Miguel de Lemos Dias Rosa Anciães

Bachelor of Computer Science and Engineering

A Trusted and Privacy-Enhanced In-Memory Data Store

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

A Trusted and Privacy-Enhanced In-Memory Data Store

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 everyone who stood with me during my student years, to all my family and friends

Acknowledgements

I would like to thank the Faculty of Sciences and Technology of NOVA University Lisbon and all its faculty, especially professor Henrique João Lopes Domingos who supported me all throughout this dissertation and always took the time to help and motivate me.

I would also take this opportunity to thank my family, as they have supported me unconditionally not only during the college years but all my life. I thank Sofia Miranda who tolerated my mood swings and my stress during this period, calmed me down and has listen to my outbursts and thesis talk even though computer science is not her area.

I would like to thank the whole WATERDOG team, they took me in since my graduation and helped me start my adult career making me feel like one of their own since the beginning and with an especial thank you to Bruno Félix and Pedro Coutinho who picked up the slack of the days I've missed work to have time for this dissertation. They are extraordinary and amazing work colleagues.

Lastly, I want to thank my friends who went through the dissertation at the same time, especially João Reis, Tiago Sousa and Ricardo Amaral, we "suffered" together but now it is done. Also to my friends who already overcame this objective or are still to do it. You know who you are.

Thanks to all, could not have done it without you...

There is no cloud, it's just someone else's computer

Abstract

The recent advent of hardware-based trusted execution environments provides isolated execution, protected from untrusted operating systems, allowing for the establishment of hardware-shielded trust computing base components. As the processor provides such a "shielded" trusted execution environment (TEE), their use will allow users to run applications securely, for example on the remote cloud servers, whose operating systems and hardware are exposed to potentially malicious remote attackers and non-controlled system administrators' staff. On the other hand, Linux containers managed by Docker or Kubernetes are interesting solutions to provide lower resource footprints, faster and flexible startup times, and higher I/O performance, compared with virtual machines (VM) enabled by hypervisors. However, these solutions suffer from software kernel mechanisms, easier to be compromised in confidentiality and integrity assumptions of supported application data. This dissertation designed, implemented and evaluated a Trusted and Privacy-Enhanced In-Memory Data Store, making use of a hardware-shielded containerised OS-library to support its trust-ability assumptions. To support large datasets, requiring data to be mapped outside those hardware-enabled containers, our solution uses partial homomorphic encryption, allowing trusted operations executed in the protected execution environment to manage in-memory always-encrypted data, that can be or not mapped inside the TEE.

Keywords: Hardware Security; Privacy-Enhanced Data Store; Homomorphic Encryption; Isolated Environments; Trusted Computing; Cloud Computing; Virtualisation; Containerisation; Availability; Reliability.

Resumo

Os recentes avanços de ambientes de execução confiáveis baseados em hardware fornecem execução isolada, protegida contra sistemas operativos não confiáveis, permitindo o estabelecimento de componentes base de computação de confiança protegidos por hardware. Como o processador fornece esses ambientes de execução confiável e "protegida"(TEE), o seu uso permitirá que os utilizadores executem aplicações com segurança, por exemplo em servidores *cloud* remotos, cujos sistemas operativos e hardware estão expostos a atacantes potencialmente maliciosos assim como administradores de sistema não controlados. Por outro lado, os *containers* Linux geridos por sistemas *Docker* ou *Kubernetes* são soluções interessantes para poupar recursos físicos, obter tempos de inicialização mais rápidos e flexíveis e maior desempenho de I/O (interfaces de entrada e saída), em comparação com as tradicionais máquinas virtuais (*VM*) activadas pelos hipervisores. No entanto, essas soluções sofrem com software e mecanismos de kernel mais fáceis de comprometerem os dados das aplicações na sua integridade e privacidade.

Esta dissertação projectou, implementou e avaliou um Sistema de Armazenamento de Dados em Memória Confiável e Focado na Privacidade, utilizando uma biblioteca conteinerizada e protegida por hardware para suportar as suas suposições de capacidade de confiança. Para oferecer suporte para grandes conjuntos de dados, exigindo assim que os dados sejam mapeados fora dos *containers* seguros pelo hardware, a solução utiliza encriptação homomórfica parcial, permitindo que operações executadas no ambiente de execução protegido façam gestão de dados na memória que estão permanentemente cifrados, estando eles mapeados dentro ou fora dos *containers* seguros.

Palavras-chave: Segurança de Hardware; Armazenamento de Estrutura de Dados em Memória Confiável e Focado na Privacidade; Encriptação Homomórfica, Ambientes Isolados; Computação Segura; Computação em *Cloud*; Virtualização, Containerização; Disponibilidade; Confiabilidade.

Contents

List of Figures

List of Tables

Listings

Acronyms

ACL	Access Control List
AES	Advanced Encryption Standard
AIK	Attestation Identity Key
API	Application Programming Interface
AWS	Amazon Web Services
CA	Certification Authority
CAS	Configuration and Attestation Service
CPU	Central Processing Unit
CSV	Comma Separated Values
DBMS	Database Management System
DDoS	Distributed Denial of Service
DoS	Denial of Service
Ecall	Enclave call
ECB	Electronic Code Book
EK	Endorsement Key
EPC	Enclave Page Cache
EPCM	Enclave Page Cache Mapping
EPID	Enhanced Privacy ID
GB	Gigabyte
HSM	Hardware Security Module
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
I/O	Input/Output

IaaS	Infrastructure as a Service
IoT	Internet of Things
IP	Internet Protocol
JVM	Java Virtual Machine
JWT	Json Web Token
KVM	Kernel Virtual Machine
KVS	Key-Value Store
LAS	Local Attestation Service
LRU	Least Recently Used
LSM	Log-Structured Merge Tree
MB	Megabyte
MIM	Man-in-the-middle
Ocall	Out call
OOM	Out Of Memory
OS	Operating System
OTP	One-Time Password
P2P	Peer to Peer
PCR	Platform Configuration Register
PRM	Processor's Reserved Memory
RAM	Random Access Memory
REST	Representational State Transfer
RSA	Rivest–Shamir–Adleman
RSS	Resident Set Size
SaaS	Software as a Service
SASL	Simple Authentication and Security Layer
SDK	Software Development Kit
SGX	Software Guard Extensions
SQL	Structured Query Language
SSL	Secure Sockets Layer

syscall	System call	
TB	Terabyte	
TCB	Trusted Computing Base	
TCE	Trusted Computing Environments	
TCG	Trusted Computing Group	
TEE	Trusted execution environment	*
TLS	Transport Layer Security	
TPM	Trusted Platform Module	
USB	Universal Serial Bus	
VM	Virtual Machine	

Chapter 1

Introduction

In this chapter, it's presented the context and motivation for this thesis, the main problem statement followed by the goals and objectives and all the planned contributions. In the end, it is presented the structure used in the following chapters of the document.

1.1 Context and Motivation

Cloud computing has gone through many steps, that include grid and utility computing, application service provision and software as a service before reaching the level we know these days. The concept of delivering continuous resources through a global network is rooted in the 1960s. Some experts credit the professor and computer scientist John McCarthy [77] with proposing the concept of computation being delivered as a public utility.

Then, around the 1970s the concept of the virtual machine (VM) started to gain popularity as it permitted multiple distinct computing environments to reside on one physical machine.

One of the first major cloud computing moments was the arrival of *salesforce.com* that pioneered the concept of delivering enterprise applications via a simple website. Later, around the 2000s, current big names like Oracle, SAP, Google, Amazon and Microsoft joined the trend and made the cloud world as it is today [2, 3].

Over the past decade, cloud computing has evolved from something service providers told companies they should adopt, to becoming the technology heart of not only major companies but medium-sized enterprises, small start-ups, personal projects and pretty much anyone who works in the computer science world.

Recent studies are foreseeing that 80% of enterprise IT will move to the cloud by 2025 [1]. The array of services provided now are endless and the costs are attractive to

businesses. These services allow developers to only pay for resource usage, and to take advantage of all the power of very large companies. Scalability at request, reliability with daily backups and seamless integration with a lot of other services are some advantages of moving to the cloud. And all of these functionalities without having to manage big infrastructures and a lot of servers, networks, disks, etc... [95].

All of this data and processing happening in someone else's machine started to raise privacy and security concerns. It has become a very attractive target for malicious hackers to attack cloud providers due to the amount of data they process and hold on their services. The best security researchers are always working with the providers to try and mitigate all bugs and vulnerabilities on their very large platforms which have become also a big attack vector. It has been reported by Microsoft, that *"There was a 300 per cent increase in Microsoft cloud-based user accounts attacked year-over-year (Q1-2016 to Q1-2017)."* and also *"The number of account sign-ins attempted from malicious IP addresses has increased by 44 per cent year over year in Q1-2017."* [72]. Another example published on the Washington Post describes a sophisticated Man-in-the-Middle (MIM) cyber-attack that has targeted Apple's iCloud service in China, in an apparent attempt to collect user names, passwords and other private information [10]. Also, Amazon Web Services has been in 2019 hit by a massive DDoS (Distributed Denial of Service) attack that kept the system down for about 8 hours straight, which can mean thousands of dollars lost by clients [21].

The use of virtual machines to lodge different computing environments on the same physical machine can also raise problems, as explained by the publication *"Seriously, get off my cloud! (...)"* where the researches were able to exploit and obtain RSA (encryption) keys from other VMs deployed in the same physical machine of Amazon EC2 service. This work affirms the need for stronger isolation techniques in public clouds [49].

Docker containers and Kubernetes clusters, used instead and/or alongside traditional VMs, are two of the most popular technologies among cloud providers and cloud server environments these days, and if not managed correctly can become attack vectors. They are already being exploited, most known by the report of Tesla Motors [94], suffering a breach because of an exposed Kubernetes instance [29, 30].

The well known Cambridge Analytica scandal [96] gave the world another perspective about the security guaranteed by the cloud providers, social media and every platform that keeps user's data in a non-secure manner. It shows how it can be exploited, sold and manipulated without the owner's consent.

As for Redis as a storage solution, and explained in depth in section 2.1.2, *"Redis should not be publicly exposed as it has no default authentication and all the data is stored in clear text"* and so, according to some studies, there are around seventy-two thousand Redis servers available online today, and over 75% of them were compromised and infected with some kind of malware [15, 31, 74].

It is certain that as the cloud environment grows, the motivation for malicious hackers to attack and try to steal information will grow with it, and with all these security

breaches increasing year after year, a previously mentioned study also reflects that *"66% of IT professionals say security is their greatest concern in adopting an enterprise cloud computing strategy"* [1].

1.2 Problem Statement

The problem behind the goals and objectives of this dissertation can be summarised in designing a system, answering the following questions:

Is it feasible to implement a solution for a remote trusted and privacy-enhanced cloud-based key-value store system providing strong security and privacy features in a trustworthy solution? Can we design and implement those features with a good trade-off with operational and performance criteria under scalability and reliability guarantees? Can we address the trustability by minimising the trust-computing base using protected components running as isolated trusted bases in hardware-shielded trust execution environments? Is it possible to remove the threat of administrators of a cloud platform breaking data privacy? Can we combine the security guarantees with privacy-enhanced in-memory operations supporting big data sets and grained data-structures?

1.3 Objectives and Planned Contributions

The main goal of this dissertation is to design implement and evaluate a privacy-enhanced in-memory key-value store to be used as a trusted cloud service with hardware-based security features running in a trusted execution environment supported by Intel's SGX technology. To overcome the protected memory limitations and coarse-grained paging as supported in SGX memory-management facilities, our solution must support flexible big data sets in a hybrid approach using SGX-mapped protected memory and unprotected memory. Data in unprotected memory will be encrypted and operated in the encrypted form, using partial homomorphic encryption techniques. For implementation purposes, we will address our solution to be leveraged from the REDIS technology, enhanced in an architecture using isolated and containerised services running in isolated Intel-SGX trusted-execution-enclaves but supporting all the variations of REDIS-based architectural deployments, e.g.: as a single REDIS server instance, or using replicated instances (with master-slave and master-slave tree-chains, as well as, clustered instances). We will analyse and compare the designed solution in terms of the introduced security benefits and measuring the overheads introduced by the additional security, privacy and trustability guarantees.

In this thesis we plan to achieve the following contributions:

- **Design and implementation of the cloud-enabled privacy-enhanced solution**, with all-in-the-box planned features as described above, and able to be used as a "cloud-platform as a service" solution, providing:

  - **Trust-ability, security and privacy** properties with the following guarantees: high availability, built-in replication, LRU eviction model, in-memory operations on encrypted big data sets and complementary options for protected on-disk persistence.
  - **Software attestation** guarantees provided to clients (users) to validate the correctness and integrity of the remote software stack providing the solution.

- **Multiple Replication Mechanisms** based on the same secure solution to analyse how these types of replication (centralised solution, a Master-Slave architecture and a clustering solution) will be impacted by the additional security features.

- **Drastically reduce TCB** in the remote cloud provider by removing the millions and millions of lines of code implementing the hypervisors and operating systems used in their infrastructures thus creating a **truly isolated system** by leveraging Intel's SGX technology to create a shielded and trusted execution environment in a remote cloud provider.

- **Complete analysis report** of the different solutions of replication and security levels, comparing a normal non-secure solution with the privacy-enhanced implementation along with an evaluation of overheads and trade-offs introduced by the additional security mechanisms.

1.4 Report Organisation

The remaining of the report is organised as follows:

Chapter 2 presents the topic background, related work and initial research performed for this thesis, including relevant contributions and similar solutions existing in current days.

Chapter 3 discusses the elaboration phase by describing the implemented system model and architecture. It provides an in-depth explanation of how the proposed system achieves the goals and objectives of this dissertation.

Chapter 4 details the implementation of the previously presented system model. It shows how the system was implemented, the details of the design and important system features and how they were implemented. It also references all technologies used and their versions.

Chapter 5 explains how the system was tested, and the relevant metrics and information in order to calculate the performance and system resource usage of the system, always providing a comparison between the implemented the system and a normal implementation.

Chapter 6 summarises and concludes the dissertation by referring the achieved contributions, system limitations and future work directions.

Chapter 2

Related Work

This chapter presents and briefly discusses the related work and the study performed beforehand in order to guide and give some context to the reader. It will present work that was used as the basis of this thesis, existent technologies and their relation with this project, and some comparisons between those existing technologies, the problem addressed in this thesis and the solutions proposed to solve, or better address, those very same problems.

First, in section 2.1 we explain and discuss for the first time the definition of a Key-Value Store. We present some use cases, current technology available, their differences and most importantly their security models and concerns. Having discussed the software, section 2.2 will then address the environment on where that software will run - hardware. It explains and presents the different ways to secure and authenticate the hardware, prevent hardware-based attacks and discuss some of the current products available and how they will be used across this thesis. Section 2.3 will then make the bridge between software and hardware. It explains how Key-Value stores are currently being run on secure environments. It discusses how software and hardware work together to achieve a secure environment for an application to run. Section 2.4 will discuss how virtualisation can allow for faster deployments and easier implementation of secure code for the complicated frameworks of secure hardware such as Intel's SGX processors. To conclude the chapter, section 2.5 will combine the information of every sub-chapter and analyse it with a bigger perspective and better knowledge of the theme.

Section 2.3 is considered to be the **main core** investigation and directly related to the work planned for this thesis. As for the other sections, they provide a piece of background knowledge necessary for the understanding of the core of this dissertation.

Along through the next chapter, we summarise the main relevant ideas that can be retained from each section for our objectives and expected goals.

2.1 Key-Value Stores

Key-value stores are the simplest form of what computer scientists call a database. The simplicity lies on associating a value to a certain key and storing that pair, as well as retrieving the values of known keys. [54]

Listing 2.1: Redis Set & Get

```
redis> SET mykey "Hello"
"OK"
redis> GET mykey
"Hello"
```

Is this simplicity that makes this technology very attractive to developers. The ease of use, their high performance and speed are key aspects in favour of these technologies. However, simply working with keys and values might not be enough to more complex applications, and that is why Key-Value store product developers are introducing new features to make them appealing to a broader mass of users, always keeping them lightweight and fast.

For that lightweight and fast attributes, most of the key-value stores work in computer memory. This allows fast read and write operations as opposed to persistent disk storage. Although they work mainly in memory, most of the solutions offer some persistent mechanism so we can make use of its performance but still persist data in case of a disaster, server failure or any crash.

KVSs have been evolving for years and some are now more than a single key-value store module. A lot of them are now supporting a multi-model storage. Meaning that a value can be more than a single integer or a string. For example, Redis [78] as a multi-model store is not only a key-value store, but also [80]:

- **Document Store** - *"nonrelational database that is designed to store and query data as JSON-like documents"* [36]
- **Graph DBMS** - *"Graph databases are purpose-built to store and navigate relationships. Use nodes to store data entities, and edges to store relationships between entities"* [42]
- **Search Engine** - *"nonrelational database that is dedicated to the search of data content. Use indexes to categorize the similar characteristics among data"* [85]
- **Time Series DBMS** - *"Provides optimum support for working with time-dependent data. Each entry has a timestamp, the data arrives in time order and time represents a primary axis for the information."* [97]

So, the KVS world is becoming more and more versatile as the years pass.

In the next subsections, its discussed and presented the overview of the current KVS technology. We picked some top KVSs technologies nowadays according to db-engines [55] website.

2.1.1 Memcached

Memcached [67] is a free and open-source key-value store released in 2003. It is described as a high performance distributed memory object caching system.

It is designed to hold small chunks of data (strings and objects) to work as a cache for results of database calls, API calls, or page rendering. Its biggest use case is for use in speeding up dynamic web applications by alleviating database load.

This system lies on the simpler key-value store spectrum. It takes advantages of the simplicity of a key-value store to edge ease of development and solving many problems facing large data caches. Its API is available for most popular languages. It has a LRU eviction technique which means that items will expire after a specified amount of time if not used.

When it comes to system replication, availability and reliability, Memcached has an interesting approach. To keep it blazing fast, there is no communication between server instances in a cluster. Memcached servers are unaware of each other. There is no crosstalk, no synchronisation, no broadcasting, no replication. Adding servers will only increase the available memory.

As for its security context, Memcached spends very little, if any, effort in securing the systems for random internet connections. The servers only have support for SASL [83] authentication mechanism. This method of authentication is not implemented as end-to-end encryption, it only provides restriction access to the daemon, but it does not hide communications over the network. That means it is not meant to be exposed to the internet or any untrusted users [68].

