimental evaluation over the entire TREDIS solution, we opted to used Jmeter version 5.3 as a way to reach our Proxy endpoint, thus benchmarking the behavior of our solution without needing to exclude any component. Also, by being designed by a different organization than the provider of the KVS we are studying, it gives an extra level of guarantee in our results, just in case.

With the clients defined as above, we were able to evaluate our system as we intended. We made simple CRUD operations to the in-memory Redis KVS instances running inside the TREDIS solution, either through the Proxy or directly to Redis instances, along with some other specific tests that we will detail later in this thesis, in order to analyze performance levels, scalability capabilities, resource usage, and other metrics we found we needed to evaluate.

4.3 Summary

We implemented our system model defined previously in two environments. One to emulate a client running in realistic conditions, running in a local machine with commodity software, that only makes requests to our TREDIS solution. Another inside a Cloud host, to emulate the use case we intend to approach, which is the main focus of this thesis: to assure integrity and confidentiality to applications (in this case a Redis KVS) running on a third party system host, by leveraging trusted technology properties without inducing in major performance overheads.

For the client component running in our local machine, we used redis-benchmark and Jmeter v5.3 as benchmark applications. The first one only when making requests directly to the Redis KVS, but since we implemented a Proxy component that redirects client requests to the Redis by exposing an API, redis-benchmark shows limitations when working with a mediator to its requests to the KVS. Thus, we adopted Jmeter to evaluate the solution with the API working as an entry-point to the system. By working with this two different client applications, and although one can not be used for every scenario, we can evaluate the system in a better, less biased way, since redis-benchmark and the KVS whose behavior we are studying share the same developer entity.

For the server component running in the Cloud environment, which is where our TREDIS solution really is, allowed us to run our components as we intended to: inside containers and leveraging SGX security properties with the help of SCONE, with the exception of the Authentication Server, which runs in a regular docker container. Our Proxy was implemented in Java 1.8 with the help of Spring Boot v2.3.1. We used Jedis v3.3.0 to reach the KVS in order to translate client HTTP requests into operations over the in-memory Redis database. We established a two-way authentication TLS connection between these two components, Proxy and Redis KVS.

We used Keycloak v11.0.2 to implement the Authentication Server. For this, we run a container with Keycloak's docker image, where we specify access control policies to our system in an in-memory database. This server authenticates Clients upon arrival, and grants access tokens if they have permission to reach the system, which the Clients will have to use to interact with the Proxy. In order for the Proxy to validate the tokens, we established a secured TLS connection between these two components.

The attestation mechanism works as a way to make sure our system components that run on private memory regions of SGX, run indeed inside them. We use SCONE's remote attestation mechanism, which consists of a remote entity CAS, provided by SCONE itself, that manages the secrets of the applications supposed to run in enclaves. For an enclave to reach those secrets and thus prove their identity in order to run the desired application, it needs a signature from the quoting enclave LAS, running locally in our system in a secured container. With that signature, the enclave proves to CAS its identity as an attested enclave, and CAS matches the enclave's hash value with the one specified to run the application. If the hashes match, CAS hands over the application secrets to the enclave, allowing it to run the application.

Finally, the KVS was implemented with Redis v6.0-rc1 images incorporated with SCONE's image, in order to configure KVS instances as we intend to, but also to establish secured TLS connections with other components over the network, by using our own set of TLS keys and certificates. We implemented various Redis configurations to run in our TREDIS solution, in order to evaluate their behavior: Standalone, Master-Slave and Cluster. The Master-Slave was configurated to run with three replicas, one Master and two Slaves, while the Cluster was set to run with six replicas, three Masters and three Slaves. Slaves are read-only nodes.

In the next chapter, we start our experimental evaluation of the system, going into detail about what tests we performed while making a practical analysis of the values obtained.

Chapter 5

Experimental Observations and Validations

In this chapter, we describe the work we conducted for the validation and evaluation of our proposed in-memory TREDIS solution, detailed in 4. We will mostly evaluate the impact of the biggest and most important components, such as the KVS layer and the Proxy, while also analyzing the impact of the attestation component. However, as for the component responsible for the authentication of the clients, we consider it to be out of the scope of our study.

Here, we start by presenting the metrics we intend to evaluate, along with the test benches we defined to test our prototype. We then go deeper about each test bench, describing the results obtained during the experimental evaluation of each one of those scenarios, leading to a discussion later in this chapter aiming to compare those results and, finally, ending with a summary of the chapter.

5.1 Criteria for Experimental Observations

Our evaluation process is simple: track our TREDIS solution's behavior through all the different configurations, incrementally adding more layers to the system. Thus, we intend to evaluate our TREDIS solution on each possible configuration, starting with a basic model and slowly add more components to the system, while also making experimental observations about the impact they have.

During the evaluation process, we focus on measuring: 1) the performance impact each component has in the system while running with or without SGX, through a latency and throughput analysis and 2) resource allocation during runtime, including memory and CPU usage.

There are other particular measurements that we found useful to introduce, that we detail later while describing the test benches individually. Adding to that, we also analyze the system's behavior under different client-workloads, by scaling up the number of requests and varying the typology of requests made to the system.

5.2 Deployment of Testbench Environments

As said before, we intend to evaluate our in-memory TREDIS system behavior by incrementally adding components to it, which will add security to the whole system, and see what impact they have on it. In order to do that, we define a list of Testbench (TB) scenarios to help us evaluate the system gradually.

First, our idea is to benchmark the KVS Redis layer running normally inside our cloud server, in order to understand the behavior of an in-memory Redis KVS running normally, and also to use it as a reference point to our tests. For that, we define Testbench 1 (TB1) as a default version of Redis, with TLS support and all its configurable security features, running in our OVH cloud server.

As the next step, we analyze the impact that SGX has in an SGX-enabled Redis. For that we define TB2 as our cloud server running the Redis KVS component mentioned in TB1, but this time inside an SGX-enabled SCONE container.

In TB3, we test a scenario with a default Redis instance, running along with a Proxy instance, in order to simulate our system's environment. This scenario will help us understand the exact impact SGX has in the KVS layer running in our system.

TB4 assumes the addition of SGX support to the Redis instance of TB3, in order to benchmark its impact on the system. Thus, we define TB4 as a Redis component running inside an SGX-enabled SCONE container, along with a Single Proxy instance.

In TB5, we deploy the Proxy component on top of SGX, resulting in benchmarking a system composed by a Redis KVS running inside an SGX-enabled SCONE container, along with a Single Proxy instance also running inside SGX.

For TB6, we added the attestation property to the components, used to assure that they run in private memory regions on top of SGX and that they are only executed by the right enclave. Here we measure the impact it has to attest each component upon start.

With all the system model defined in Chapter 3 in place, we define two more test benches TB7 and TB8 to evaluate the whole system's behavior against different client request overloads (increased number of requests plus different size payloads) and against different typologies of requests (i.e., 10% Writes : 90% Reads), respectively.

This are the testbench scenarios we follow in order to evaluate our solution while running the Redis layer in all the three configurations we detailed in previous chapters: Standalone Redis, Master-Slave Redis, and Clustered Redis.

5.3 Observations with Cloud-based Standalone REDIS

In this section, we analyze our solution running the in-memory Redis component configured as a single instance KVS. The experimental evaluation will be done following the test benches defined above, to access the impact of SGX in our system, analyzing both the performance and resource allocation impact that each secured component has in the system.

As we detailed in 4.1, we run our solution on a cloud system with SGX-enabled hardware, while our client benchmark applications run on a local machine with commodity hardware, in order to simulate a real-world use-case where the network has a major impact on the performance of a system.

5.3.1 Latency Impact of SGX-Enabled REDIS

To study and compare the latency levels of TREDIS with and without SGX, we evaluate our solution by complying with the testbenches TB1, TB2, TB3, TB4, and TB5 (see 5.2 for details) definitions, with network conditions of ≈116Mb/s Download and ≈114Mb/s Upload speed. It is important to mention that, for the first two testbenches in which our client application points directly to the Redis KVS layer, we used redis-benchmark to make the requests and benchmark the solution. However, with the addition of the Proxy layer in TB3, we had to switch to an HTTP-enabled client, Jmeter.

Thus, we start to benchmark our solution according to TB1, which results in an average 33,97 millisecond response time that we can use as a base value. By comparing it to the value of TB2, we note that the addition of SGX security properties to the KVS component induced a latency penalty of ≈5%, as we can see in Table 5.1. This value is expected since, as we studied in previous chapters, SGX induces in performance overheads caused by the time it takes to deal with heavy cryptographic functions and mechanisms in order to keep the integrity and confidentiality of the data.

Table 5.1: Latency impact of SGX in Standalone Redis

Configuration	Latency
Redis	33,97ms
SGX-enabled Redis	35,63ms
Redis + Proxy	35,40ms
SGX-enabled Redis + Proxy	37,3ms
SGX-enabled Redis + SGX-enabled Proxy	44ms

As we detailed before in previous chapters, our TREDIS solution includes a Proxy component that also adds overhead to the system, and it is expected to add even more when running inside SGX. For that analysis, we run TB3 and TB4, in order to evaluate Proxy's impact on the overall system, and TB5 to observe how SGX impacts its performance. With the addition of this component, we observe an additional 4% overhead compared to TB1 and 4,5% to TB2. This value is something we can deal with due to the utility we give to this component in particular. However, the ≈18% latency increase that SGX imposes on the Proxy component can lead to a subjective conclusion. This overhead is covered by the SGX impact, but also by it being a more I/O-intensive component, thus resulting in a higher probability of making system calls.

5.3.2 Generic Throughput Observation

In order to measure the impact that enabling SGX has on the throughput of our solution, we follow the same test benches as the ones used in the latency test - TB1, TB2, TB3, TB4, and TB5. Our following evaluation is based on the average measurements of a set of identical tests, each one consisting of one client making 10 000 requests with 10 Bytes worth of data over the network.

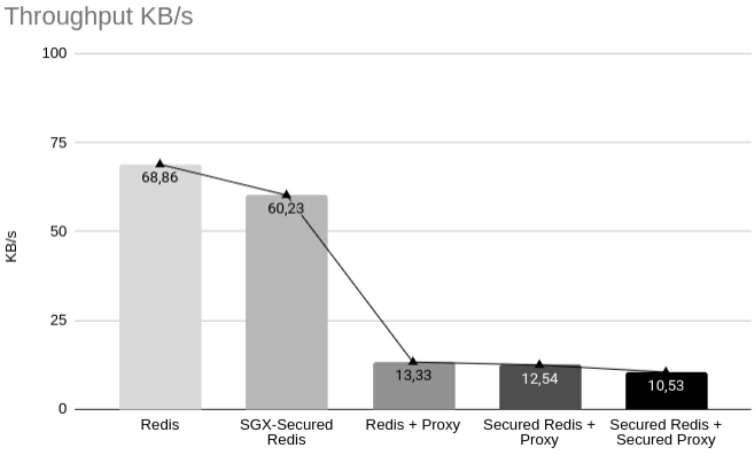


Figure 5.1: Throughput impact of SGX in Standalone Redis

Here, the addition of SGX to the Redis component induces a 12,5% overhead, that we observe in Figure 5.1 in the tests made via redis-benchmark, in which the client application connects directly with the KVS layer itself, via TCP.

However, in TB3, TB4 and TB5 results, we can see that adding the Proxy component to the system drops significantly the solution's throughput levels. This is expected since the requests start to be done via HTTP, which induces losses of ≈80% compared to TCP. With the requests being now done to the Proxy, we observe only a 6% throughput penalty on running the KVS on top of SGX, along with a 16% drop when enabling the Proxy layer to also execute on top of SGX. The reasons are related to the performance overheads know to be induced by SGX itself. Note that the Proxy component is an application written in Java, making the inclusion of JVM and other java libraries fundamental, thus the image running in the SCONE container on top of SGX is heavier, resulting in having to leave some Java code outside the enclave.

5.3.3 Evaluation of Specific Benchmarks and Operations

As a way to evaluate our solution's behavior facing more specific benchmarks, we test it by following the TB7 and TB8.

Here we present how different operation ratios influence the system throughput. By looking at Figure 5.2, we can see that the results on how TREDIS handles these different combinations of requests with small payloads do not vary much. This absence of difference might be related to Redis high-performance levels, especially with small payloads, where it takes almost no time to compute.

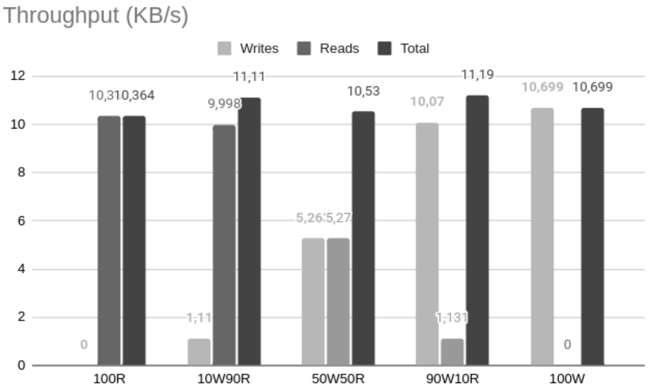


Figure 5.2: Throughput with different sets of operations

However, with the increase of payload size, we start to see significant drops in the number of operations done, along with higher response times, as we can see in the Table 5.2. This is due to the Redis-server not handling big payloads very well since it is mostly single-threaded, thus needing more time to handle bigger requests one at a time.

Table 5.2: Throughput values with different size payloads

Payload Size	Operations p/sec	Latency
10B	≈22	≈44ms
10KB	≈20	≈46ms
100KB	≈12	≈57ms

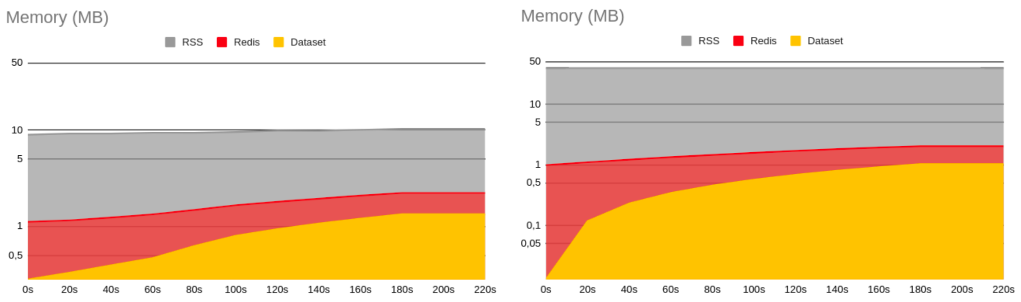
Also, to comply with TB7, we scale the number of requests made to the server, in order to see differences in the behavior of the system, along with their size so it is easier to populate the enclave. By doing that, we encounter an unexpected problem: when reaching the maximum RSS size defined for Redis upon start, the container crashes. We later found out that this problem is targeted in SCONE's website[1], where they explain it to happen due to the SGX version (SGX v1) we are using not supporting dynamic allocation of memory, thus *"enclaves must allocate all memory at startup since enclaves are fixed"*. This causes the memory usage of Redis to be higher than it needs to be since to prevent it from crashing we need to allocate memory upon start that we do not know we will need, leading to larger startup times. However, SCONE also affirms that the next version of SGX (SGX v2) will support dynamic allocation, which tackles this problem.

[1] https://sconedocs.github.io/faq/

5.3.4 Standalone REDIS System Resources

Here we evaluate our solution's memory consumption and CPU usage during runtime. For the purpose of this test, our client application made requests to the Proxy during 180 seconds with 1 KBytes worth of data.

In the graphs we present in Figure 5.3, we observe some changes in the system when running it on top of SGX and outside of it. First of all, we notice that the dataset size increases faster if running without SGX support since the system is faster and its throughput levels are higher, resulting in more operations made over the dataset in the same time period. We can also see that the Resident Set Size (RSS) in one case is dynamic, while in the other is static. This calls back to what we stated in the section before, where we mentioned that SGX v1 does not support dynamic allocation of memory, thus only relying on the memory size specified upon creation which remains static throughout execution. In Figure 5.3b we see just that, a static RSS value during the entire evaluation. Note that this RSS value defines the memory available for a Redis instance to scale during its execution. Thus, running Redis inside SGX might induce into memory problems if using SGX v1, since either the system reaches the point where it is left with no memory available to run and stops, or it is created with huge amounts of memory, which might not be necessary, inducing it to big startup times since it needs to allocate more memory upfront.