2.1.2 Redis

Redis [78] is an in-memory data structure store that can be used as a database, cache and also a message broker. Redis focuses on performance, so most of its decisions prioritise high performance and very low latency.

It has been benchmarked as the world's fastest database [81] and together with their multi-model and its rich set of operations that can be performed over data it has been the leading key-value store according to use and popularity for a multiple set of years [55].

Listing 2.2: How Fast is Redis

```
$ redis-benchmark -t set -r 100000 -n 1000000
====== SET ======
1000000 requests completed in 8.78 seconds
50 parallel clients
3 bytes payload
keep alive: 1

99.59% <= 1 milliseconds
99.98% <= 2 milliseconds
```

```
100.00% <= 2 milliseconds
113934.14 requests per second
```

As said before, Redis is now not a simple KVS. It supports data structures such as strings, hashes, lists, sets, sorted sets with range queries, bitmaps, hyperloglogs, geospatial indexes with radius queries and streams. It also has built-in replication, server-side scripting, LRU eviction, the concept of transactions and different levels of persistence. It provides high availability and automatic partitioning as well.

Redis provides two modes of replication/availability. The master-slave form of replication works with a single node (master node) where all writes occurring will be replicated to the other Redis instances (slave nodes). Writes on nodes other than the master will not be replicated. Redis provides a read-only setting that can be applied to slave nodes to prevent state differences between instances. The cluster mode of replication consists of partitioning the data between multiple master nodes. This mechanism does not provide replication as each data object is stored on a single master node but will allow for lower dataset sizes on each node and therefore faster response times. This method is usually mixed with the first one to provide both performance and replication features, by adding slave nodes to each of the masters.

Security is not Redis' primarily concern (just like others). *"In general, Redis is not optimised for maximum security but for maximum performance and simplicity"* [82]. It is designed to be accessed by trusted clients inside trusted networks. This means that it is not supposed to be publicly exposed. Redis (in its latest release 6.0.6 [79]) now implements an ACL policy that allow the configuration of multiple users with different permissions over the dataset. It also added support for SSL network communication security.

There are a few other security concerns that Redis addresses, but as we can now start to see, in these types of stores, security falls behind performance and usability.

2.1.3 Amazon Dynamo DB

Amazon Dynamo DB [6] is a fully managed NoSQL database service. It is a key-value store and a document store that is built based on the dynamo paper [32]. This paper describes a P2P (peer-to-peer) network with high availability, eventual consistency and very easily scalable. It also successful handles server and data centre failures and network partitions.

Amazon builds on this paper and offers DynamoDB as a service in its platform. It is a hosted system in the Amazon Web Services [8] infrastructure and it is fully managed. That means no need for low-level server configurations or maintenance. It is all managed by the AWS team and offered to the user with a nice configuration interface. It also means that it has built-in security, backup and restore and in-memory caching for internet-scale applications. Also, it offers seamless scalability by increasing the number of nodes/servers according to current traffic received by the application on any given time.

This technology focuses more on high availability but also achieves very high performances and very low latency and being fully managed it also takes advantages of the AWS infrastructure full power. It currently sits second on the db-engines [55] most popular ranking.

2.1.4 Microsoft Azure Cosmos DB

Microsoft Azure Cosmos DB [70] is a fully managed database service provided by Microsoft Azure [71]. This service provides a globally distributed, horizontally scalable, multi-model database. Its multi-model architecture can work as a key-value store, a Document Store, a graph DBMS and a wide column store.

It's very proud and excels in the ease of global scale with the system call *Turnkey global distribution*, providing transparent multi-master replication and a set of user-configurable consistency options. It also strongly advertises a *Multi-Model Multi-Api* feature where you can use multiple data types on this single database service. Cosmos DB automatically indexes all data and allows the user to use various NoSQL APIs to query the data.

As a fully managed service, Cosmos DB makes use, in the background, of the large infrastructure with almost unlimited resources and capabilities provided by Microsoft, which means it also has built-in security, fail-over mechanisms for disaster recovery, and high performance with single-digit read and write latencies.

2.1.5 Microsoft Azure Cache for Redis

Microsoft Azure Cache for Redis [69] is a service provided by Microsoft Azure that joins the open-source world of Redis with the commercial side of a fully managed and hosted platform.

It uses at its core the Redis server technology and provides ease of deployment and management, built-in global replication, Azures' infrastructure security and flexible scaling and Redis superior throughput and low latency performance.

Being in the Azure ecosystem provides nice integration with all Azures' services as shown in figure 2.1.

2.1.6 Aerospike

Aerospike [4] is an enterprise-grade, high-performance Key-Value Store. It is another KVS technology currently available today. It promises a philosophy of *"no data loss"* through Strong Consistency. Normal systems trade requiring this type of consistency usually trade performance for data integrity but Aerospike allows it with minimal performance loss. That means it can be used for example in banking payments, retail and telecommunications use cases.

It also provides dynamic cluster management and unique flexible storage. That enables very easy deployments and particularly very easy scalability, so it is able to meet

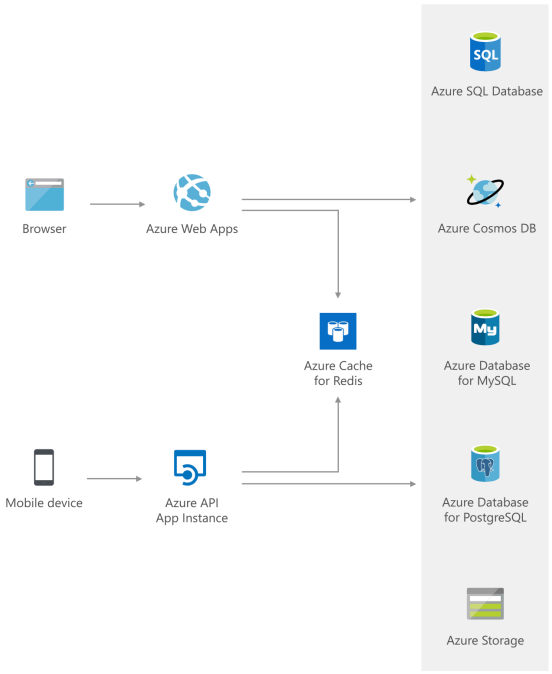


Figure 2.1: Azure Environment Integration

any data volume needs and still maintaining low latencies across that wide range of data volumes, from low volumes until hundreds TB of data.

As for security, it includes (the enterprise version) a database access management and audit trail logs. It also includes transport-level encryption for client-server traffic and cross-data center traffic [5].

2.1.7 Discussion

In this chapter, we gathered information about the overview of the current Key-Value Store state of the art. It can be concluded that the most important feature of this technology is the performance and all of the above products mentioned do focus on that characteristic. Some of them even compromise in other features to achieve the best performance possible. Security is not the main concern and the most used measures in the current technologies being securities implementations at the network and transport level by using TLS and also full disk encryption.

Network and transport layer security is a must when implementing any system, and this thesis will also use those standards.

As for full disk encryption on the server, it opens up some attack vectors. Full disk encryption means that random users will not be able to query the data but credentialed users can. Although, anyone with full access to the database, for example, database operators or/and administrators, can decrypt and access all information. This creates a risk of privacy breaking due to hackers wielding stolen credentials, rogue insiders who have

been granted more access than they need or the well known honest-but-curious adversary model, where an administrator with full credentials does not have bad intentions, but, driven by curiosity, access information, therefore, breaking data privacy. A cloud-based KVS service like the ones talked above, this type of vulnerabilities can be a major concern for a use case with very sensitive data since the server would be off premises, there is no control over it when it comes to the privacy of data.

This thesis implemented a system based on Redis, the most popular and used Key-Value Store currently used and will try to solve some of the problems with security described above. It will compare the principal feature of a KVS, the performance, of a simple and normal Redis server and a privacy-enhanced Redis solution so the user can calculate the trade-off between performance and security and applied the correspondent solution to their own use case.

2.2 Trusted Computing Environments

Modern data processing services hosted in the cloud are under constant attack from malicious system administrators, server administrators and hackers who exploit bugs on applications, operating systems or even the hypervisor. However, current days shows a massive trend of business moving to the cloud infrastructure looking for easy deployment, managed services with built-in replication and fault tolerance, fast and trivial scaling and predicted costs.

With more and more data exposed in the cloud, hackers have a bigger desire to exploit and look for vulnerabilities. This results in frequent data breaches that reduce trust in online services. The need for cloud providers to ensure a level of security and trust to make the user comfortable of moving its data to the cloud has never been bigger, and with that need, some solutions in the form of Trusted Computing Environments (TCE) appeared.

Trusted Computing is a concept that strives to provide strong confidentiality and integrity guarantees for applications running on untrusted platforms. It forces a certain machine to behave an expected way even if running on a remote or machine that is out of our control.

TCE will also provide a decrease of the Trusted Computing Base (TCB) - the number of components that the application needs to trust in order to run smoothly. By isolating the service running on this trusted environments (limiting the set of instructions available and encrypting data), it prevents the operating system, the hypervisor and even malicious system administrators (three components normally on the TCB) to break data confidentiality and integrity within this environments.

There are a few hardware/software-based solutions to achieve a trusted computing environment, and they will be explained in the next sections.

2.2.1 TPM – Trusted Platform Modules

A Trusted Platform Module, also known as a TPM, is a technology proposed by the Trusted Computing Group (TCG) designed to provide hardware-based security-related functions. It's a chip embedded into the motherboard and includes multiple security mechanisms to make it tamper-resistant to physical harm and malicious software is unable to mess with its security features [98]. Some key advantages of using TPMs are:

- Generate, store, and limit the use of cryptographic keys
- Platform identity by using the TPM's unique RSA key, which is burned into itself also known as Endorsement Key (EK) and never leaves the TPM.
- Help ensure platform integrity by taking and storing security measurements.

Figure 2.2 shows the mains components and services provided by a TPM module. As shown in the figure, all of them only have one access point I/O which is protected and safely managed by the TPM execution engine.

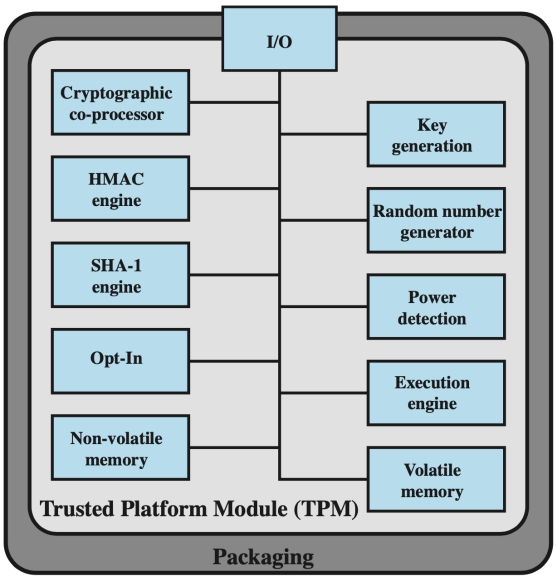


Figure 2.2: TPM insides

With all the components described by figure 2.2, TPMs provide there main TPM features: Encryption, Authenticated Boot and Attestation.

The first feature is used for every security and confidentiality aspects, mainly generating cryptographic keys, encrypting, signing and hashing data with secure standard algorithms melted in the module.

Authenticated Boot is the ability to boot the OS in stages, assuring that each portion of OS, as it is loaded, is a version trusted and approved for use, detecting hardware and software changes on every stage to verify if the code loaded can be trusted. This boot

sequence happens with the help Platform Configuration Registers (PCR) that store the trusted software hashes.

The attestation feature is a way for a client to remotely check the state of a machine and will be further explained in the next subsection.

2.2.2 TPM - Enabled Software Attestation

The remote attestation feature of a TPM is the ability of a program to authenticate itself against external verifiers. Is a mechanism that allows a remote party to verify the internal state of the OS or another software and decided whether or not that piece of software is intact and trustworthy. The verifier can trust that the attestation data is accurate and not tampered with because it is signed by the internal key of the TPM, a special key known as the Attestation Identity Key, known from now on as AIK [20].

A remote attestation procedure is described in image 2.3 [19]:

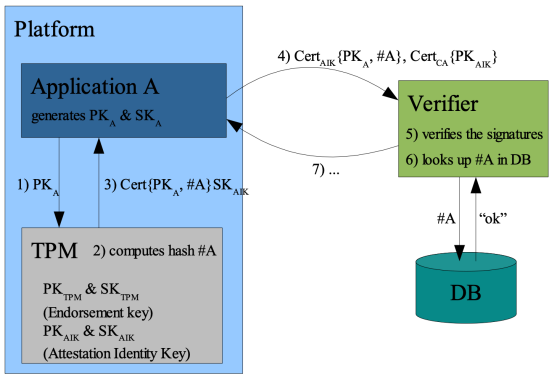


Figure 2.3: Remote Attestation Procedure

1. The application "A" generates a public/private key pair PK_A & SK_A and asks the TPM to certify it.
2. The TPM computes a hash value #A of the executable code of program "A".
3. The TPM creates a certification including PK_A and #A and signs it with the attestation identity key SK_{AIK}.
4. When application "A" wishes to authenticate itself to a remote party, it sends the certificate of its public key and hash value #A along with a certificate issued to the TPM by a trusted certification authority (CA).
5. The remote party verifies the certificate chain.
6. The remote party looks #A up in a database which maps hash values to trust levels.
7. If application "A" is deemed trustworthy, we continue the communication, probably by using PK_A to establish a session key.

2.2.3 HSM – Hardware Security Modules

An (HSM) hardware security module is normally an external module that can be added to a system, in form of a USB device or a component living in a secure network as a trusted server, instead of being embedded into the motherboard like a TPM. It provides a dedicated system of hardware enable accelerated cryptographic functions like encryption, decryption, key generation and signing capabilities [47]

What makes these devices so secure, like the TPM, is it can't be interfered with by external code, and it provides an array of protective mechanisms to detect and prevent external physical tampering like drill protection foil, resin-embedded chips as well as temperature and voltage sensors. Any detection of tampering will result in an alarm as well as countermeasures by the applications installed inside. [25].

HSM can have various applications and can be used in simple forms, for example, a specific bank dongle that generates OTP (one-time password) for accessing your account or be a big corporation and enterprise appliance in various industries, e-health, automotive and IoT systems.

2.2.4 Trusted Execution Environments

A Trusted Execution Environment (TEE) is an abstraction that describes a machine capable of executing a given program P in isolation, i.e. whose output is determined by the initial state of P and a set of defined inputs given into the TEE (Barbosa et al., 2016).

It is a secure area of the main processor that ensures sensitive data and code loaded inside is stored, processed and protected in an isolated and trusted environment. As such, it offers protection from software attacks even the ones generated in the operating system.

A TEE guarantees that:

- The code loaded in the environment is authentic and was not tampered by an attacker.
- All system state is correct (CPU registers, memory and sensitive I/O).
- The code, all data generated and runtime state is confidential and stored persistently.

The threat model of a TEE should include all software attacks and the physical attacks performed on the main memory and its non-volatile memory.

"There are many interpretations of what is meant by Trust. In the TEE it is used to imply that you may have a higher level of trust in validity, isolation and access control in items (assets) stored in this space, when compared to more general-purpose software environments"[102].

2.2.5 Intel SGX

"Intel® Software Guard Extensions (Intel® SGX) is a set of instructions that increases the security of application code and data, giving them more protection from disclosure or modification."[50].

These set of instructions are one of the latest iterations of trusted computing solutions and designs that tries to tackle the problem of securing remote computations by leveraging secure hardware on the remote host machine. The SGX processor enables a secure container called enclave which protects the confidentiality and integrity of the execution, such as code and data while relying on software attestation mechanisms.

A SGX can be though as a reverse sandbox. With a sandbox, you are trying to protect the system from your application, but with SGX you are trying to do the opposite and protect the application from the system. The system can be the OS, the hypervisor, the BIOS, the firmware or even the drivers [90].

A SGX enabled application is broken into two parts, the untrusted and trusted parts. The trusted part of the application is all the processing that deals with any sensitive data the application is handling. This part will be run inside enclaves and be stored in protected memory. The rest will live in normal memory and not be protected.

It provides this kind of security from the hardware by isolating all the private data from the outside, placing it into a restricted area of the memory called the PRM more specifically in the EPC (Enclave Page Cache) as shown on figure 2.4. The PRM is a zone of the RAM with guaranteed access management by the CPU where it will deny every external access and only allow access through the associated enclave. This region of the memory is also known as the private/protected memory or the trusted part of the application. The data managed in EPCs is decrypted and verified when loaded to CPU caches and handled on plaintext while is present there but is encrypted and integrity-protected when it leaves the CPU cache and is mapped in the external memory.

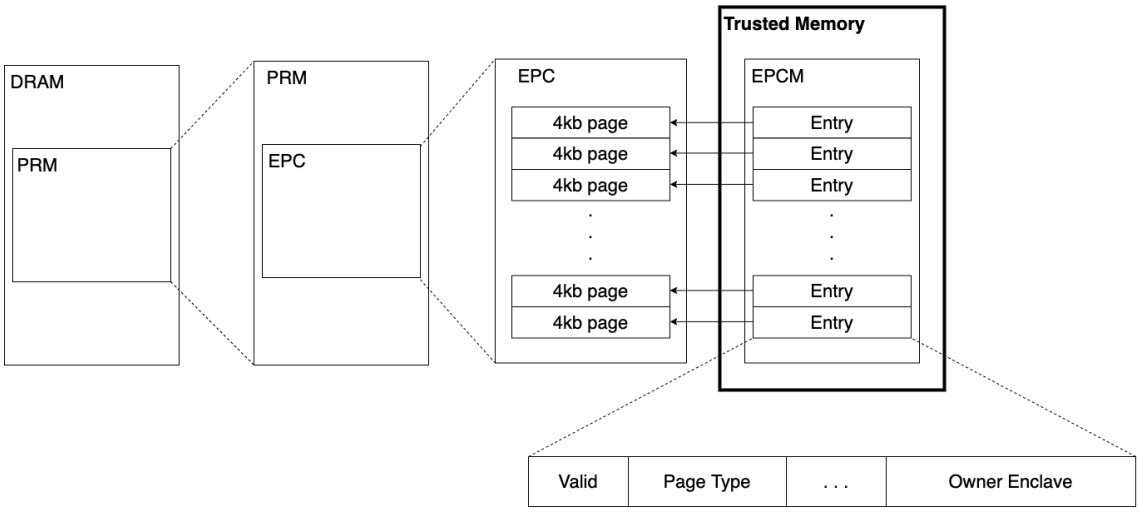


Figure 2.4: SGX Memory Architecture [46]

Because the SGX enclaves execute within the virtual address space of a process, the translation of enclave addresses must be trusted. However, since it is the OS that manages the translation between physical and virtual addresses (and the OS cannot be trusted), SGX maintains an internal data structure called the EPCM (Enclave Page Cache Mapping) which tracks the referred mapping as well as the information described on figure 2.4 [73].