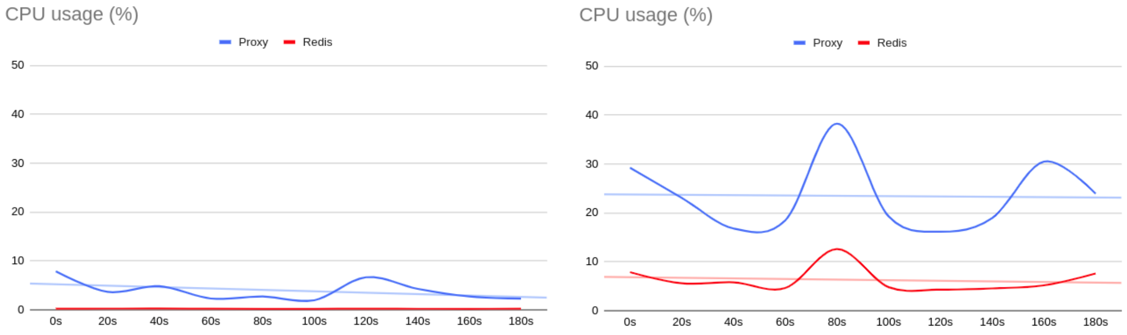


a) no SGX

b) with SGX

Figure 5.3: Standalone Redis memory consumption

As for the CPU usage, we show in Figure 5.4 the impact that SGX has. On the left, we see the CPU resources that go into the execution of the solution without this extra layer of security, remaining near 0% for Redis while the Proxy shows values of around 5-8%, due to all I/O operations it handles. On the right, we see more expressive results, where the addition of SGX results in higher CPU usage, especially by the Proxy component, due to its size causing it not to fit entirely inside the EPC. However, since this test also induces a high density of requests, we do not consider this behavior to be unexpected.



a) no SGX

b) with SGX

Figure 5.4: Standalone Redis CPU usage

5.4 Observations with Cloud-based Master-Slave REDIS

Here we present our observations while testing our solution with the Redis layer in a Master-Slave configuration. We run the KVS with three replicas, one being a master node, and the other two being read-only slave nodes.

Hereupon, this experimental evaluation follows the same test benches that we used in Section 5.3, to analyze the impact that enabling SGX support to our components has in the whole solution.

We run the test scenarios in the same environment previously used, in a cloud system with SGX support, while making the requests in a local machine over the network, with the network speed being of ≈117Mb/s Download and ≈119Mb/s Upload.

5.4.1 Latency Impact of SGX-Enabled Master-Slave REDIS

In order to evaluate the latency impact of SGX in our solution, as we just mentioned, we intend to execute the same five scenarios, incrementally adding components to the system, while enabling SGX support to each one of them along the way.

We start the test by making requests to the server with redis-benchmark as the client application when communicating directly to the Redis-server and then switching to Jmeter to communicate through HTTP with the Proxy component.

Table 5.3: Latency impact of SGX in M-S Redis

Configuration	Latency
Redis	32,48ms
SGX-enabled Redis	32,97ms
Redis + Proxy	33,05ms
SGX-enabled Redis + Proxy	34,45ms
SGX-enabled Redis + SGX-enabled Proxy	41,3ms

In Table 5.3 we can observe a similar behavior compared to the values we saw for our solution running a single instance Redis, as the latency time increases with the components and extra security we add to the system. Here we can see a 1,5% drop with the inclusion of SGX support to the KVS layer. Connecting the Proxy, however, costs the system running unsecured Redis ≈ 2%, while for SGX-enabled Redis around 4,3%. Adding SGX support to the Proxy induces a 20% latency drop, again originated from the fact that it is an I/O-intensive component, which leads to more system calls being done by it while inside the enclave, which by SGX' definition is a major influencer in performance dropping. Adding to that, it is a Java application, resulting in a bigger image which can lead to some of the code having to be placed outside the enclave.

5.4.2 Generic Throughput Comparative Observations

For our throughput evaluation, we use the same configuration as we did for Standalone Redis, consisting of one client thread doing all the 10 000 requests with a payload of 10 Bytes while registering the average results of multiple tests.

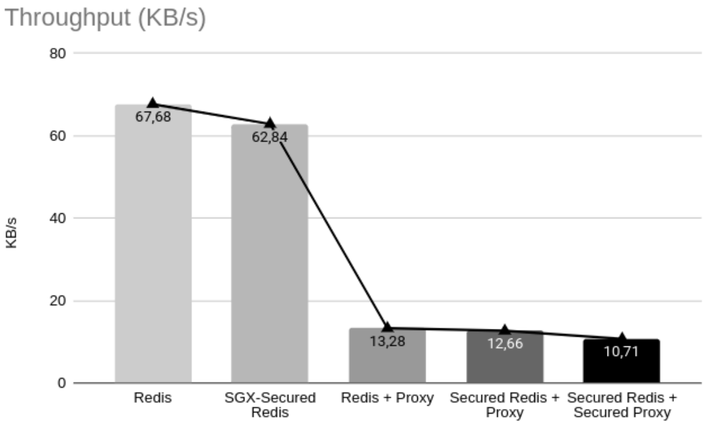


Figure 5.5: Throughput impact of SGX in M-S Redis

In Figure 5.5 we can observe a 7,15% drop by securing the KVS layer with SGX, which is a slightly smaller loss than the one we observed in the Standalone tests. However, we observe once again a considerable penalty with the addition of the Proxy layer, which along with the switch from TCP requests to HTTP requests causes a huge throughput drop of around 80%, even without any SGX inclusion. The inclusion of this extra layer of security has shown the same tendency we observed in the Standalone results, causing throughput losses of 5% and 15%, for both the protection of the KVS layer and the Proxy, respectively.

5.4.3 Throughput with Specific Benchmarks and Operations

Here we test the behavior of our Master-Slave configured solution against different operation ratios and different payload sizes. In Figure 5.6 we can see pretty much the same results we had with Redis running in Standalone mode, either while running 100% writes, or 100% reads, or any other combinations we present here.

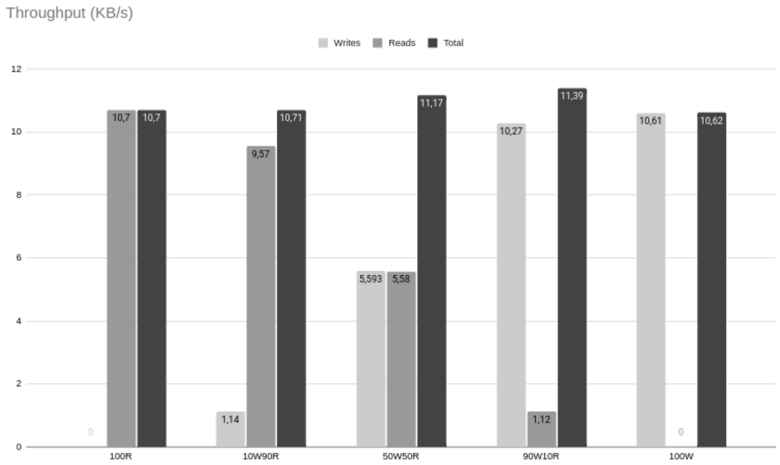


Figure 5.6: Throughput with different combinations of operations

In the evaluation regarding the increment of the payload of the requests, shown in Table 5.4, the trend shows to be the same as with Redis Standalone, where bigger payloads lead to less performance. However, with a Master-Slave configuration, we got slightly higher results while working with smaller size payloads.

Table 5.4: Throughput differences with different size payloads

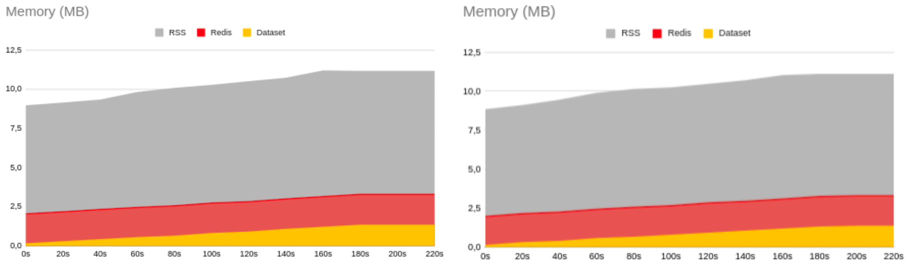
Payload Size	Operations p/sec	Latency
10B	≈23	≈42
10KB	≈21	≈43
100KB	≈12	≈56

We can see that the results are pretty similar in all the aspects, in the tests of both configurations (Standalone and Master-Slave). This happens because upon receiving a request from the client, the Redis-server responds immediately after it completes the request. All the replication added by the Master-Slave configuration only takes place in the background. Therefore, the throughput results from a Master-Slave Redis should be, in theory, the same as a Standalone Redis.

5.4.4 Master-Slave REDIS System Resources

To perform a system resources evaluation during our solution's runtime, we set our client to make requests with 1 KBytes to the system for a total of 180 seconds.

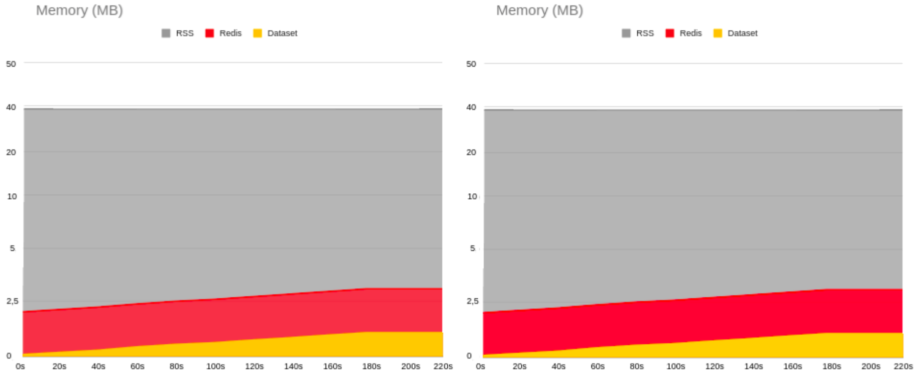
The graphs we present in Figure 5.7 and 5.8 show exactly how the server manages the memory of our Redis layer, either with or without the SGX security properties. In the first one, we can observe the behavior of both the master node and the slave nodes, in which we see a slightly faster memory increase on the master since it is the only replica responsible to perform writes in the KVS in this configuration. Thus the master receives the data first, whereas the slaves have to wait for the replication to happen, in order to update their own dataset. They behave similarly throughout the execution of the test, achieving eventual consistency by the end, thus ending the test with the same exact dataset.



a) master node

b) slave node

Figure 5.7: M-S Redis memory consumption



a) master node

b) slave node

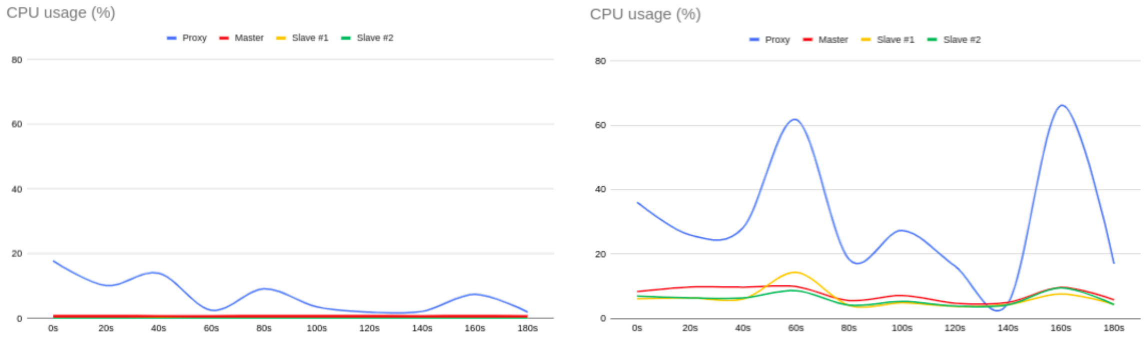
Figure 5.8: SGX-enabled M-S Redis memory consumption

Note that without SGX support, the memory allocated to run the Redis layer (shown as grey in the graphs - RSS) follows dynamically both the rise of the Dataset size and the Redis instance size. This means that, without SGX, this layer can allocate memory on-demand, optimizing its memory consumption during runtime. However, as we have mentioned before, the same does not happen for SGX-enabled components, since SGX version v1 does not include this dynamic allocation of memory, thus failing to scale the application's memory size, while running inside the enclave. This is why we see a static RSS value in both graphs presented in Figure 5.8. Besides that, the values shown are similar to the ones shown in the Standalone evaluation, as the Dataset memory size increases gradually alongside the Redis memory size.

The memory consumption values we show in the figure covering the SGX-enabled solution are lower at the end of the 180 second test, which is expectable since the throughput numbers are inferior, therefore less writes are made in that same period of time.

For the CPU, we can see what resources go into the execution of the tests in Figure 5.9, where we notice once again almost 0% CPU usage when dealing with a Redis KVS without SGX, whereas if we enable SGX support, this value rises to ≈8%, in which the master node shows to be the one node needing more resources, due to being in charge of replicating the data to all the slave nodes in the system.

As for the Proxy impact in the CPU, we can see a similar behavior to what we observed with Standalone Redis, since not much has changed for this component. We see the same ≈8 to 10% when running without SGX security properties, while more expressive results when executed on top of SGX.



a) no SGX

b) with SGX

Figure 5.9: M-S Redis CPU usage

5.5 Observations with Cloud-based Clustered REDIS

In this section, we share our evaluation made for the system while running the KVS layer in Cluster mode. We use the default configuration that Redis has for the cluster, consisting of three master nodes with one slave each, resulting in six nodes total. The writes are made to the master nodes and later replicated to their slave replica, which eventually ends up having the same dataset as their master instance.

Here we follow the same scenarios we used in the previous tests, in order to conduct an evaluation about the SGX impact over the solution however this time running in cluster.

Our test conditions are the same, where we run the client application locally while communicating with a cloud server with the possibility to enable SGX support. However, our network conditions show to be slightly worse than before, consisting of ≈108Mb/s Download and ≈113Mb/s Upload speed.

5.5.1 Latency and impact of SGX-enabled REDIS Cluster

To study the latency of our system, we test our solution running according to the same defined testbenches as we did in both Section 5.3 and 5.4, where we start with only the KVS layer, following it by deploying the KVS on top of SGX, then adding the proxy, and ending with both layers working together while running on top of SGX.

The client applications we use to test each scenario are the same specified in the previous tests: redis-benchmark for the first two scenarios shown in the table below, and Jmeter for the remaining ones that make use of the Proxy layer.

Table 5.5: Latency impact of SGX in Cluster Redis

Configuration	Latency
Redis	33,26ms
SGX-enabled Redis	35,29ms
Redis + Proxy	35,98ms
SGX-enabled Redis + Proxy	36,67ms
SGX-enabled Redis + SGX-enabled Proxy	43,3ms

In Table 5.5 we observe the same tendency shown in tables 5.1 and 5.3 from the previous tests, where more security and complexity means more response time, affected by the increase of processing power needed. Here we see a 6% delay caused by enabling the Redis to SGX support without the Proxy, followed by a ≈1% upon the addition of the Proxy layer. Finally, running the Proxy on top of SGX results in a 17% penalty.

5.5.2 Generic Throughput Comparative Observations

We continue our analysis with the same exact testbenches, although this time in order to evaluate our solutions throughput. In Figure 5.10 we observe the results we registered doing 10 000 requests with a size of 10 bytes each. Here we see throughput values dropping 6% with the addition of SGX to the KVS cluster, followed by ≈80% drop induced by the Proxy supported environment. There we see the impact of SGX to be of 2% for the KVS and 15% for the Proxy.

As for different sets of operations performed over the cluster, the solution shows to also handle them evenly, as it is noted in Table 5.6.

Table 5.6: Throughput with different sets of operations

Op. Ratio (R:W)	Throughput
1R : 0W	10,8KB/s
10R : 1W	10,12KB/s
1R : 1W	10,93KB/s
1R : 10W	10,52KB/s
0R : 1W	10,8KB/s

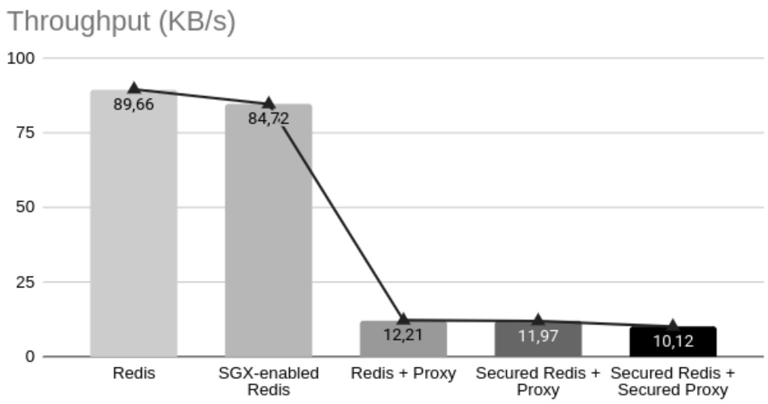


Figure 5.10: Throughput impact of SGX in Clustered Redis

By comparing these values to the ones presented for the previous two configurations, Standalone and Master-Slave, we can conclude that using our solution in cluster mode does not induce in significant overheads. Furthermore, in cluster, the solution is actually able to offer similar results while dealing with bigger size requests, as shown in Table 5.7. Here we expect this behavior since the operations are handled exactly like a single Redis instance, whereupon the KVS server receiving the request, it responds right after it finishes executing it. The extra workload induced by the replication of the data through the cluster does not impact the time it takes for the KVS to respond back to the client.