With all this information about enclave pages, the processor can now perform and control access management to the enclave page cache described in figure 2.5, where it will deny access not only from outside the enclave but as well as from enclaves that do not own the page of memory request creating then, an isolated memory region.

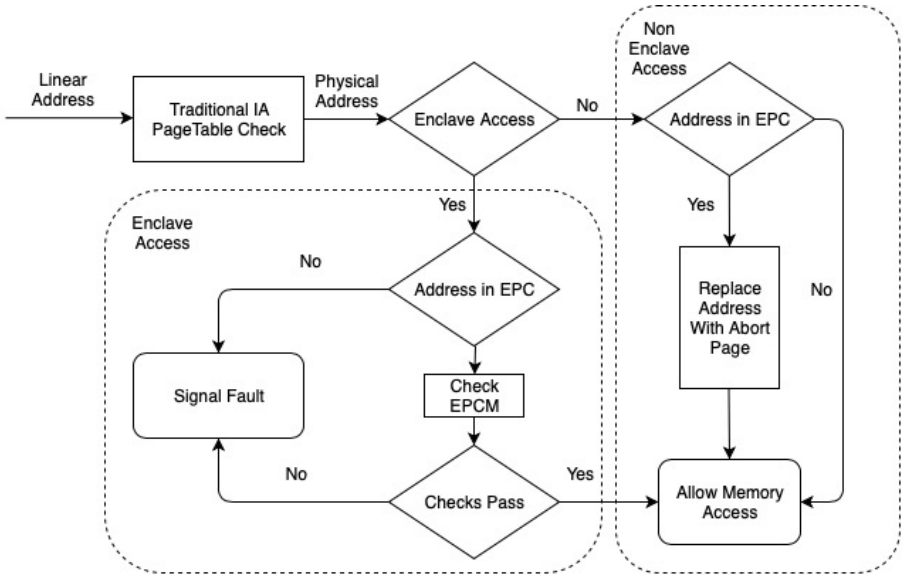


Figure 2.5: SGX Access Control [46]

The SGX will create an enclave when sensitive code needs to run by a specific SGX CPU instruction (ECREATE) and will create a unique instance of an enclave, establishing the linear address range and load the sensitive code into an EPC inside the protected memory. Once all pages are loaded into the EPC, and the loading is complete, an authentication hash is computed and is available for remote attestation so a user can verify that the code running in the enclave has not been tampered with.

An enclave must expose an API for the application to call in and advertise what services provided by the untrusted domain are needed. This is the definition of an interface boundary between the untrusted part of the code and the enclave and it is how they communicate. An Ecall is a function that the untrusted part application can call to execute some code inside the enclave and since it exposes a sensitive interface, to reduce the enclave attack surface, the number of Ecalls should be limited. On the other hand, the Ocall is a function that an enclave can call to reach a service/interface outside the enclave, on the untrusted OS. Again, calling some service out of the enclave can carry additional security risks and should be as minimal as possible [52].

When running, the execution always happens in protected mode, and to prevent data leaking, the CPU will not directly address an interrupt, fault or VM exit, but will instead emit another specific instruction (EEXIT) to properly exit the enclave, save CPU state into the enclave and only then will service the fault.

With all these properties, Intel® through SGX set of instructions and implementation tries to achieve a secure and trusted environment with guarantees of code and data isolation, confidentiality and integrity from attackers such as the OS, hypervisor, any hardware and even physical attacks [27].

2.2.6 Sanctum

Sanctum [28] is an open-source project that shares the same as goal as Intel SGX, providing strong provable software isolation to protected the data from external hardware and software, but claims to be simpler and protect against indirect attacks called side-channel attacks [59] such as cache timing attacks [24] that have been known to exist in SGX [41, 84]. These are additional software attacks that can infer private information by analysing a program's memory access patterns.

Following the minimal and simple concepts, it uses minimal invasive hardware and it does not require any modifications to the CPU major blocks, but only adds hardware to the interfaces between blocks. This allows for a respectable overhead by maintaining normal clock speeds as it does not modify the CPU core critical execution path.

Sanctum project builds on the SGX programming model and implements an architecture that deviates as little as possible from the one built by Intel. Although it differs from SGX by implementing the enclaves via a small combination of hardware extensions to RISC-V (an open-source set of CPU instructions [101]) and a trusted piece of software called the security model, as SGX implements them via hardware microcode and presents a set of CPU instructions to manage the enclaves.

This security monitor is the core of the project and configures the hardware to enforce low-level rules that control the enclaves' access policies. As explained in the Sanctum paper, "*the security monitor checks the system software's allocation decisions for correctness and commits them into the hardware's configuration registers*". One of the examples and the main points of an upgrade compared to SGX is that Sanctum keeps the enclave page tables inside enclaves memory, protecting the system against the timing attacks referred above by keeping the page table dirty and accessed bits private. Their hardware extensions make sure that enclaves page tables only point to enclave memory and untrusted OS tables only point to OS memory regions and never to enclave private memory.

Sanctum is also open to the public which makes easier for security researchers to audit and find vulnerabilities and to further encourage the analysis of the code, Sanctum security monitor is written in portable C++ code and can be used across different CPU implementations.

2.2.7 ARM Trust Zone

ARM Trust Zone [12] is a technology that offers a system-wide approach to security based on hardware-enforced isolation built into the CPU [11]. The principle of the technology is to separate the trusted and untrusted by two virtual processors backed by hardware access control. The two states are referred as worlds, where the first is called the secure world (SW) and the other is the normal world (NW) like figure 2.6 shows.

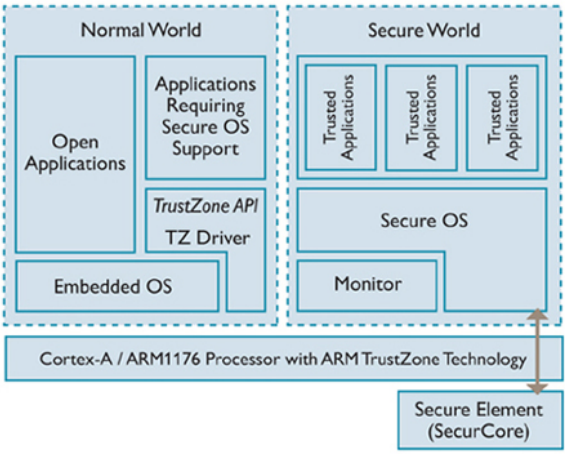


Figure 2.6: Arm TrustZone Stack [13]

The non-secure world (or normal world) is where the OS and most of the software and applications will be running, as for the secure world is where more secure and sensitive software will run and will ensure that vital information is not intercepted by a third party. The security is enforced because each of the worlds acts isolated from the other as a runtime environment with separated resources such as memory, processor, cache, controllers, interrupts. The ARM hardware has separate copies of the registers for each world and cross-world register access is blocked. However, the Secure Monitor shown in figure 2.6 can access non-secure registers while running in the secure world. This means that the monitor can then implement context switching between both worlds.

When in Normal World, the application calls a specific ARM instruction call SMC (Secure Monitor Call) to call back inside the secure world and securely execute the code.

By keeping the worlds separated from each other, the ARM TrustZone can keep applications running in secure mode isolated from the normal world applications such as the OS and thus achieving another implementation of a TEE.

2.2.8 Discussion

The TCB (Trusted Computing Base) is the set computer components (hardware), software and data that we need to trust and deem as not malicious in order to use a system. It's a group of various elements that are critical to a systems security in a way that any bug or vulnerability occurring from inside the TCB components might compromise the security

and privacy of the entire system. On the other hand, security flaws and bugs from outside of the TCB should not become a security issue.

Hardware security like TPMs and TEE technologies have the goal to drastically reduce the TCB of a system, for example, remove the OS from the trusted base to make sure that a remote compromised machine vulnerability does not affect the security and privacy of an application or system.

By isolating the program from outside and uncontrollable sources like cloud infrastructures OSs, hypervisors and hardware we can more safely deploy sensitive applications to those cloud providers. That is the goal of all different implementations of trusted execution environments like the Intel SGX and the ARM TrustZone.

Although not free of some problems, SGX implementation of a TEE seems to be the most accepted technology, with serious and skilled developers and security researches always working to mitigate any vulnerability in order to create a truly trusted and isolated environment.

2.3 TEE/SGX Enabled Key Value Stores

There has been an increasing trend from developers to move their applications to the cloud. It provides dynamically and almost seamlessly scaling with predict cost. Although it also means that users need to rely on the cloud providers for securing and maintaining the integrity of their applications. That means the user must trust not only the provider's staff but also its globally distributed software and hardware not to expose their private data. Today's cloud providers only aim to protect their privileged code from the untrusted code (the user's code) and do not provide any guarantees about the opposite scenario.

To mitigate this use case, and after studying and discussing the Key-Value store technologies and also the trusted platform modules as well as the trusted execution environments, in this chapter, it will be presented how are this two topics being combined and used together.

It will be more focused on the Intel SGX platform as it is the one that will be used throughout this thesis. Currently, several databases that leverage this technology to provide a more secure environment and service. In this chapter, it's presented how they work and operate, discussed the differences between them and also the how the work planned to be performed on this thesis will solve some of the problems and caveats.

2.3.1 Trusted Execution with Intel SGX

As explained before, Intel SGX provides a trusted execution environment by running code inside the enclaves. It creates an isolated environment where we can run some instructions as securely as possible, without OS intervention.

Key-Value Stores and other database type systems can leverage this secure and isolated environment to perform queries on very sensitive data that would otherwise be vulnerable to some attacks. There are a few techniques currently implemented to use isolated environments. Maintaining an encrypted database and using enclaves cryptographic capabilities to decrypt data and perform queries on plain text with the assurance of no data leaking is a possible use case. Also, maintaining a database fully on enclave memory, where it cannot be accessed by anyone other than the CPU is another way to keep the data secure by leveraging isolated and trusted execution environments. Different techniques will be furthermore discussed below.

As we can see, isolated and trusted execution environments are an important feature when it comes to protecting the data from the OS and Key-Values Store systems do benefit from them.

2.3.2 Circumvention of SGX Limitations

There are a few limitations and challenges of the SGX platform that we address when programming for such technology.

It starts with a big challenge of choosing and defining what parts of the program can benefit from the SGX security. As it is known, it works with two major application components, the trusted and untrusted modules of or program. The limitations have to be thoroughly analysed so we can make that definition.

The main limitations are:

- Performance
- Memory
- I/O
- syscalls

In the KVS world, as we extensively covered, performance is the major concern and there is no real way around this limitation. Using secure enclaves will decrease the supposed blazing fast performance. Although, with an intelligent partition between the untrusted code, which will be fast, and the trusted instructions, which can introduce overhead, we can limit the performance decrease. By separating and clearly defining both modules of the application, we can decrease the code that needs to run securely and find a fine compromised between security and performance. Homomorphic encryption is also a mechanism to speed up performance by performing queries directly over encrypted memory. Although fully homomorphic encryption is not a possibility yet, so, by compromising some set of operations that are not possible, partial homomorphic encryption will speed up the performance.

With the SGX base support the access to EPC from the owner enclave is efficiently processed by hardware en/decryption logics at cache-line granularity. This means that

when the cache-line is brought from EPC to the processor, it is decrypted. The hardware logic calculates the keyed hash value of the cache-line and verifies it against the stored hash value of the address. Internally the integrity hash values are organised in data-structures similar to Merkle Trees, to allow sub-trees to be evicted from the on-chip storage securely [45]. Due to space and time overhead of storing and processing security metadata at fine-grained cache-line granularity for EPC, the EPC capacity is unlikely to increase significantly. For example, a huge Merkle tree for tens gigabytes of main memory at cache-line granularity will necessarily increase the integrity verification latency intolerably, as an inefficient solution that will sacrifice throughput and latency [93].

Memory sizing is also a limitation when using enclaves in SGX technology. The amount of private secure data that can be maintained by the enclave is limited to the size of the enclave cache, which is around 128 MB, being that only about 94 MB are available to the application, with the rest reserved to metadata. However, for some operation systems, mainly Linux because of paging swap support, it can be increased up to all the memory available in the system [89] by swapping pages from the EPC to main untrusted memory, with guaranteed of confidentiality, integrity and data freshness. When evicting pages from the EPC, it is assigned a unique version number which is recorded in a new type of EPC page and the contents of the page, metadata, and EPCM information are encrypted and written out to system memory and to prevent address transaction attacks, the eviction protocol interrupts all enclave threads and requires a flush to the translation lookaside buffer (TLB) [1] [14]. When reloading a page back into EPC the page is decrypted and has its version and integrity checked to make sure it was not tampered with.

Although, page eviction to main untrusted memory introduces a big overhead because of encryption and decryption and integrity checks (2x - 2000x) [14]. Clever partitioning of the application into the untrusted and trusted modules will reduce the enclave memory necessary, reduce the number of system calls and page evictions and help to overcome this limitation, as described in the next sections. Another issue is EPC fault handling because EPC limit requires exiting the enclave improving more the cost of paging.

I/O and syscalls are limited by default on the enclave for security purposes, so it can't affect or be affected by the OS. There is a way to perform and access I/O and syscalls through the aforementioned Ecalls and Ocalls (section 2.2.5 of this thesis), but they have to be accounted for when implementing the application. To address the problem, recently proposed solutions try to reduce the frequencies of enclave exits for system calls by running threads in untrusted execution for the interaction with the operating system, communicating with the enclave thread by sharing memory. Also, Ecalls and Ocalls require exiting and entering the enclave and that caries a big performance overhead, as encryption and decryption cycles must be done to maintain security guarantees, as well as integrity checks.

[1] "A TLB is part of the chip's memory-management unit (MMU), and is simply a hardware cache of popular virtual-to-physical address translations; thus, a better name would be an address-translation cache"[16]

2.3.3 SGX-Enabled Secure Databases

Database management service developers are now implementing secure databases ready to take advantage of Intel SGX hardware. It differs from normal databases because it runs on top of protected and encrypted memory so it can work with minimal TCB.

Next subsections present and discuss the overview of the current technology that leverages SGX to provide a secure database.

2.3.3.1 EnclaveDB

EnclaveDB [76] is a privacy-enhanced and secure database that works alongside with Intel SGX and provides a Structured Query Language (SQL). It uses its technology to maintain **all** sensitive information inside SGX enclaves in order to keep them secure from a threat model of strong adversaries that can control the entire software stack on the database server. It resists attack from the administrator server, the database administrator and attacker who may compromise the operating system, the hypervisor or the database server.

Following Intel's application guidelines, EnclaveDB has a two-part architecture: trusted (running on the enclave) and untrusted modules. The enclave hosts a query processing engine, natively compiled stored procedures and a trusted kernel which provides API's for sealing and remote attestation. The untrusted host process runs all other components of the database server. Figure 2.7 shows the architecture of the enclaveDB server-side.

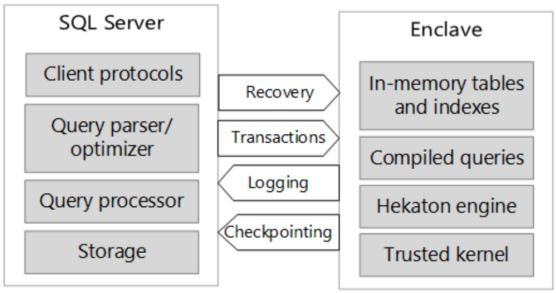


Figure 2.7: Server-side components of EnclaveDB

Leveraging TEE, EnclaveDB then provides a database with a SQL interface and guarantees confidentiality and integrity with low overhead. With its design, it also reduces the TCB to a smaller set than any other "normal"database.

2.3.3.2 Pesos DB

Pesos [62] is a secure implementation of object storage services like Amazon S3 [7], Azure Blob Storage [22], Google Cloud Storage [40] among others. In these current large-scale

services, due to their complexity, the risk of confidentiality and integrity violations increase significantly. These storage systems are characterised by multiple layers of software and hardware stacked together which means the access policies for ensuring confidentiality and integrity are scattered across different code paths and configurations, thus exposing the data to more security vulnerabilities. Furthermore, untrusted third-party cloud platforms expose an additional risk of unauthorised data access by a malicious administrator.

Pesos allows clients to specify per-object security policies concisely and separately from the remaining storage stack. It also provides cryptographic attestation for the stored objects and their associated policies to verify policy enforcement mechanisms.

It enforces these policies by leveraging the Intel SGX for trusted execution environments and Kinetic Object Storage [61] for trusted storage (secure persistent storage - not the focus of this thesis). It structures a policy-compiler, its binary-format interpreter, per-object policy metadata, and the enforcement logic into a single layer of the storage stack. With this unification, it drastically reduces the TCB when compared to the order cloud services. Then it uses the trusted execution environment provided by SGX to connect directly Kinetic disk through an encrypted Ethernet connection allowing for object transfer and policy enforcement securely without any intermediate layers in the storage stack.

2.3.3.3 Speicher

Speicher [23] is a secure LSM-based (Log-Structured Merge Tree) Key-Value store that uses Intel SGX and it ensures not only strong confidentiality and integrity properties, but also data freshness to protect against rollback/forking attacks. It leverages SGX technology to achieve those security characteristics focusing on providing a **persistent** service, tolerant to system faults and securely recovering from crashes. It also tackles in interesting ways, two of the major limitations of SGX: Memory Limits and Performance.

Implementing a Key-Value Store has a major requirement - High performance and low latency queries for big data structures. As already discussed, SGX has some memory limits. The enclave memory is located in the Enclave Page Cache (EPC) which is limited to 128 MB with about 94 MB available for application use (the rest being reserved for metadata). To allow the creation of enclaves with bigger size than EPC, the OS can use secure paging mechanism where it evicts pages to untrusted memory. Although with page encryption, decryption and integrity checks, this solution introduces high overheads (2× - 2000×) [14].