Table 5.7: Throughput differences with different size payloads

Payload Size	Operations p/sec	Latency
10B	≈23	≈43ms
10KB	≈22	≈44ms
100KB	≈13	≈54ms

And although it does not assure a better performance, it still offers better availability, scalability, and replication of data, along with other properties, then the previously tested configurations. Thus, we prove it to be a valid approach to adopt when a Standalone or Master-Slave options are saturated, and we need to scale our solution horizontally.

5.5.3 Clustered REDIS System Resources

In order to evaluate our Clustered solution's use of resources, we will register the cloud system's behavior while handling 1 KBytes size requests for a duration of 180 seconds.

Upon starting the cluster, the Redis instances rearrange themselves to a configuration based on the one shown in Figure 5.11, with multiple sets combining a master node and one or more slaves. Then, each master node becomes responsible for a set interval of values, that will be used to forward requests upon arrival, based on the request's hash value. Thus, the cluster balances the load between master nodes, which will eventually propagate their data to their respective set of slaves, replicating it.

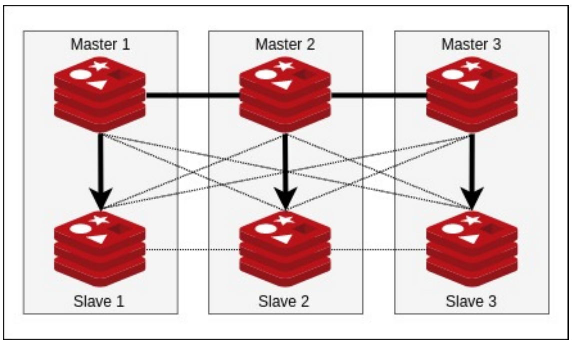
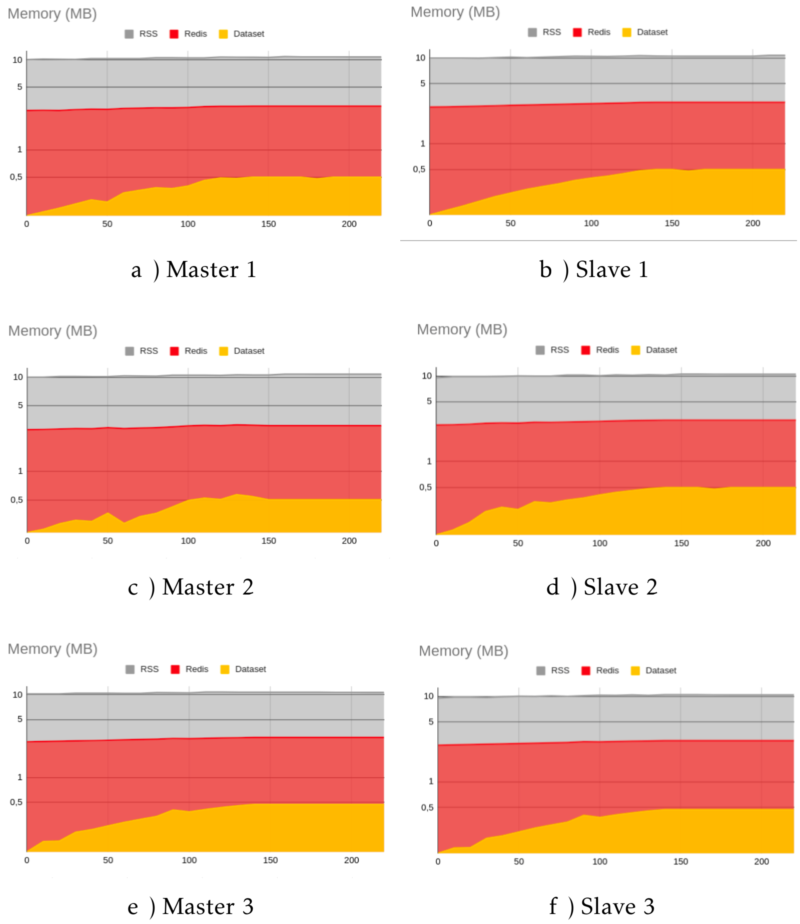


Figure 5.11: Cluster configuration

With the cluster assembled according to the model discussed just now, we proceed to register and evaluate the system memory and CPU usage, for both SGX-enabled and not.



a) Master 1

b) Slave 1

c) Master 2

d) Slave 2

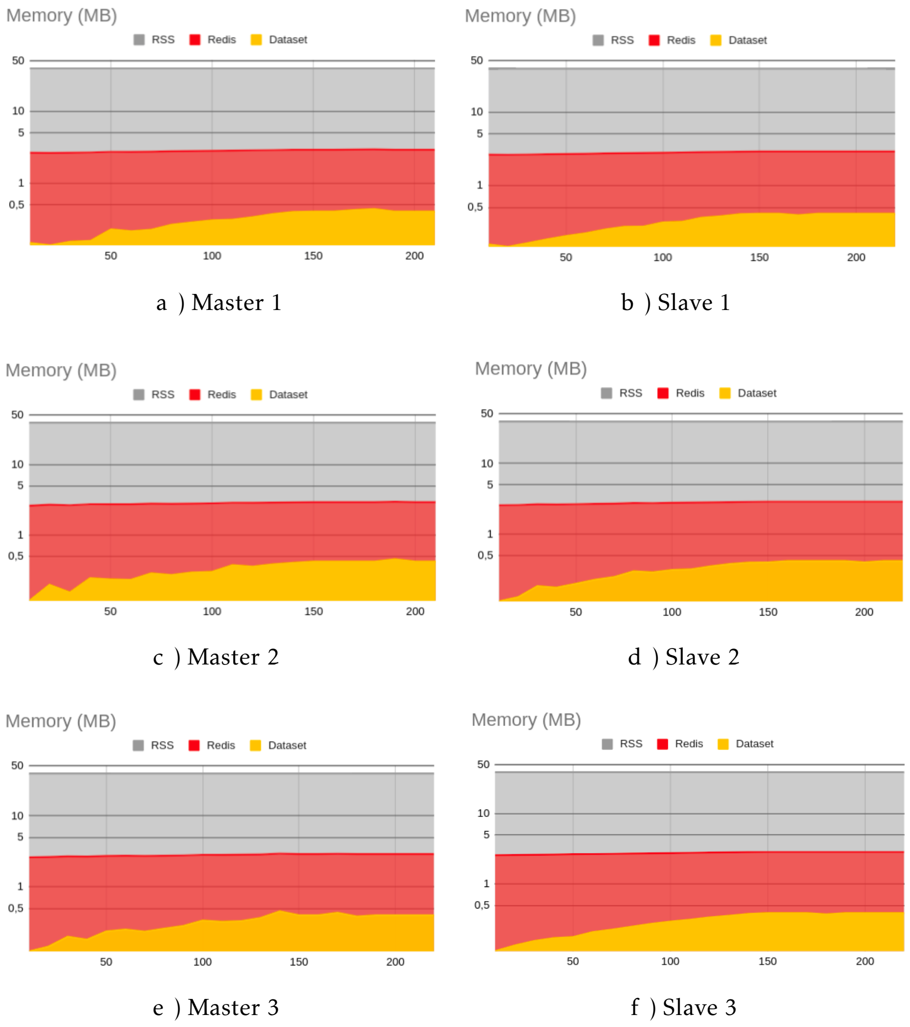
e) Master 3

f) Slave 3

Figure 5.12: Redis Cluster memory consumption

Starting with the analysis of the solution without SGX support, we observe throughput values in the order of ≈27 operations per second, which results in a total of around 1000