To address this performance and memory problems, the developers of Speicher implemented the following custom features (from Speicher public paper):

- *"**I/O library for shielded execution**: Direct I/O library for shielded execution. The I/O library performs the I/O operations without exiting the secure enclave; thus it avoids expensive system calls on the data path."*

- *"**Asynchronous trusted monotonic counter**: Trusted counters to ensure data freshness. The counters leverage the lag in the sync operations in modern KVS to asynchronously update the counters. Thus, they overcome the limitations of the native SGX counters."*

- *"**Secure LSM data structure**: Secure LSM data structure that resides outside of the enclave memory while ensuring the integrity, confidentiality and freshness of the data. Thus, the LSM data structure overcomes the memory and I/O limitations of Intel SGX."*

The technology leverages SGX with a clever partition between trusted and untrusted modules of the application. By maintaining the encrypted data on untrusted memory hardware it addresses the memory and persistent limitations, and by keeping some information in secure enclave memory and with a good I/O library it overcomes (to an extent) the performance issues.

2.3.3.4 ShieldStore

ShieldStore [60] is a *"(...) shielded in-memory Key-Value Storage with SGX"*. It aims to provide a very fast and low latency queries over very large data trying to overcome the SGX memory limitation. It accomplice's it by maintaining the majority of the data structures in the non-enclave memory region, addressing as well the performance issue by not relying on the page-oriented enclave memory extension provided by SGX.

ShieldStore runs server-side in the enclave to protect encryption keys and for remote attestation and it is used to perform all the KVS logic. It uses a hashed index structured but places it in the unprotected memory region instead of the enclave EPC. As the main data structure is not protected by the SGX hardware, each data entry must be encrypted by ShieldStore in the enclave and written to the main hash table.

The main flow and architecture are as described in figure 2.8.

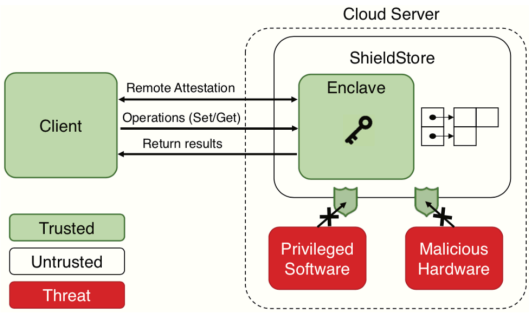


Figure 2.8: Overview of ShieldStore

First, the client remote attests the server-side (1) verifying SGX support of the processor, the code, and other critical memory states of an enclave. In a second step, the client and the server exchange sessions keys (2) in order to establish a secure connection, using

Intel SGX libraries to do so. Using this newly generated session key, the client sends a request for an operation (3). The server deciphers and verifies the request and accesses the Key-Value Store (4). Clients do not access the server-side ciphertexts neither need to know the encryption key used by the server to encrypt the values. The server will then decrypt the data from the storage, encrypted it again with the session key and reply to the client (5). All accesses to the KVS have integrity checks.

2.3.4 Discussion

Concluding, EnclaveDB (section 2.3.3.1) and Pesos (section 2.3.3.2) presents secure databases and objects storage systems respectively, using SGX, but EnclaveDB assumes that SGX supports large enclaves whose size is an order of several hundred GBs and Pesos restricts the size of data structure to the size of EPC. On the other hand, Speicher (section 2.3.3.3) and ShieldStore (section 2.3.3.4) proposes a store that alleviates the memory limitation of Intel SGX by storing encrypted data on untrusted memory regions. Speicher and ShieldStore have similar architectures, but the former is primarily designed for persistence storage and the latter is focused on a fast in-memory key-value store.

We can now conclude that the clever partitioning of the application into trusted and untrusted parts is really important when programming with Intel SGX. It directly affects syscalls and I/O, performance and memory of the service.

The long-term goal of this thesis is to implement a system with characteristics from the databases present above. In terms of performance, we implemented partial homomorphic encryption, so it allows to perform operations directly over in-memory encrypted data. This will be a challenge, as fully homomorphic encryption is not yet practical [39], so adaptations must be made, but performance increases are expected over the databases presented above, by not needing to decrypt the data in secure execution. Persistence will also be a requirement just like some of the databases presented.

For trusted execution with SGX, extensive research is needed to partition the application in the two necessary modes to circumvent persistence, performance and memory SGX limitations. It will also be researched and tested the ability to provide built-in replication and availability with SGX.

2.4 SGX Virtualisation Frameworks

Virtualisation is the mechanism that relies on software to simulate hardware components and with the objective of creating a simulated ("virtual") computer system. It means that multiple virtual machines can run inside the same physical machine, and although sharing psychical resources between them - CPU, memory, disks, run completely separated from one another.

With the broader adoption of virtualisation and SGX over the years, developers are working on ways to leverage virtualisation with these secure enclave technologies.

Also, container-based virtualisation has become very popular in the last few years. Containers can use the capabilities of a secure and trusted environment to achieve a fast, highly portable, faster to deploy, and small SGX environment that can run mainly unmodified applications without a big performance overhead.

2.4.1 KVM-SGX

"KVM (for Kernel-based Virtual Machine) is a full virtualisation solution for Linux on x86 hardware containing virtualisation extensions (Intel VT or AMD-V). It consists of a loadable kernel module, kvm.ko, that provides the core virtualisation infrastructure and a processor specific module, kvm-intel.ko or kvm-amd.ko" [57].

The KVM framework allows the running of multiple virtual machines with unmodified Linux or Windows images. Each VM will be assigned their own virtual private resources like the network cards, disks graphic adapters, etc...

The KVM-SGX project is a module KVM/Linux module to support SGX virtualisation on KVM. It exposes a private virtual SGX that can be used by the guest OS. To fully virtualise SGX, EPC must be assigned to the guest OS. Although, since the EPC is a system resource, the management of this physical EPC - that backs the virtual EPC, its owned by the kernel. The kernel exposes a device /dev/sgx/virt_epc that allows the guest VM to make a system call that creates and assigns its own virtual EPC section.

To recap, the guest VM does not have to deal with PRM or even EPCM because those resources are sitting below it on the hardware. The guest has a virtual Enclave Page Cache managed by the hardware that it can use at will to store secure code and data.

The Linux SGX KVM module is still in its development phase but can already be used to virtualise SGX platforms onto virtual machines [26, 51, 91].

2.4.2 Graphene-SGX

Graphene [43, 99, 100] is a guest OS design to run a single unmodified application with minimal host requirements. It bridges the gap of portability by abstracting the OS making it easier to port applications to different OSs. Like a VM, Graphene provides an isolated environment for the application to run in, and, by not running a complete operating system it has a much lighter footprint, and generally low memory requirements and performance overheads. Also, developers are working to supply graphene as a containerised application [44].

As an extension, Graphene is working to provide integration with Intel's SGX processors where applications could run unmodified with the additional security guarantees of the secure enclave technology [99].

Since it is a library OS, Graphene takes advantage of its lightweight form factor to serve as a compatibility layer between the hardware and the application where normally, they do not work out of the box, with small performance overheads high portability and minimal or even no code modifications.

2.4.3 SCONE

The SCONE [14] (Secure CONtainer Environment) platform facilitates always encrypted execution using Intel's SGX secure enclaves to run encrypted code not even accessible to system administrators or root users. It supports this kind of execution running inside Docker containers and even managed Kubernetes clusters.

Although containers can have a big TCB and weaker isolation, the design of SCONE leads to a small TCB by exposing a small C standard library that allows for system call to execute outside the enclave but maintaining security guarantees by seamlessly and transparently encrypting/decrypting application data. Files stored outside of the enclave are therefore encrypted, and network communication is protected by TLS. SCONE also implements a *user-level* threading that maximises the time that the thread spends inside the enclave. Combined with the asynchronous system where OS threads outside the enclave execute system calls, it minimises a big enclave overhead - **enclave entry/exit**.

It also provides seamless and transparent attestation of enclaves, to make sure applications run inside secure enclaves and also, that the process running is secure and correct. Combined with the attestation feature, it also provides a secret management API that can store or even generate secrets (encryption keys, TLS certificates, passwords and others) keeping them secure, also stored inside enclaves, and invisible to any human, and guarantees that only the correct enclave can access them.

SCONE also ensures better compatibility with Linux than library OSs and is hardware independent meaning that their implementation can easily be portable to another TEE without any user application code changes.

SCONE is a tool that helps to run **unmodified** applications by abstracting the TEE technology behind and helps to better integrate a user's custom application to a secure environment without application changes, guaranteeing an extra layer of security with a small performance overhead.

2.4.4 Asylo

Asylo [17] is an open and flexible framework for developing enclave secured applications. Developed by Google, Asylo provides [18]:

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

Asylo does not lock their technology with Intel's SGX but instead leaves it open to multiple secure enclave frameworks. In Asylo, the majority of user-developed logic lives inside the enclave. However, due to security and portability reasons, this framework does not support direct interactions between the enclave and the OS. All of the enclave-to-OS interactions must be mediated through code that runs on the outside of the enclave but Asylo provides most of the code for creating, exiting and interacting with the enclave and the OS.

2.4.5 Discussion

Virtualisation was a game-changing technology when it was first created and, to this day, it is still evolving and it has become the core of cloud computing. With each iteration, more and more components can be virtualised and simulated to be used freely by their guest OSs and virtualised SGX's processors are starting to appear. The KVM SGX module and Graphene are more low-level types of virtualisation that can use SGX to provide an even more isolated an secure environment for the application to run. However, with its fast shipping and deployment and highly portable applications, containerisation follows the same path, as it becomes more and more popular to developers and SCONE and the Asylo frameworks work beautifully to combine the power of a secure enclave with the characteristics of a container-based application.

In this thesis, we will use SCONE technology to run an unmodified Redis system secured with Intel's latest Software Guard Extensions (SGX) to expose a seamless secure application to the client with fast deployment speeds and high portability and scalability.

2.5 Related Work Balance and Critical Analysis

In the current days, computer scientists are always looking for a secure, fast and cheap environment to develop applications. As we know, it is not feasible to have all three of these elements working flawlessly without any compromises. However, by combining in-memory key-value stores, trusted remote execution environments and cloud providers, developers can now have a practical example of what would be to develop for a privacy-enhanced secure system with reduced costs by using cloud providers and with better reassurances that the hardware and software that it's out of the control of the user will have a minimal impact on sensitive data and code of an application. By adding the performance benefits of an in-memory key-value store and all of its technology, like built-in security, built-in replication and persistence we can in the best of our abilities today, combine the best of the three worlds without compromising too much on any of them.

In this thesis, we will compare different kinds of approaches to implement a fast system with the assurance of a secure data flow that can easily be deployed into the cloud without fear of any components out of our control.

Chapter 3

System Model and Design Options

In this chapter, we provide an overview of the system model and implemented architecture along with all of its components. It is explained how all components interact with each other and each component purpose and how they work.

It is also explained how the different security features are implemented into the system in order to provide a secure overall system and achieve the contributions planned for this thesis.

Section 3.1 provides a small recap and refinement of the objectives and contributions of this thesis and the used TEE technology.

In section 3.2 we describe the basic security assumptions need for our system to work securely. The threat model is a very important part of any security-related project as it provides a clear overview of what attacks the system is able to protect against, presents the trustability chain and TCB as well as what components and security properties are out of scoped and not addressed in the project. Sections 3.3 and 3.4 presents an high level view of the system model and implemented architecture of the system. Then, in section 3.5 and 3.6 we expose the application-specific operations supported both key-value store specific operations and custom operations implemented by the proxy and also an interaction flow that represents a user interaction with the system.

Finally, as always, section 3.7 we summarise all the findings and provide a clear transition into chapter 4, which will present all implementation-specific details performed to achieve the described system.

3.1 Refinement of Objectives and Contributions

The goal of this dissertation as explained on chapter 1, is the design, development and validation with experimental evaluation of a secure in-memory storage (based on a "key-value"model), supported by a hardware-enabled trust computing base.

Regarding the security assumptions, the solution will provide: (i) hardware-isolated in-memory processing engine, designed as a hardware-isolated container facility, enclaved within the Intel-SGX protection guarantees; (ii) hardware-isolated communication endpoints for client access, providing TLS tunnelling with strong TLS 1.3 endpoint encryption parameterisations and support for mutual client/server authentication, and (iii) privacy-enhanced operations to be directly processed on encrypted data sets in memory. The former facility is particularly interesting to combine the possibility to manage protected memory for small data sets and also searchable encrypted data sets that are far larger than the protected memory limits imposed by the SGX memory mapping facility. Furthermore, the solution will target main data structures commonly use fine-grained data items that can include pointers, complex composite types and keys, which do not match well with the coarse-grained paging of the SGX memory extension technique.

In the next subsections, we will align the implementation ideas starting by refining the circumvention of limitations in SGX and the threat model assumptions to address our solution.

3.1.1 SGX Limitations Refinement

Section 2.3.2 describes in detail the limitations of SGX. In our approach we will intend to design a solution that can be leveraged from conventional reference KVS technology, giving the possibility to manage small datasets but also larger datasets (directly mapped in non-protected memory). To circumvent the problem our solution must combine the possibility to use the internal capabilities native to SGX with the possibility of supporting operations managing datasets encrypted in memory, with such operations executed directly over encrypted data. This facility will be provided by the use of partial homomorphic encryption constructions, with data initially encrypted and submitted to the key-value-store solution with cryptographic keys only managed in the client-side. Our solution will be designed in order to be possible the support for fine-grained key-value encryption, driven form the application requirements. Our target is the support of a variety of operations provided in a typical KVS API, taking REDIS as the reference solution and to support a considerable number of queries currently used by many REDIS-supported applications.

3.2 Threat Model and Security Properties

The threat model and security properties definition describe the conditions of how we define a secure system. However, to ensure a basic secure system, we must achieve a few

key goals and objectives:

Data Privacy - Data must remain private to its owner.

Data Integrity - Data must not be compromised, modified or corrupted.

Authenticity - Data and system interactions should be authentic and not spoofed by unauthorised users.

Subsections 3.2.1, 3.2.2 and 3.2.3 explain under which conditions and assumptions the system will achieve these three main parameters and implement a secure system.

3.2.1 Adversarial Model Definition

As the baseline, our threat model will lie on the protection overview stated in SGX's paper [66]: *SGX prevents all other software from accessing the code and data located inside an enclave including system software and access from other enclaves. Attempts to modify an enclave's contents are detected and either prevented or execution is aborted*, which falls in the following adversary model:

Isolation by trusted containerisation from malicious code: The system performs and protects its data from an attacker capable of compromising the system through another application installed on the same system or malicious code existent or injected in the OS or OS hypervisor layers;

Privacy protection against insider "Honest but Curious"System Administrators: The system must be able to protect from an attacker with root access to the machine, with permissions to access and monitor memory-mapped data. This is relevant because we will target our solution as a candidate solution to protect data privacy in a cloud-based key-value store solution as a service, preventing data-leakage vulnerabilities exploitable by insider incorrect users or system-administrators.

Network Attacks: All communication to and from the system (supporting client-service operations) should be secure, using proper strong cryptographic parameterisations for TLS 1.3, mutually authenticated handshakes and TLS endpoint executions isolated in SGX-enabled TLS tunnels in communication containers, avoiding attacks against the authentication of the service endpoints, as well as, attacks against the integrity and confidentiality of data flows supporting REST/HTTPS operations.

File system and memory access attacks: All sensitive data residing outside protected memory should be encrypted and operated in the encrypted form. An attacker can access the physical disks and hardware without the sensitive data being exposed.

3.2.2 System Assumptions

With the above security baseline considered for the threat model assumptions, the solution must be resilient to malicious privileged attacks and certain physical attacks. With a controlled and reduced hardware-shielded trust computing base, we want to design a solution that does not rely on the security of the operating system managed by cloud

providers. Furthermore, our solution must be also resilient to direct conventional physical attacks, such as cold boot attacks, which attempt to retain the DRAM data by freezing the memory chip or even bus probing to sense and to read exposed memory channel between the processor and memory chips. The only weaknesses not covered in our concerns with the SGX platform is the lack of protection for side-channel attacks.

The system planned has certain assumptions and aspects that are considered to be out of scope for this dissertation:

- **Trusted Client** - The client side is assumed to be completely trusted and correct.
- **DoS and DDoS** attacks are out of scope.
- **Side Channel Attacks** - It is out of scope any side channel attacks or any related attack not present in SGX's threat model.
- **Physical and Hardware attacks** exploring the SGX processing model and its isolation guarantees are out of scope, namely those presented above initially addressed in chapter 2, section 2.2.

3.2.3 Countermeasures for Privacy-Preservation

Under the described threat model and system assumptions, the system has implemented and deployed several measures to achieve the desired security and trustability level.

All instances of the datastore are running in a containerised solution which means that each container is not only isolated from the host but they are also isolated from each other. Data is kept in memory at all times unless persistent disk storage is turned on. Data in main memory is secure when running both in protected and unprotected mode.

Privacy, integrity and authenticity when running in protected mode are always ensured by the SGX's technology under their threat model and assumptions, but when the storage server is running in unprotected memory, outside the trusted execution environment, the system will always keep data encrypted with strong and standard state of the art cryptography algorithms, therefore preserving the privacy of data.

Although data might be private, the preservation of the integrity of that data is also very important and to assure data integrity, all values are appended with standard checksums calculations and integrity check algorithms. Authenticity is preserved by performing strong and standard cryptographic digital signature algorithms.

When it comes to the overall security of the exposed application, the system provides an API that only allows authenticated requests and it contains a role-based segmentation authorisation with secure and strong passwords and cryptographic keys. This system also applies the principle of least privilege [65] to all actions, where users never have more privileges than they require.

The same security principals must also be guaranteed on an in-transit level and so, all communications are secured with the use of the strongest transport layer security algorithms currently available and established trust between all components with a trusted certificate chain.

Security by Design means that security is the foundation of a project, and its tactics and patterns should be used as guideline principles for developers, baked into the project initial state, and enforced by the architecture design.

The measures adopted to achieve data privacy, integrity and authenticity are a standard that the whole project followed since the beginning of the implementation, meaning that the security by design principle was adopted, and the whole architecture was shaped around security in order to achieve a secure and correct system.

3.3 System Model

The main goal of this project is to implement a system that follows the model presented in figure3.1. As shown, the system can be divided in four main components - the **client**, the **proxy server**, the **key-value storage server** and the **authentication server**.