KBytes written into the system. By following the cluster design, we see on the left-side graphs present in Figure 5.12 that the data is equally split between the master nodes, whereas in the right-side graphs, we can see that the slaves end up being replicated by their masters, thus achieving consistency. However, we can note that the sum of the dataset sizes surpasses the total number of data written during the test. This happens because each node in the cluster has an internal table where they map each key existent in the cluster, in order to indicate where any key is in the cluster, in case it is needed. Although they do not include any values for those keys, this table can add some overhead to the system, especially with a huge number of keys.



a) Master 1

b) Slave 1

c) Master 2

d) Slave 2

e) Master 3

f) Slave 3

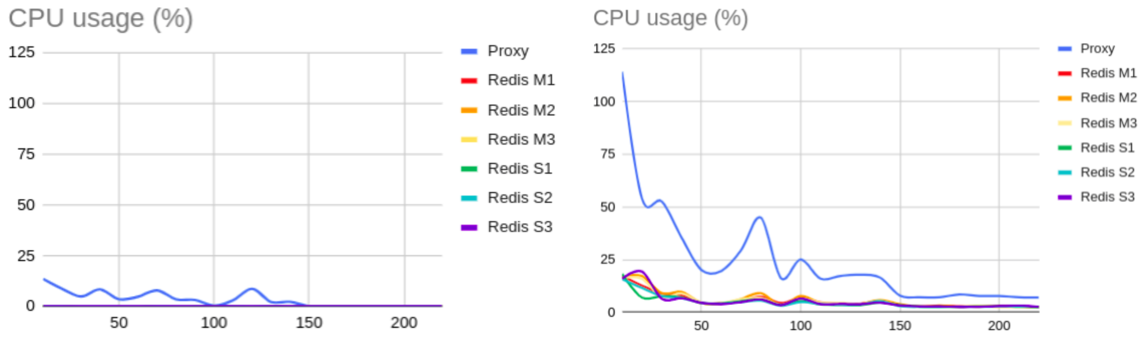
Figure 5.13: SGX-enabled Redis Cluster memory consumption

When running our solution in an SGX-enabled environment, we achieve results of around 22 operations per second, thus resulting in a slightly smaller Dataset size of ≈800 KBytes. In Figure 5.13 we see that exact same behavior, with the partitioning of the data also taking place between the master nodes, while propagating those partitions to their slave replica. However, and as we have seen in 5.3.4 and 5.4.4, the RSS value remains static throughout the evaluation.

Comparing both scenarios where we run the cluster outside SGX and inside it, we see a difference of ≈75% more memory allocated by the system for two identical Dataset sizes. This limitation results in having to allocate more memory than what is necessary, in order to prevent the KVS instances to run out of memory, and thus stop working.

Lastly, Figure 5.14 shows the CPU work that goes into the execution of our solution, either inside or outside a trusted execution environment.

Here we see that nearly 0% of the system's CPU resources go into the execution of the Redis instances while running without SGX support. However, this value rises to nearly 10% in the second graph 5.14b. This increased value is due to the extra overhead SGX gives to the system, along with the data replication among replicas in the cluster.



a) no SGX

b) with SGX

Figure 5.14: Cluster Redis CPU usage

As for the Proxy, its behavior is identical to the previous tests, where we see it going from ≈5-10% to more expressive values when running on top of Intel-SGX hardware.

5.6 Attestation Impact

In order to measure the attestation impact in our components, we comply with the scenario described for TB6 defined earlier in 5.2, where we add the functionality for the SGX-enabled components to attest themselves and prove that they are indeed running on private memory regions on top of Intel-SGX.

To assure attestation to our components, we use the mechanism provided by SCONE, consisting in defining secrets to applications on a remote component CAS, where a validation is performed upon starting with the help of a local attestation component LAS. Note that we detailed this process in 4.2.1. We define secrets to be pairs of TLS keys and certificates that applications need upon start to establish TLS connections with other components. Therefore, if an enclave fails to attest itself, it will not have these secrets, thus failing to establish TLS connections, and not starting. These secrets are defined in a YAML file like the one we show in Appendix A, and sent to a public instance of CAS.

Our solution's attestation mechanism only allows the SCONE components to be attested upon start and not on-demand. Thus we compare the time it takes a component running in a SCONE container to boot with attestation, comparing it to the time it takes to boot without it. In Table 5.8 we observe just that, where we conclude that adding this attestation mechanism to a Redis instance container induces in ≈ 1,23 extra seconds, while for the Proxy ≈ 1,05 seconds.

Table 5.8: Attestation impact in boot time

	No Attestation	**Attestation**
Redis	0,14s	1,37s
Proxy	62,07s	63,12

5.7 Main Findings From the Experimental Observations

During the evaluation in this chapter, we were able to analyze our solution in three different configurations, Standalone, Master-Slave, and Cluster, by following multiple testbenches that were previously described, in order to benchmark the impact that securing our solution's components with SGX has overall.

After going through all the results for each configuration, we see that although the system has lost some performance with the addition of SGX, its impact can be looked at as quite modest, varying from low to medium on all components involved in the evaluation, regardless of the mode the KVS is set to execute.

However, after comparing the different evaluations made over the three Redis configurations, we see no considerable difference in the results. Therefore, we can note that all configurations are able to work in a secured environment supported by SGX. This shows to be great since the configuration used for Redis can be picked based on each application scenario, and not based on performance differences. Thus, if the intention is to use a simple Redis instance with SGX support, use the Standalone mode. If we want redundancy and fault-tolerance, use the Master-Slave configuration. And if we want to scale the application to another level, with scalability, redundancy, high availability, among other properties, use the Cluster.

Keep in mind that the Cluster can also be optimized if running each instance in independent machines. By doing so, each instance works as a single-threaded instance, thus achieving an optimized performance.

As for the Proxy, since our intention is to recreate a real-world scenario where the TREDIS can be used as a normal application in the cloud, it is essential that our solution is enabled to receive requests via HTTP. Thus, the ≈80% impact it has on the performance caused by adding the Proxy alone has to be taken as a necessity. However, there is room here for improvement, since our Proxy was designed in Java, which led to more code being needed.

5.8 Summary

In this chapter, we evaluated the solution we proposed for this dissertation, an in-memory TREDIS solution, introduced in Chapter 3 and detailed in Chapter 4. Here we conducted a study about the impact that SGX has on the components that make up our solution, in order to analyze if the tradeoff between performance and security is worth it or not.

To access its impact, we benchmarked the solution according to defined testbenches, each one representing a different scenario, that we used to perform a structured incremental analysis where we added system components one by one, while also adding security to them one by one, starting with the Redis layer unprotected and gradually building up to the point we had the whole solution running on top of SGX. We also defined some extra testbenches in order to evaluate other more specific measurements. All the testbenches were repeated for the three Redis configurations we mentioned, Standalone, Master-Slave and Cluster, in order to see their behavior running in an SGX-enabled environment.

During the evaluation, we concluded that while the overall tendency of using Intel-SGX results in overheads, the values go from having low to medium impact overall, never going past 20%. For the KVS, we see similar results through all three configurations evaluated. Thus, we can safely assume that our KVS is capable of making use of SGX properties in every configuration. This proves to be good since the configuration can be picked only based on the properties we want for our KVS layer.

The biggest impact to our solution comes from the Proxy component, whether by enabling it to SGX support or by its inclusion in the solution itself. However, we found this component to be essential for our solution to work as we intend, not only because of the purposes we assigned to it, which are described in 3.3.2, but also because it makes our solution more practical in real-world scenarios, working as an API that can be exposed to clients that they can reach through the web, via HTTP.

Chapter 6

Conclusion

6.1 Main Conclusions and Remarks

As addressed in Chapter 1, our dissertations objective was to study how unmodified applications can be deployed to a Cloud server, to be executed on top of trusted hardware, while still offering decent levels of performance compared to applications running without this extra layer of security.

To address this objective, we proposed our solution: TREDIS, a system running on top of Intel-SGX trusted hardware, assembled around an in-memory Redis KVS layer, with the idea of assuring integrity and confidentiality to the data executing and being stored in the system.

However, running code on top of SGX, or any other TEE, generally leads the system to huge performance overheads. This is due to the private regions where the code runs being particularly small, resulting in a lot of encryption functions and security checks to take place, in order to keep the integrity and security of the data. To mitigate this problem we adopted in our solution a secured container mechanism - SCONE - capable of running Docker containers on top of Intel-SGX. With it, we were able to run and secure individual containers for each of our components, with fewer performance losses.

Thus, we deployed Redis instances inside SCONE containers, to be used as the central layer of our solution. To it, we added a Proxy layer, working as an entry-point to the whole system, followed by an Attestation component, responsible for attesting each component that run on SGX enclaves upon startup, only enabling them to boot if they prove their identity to be valid, and finally an Authentication Server, which works together with the Proxy in order to grant access to clients that try to reach the system.

We implemented a prototype designed to run the Redis KVS in three distinct configurations, Standalone, Master-Slave, and Cluster, in order to understand their own individual behavior when running on top of SGX trusted hardware. The Proxy was designed in Java, working as a Spring API deployed to a SCONE container that enabled the clients to make requests to the system via HTTP, while imposing access control policies with the help of the Authentication server, consisting of a Keycloak authentication server instance. Lastly, for the attestation process, we used a mechanism provided by SCONE itself, that can be set individually over each container.

We used our prototype to conduct an experimental evaluation for validation purposes. In the evaluation, we tested how SGX support impacts our solution, through performance testing and system resource analysis. We tested the solution in all the three KVS configurations mentioned before, in order to benchmark their behavior on top of SGX.

In conclusion, we addressed all the objectives and goals proposed for our dissertation. From our observations, we conclude that, although we registered some overhead induced by Intel-SGX, the system performs with a good balance between privacy concerns, trustability assumptions and operation performance, showing that is possible to have a solution using an in-memory KVS that can take advantage of these trusted execution environments without becoming inefficient to deal with realistic scenarios.

We have shown that the impact on the possible loss of performance for the KVS layer is mainly in the range between 5 to 10% of overhead, comparing with a similar solution without these privacy and trustability considerations, for all configurations mentioned. However, for the Proxy layer the values increase to around 17%, which still results in a decent tradeoff, although there can be room for improvement in the way we implemented this component, in order to take advantage of SGX security properties with more effectiveness.

6.2 Open Issues and Future Work

Although the solution we developed showed interesting results, there is still a few points where the solution can improve, either by optimizing some of the components we have used or by extending it in some way.

Starting with our KVS layer, we think that optimizing our TEE-enabled solution with a version of REDIS designed specially to work in this kind of environment, thus avoiding fitting the entire REDIS solution inside the enclave, would lead to better results than an unmodified version of REDIS. This work direction could minimize the impact of SGX performance issuers, such as SCONE and other TEE frameworks. However, it forces to an accurate reengineering process of the REDIS implementation, in order to be more suitable for this purpose. Another point of work is to partition the solution into multiple machines to achieve better performance levels by separating the workloads physically. This also includes running the cluster instances in different physical machines, in order to achieve optimal cluster performance. Persistency of data can also be added as an extra feature in the future, where the data would transit from being encrypted in-memory to also being stored encrypted in disk, thus keeping the privacy of the data intact throughout all the application.

As for the Proxy, evaluate if using different technologies to implement its behavior improves the overall performance of this component when executed on top of SGX. There is also room for optimization when the SGX version used starts to support dynamic allocation of memory during runtime, which can help to avoid big startup times, unnecessary memory use and failures.

Another future work direction we thought to be interesting to evaluate is the impact of different existing container-based solutions that leverage SGX security properties (i.e., Graphene, Graphene GSC, etc.) in a system, comparing it to the impact of SCONE in ours. Adding to that, evaluating how different database technologies, either already implemented to work without trusted hardware or a native KVS technology designed from scratch to be deployed and used on top of SGX would behave in a similar environment.

Bibliography

[1] *A persistent key-value store for fast storage environments*. Accessed: 18-01-2020. URL: https://rocksdb.org/.

[2] *AMD MEMORY ENCRYPTION*. Accessed: 03-07-2019. URL: http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf.

[3] A. Arasu, S. Blanas, K. Eguro, R. Kaushik, D. Kossmann, R. Ramamurthy, and R. Venkatesan. "Orthogonal Security With Cipherbase." In: *6th Biennial Conference on Innovative Data Systems Research (CIDR'13)*.

[4] M. Armbrust, A. Ghodsi, M. Zaharia, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. Bradley, X. Meng, T. Kaftan, and M. Franklin. "Spark SQL." In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data - SIGMOD 15*.

[5] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O Keeffe, M. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer. "SCONE: Secure Linux Containers with Intel SGX." In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*.

[6] J. Attridge. *An Overview of Hardware Security Modules*. 2002.

[7] M. Bailleu, J. Thalheim, P. Bhatotia, C. Fetzer, M. Honda, and K. Vaswani. "SPEICHER: Securing LSM-based Key-Value Stores using Shielded Execution." In: *17th USENIX Conference on File and Storage Technologies (FAST 19)*. 2019.

[8] A. Baumann, M. Peinado, and G. Hunt. "Shielding applications from an untrusted cloud with Haven." In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*.

[9] S. Brenner, C. Wulf, and R. Kapitza. "Running ZooKeeper Coordination Services in Untrusted Clouds." In: *10th Workshop on Hot Topics in System Dependability (HotDep 14)*.

[10] S. Checkoway and H. Shacham. "Iago attacks: Why the system call API is a bad untrusted RPC interface." In: *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS* (2013).

[11] V. Costan, I. Lebedev, and S. Devadas. “Sanctum: Minimal Hardware Extensions for Strong Software Isolation.” In: *25th USENIX Security Symposium (USENIX Security 16)*.

[12] J. Criswell, N. Dautenhahn, and V. Adve. “Virtual Ghost: Protecting Applications from Hostile Operating Systems.” In: *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems - ASPLOS 14*.

[13] $db_bench - MainbenchmarktoolforRocksDB$. Accessed: 18-01-2020. URL: https://github.com/facebook/rocksdb/wiki/Benchmarking-tools.

[14] J. Dean and S. Ghemawat. “MapReduce: Simplified Data Processing on Large Clusters.” In: *Communications of the ACM* (2004).

[15] *Field-Programmable Gate Array*. Accessed: 13-02-2020. URL: https://en.wikipedia.org/wiki/Field-programmable_gate_array.

[16] B. Fuhry, R. Bahmani, F. Brasser, F. Hahn, F. Kerschbaum, and A.-R. Sadeghi. “HardIDX: Practical and Secure Index with SGX.” In: 2017.

[17] T. C. Group. *Trusted Platform Module (TPM) Summary*. Accessed: 20-06-2019. URL: https://trustedcomputinggroup.org/wp-content/uploads/Trusted-Platform-Module-Summary_04292008.pdf.

[18] O. Hofmann, S. Kim, A. Dunn, M. Lee, and E. Witchel. “InkTag.” In: *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems - ASPLOS 13*.

[19] “How much data do we create every day?” In: *Forbes - May ’18* (). Accessed: 20-01-2020. URL: https://www.forbes.com/sites/bernardmarr/2018/05/21/how-much-data-do-we-create-every-day-the-mind-blowing-stats-everyone-should-read/#77e3686860ba.

[20] “How much data is created on the internet each day?” In: *Microfocus* (). Accessed: 20-01-2020. URL: https://blog.microfocus.com/how-much-data-is-created-on-the-internet-each-day/.

[21] P. Hunt, M. Konar, F. Junqueira, and B. Reed. “ZooKeeper: Wait-free Coordination for Internet-scale Systems.” In: *In USENIX Annual Technical Conference*.

[22] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel. “Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data.” In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*.

[23] *Intel Software Security Guard Extentions*. Accessed: 04-07-2019. URL: https://www.intel.com/content/www/us/en/architecture-and-technology/software-guard-extensions.html.

[24] “Internet Growth Statistics.” In: *InternetWorldStats* (). Accessed: 13-01-2020. URL: https://www.internetworldstats.com/emarketing.htm.

[25] P. Jain, S. Desai, S. Kim, M.-W. Shih, J. Lee, C. Choi, Y. Shin, T. Kim, B. Kang, and D. Han. "OpenSGX: An Open Platform for SGX Research." In: 2016.

[26] S. Kamara and C. Wright. "Inference Attacks on Property-Preserving Encrypted Databases." In: 2015.

[27] S. Kim, Y. Shin, J. Ha, T. Kim, and D. Han. "A First Step Towards Leveraging Commodity Trusted Execution Environments for Network Applications." In: *Proceedings of the 14th ACM Workshop on Hot Topics in Networks - HotNets-XIV*. 2015.

[28] T. Kim, J. Park, J. Woo, S. Jeon, and J. Huh. "ShieldStore: Shielded In-memory Key-value Storage with SGX." In: *Proceedings of the Fourteenth EuroSys Conference 2019*.