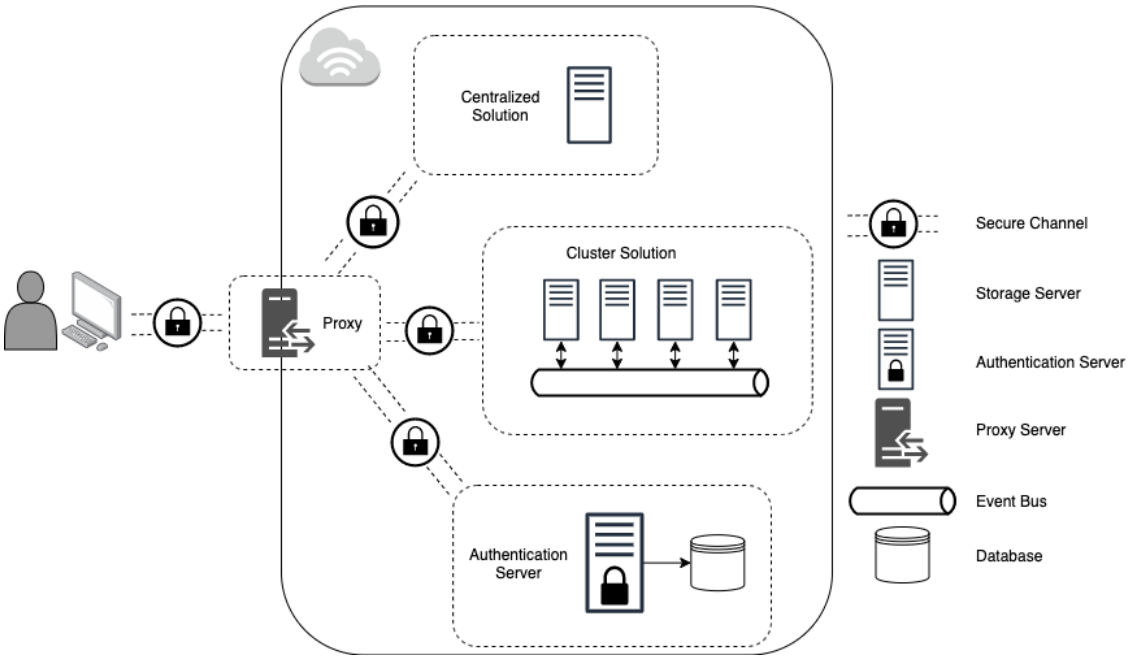


Figure 3.1: System Model Overview

The system complies with the adversary and threat model explained in section 3.2.1. The user is not aware of any implementation details and has seamless interaction with the system despite the architecture and implementation provided by the backend, meaning that all solution exposes the same API and support an interface as equal as possible to the unprotected version of the key-value store.

All of the four main components will be explained in the subsections that follow.

3.3.1 Key-Value Storage Server

The storage server is a key-value store meant to hold data required by the user. This data is kept in memory so it allows fast read, writes, updates and deletes.

To circumvent the SGX memory limitation, the dataset will be split into two different configurations. The **secured** and **unsecure** and the system model of the storage server is shown in figure 3.2. This nomenclature does not reflect the privacy, integrity and authenticity security properties as they are preserved on both components, but it reflects the environment which they will run on.

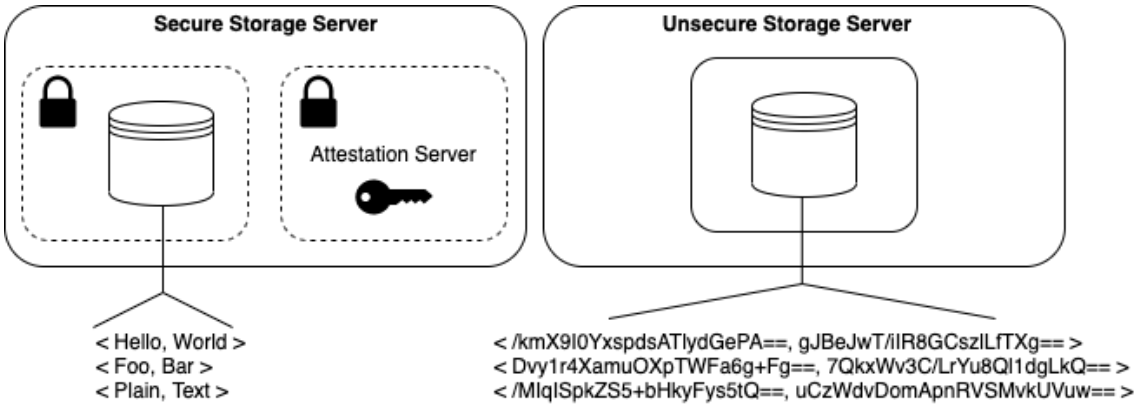


Figure 3.2: Storage Server Model

The **secure** storage runs on a trusted execution environment (trusted hardware) protected by a secure physical processor present on the host machine. This processor provides the security features that allow for memory to be operated in plain text without breaking privacy or data security as explained in section 2.2.

As for the **unsecure** storage, it describes a storage service that runs on unprotected memory regions and untrusted hardware. This means that data must be actively secured by encryption, integrity checks and authentication features in order to provide and preserve privacy.

We must mention that the above system model addresses the replication facilities as leveraged from the Redis solution and tries to offer the same availability conditions as the original Redis service. Both secure and unsecure configurations will provide availability by running replicated Redis instances in a cluster. This availability provided by the replication of Redis instances that can run as cloud service instances (that can be distributed among different cloud providers and geo-replicated data centers) only offers consistency guarantees under a fail-stop model (not extended to byzantine fault tolerance or byzantine intrusion tolerance, that are conditions out of the scope of our planned dissertation).

Every Redis instance running on a secure configuration will also provide an API for remote attestation, going forward known as the attestation service, so their hardware stack and software can be attested by remote parties, to provide trust to the users.

3.3.2 Proxy Server

The proxy server serves as a gateway to the storage instances. It is a single point of entry to the system which as some advantages. Not only it provides an abstraction to a backend which can be replicated across multiple instances across multiple geo-locations, but it also can add additional features to the system.

With a centralised gateway, we have centralised access management with authentication and authorisation features. Not only that, but the connection to multiple instances of secure data storages with different authentication mechanisms and secrets can also be centralised in one place, which allows the user to have just a single username/password pair to connect to all the storage instances.

The proxy server manages security properties when connecting to instances on an unsecured storage configuration. It is the proxy that handles security features like encryption and decryption, integrity and authentication checks, and provides a completely encrypted and private storage service. Trying to soften the performance overhead of encryption and decryption, the proxy enables operations directly into encrypted data, using homomorphic ciphers.

It also exports a REST API which can be used by multiple clients with different implementations and also offers custom features that the standard data storage system does not. Custom operations will be explained below, on section 3.5.

The proxy server serves as a single point of access for the multiple storage instances remote attestation features. It attests every instance and feedbacks the user on the hardware stack and software state of each instance. Furthermore, the proxy server itself runs on a trusted execution environment leveraging secure trusted hardware and can also be attested by the clients.

3.3.3 Authentication Server

An external authentication server is required to centralised user management. It is responsible for user authentication and verification. A user authenticates against the authentication server receiving an access token, that it provides to the proxy on every request. The proxy can verify the token and check if it is valid and what permissions this user has, and can authorise or reject the access to a particular endpoint or system functionality.

An external authentication server relieves the proxy of user authentication and user management and outsources it to a standardised open-source system that implements security standards of identity and access management. This is also important so that the proxy server can, if needed, be replicated.

3.3.4 Client

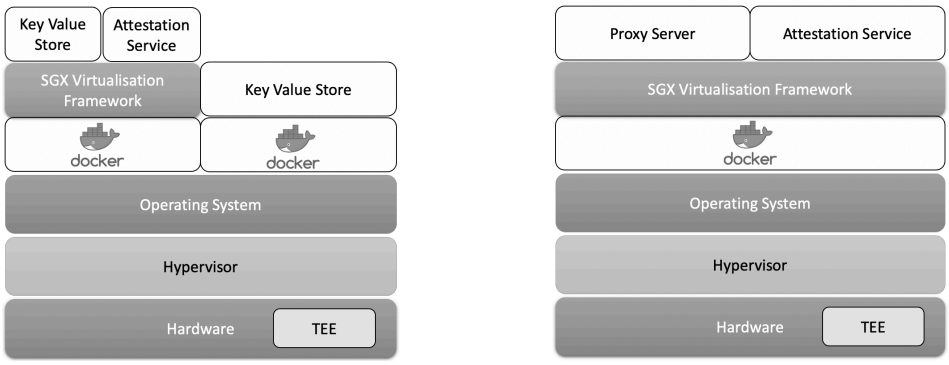
The client, or user, is the one that will consume the exposed APIs by the proxy server. In the case of this project, the client will be a representative benchmark tester that will use the exposed endpoints to record benchmark times and various relevant evaluation criteria so all storage configurations and replication mechanism can be compared.

Although just a simulated client/tester, the APIs are consumed the same way a real user would consume them, so, benchmarks are a representation of real system usage.

3.4 System Architecture

Figure 3.3 describes the hardware and software representative stack of the infrastructure [1]. Figure 3.3a shows how the storage service is deployed onto the cloud provider's machines. On the machine's hardware, it is provided a very specific and physical processor that implements a trusted execution environment. Running on the operating system, a containerisation solution runs the TEE virtualisation framework in order for the container running both the key-value stores and the attestation services being able to access the TEE. On the right-hand side of this figure, we can see a storage service that runs with no TEE virtualisation system, and that is the deployment of an unprotected key-value store configuration.

On figure 3.3b the stack provided describes a system also running in the confinements of a trusted execution environment and therefore, also providing an attestation service.



(a) Storage Stack (b) Proxy Stack

Figure 3.3: System Architecture Stack

[1] The described stack is a representative stack of the machine layers and it does not represent a true overhead ladder

3.5 Supported Operations

The exposed API can perform some operations over the storage system. Some operations are out of the box key-value stores operations, and some were implemented in the proxy so to give a customised and more complex operations.

The next subsections will iterate and explain the supported operations, and we need to indicate that all operations listed can work on both main data storage configurations (protected and unprotected), where operations are performed whether data is maintained on clear text, securely inside a trusted execution environment or maintained encrypted in unprotected memory regions.

3.5.1 Role-Based Authorisation

The system by default rejects all unauthenticated requests, which means, users need to be registered and have credentials to access the system.

As explained before, user management is performed by the external authentication server, and each user has a role assigned. Different roles can do different actions on the system and role-based authorisation can be completely customised.

In this thesis, we operate with the principle of least privileges and came up with two different roles: a **BasicUser** which can only perform read operations and cannot alter the system state (basically a read-only user) and an **Administrator** which can perform all actions on the system. These roles are just a representation of the role-based authorisation that the system performs before accepting a request into the server and can be further configured (or new roles added) for a finer-grained authorisation mechanism.

3.5.2 Key-Value Storage Operations

The system is ready to perform a set of operations from the key-value storage of choosing, without modification. However, this is not a full-fledge solution, and accommodations had to be made for the two main configurations of the storage system: protected and unprotected.

The supported operations are as follow:

- **Set** - Stores a value into the database associated with a key.
- **Get** - Retrieves a value associated with the provided key.
- **Set on List** - Creates a list of values associated with a key.
- **Get List** - Retrieves all values associated with the given key.
- **Set on List with Score** - Creates a list of values associated with a key. Each value has associated a score (integer)
- **Get on List between Score** - Retrieves all values associated with the given key with scores between provided scores.

3.5.3 Proxy Enabled Operations

The proxy server implementation allows for the implementation of another set of operations not supported out of the box from the key-value store:

Sum, given a key and a number, the server can fetch the value associated with a given key and add it to the number provided.

Subtraction, given a key and a number, the server can fetch the value associated with given key and subtract it to the number provided.

Multiplication, given a key and a number, the server can fetch the value associated with given key and multiply it to the number provided.

Search on List, where the server takes a search term, match it against each value of a provided key of a list and return only the matching values.

3.5.4 Attestation

Attestation is the method provided by the processor implementing a trusted execution environment to establish trustability assumptions to a remote challenger. It proves to a third party that the system is running the correct application with the correct system configurations in the expected physical hardware. This thesis implements two different kinds of attestation: An hardware and TEE attestation and an on-demand software and system stack attestation.

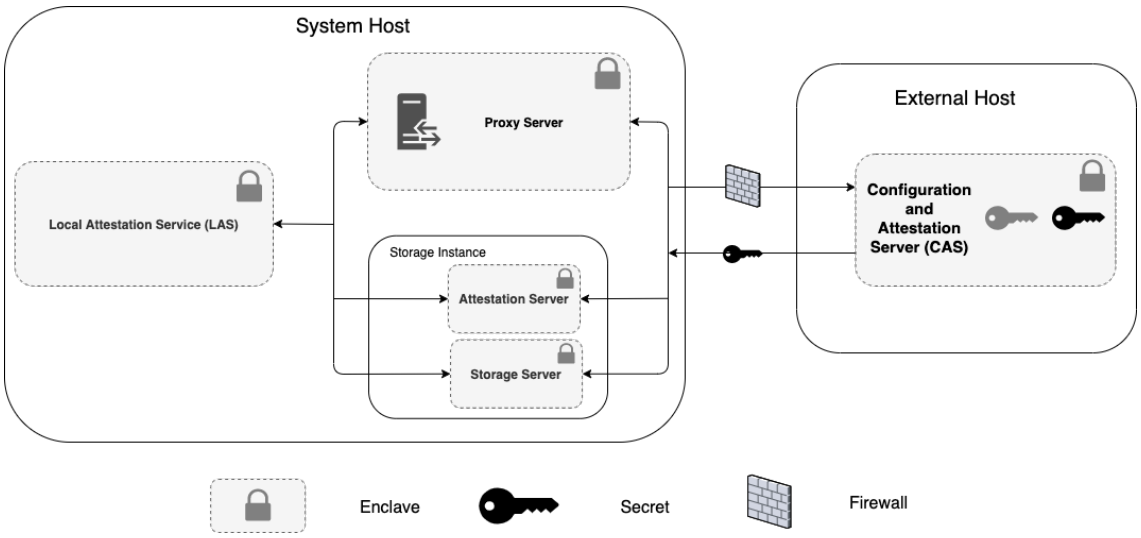


Figure 3.4: Attestation Model

Figure 3.4 describes the TEE hardware attestation infrastructure. It happens at application startup and is the attestation that relies on the secure processor and it requires an external Configuration and Attestation Server (CAS) that holds some kind of secret necessary for the application startup. However, the CAS will only allow and provide the secret if the application enclave can prove itself to be correct, i.e is running the correct source code in the correct system. The application contacts a Local Attestation Server

(LAS) to obtain a proof of their correctness and then try to fetch their secrets in the configuration server. If the application is deemed correct, it can access those secrets and can start.

On the other hand, users can also request the remote attestation (although not the attestation specifically provided by the TEE) of the complete system software stack and hardware information at any point on the user's lifecycle. The proxy server is responsible to contact each storage server's attestation service and request attestation quotes. Each quote is then gathered in the proxy, which attests itself and returned to the client.

The correctness of the system is then determined by the client which analyses the quotes provided by each system and decides whether or not the system is correct and protected and can continue normal use.

3.6 Operation Flow

Figure 3.5 shows an example of a flow that can occur between all system components.

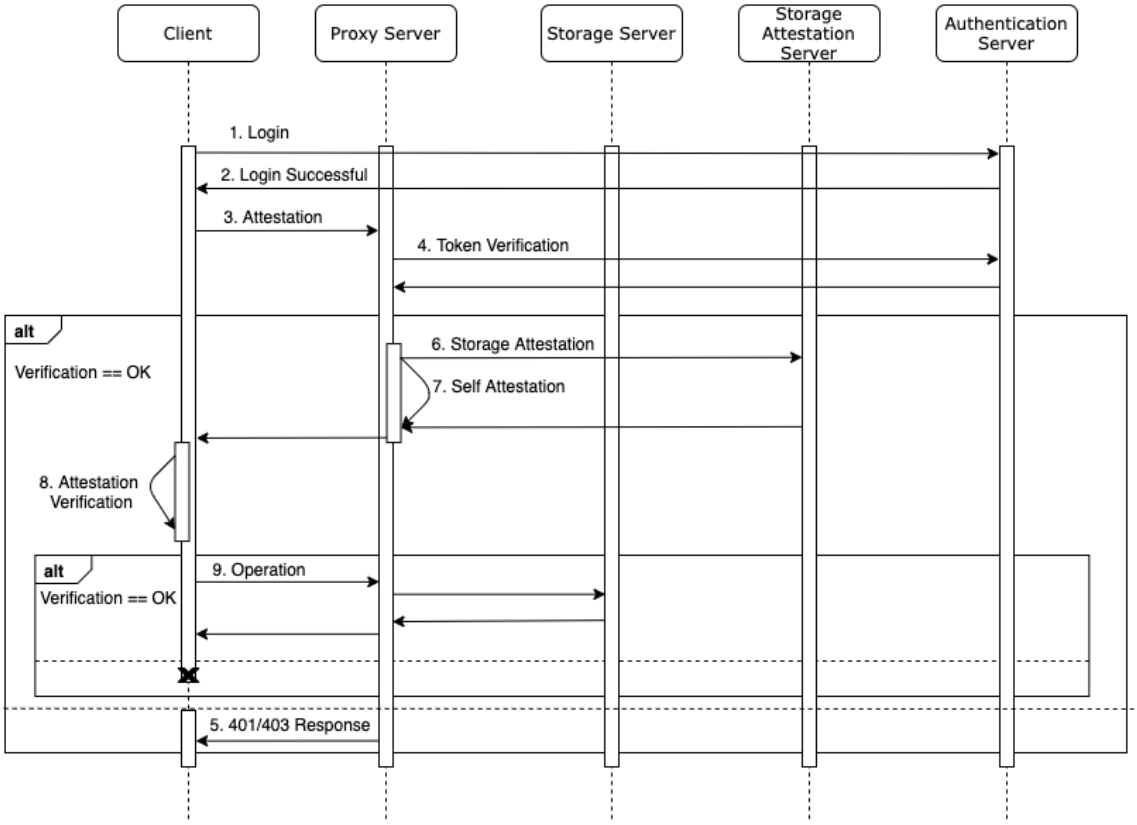


Figure 3.5: Operation Flow

Following the flow, we can see that the first interaction is with the external authentication server by providing the user credentials (**1**) and (if login is successful) receiving an access token in return (**2**).

Then, the user uses the access token to attest the system and make sure that hardware and software stacks are working as expected. It requests an attestation to the proxy (**3**) with the access token retrieved on login. The proxy checks the access token with the authentication server (**4**) and verifies if it is correct and has the necessary roles to access the attestation endpoint. If the verification fails, a proxy returns an authentication/authorisation fail-back to the client (**5**).

If it succeeds, the attestation can proceed and the proxy server contacts all storage server attestation services and gathers all quotes (**6**), and while it's doing that, also starts a self-attestation process (**7**) and returns all quotes to the client.

Now, the client must analyse the quotes from the attestation and decide whether or not it trusts the system to continue (**8**). If it deems the system incorrect, it fails but if the client trusts the system, it can carry on with normal operations to the proxy and to the storage server (**9**).

3.7 Summary

To achieve the objectives and contributions of this thesis, the system should rely on a physical hardware processor that can implement an isolated execution environment, to protect the system against insider threats, operating system and hypervisors vulnerabilities, and several other attacks already explained in the adversary and threat model of this project. That processor, as shown on system architecture section is present in the used machines, and to allow for a lightweight, easily portable and deployable system, a TEE virtualisation technology was used alongside a containerised application that can take advantage of the physical processor.

However this technology still has some limitations, and to address the memory limitations, the system also allows for a storage system to be placed in unprotected memory and still maintain security, privacy and integrity properties.

This way, we can maintain a system, running an unmodified key-value storage application and deploy it safely on the cloud, or any other machine provider and be sure that data is as protected as it can be.

Not only that, but all the hardware stack and software can be attested so that, on an unlikely event of an attack, data corruption or leaked vulnerability occurs, the user can always, to a high percentage of trust, define whether or not the underlying system is correct and can be trusted.

Chapter 4

Prototype Implementation

This chapter presents a detailed explanation of the implementation of the prototype - A Trusted and Privacy-Enhanced In-Memory Data Store, and all the implementation details that helped the system to achieve a secure state according to the adversary model.

Section 4.1 explains the system model presented on figure 3.1 from a developer view, and presents all used technologies, programming languages and implementation details used to achieve the desired system.

Section 4.2 presents some general additional security and implementation features also worth mentioning and in section 4.3 it is explained some tradeoffs decided in the implementation of the prototype, and why were they made.

To finalise, there is a general summary of the chapter in section 4.4 that gathers all the important implementation features from all components.

The implemented prototype source code is available publicly on GitHub, secure datastore [87], the proxy [88] and the client/tester [86] and a list of all technologies and corresponding versions are present in annex I.

4.1 Architecture and Implementation Options

To achieve the goal of deploying the system in a cloud, we had to find a provider that has and provides host machines with the pretended TEE technology - Intel's Software Guard Extensions (SGXv1) version 2.5.0. Although not globally available, some cloud providers are starting to make them available and for this thesis, the cloud provider used is OVH Cloud [75].

For this thesis, OVH provided an IaaS stack machine running Ubuntu Server version 18.04 with kernel 4.15.0-101-generic, which means that we have control over all host's stack but the hardware, from the operating system, networks, runtime and applications.

The used machine configurations are listed on listing 4.1.

Listing 4.1: Machine Specifications

```
Dedicated Server Node
Processor: Intel 2x Xeon Silver 4214 - 24c/48t - 2.2GHz/3.2Ghz
Memory: 128 GB
Hard Drive: NVMe, SATA available
Public Network: Beginning at 1 Gbps
Private Network: Beginning at 2 Gbps
CloudLinux (Ubuntu 18.04 LTS Server 64 bits)
```

This particular Intel processor offers SGX with an 128MB of enclave page cache (EPC) with about 94MB being available for application use like explained in section 2.3.2 and all SGX linux drivers and SDKs were installed [53, 63].

All components of the application will be deployed using Docker v19.03.6 [33] and the Docker Compose tool v1.17.1 [34]. To integrate and run unmodified applications with SGX, the SCONE v4.2.1 [14] technology was used and will wrap all components that need to run within a secure and isolated environment.

4.1.1 Secure Redis

Redis [78] is the key-value storage server used by this thesis. Redis instances will run in two different modes, as explained in section 3.3.1. Unsecure Redis configuration will run on unprotected memory on a docker image based on the official Redis Docker repository [35]. For the secure configuration, SCONE framework already provides a curated image from their repository which contains a Redis server version 6.0.8 ready to run on an isolated environment, in this case, Intel's SGX. The SCONE version used is the SCONE 4.2.1 to match across all the SCONE components.

Although all Redis servers run behind a proxy all the necessary security features provided natively by the server are used. Only communications incoming from the proxy server are allowed and all are encrypted with strong TLS 1.3 [1] protocols with enclave termination. The non-encrypted communication port is disabled, and mutual TLS authentication is turned on, which means that all clients are required to provide a certificate signed by the thesis CA in order to establish a connection.

Access Control is also enabled through an explicit ACL [2]. Following the principle of least privilege, users are defined via a username and a strong password and have permissions to access only the operations that they require to function.

When running in a replicated environment, master-slave or cluster, the same principles apply. Communication between replicas is also always through mutual TLS authentication, even in cluster mode where an event bus is necessary for replica synchronisation. Replicas are read-only and since they can connect to the master instance, they use

[1] TLS is a new feature released in Redis v6.0
[2] Redis ACL is a new feature released in Redis v6.0

a specific user with permissions to perform just the operations that the replica needs to synchronise, and cannot alter the state of the master instance. On a cluster environment, data is *sharded* between masters which means that data is partitioned between hosts. Each key is hashed, and the hosts are responsible to handle keys for a given hash range. On an event of a crash or terminal fault, a slave instance can be promoted to master and automatically take over the failed master's hash range. The synchronisation between replicas and masters, and communication between masters are performed over an event bus, and also always through encrypted TLS/SSL strong secure channels.

4.1.2 Proxy Server

The proxy server is the component that abstracts the Redis configurations in the backend. Proxy is a spring boot starter, version 2.3.0.RELEASE, web server application written in Kotlin v1.4.10, a language that runs on the JVM with Java OpenJDK version 1.8.0_222.

This component serves as a single point of entry to the system, and clients connect to it via an exposed HTTP API via a SSL/TLS encrypted channel (HTTPS). This connection only authenticates the server, but users need to provide an authorisation header to access the server. The bearer token, on the format of a JWT (Json Web Token), must be provided by the external authentication server as the proxy will check with it to validate the request. Not all users can access all endpoints, and the proxy decides the access control based on the role presented in the token.

Using an external configuration file, the proxy is able to communicate with multiple configurations of Redis instances. Communication with the instance is performed via Jedis v3.3.0 [56], a simple and lightweight java Redis client. When connecting to a protected Redis instance, an instance secured by SGX processor and running inside an enclave, the proxy passes through the keys and values to the instance without any modification, meaning that values are secured inside the enclave even though they are handled in plaintext. However, all data residing on unprotected memory should be encrypted and by enabling a flag in the configuration file, the proxy will encrypt, sign and perform integrity checks on all keys and values. The homomorphic encryption is performed with the help of the Hlib v1.2r2 [92], an homomorphic encryption library implemented by the NOVA LINCS developers.

Keys for all value formats (simple, lists, etc..) are always encrypted with the Homomorphic Deterministic (HomoDet) cipher, that guarantees the same encrypted string for the same clear text value. This allows for Redis to match a given key with one present in the storage without revealing the actual value of the key.

The value from the key-value pair is encrypted in a more complex way than the keys, and their format is detailed in 4.1, 4.2 and 4.3.

$$EncryptedValue = [value]_{Ks} \tag{4.1}$$

$$CompositeValue = [value]_{Ks} \mid [EncryptedValue]_{Ksignature} \tag{4.2}$$

$$[value]_{Ks} \mid [EncryptedValue]_{Ksignature} \mid [CompositeValue]_{KHmac} \tag{4.3}$$

The value is encrypted in one of two ways - refering to 4.1:

- **If the value is a string**, it is encrypted with a strong AES cipher working on a ECB (Electronic Code Book) mode with a *PKCS5* Padding (*AES/ECB/PKCS5Padding*) from SunJCE provider, with a 256 bit key.
- **If the value is an integer/long/double**, the value is encrypted with the Homomorphic Addition (HomoAdd) cipher from the Hlib library with a Paillier Key.

Strings are encrypted with the strongest cipher because no operation will be performed over the encrypted value, but for arithmetic values, the HomoAdd cipher is used, to allow for the addition, subtraction and multiplication operations over the encrypted value.

Regardless of the encryption cipher, the encrypted value is signed with a standard SHA512 with RSA signature algorithm from the *SunRsaSign* provider - 4.2. The signature is performed over the encrypted value so it allows for the results of the homomorphic operation being sign without having to decrypt the value. The signature is then appended to the encrypted value and the result is hashed with an HMacSHA256 algorithm to provide a rapid integrity check. The result of all security operations, encryption, signature and hashing is appended into one string and set into the Redis database as a single value.

When setting a value in a list with a score, the keys and values are secured like explained above, but the scores can also pretend to sensitive data and are also encrypted. However, to provide the capabilities of fetching values between scores, a score is encrypted with the HomoOpeInt cipher from Hlib, an order-preserving cipher. This cipher keeps the scores confidential but still preserves order and can be searchable on an inequality operation, for example, $x < score < y$, that fetches all values with scores between an x and a y number.

For demonstration purposes, when adding a value to a set, on the respective endpoint, the value is encrypted with the HomoSearch cipher from Hlib. This cipher allows searching an encrypted value for a specific substring provided by the client.

The API, endpoints and parameters, are completely documented and available on an OpenAPI v3.0.0 *yml* format.

4.1.3 Client-based Benchmarks

As explained before, the client is going to be emulated by a tester. Benchmarks were performed in two different ways: directly against the Redis instances, using the **redis-benchmark** [48] tool, and through the proxy API.

This proxy API tester is implemented using the Gatling v3.2.1 [38] framework. Gatling is a load and performance testing tool that is configurable as code. The configuration of the tool is written in Scala v2.12.3 and provides a very configurable API. It has the ability to generate fields, that will be used to populate the database, and perform as many requests as needed or perform request during a certain amount of time. This framework also allows performing the necessary login request to the external authentication server in order to provide the access token to the proxy API.

For performance and load testing, we can also make various simultaneous users perform actions at the same time and set up ramp-up periods of higher load.

The benchmarks are doing requests to every endpoint available and will be compared against other proxy and database configurations and Gatling provides a detailed report for each one, exposing different metrics that will be presented further down this document in chapter 5.

Also, the same tests were configured on the Apache Jmeter v5.3[9] load testing platform to corroborate some results.

To record memory and CPU statistics, a script, written in bash, was implemented that makes use of the Docker command *docker-stats*, the *ps* and *top* Linux commands and also the Redis Info available from the server, to record the statistics in real-time and write it to disk in a CSV format. Scripts and tests are present on GitHub [86].

4.1.4 Authentication Server

The external authentication server is Keycloak v10.0.2 [58], an open-source identity and access management system from Red Hat. This service implements the current standards for single sing-on, authentication and authorisation security.

Communication is performed over encrypted HTTPS channels, and the client logins directly against this server to obtain the access token necessary to access the proxy.

User and roles management is done via the Keycloak user console and the platform allows for the configuration of different access token signing keys, their expiration date and supports token revocation and key rotation.

4.1.5 Attestation

Attestation is the mechanism responsible to provide the user with a trusted indication of the complete system state and is performed with two different methods, in two different scenarios. The hardware and SGX attestation and an on-demand software and system stack attestation

SGX attestation is performed at startup and its transparent and automatic using physical SGX EPID-based attestation between the application, a locally deployed attestation service (SCONE LAS) and the remote SCONE CAS.

The SCONE Configuration and Attestation Service (CAS) is a part of the system's infrastructure and it is meant to hold application secrets and injected into authenticated

and attested enclaves. This service is also running inside an enclave so, any secrets stored in the server are also isolated and protected from outsiders. Each secret has a strong access policy attached and guarantees that only the right enclave, running the correct code in the correct system can access them. On application start-up, the created enclave communicates with the CAS to fetch the necessary secrets for the application to start. This is the first step of the application process and it is described in figure 4.1.

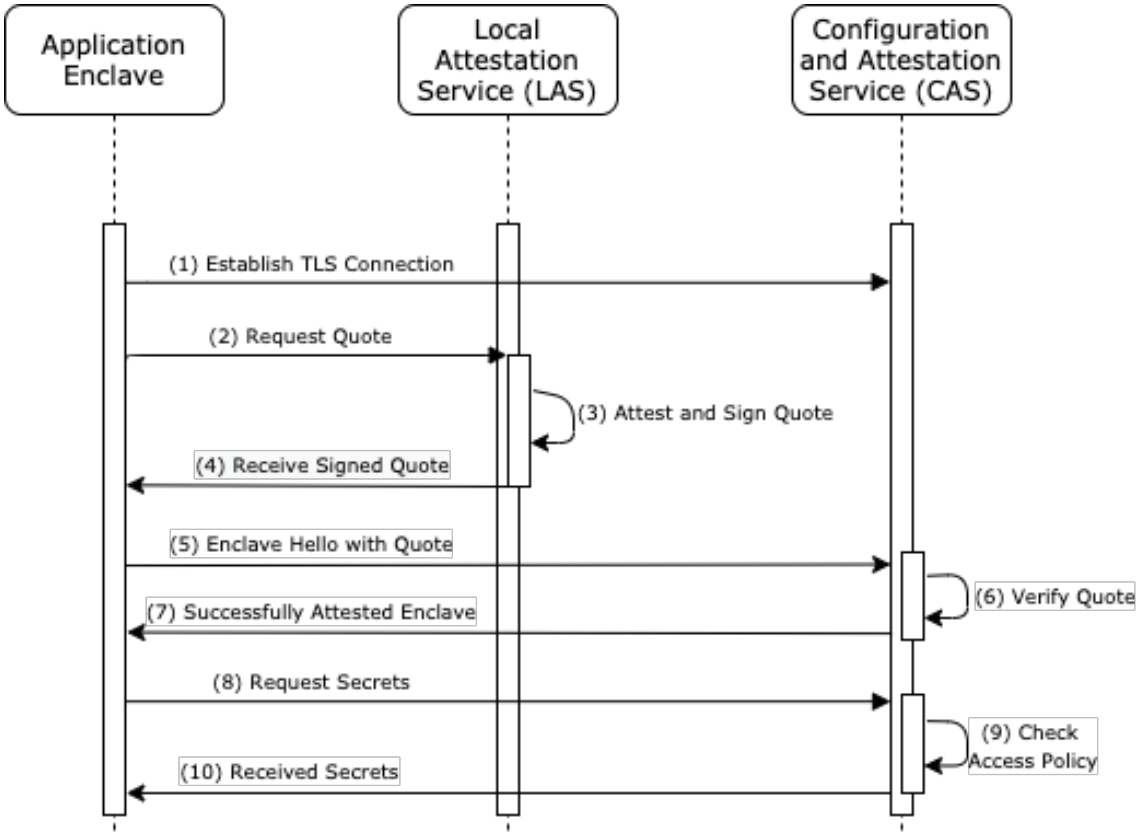


Figure 4.1: Attestation Flow

Having established a connection with CAS (**1**), the application now contacts the SCONE Local Attestation [3] Service in order to receive a signed quote from the trusted LAS (**2, 3, 4**) that then sends to the CAS (**5**). This quote contains information about the application enclave signed by the trusted LAS, including and most importantly, the enclave hash, an hash determined by the content of the pages of the enclave. CAS can now verify the quote provided (**6**) and return a success message (**7**) if the quote is deem correct and signed by LAS. Then, the application requests the secrets from CAS (**8**) and if the request passes all access policies (**9**), the secret is then returned (**10**) and injected into the enclave. These secrets can be injected as an environment variable or a file in a provided path, but whatever the format, they always stay inside the enclave protected memory, which means that no human or process, regardless of their roles in the system will be able to access them. On

[3] Check local attestation process on annex II

enclave destruction, the secrets are also destroyed and never written to disk or persisted anywhere.

In case of the Redis storage server, the TLS private keys and certificates are stored in CAS and not in the container. This guarantees that, if it is able to provide the right certificates on a HTTPS connection, the enclave was attested and it is correct. For the Redis custom attestation service, the secrets hidden in CAS are the quote signing keys and for the proxy server, the TLS and the attestation signing keys as well, and the same trustability principles apply.

The application software and stack attestation relies on the first type attestation in order to access the keys necessary to operate, and it is performed at user request. This attestation is meant to attest binaries, config files, hardware and OS information to the client by providing signed hashes of those resources. The client then decides if they match and pass the expected integrity checks, and if sign with a key provided by the CAS.

This attestation service was implemented in C++14 for the Redis instances, compiled using *Linux Musl g++* GNU Compiler Collection (GCC) version 10.2.0 using the Linux Chilkat C/C++ Library v9.5.0.84 statically linked to the code in order to run totally inside the enclave, and Kotlin for the proxy server. An example of the attestation response is presented in annex III.

4.2 Additional Details

4.2.1 Protected Memory Check

SCONE framework says that it is running the applications inside secure enclaves and application memory are protected and there are two ways to verify it. By running an application with secrets in the SCONE configuration and attestation server (CAS), we can be sure that the program is running inside an enclave since CAS requires a specific SGX attestation mechanism.

On a lower architecture level, we can actually inspect the memory being used by the application. To test this, we can write a simple C program that runs on an enclave and holds a secret on an array in memory and then inspect the program's memory by dumping it via the */proc* filesystem. The */proc/<pid>/map* shows the different memory regions of the process and */proc/<pid>/mem* holds the memory. By dumping all the memory of the process externally to the enclave, and analysing this file, we can check that the secret is **not** present in the dumped memory since, externally to the enclave, we do not have access to that memory. On the other hand, running the same program on unprotected memory, by again dumping the process memory to a file, we can examine it and find the on-memory secret, since its memory is not protected by the enclaves and the secure processor [37].

4.2.2 Protected Heap and Stack Memory

The application stack and heap memories are a core feature of any program. The stack is a memory region that stores temporary variables and data for a single computing task or function. In this memory space, variables are declared stored and initialised during runtime and are automatically erased after the code block is complete. The heap, on the other hand, is a bigger region of memory, mostly allocated at startup and it stores long-livedglobal variables, class references and other data necessary for the complete lifecycle of the program.