[29] Y. Kwon, A. Dunn, M. Lee, O. Hofmann, Y. Xu, and E. Witchel. "Sego." In: *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS 16*.

[30] D. Lie, C. Thekkath, and M. Horowitz. "Implementing an untrusted operating system on trusted hardware." In: *Proceedings of the nineteenth ACM symposium on Operating systems principles - SOSP 03*.

[31] Y. Liu, T. Zhou, K. Chen, H. Chen, and Y. Xia. "Thwarting Memory Disclosure with Efficient Hypervisor-enforced Intra-domain Isolation." In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security - CCS 15*.

[32] J. M. McCune, B. Parno, A. Perrig, M. Reiter, and H. Isozaki. "Flicker: an execution infrastructure for TCB minimization." In: *EuroSys'08 - Proceedings of the EuroSys 2008 Conference* ().

[33] F. McKeen, I. Alexandrovich, I. Anati, D. Caspi, S. Johnson, R. Leslie-Hurd, and C. Rozas. "Intel Software Guard Extensions (Intel-SGX) Support for Dynamic Memory Management Inside an Enclave." In: *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016 on - HASP 2016*.

[34] *Memcached: A high performance, distributed memory object caching system*. Accessed: 28-10-2019. URL: http://memcached.org/.

[35] *Microsoft Palladium: Next Generation Secure Computing Base*. Accessed: 10-10-2019. URL: https://epic.org/privacy/consumer/microsoft/palladium.html.

[36] S. Mofrad, F. Zhang, S. Lu, and W. Shi. "A comparison study of intel SGX and AMD memory encryption technology." In: *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy - HASP 18*.

[37] T. D. Ngoc, B. Bui, S. Bitchebe, A. Tchana, V. Schiavoni, P. Felber, and D. Hagimont. "Everything You Should Know About Intel SGX Performance on Virtualized Systems." In: 2019.

[38] “Over a third of firms have suffered cloud attacks.” In: *Infosecurity Magazine* (). Accessed: 20-01-2020. URL: https://www.infosecurity-magazine.com/news/over-third-firms-have-suffered/.

[39] “Playstation Network attack.” In: *The Guardian 2011* (). Accessed: 20-01-2020. URL: https://www.theguardian.com/technology/2011/apr/26/playstation-network-hackers-data.

[40] R. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan. “CryptDB: Protecting confidentiality with encrypted query processing.” In: *SOSP’11 - Proceedings of the 23rd ACM Symposium on Operating Systems Principles* ().

[41] D. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. Hunt. “Rethinking the Library OS from the Top Down.” In: *Sigplan Notices - SIGPLAN* (2011).

[42] C. Priebe, K. Vaswani, and M. Costa. “EnclaveDB: A Secure Database Using SGX.” In: *2018 IEEE Symposium on Security and Privacy (SP)*.

[43] H. Raj, S. Saroiu, A. Wolman, R. Aigner, J. Cox, P. England, C. Fenner, K. Kinshumann, J. Loeser, D. Mattoon, M. Nystrom, D. Robinson, R. Spiger, S. Thom, and D. Wooten. “fTPM: A Software-Only Implementation of a TPM Chip.” In: *25th USENIX Security Symposium (USENIX Security 16)*.

[44] *Redis*. Accessed: 28-10-2019. URL: http://www.redis.io/.

[45] A. Ribeiro. “Management of Trusted and Privacy Enhanced Cloud Computing Environments.” Master’s thesis. FCT, Universidade Nova de Lisboa, 2019.

[46] M. Roesch and S. Telecommunications. “Snort - Lightweight Intrusion Detection for Networks.” In: 1999.

[47] *Running Java Applications in SCONE with CAS-Policy*. Accessed: 10-11-2020. URL: https://sconedocs.github.io/Running_Java_Applications_in_Scone_with_remote_attestation/.

[48] N. Saboonchi. “Hardware Security Module Performance Optimization by Using a “Key Pool”.” Master’s thesis. KTH, Royal Institute of Technology in Stockholm, 2014.

[49] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. “VC3: Trustworthy Data Analytics in the Cloud Using SGX.” In: *2015 IEEE Symposium on Security and Privacy*.

[50] *SCONE Configuration and Attestation Service (CAS)*. Accessed: 07-11-2020. URL: https://sconedocs.github.io/CASOverview/.

[51] M.-W. Shih, M. Kumar, T. Kim, and A. Gavrilovska. “S-NFV.” In: *Proceedings of the 2016 ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization - SDN-NFV Security ’16*.

[52] S. Shinde, D. Le Tien, S. Tople, and P. Saxena. “Panoply: Low-TCB Linux Applications with SGX Enclaves.” In: *Proceedings 2017 Network and Distributed System Security Symposium*.

[53] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. “The Hadoop Distributed File System.” In: *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*.

[54] R. Sinha and M. Christodorescu. “VeritasDB: High Throughput Key-Value Store with Integrity using SGX.” In: *IACR Cryptology ePrint Archive* (2018).

[55] D. Tian, J. Choi, G. Hernandez, P. Traynor, and K. Butler. “A Practical Intel SGX Setting for Linux Containers in the Cloud.” In: *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy - CODASPY 19*.

[56] *Trusted Computing Platform Alliance (TCPA)*. 2002.

[57] C.-c. Tsai, D. Porter, and M. Vij. “Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX.” In: *2017 USENIX Annual Technical Conference (USENIX ATC 17)*.

[58] O. Weisse, V. Bertacco, and T. Austin. “Regaining Lost Cycles with HotCalls.” In: *Proceedings of the 44th Annual International Symposium on Computer Architecture - ISCA 17*.

[59] “Who coined cloud computing?” In: *TechnologyReview* (). Accessed: 20-01-2020. URL: https://www.technologyreview.com/s/425970/who-coined-cloud-computing/.

[60] N. Zhang, M. Li, W. Lou, and Y. Thomas Hou. “MUSHI: Toward Multiple Level Security cloud with strong Hardware level Isolation.” In: *MILCOM 2012 - 2012 IEEE Military Communications Conference*.

[61] W. Zheng, A. Dave, J. Beekman, R. Ada Popa, J. Gonzalez, and I. Stoica. “Opaque: An Oblivious and Encrypted Distributed Analytics Platform.” In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*.

Appendix A

Attestation Secret Example

Listing A shows an example of the YAML file stored in CAS that holds the secrets needed for an application to run properly, only upon attestation.

```
version: 0.3
name: exampleThesis_joaoreis
services:
   - name: redis
     mrenclaves: [3f33b7dd2f6997a9547b9cc2983502a3853802d5b52838d8b47226ed7173a15d] #
          ↪ MRENCLAVE = Enclave hash: from execution with SCONE_VERSION=1
     command: "redis-server␣/usr/local/etc/redis/redis.conf"
     image_name: redisKey_image redisCrt_image
     pwd: /usr/local/etc/redis/
     environment:
        LD_LIBRARY_PATH: "/usr/lib/jvm/java-1.8-openjdk/jre/lib/amd64/server:/usr/lib/jvm/
             ↪ java-1.8-openjdk/jre/lib/amd64:/usr/lib/jvm/java-1.8-openjdk/jre/../lib/
             ↪ amd64"
        JAVA_TOOL_OPTIONS: "-Xmx256m"
        TMP_SECRET_VAR: "This␣is␣a␣protected␣secret␣distributed␣by␣Scone␣CAS!"

secrets:
  - name: redisKey_secret
    kind: private-key
    value: |
       -----BEGIN PRIVATE KEY-----
                  <KEY>
       -----END PRIVATE KEY-----

  - name: redisCrt_secret
    kind: x509-ca
    private_key: redisKey_secret
    value: |
       -----BEGIN CERTIFICATE-----
```

```
                  <CERT>
       -----END CERTIFICATE-----

images:
  - name: redisKey_image
    injection_files:
      - path: /usr/local/etc/redis/redis.key
        content: |
          $$SCONE::redisKey_secret$$

  - name: redisCrt_image
    injection_files:
      - path: /usr/local/etc/redis/redis.crt
        content: |
          $$SCONE::redisCrt_secret$$

security:
   attestation:
     tolerate: [debug-mode, outdated-tcb]
     ignore_advisories: ["INTEL-SA-00220", "INTEL-SA-00270", "INTEL-SA-00293", "INTEL-SA
          ↪ -00320", "INTEL-SA-00329"]
```