Running on an enclave, all of this memory must be protected and isolated from the all the other system software, hardware and even users with privileged accounts, by being placed inside the EPC. Since we are running Intel's SGX version 1 processor, all protected memory must be allocated at enclave startup. SCONE provides environment variables that can be set when running an application to adjust the size of the heap and stack memory regions, the SCONE_HEAP and SCONE_STACK variables.

Version 1 of SGX technology means that no on-demand dynamic scaling or paging is available on the enclave allocated memory (which will be fixed by SGXv2) and that SCONE cannot estimate how much memory the application will need. However, swapping enclave pages to main unprotected memory is still available, meaning that we can allocate more memory than the psychical EPC limit of 128MB, but also means that all memory must be allocated at enclave startup which can generate a higher startup time, and some OOM (Out Of Memory) errors if the application reaches Stack or Heap allocated memory limits.

4.2.3 TLS, HTTPS and Certificate Chain

All communications in the system, both from outside or inside the system are performed over HTTPS, TLS v1.2 or v1.3, with custom and different certificates for each component. Moreover, we can also establish a chain of trust by signing all certificates with a root custom certificate authority (CA). A custom CA was created, and signed all certificates and keys and marked as a trusted certificate authority for the entire system.

4.2.4 Logging and Auditing

All operations in the system, being logins, proxy standard operations or even attestation requests are logged. A log line contains a timestamp, remote IPv4 address, request information such as path, method, and response status, and the request owner, the user. However, to protect preserve privacy, the username is hashed. A system administrator can access the logs in order to audit the system.

4.3 Tradeoffs on the Implementation Options

Tradeoff is the loss a system property in exchange for another. When implementing additional security properties in a system, there will always be a performance impact. It is then up to testing and use-case evaluation to decide whether or not the security increase compensates the performance decrease.

When it comes to replication, there is also a decision to make referring to a consistency-performance tradeoff. Being a key-value store, Redis is always focused on performance and that is why, it follows an eventual consistency model on replication, meaning that a Redis master node will propagate changes to their slave nodes but will not wait for slave acknowledge to respond to the client.

SGX technology also makes several tradeoffs between performance and security, although it defaults to the latter. Protected memory size is a big issue to an SGX enabled application, because, as explained in section 2.3.2, the protected memory is limited, and when it is exceeded, protected memory is swapped to main memory. With additional encryption and decryption cycles as well as integrity checks, that as a penalty performance that can reach over 2000x.

To explain this problem, we can use the OpenSSL library as an example: Dynamically linking a library to an enclave will incur a security level penalty because, not only the library cannot be trusted as it is part of the operating system and can be compromised, but also TLS termination and decryption would be performed, outside the enclave. On the other hand, statically linking the OpenSSL libraries (option taken on this project) means that the enclave has the necessary libraries to perform TLS termination inside the enclave. However, it will increase the size of the application, reducing the amount of memory that can be present inside the enclave and having to be being swapped into main memory.

Another implementation detail on the performance-security tradeoff topic is the homomorphic ciphers that the proxy implements on an unprotected Redis instance. Homomorphically encrypted values do leak some minor information to an attacker, although it is a good trade-off between security and performance. For example, order-preserving ciphers although they do not leak specific values, they do leak the decrypted value order.

The security-performance tradeoff is not a static line, and it moves depending on the use case. For some applications, data must be extremely secure and performance can take a hit to maintain a strong privacy-preserving system. This thesis implemented various security properties and all are evaluated in chapter 5 to provide enough information for a user to decide whether or not it is a right solution to its own use case.

4.4 Summary

To recap, Redis will serve as a storage server and runs on an isolated environment using Intel's SGX secure processor and also runs on normal unprotected memory. However,

both solutions preserve privacy and integrity, either by SGX's guarantees or by proxy enabled encryption and security properties. The proxy enables some Redis out-of-the-box operations, but also some additional operations. When Redis is running in unprotected mode, the proxy enables the performance of operations over encrypted memory, sparing encryption and decryption cycles.

Both Redis and the Proxy Server are attested at start-up, by requesting application secrets held by a third party server that implements strong access policies to make sure each secret is injected to the correct enclave.

Users can also request attestation of the system on-demand, by requesting it to the Proxy server. The Proxy contacts all Redis instance's attestation services, which are deployed alongside Redis, and returns binary, config files and system information that the user can analyse and have the guarantee that it is communicating with a correct system.

In regards to communication, all channels between all components are secure with the standard string TLS encryption with a system-wide CA signature.

These security properties come with some tradeoffs but they all address a necessary privacy property, and the next chapter will evaluate them and determine if there is a bearable overhead in order to implement a secure correct and trustable system.

Chapter 5

Validation and Experimental Evaluation

This chapter presents, analyses and discusses the work performed to obtain relevant evaluation criteria measured from the complete system deployed on a cloud environment, in order to validate the implemented prototype.

Section 5.1 details how the system is deployed, explains the different configurations that that system adopts in order to present relevant data to compare.

In section 5.2 explains which metrics will be evaluated for each different testbench scenario. Sections 5.3 through 5.8 presents the actual metrics taken from the performance and load testing tools and compares the different system configurations with secure and vulnerable storage instances.

The last section, section 5.9 contains a summary of all the finding and some considerations that can be taken from the presented metrics.

5.1 Testbench Environments

The prototype was evaluated in multiple different system configurations to achieve complete coverage of all the proposed solutions and all the relevant metrics.

The first test is a representative test of the overhead introduced not only of additional security features like TLS and authentication but also the SGX hardware isolation. It will run the *redis-benchmark* tool both externally to the server and internally directly against a single standalone Redis instance bypassing the proxy server. Secondly, and more real-life tests, test are run against the proxy exposed API and a single standalone Redis instance composes the storage server. Then, the same tests were run in the same environment but with a cluster of Redis instances composing the backend storage. The fourth testbench measures the performance of homomorphic operations on a standalone Redis instance

running on unprotected memory, and finally, it is presented the metrics for the two different attestation operations on Redis instances deployed on protected memory and isolated through SGX.

All tests performed on the prototype were performed on a MacBook Pro 2018 2,3 GHz Quad-Core Intel Core i5 and internet connection averaging the 500Mb/s download and 100Mb/s upload and an effort was made to maintain the same conditions on every tests. Additionally, all tests were measured several times, averaged and compared with a standard Redis deployment so the overhead of extra security can be evaluated and analysed.

5.2 Relevant Evaluation Criteria

The tests measure several different relevant metrics that can be compared with each other:

- **Latency** - Measured in milliseconds (ms) and evaluate the round trip response times between the client and the server.
- **Throughput** - Measured in operations per seconds (ops/s) indicates the number of operations the client performed in one second.
- **Startup Times** - Measured in seconds (s) is particularly important to analyse the SGX attestation that happens at startup
- **Memory Consumption** - Extracted in megabytes (MB) or a percentage, and measured alongside the tests.
- **CPU Consumption** - Extracted in megabytes in a percentage, and also measured alongside the tests.

5.3 Performance Evaluation for Redis-Benchmark tool

The first testbench is performed directly against the Redis server bypassing the proxy. This test was performed with the **redis-benchmark** [1] tool and evaluates the latency and throughput of a basic set of operations - *ping*, *set*, and *get*. It compares three different deployed Redis configurations, where the first one is completely default and open, the second one implements the built-in security features of Redis such as TLS and authentication and the third one, not only using the built-in features but also running isolated inside an SGX enclave.

[1] Since Redis TLS is fairly new, the latest official Redis-benchmark release does not support TLS test. The Redis benchmark used comes from the Redis unstable branch (commit a0576bd), but after some analyses, it appears to be a final version of the tool. All benchmarks tests used the same version.

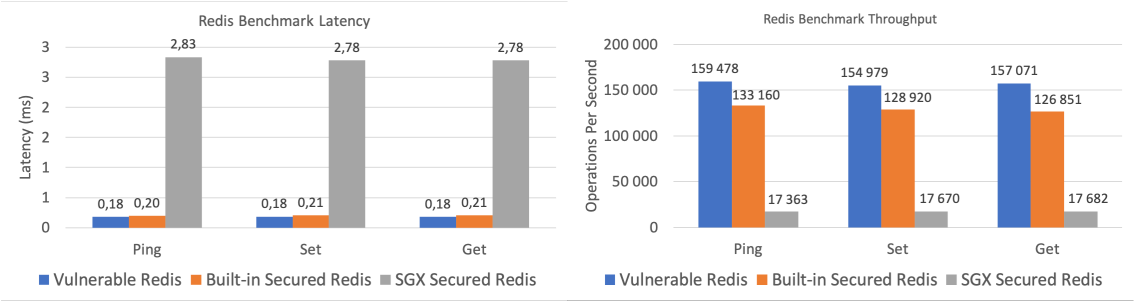
Figure 5.1 presents the latency and throughput results over one hundred thousand requests with 50 multiple concurrent clients and a 3-byte payload. This test is meant as a first reference test, to understand the sole Redis instance capacity without any gateway or proxy overhead and so, the payload size is not relevant.

(a) Redis-Benchmark Latency (b) Redis-Benchmark Throughput

Figure 5.1: Redis Benchmark External Client Metrics

The results show a performance drop for each additional security layers added to the instance deployment. The SGX deployed instance showed the biggest overhead on performance, around 8%-10%, both latency and throughput, however, due to the added security, this is an expected result.

Figure 5.2 shows the results of the same tests but instead of running the benchmark tool on a separate machine, it runs it on the same host where the server is deployed, therefore eliminating the network overhead.



(a) Redis-Benchmark Latency (b) Redis-Benchmark Throughput

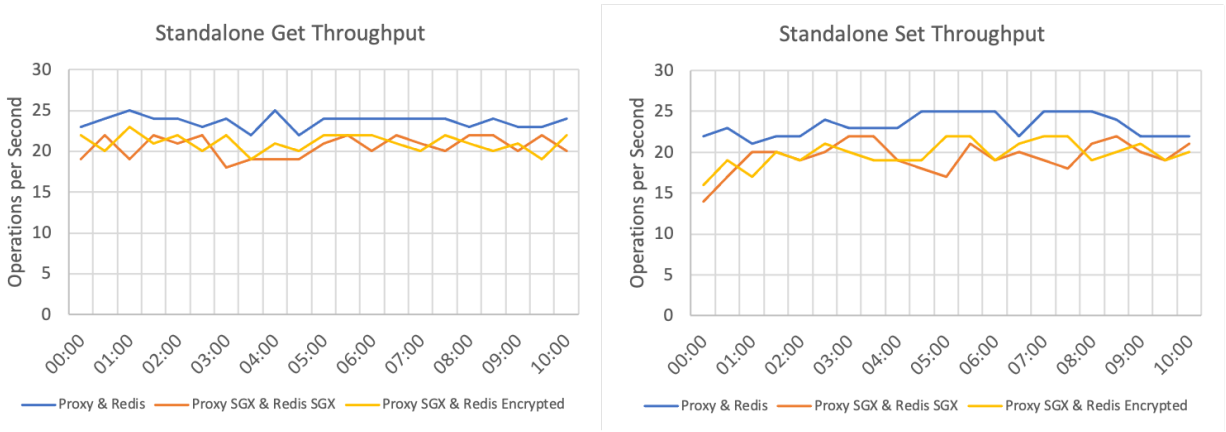
Figure 5.2: Redis Benchmark Internal Client Metrics

The results corroborate the initial tests but the differences between security levels are much more visible. The main objective of this comparison is to point out that the network jitter and latency overhead will affect the performance of the results. However, since the realistic use case relies with a client running on a different machine than the backend, all the next tests made on the system are performed with an external client to make them more realistic and close to the normal use case meant for the system.

5.4 Performance Evaluation for Standalone Redis

This testbench configuration tests a standalone architecture with a single proxy instance and a single Redis instance. Both the proxy and the Redis components will be tested in different deployment configurations, running inside and outside SGX enclaves. When Redis is running outside the isolated environment, it will run both in a vulnerable configuration, without any security properties and in its encrypted format, where the proxy server enabled encrypted values.

The tests will run one single thread, making as many *gets* and *sets* requests as possible during 10 minutes with a 20-byte key and a 100-byte value. Figure 5.3 presents the throughput obtained from the tests for the different configurations.



(a) Get throughput

(b) Set throughput

Figure 5.3: Standalone Throughput Results

The throughputs measured from the different system configurations show a higher count of operations per second on the normal and vulnerable Proxy & Redis configuration. When running the components on an SGX enclave, the performance takes a 20%-25% hit. Keeping the proxy running inside enclaves, but extracting the Redis to unprotected memory sees 18%-22% loss in performance. Even though the storage service is running faster outside enclaves, the proxy needs to perform extra work in order to maintain data privacy and integrity.

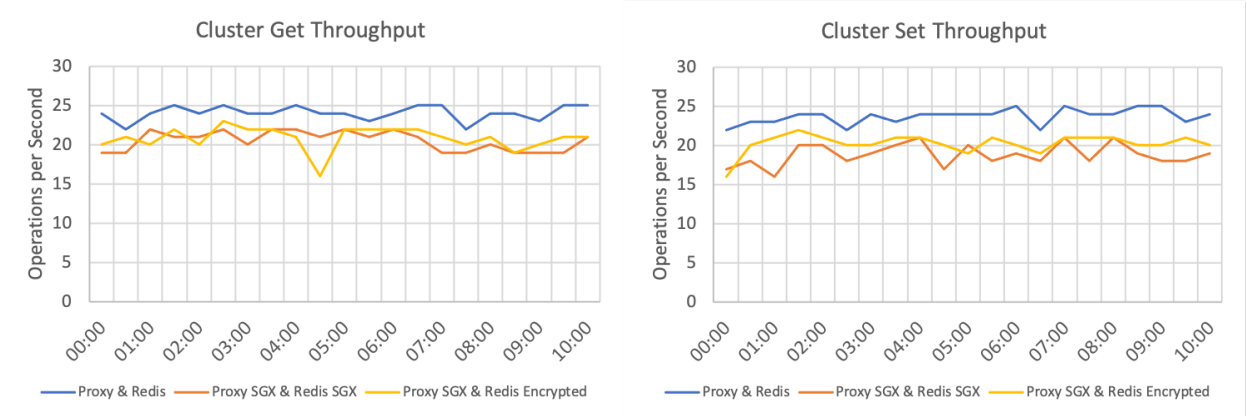
The latency results summarised on table 5.1 support the throughput measurements.

Table 5.1: Proxy Redis Standalone Results

Test	#Request	Avg Latency	σ	95%	DB Size
Proxy & Redis	14038	42ms	15ms	42ms	2,03MB
Proxy SGX & Redis SGX	12013	49,67ms	19,66ms	58ms	2,00MB
Proxy SGX & Redis Encrypted	12773	48,67ms	20,12ms	55ms	6,49MB

5.5 Performance Evaluation for Cluster Redis

Clustering on Redis is distributed out of the box on the latest Redis versions. It combines the Master-Slave model and an event bus to provide replication and sharding and coordination between the nodes. This test uses a configuration with one proxy server, and six Redis nodes, with three masters and three slaves. The tests run were the same of the standalone system configuration presented on section 5.4 to maintain consistency between tests, and it runs a single thread, making as many requests as possible during 10 minutes, using random 20-byte keys with a 100-byte random value. Figure 5.4 shows the throughput results on the sets and get operations.



(a) Get throughput

(b) Set throughput

Figure 5.4: Cluster Throughput Results

Again, as expected, the vulnerable Redis configuration is the fastest since it doesn't need to maintain any security or privacy properties. Running the Proxy and Redis server on an isolated environment incurs in a performance penalty of 18%-25%. However, running the Redis in unprotected memory but on an encrypted format seems to have a bit of an edge in performance over the SGX isolated server.

Table 5.2: Proxy Redis Cluster Results

Test	#Request	Avg Latency	σ	95%
Proxy & Redis	14219	41ms	13ms	41ms
Proxy SGX & Redis SGX	12229	49ms	21ms	53ms
Proxy SGX & Redis Encrypted	13131	46ms	16ms	46ms

Table 5.2 shows the latency results of the same tests, and it shows a correlation with the tests and the evaluation made above.

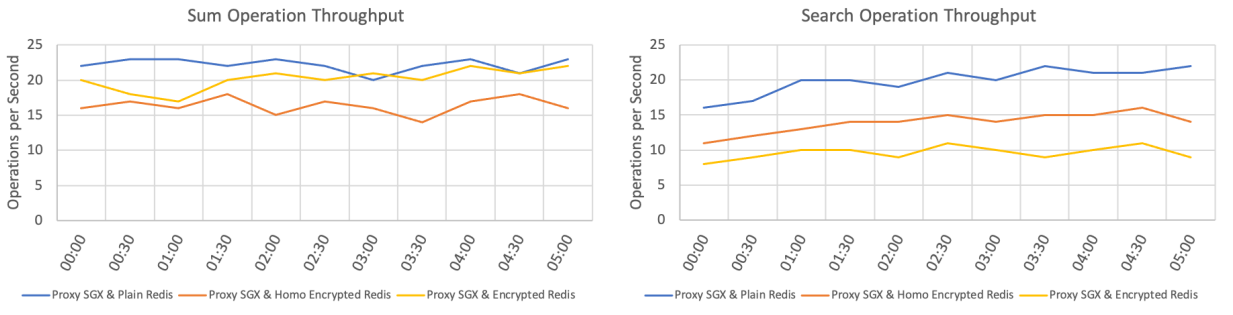
By observing both throughputs and latencies of the standalone and the cluster configurations we can compare them and observe similarities. Both tests seem to have a similar result on all system configurations and that might be due to the consistency guarantee of the cluster. Although a Redis cluster does provide replication, automatically split data

among multiple nodes and some availability during network partitions, it is not able to guarantee **strong consistency**. This means that under some specific conditions the Redis cluster can lose writes that were acknowledged to the user by the system. This happens because Redis uses asynchronous replication, where it responds to the client before replicating the results to the replica instances. This configuration explains the similarity in performance since the proxy will only communicate with one instance at a time and that instance will respond before replicating the given command, just like a standalone node.

5.6 Performance Evaluation for Homomorphic Operations

The implementation of homomorphic encryption is a way to speed up performance by performing arithmetic operations over encrypted data. This test runs a proxy server inside an SGX enclave and a single Redis instance running on unprotected memory. To maintain data privacy and integrity on an unprotected Redis instance, the proxy enables encryption and only stores encrypted data on Redis. This test compares latency and throughputs of the *Sum* and *Search* operations of the system. There are three configurations to test, one where the proxy does not encrypted data (the plain Redis), a second where the proxy enables encryption with homomorphic ciphers, and the last one, where proxy encrypts data with standard *AES* encryption. With homomorphic encryption, the operations are performed over the encrypted value, meaning that, unlike standard encryption, it does not need to decrypt the current value for the *SUM* operation to succeed.

The *Sum* tests were run over a dataset of 5000 key-pairs and over 5 minutes making as many requests as possible and the *Search* test run over a dataset with 1 list with 1000 values with about 140 bytes per line. The tests were run multiple times with multiple payload sizes and the average throughputs and latencies are presented in figure 5.5 and table 5.3.



(a) *Sum* Throughput

(b) *Search* Throughput

Figure 5.5: Homomorphic Encryption Throughput Results

As expected, a non-encrypted Redis is the fastest configuration, however, the results on the encrypted configurations of the *Sum* operation were not as anticipated. We were not able to achieve better performance by performing *Sum* operations over encrypted

Table 5.3: *Sum* Latency Results

Test	#Request	Avg Latency	σ	99%	DB Size
Sum - Plain Redis	6512	46ms	17ms	48ms	1.13MB
Sum - Homo Encrypted Redis	4853	62ms	23ms	69ms	10.578MB
Sum - Standard Encrypted Redis	6001	52ms	19ms	54ms	3.85MB
Search - Plain Redis	5718	52ms	20ms	68ms	0.21MB
Search - Homo Encrypted Redis	4238	70ms	25ms	117ms	0.49MB
Search - Standard Encrypted Redis	3016	99ms	30ms	150ms	0.27MB

data. This result may be due to the size of the *Sum* operation values, since we are working within the range of integers, standard AES encryption is very fast. As we can see on the *Search* operations since values are strings, they are much bigger than a single integer and the encryption and decryption cycles are much more costly, making its performance about 35% worst than the homomorphic encryption *Search* operation.

On the other hand, homomorphic encryption can enable operations that are not possible with the standard encryption like maintaining encrypted data in order and searching through finding values between certain boundaries. Another very important feature that is enabled by homomorphic encryption is that the plain text value is never present in memory, since the operations do not need to decrypt the value in order to proceed.

5.7 Evaluation of the Attestation Protocol

The two different attestation types are evaluated in two different ways. Since SGX hardware attestation is a SCONE feature and it happens automatically on startup, there is not an accurate way to measure the attestation time, so it is extrapolated from the startup times of different components configurations.

Table 5.4 shows the comparison between startup times of the Redis instance and Proxy instance, using a component that does not run on an enclave, the component running on enclave but not performing attestation, and one running within enclaves with startup SGX hardware attestation.

Table 5.4: SGX Hardware Attestation Results

Component	No SGX	SGX w/o Attestation	SGX w/ Attestation	≈ Attestation Time
Redis Instance	0,009s	0,2854s	1,604s	1,2916s
Proxy Instance	3,108s	66,1174s	69,146s	3,0294s

The Redis and the Proxy instances have very big startup differences and there is a reason for that. SCONE provides default Heap and Stack limits for each curated images they provide. When running Redis, SCONE provided about 64MB to the Redis enclave and to run a java program is requests about 4GB or heap memory. Since memory must

be allocated at startup, there is a big gap in size between the heap memory that is requested by the different processes and it explains the startup time and attestation time differences.

On the other attestation type, the proxy will contact the Redis instance and get a signed quote of important aspects of the system such as the binary and configuration file, OS kernel information and CPU core count and processor type. It also attests itself and returns the quotes to the client. This process was also measured by requesting as much attestation quotes in 10 minutes and collected throughputs and latency values described on table 5.5.

Table 5.5: Custom Attestation Results

Run Number	#Request	Avg Latency	σ	99%	Req/s
1st Run	1491	402ms	188 ms	549ms	2,481 ops/s
2nd Run	1494	401ms	179 ms	550ms	2,483 ops/s
3rd Run	1494	401ms	164ms	546ms	2,486 ops/s

5.8 Complementary Measurements

5.8.1 Memory and CPU Measurements

During the standalone tests, CPU and memory values of the containers and processes were measured and recorder for further evaluation. Figure 5.6 shows the average CPU loads of each system configuration during the 10 minute test performance test detailed in section 5.4 of the standalone test.

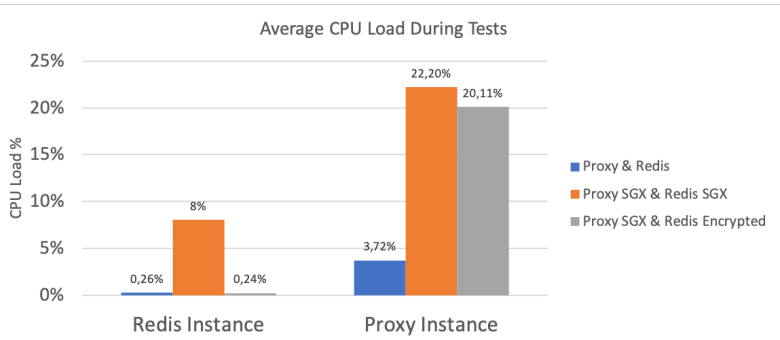


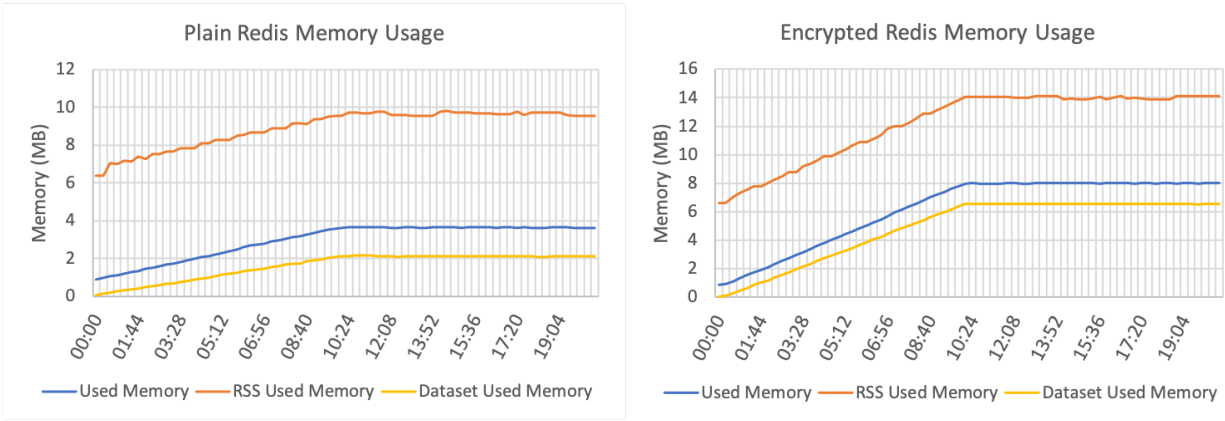
Figure 5.6: Average CPU Load

By cross-referencing the tests made directly into Redis with the help of the Redis-Benchmark tool and the ones performed through the proxy that regardless of the configuration, the network and the proxy impose a significant overhead that allows the Redis server to never really reaching a problematic CPU load. However, running the proxy inside a secure enclave also incurs an increase of CPU load during testing. Also, running alongside the protected Redis is a small attestation service that measures at about

2%-2.5% of the total 8% measured. When it comes to the Proxy server, the same conclusions were reached, where a service running protected by a secure enclave seams to run a higher CPU load during performance testing.

Higher CPU loads can be explained with encryption and decryption of data in order to maintain it secure, as well as stronger access control checks, and protected memory page swapping that requires more encryption, decryption and integrity checks and system calls that slow down the system and required exiting and entering the enclaves.

Graphs 5.7a, 5.7b and 5.8 show the memory evolution during the same tests. The metrics recorded were the Redis total memory usage, the Redis dataset memory size and the RSS (Resident Set Size) memory of the Redis process. Figure 5.7 shows the memory from both Redis configurations that do not run on a secure enclave, where figure 5.7a represents to the memory statistics of an normal Redis deployment and the 5.7b corresponds to an homomorphic encrypted Redis configuration.



(a) Plain Redis Memory Usage

(b) Encrypted Redis Memory Usage

Figure 5.7: Plain/Encrypted Redis Memory Usage

The graphs show a normal progression, with a clear distinction between the *sets* and *gets* operations on the test. All metrics climb steadily while data is being inserted into Redis, and maintain their value when the test changes to the *get* operation portion of the test. With the additional information that is stored with the value on an encrypted configuration, such as digital signatures and integrity check hashes, the memory used was expected to be higher, and the tests confirmed that expectation with a 35% to 40% memory usage increase.

Graph 5.8 shows the memory evaluation of the Redis instance running on an Intel's secure enclave, and although the total Redis memory and dataset size follow the same lines as the previous configurations, the RSS memory starts very high and maintains that value all throughout the test. This can be explained by the lack of ability that an application running inside of Intel's SGXv1 secure enclave to dynamically resize allocated memory. Nevertheless, used memory and dataset memory of the Redis seems to be about the same of an unprotected Redis running on standard RAM.

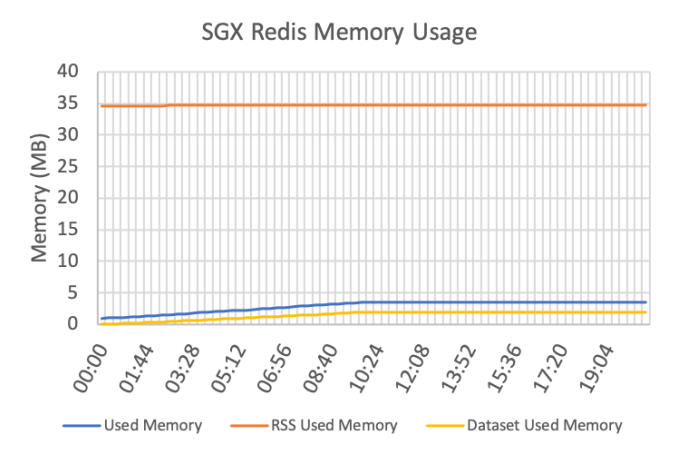


Figure 5.8: SGX Redis Memory Usage

When running Redis instances in a cluster configuration, the total dataset memory usage by each node is somewhat different and the results are presented in table 5.6.

Table 5.6: Cluster Instances Dataset Memory Usage (MB)

Cluster Node	Proxy & Redis	Proxy SGX & Redis SGX	Proxy SGX & Redis Encrypted
Master-1	1,00	0,92	2,7
Master-2	1,02	0,94	2,6
Master-3	1,01	0,93	2,6
Slave-1	1,01	0,94	2,6
Slave-2	1,02	0,93	2,7
Slave-3	1,00	0,92	2,6
Total Masters	**3,05 MB**	**2,80 MB**	**8,04 MB**
Standalone	**2,03 MB**	**2,00 MB**	**6,49 MB**

The hash of the inserted key is the metric that determines the node where the key-pair is stored, and with randomly generated keys used by the tests, we can see a very equal distribution of key-pairs among the available masters. When a key-pair is written on a mater, the operation is forwarded to the corresponding slave, and, by analysing the table, we can even determine which slave replicates each master since slaves are a complete replica of the master node, they hold the same data.

It was also determined that, although each node contains a portion of the data of the complete dataset, the sum of all the dataset memory of each node adds up to bigger in size than the same dataset [2] on a standalone configuration. This can be traced to the default metadata that each Redis node contains at startup so it can function correctly but also metadata about each node currently on the cluster, synchronisation properties and key mapping data structure. Each Redis cluster instance holds an internal data structure

[2]Datasets might not be exactly the same, because keys are generated randomly through the tests, but the key-pair size is maintained

that maps each key in the cluster to its designated slot. This carries a memory penalty for each node and the more keys the cluster holds, the bigger the overhead.

5.8.2 Exhausting Protected Memory

The performance of a process that runs inside an isolated enclave is directly dependent on the locality of memory access. When the process necessary memory does not fit in the EPC, the process will suffer page faults. When a page fault occurs, the SGX driver selects a page from the EPC, re-encrypts it for freshness and stores it in main unprotected memory. The page swap process takes some time, and the overhead of the application grows with the page fault rate. With the increasing size of an application, the page fault rate also increases, and the performance overhead will also increase correspondingly.

Also, as previously mentioned on section 4.2.2, the lack of ability of dynamically resizing and allocating memory when running on a secure enclave is an SGX1 problem, but, a second generation of the processor, the SGX2, with the objective to overcome this problem, is already a work-in-progress [103].

This means that application *heap* and *stack* memory parameters should be tuned at enclave startup and are fixed and not resizable at runtime. When running any datastore application, these parameters are very important because, depending on the use case, the size which the database size will rise is not very predictable and reaching the *heap* or *stack* limits will kill the process with a OOM (Out of Memory) error.

Through testing, we were able to confirm this limitation, by inserting data continuously until reaching the default *SCONE Heap* limit of 64MB, the container where Redis was running gets killed by a OOM error. By tuning the *SCONE Heap* parameter through the provided environment variable, Redis was able to surpass the 64MB limit and also the 128MB physical limit of the EPC.

5.8.3 Performance and Payload Size

All tests were made with a small key-value pair with about 20 bytes for the key and 100 bytes for the value. However, it is important to know how the system would handle requests with the different payload sizes, and a standalone system running the three systems configurations tested in this chapter were tested with payloads of 100 bytes, 1KB, 100KB, 500KB and 1MB and the results are presented in graph 5.9.

The graph shows the average response times in running the different configurations with the variable payload size over a 5000 request test bench. As expected, a plain Redis running on unprotected memory scaled the best, followed by the enclave and encrypted configurations that run closely throughout the tests ending up with a 25% to 35% decrease in performance.

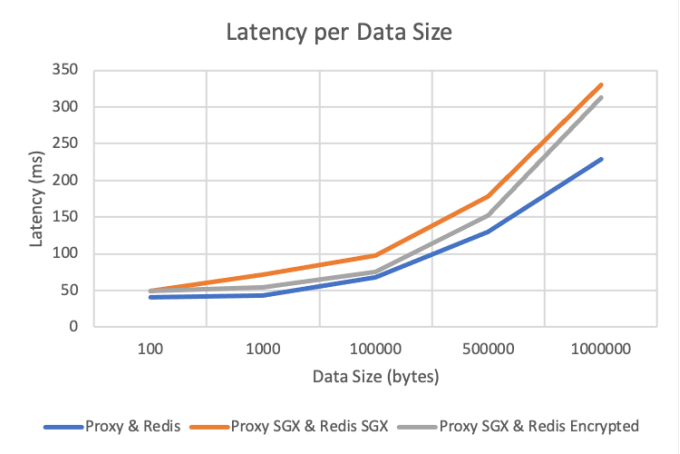


Figure 5.9: Latency per Data Size

5.9 Summary and Findings

A plain and normal configuration, an SGX enabled architecture and an hybrid system are the three most prominent configurations available and all were tested on the same test benches and on the same conditions to provide a clear conclusion of the work performed in order to achieve and reach the expected goals and contributions of this thesis, but also for those who are able to choose a system architecture based on security, can make an informed decision and tailored to their use case.

The test revealed that, as expected, additional security measures incur on a performance overhead to the system but risk managing is an important part of any project planning and the performance drawback can be worth it for some use cases.

Referencing the SGX secure enclaves, we saw a performance drop of about 18% to approximately 25%, because the performance of an application running on an enclave depends on these three main points:

Memory Access Local: If the application exceeds the EPC size limit, memory is swapped to main memory and that requires additional encryption and decryption cycles.

System Calls: System calls require enclave exiting and entering which is a slow process. However, SCONE uses an asynchronous system call interface that ensures that threads do not need to exit the enclave to perform a system call.

Threading: SCONE uses asynchronous application threading in order to allow switching to another application thread if another one is waiting for some system event, without exiting the enclave, which can speed up the performance.

We also found out that the bigger the application enclave is, the more page swapping it has to perform and also, with SGX1, the lack of dynamically resizing enclaves makes a great case for an hybrid approach, where the storage server lives on unprotected memory but is secured by encryption mechanism enabled by the proxy, which can itself run inside an enclave, making sure that the encryption and decryption cycles are hardware protected.

Chapter 6

Conclusions

This chapter resumes and summarises all the work performed on this thesis and presents final thoughts that were concluded from the related work, system model and architecture, implementation and design options and the evaluations taken from all the tests performed.

Section 6.1 recaptures the objectives and contributions and matches them against the results obtained from the system implementation. Section 6.2 details the issues and limitations of the system that were discovered during the study of the system and corresponding implementation, and section 6.3 discusses future work that can be performed in order to improve the system and continue the work presented in this thesis, as well as fixing and completing open issues and limitations presented before.

6.1 Main Conclusions

Cloud computing adoptions keeps growing with every year that passes and companies and enterprises are finding more and more attractive some of the features the cloud providers offer such as, easy deployment, automatic and seamless scalability as well as replication and pay-as-you-use payment structures and business model. However, data privacy is another trend that has arisen in the past few years, and people, now more than ever, want to hold control of their own data, keep it private and secure, and need more assurances from cloud providers that all measures are being implemented to guarantee just that.

The implementation of this dissertation had multiple objectives and contributions that were explained in chapter 1 and the main solution proposed was the **design and implementation of a cloud-enabled privacy-enhanced solution**, with all-in-the-box planned features, and able to be used as a "cloud-platform as a service" solution, providing:

- **Trust-ability, security and privacy**.
- **Software attestation**.
- **Replication Mechanism**.
- **Drastically reduced TCB**.
- **Complete analysis report**.

Given the objectives and contributions proposed, we were able to successfully design and implement a system, leveraging Intel's SGX trusted environment execution processor, and provide a trusted and privacy-enhanced in-memory data store, using container technology for the deployment and easy portability, providing remote attestation in order to guarantee software and hardware correctness, drastically reducing the TCB, maintaining data secure and private while operating on a remote machine.

Limitations of the secure enclave technology were considered, and the use of a dataset operating in unprotected memory but maintaining data private with the help of partial homomorphic encryption with operations performed over encrypted data, can be used to overcome some of those limitations. Also, the implemented secure system can be replicated over multiple instances and the data sharding and splitting can remove stress from each node of the replicated cluster, improving availability, reliability and also performance.

The SCONE framework helped to run an unmodified version of Redis and with combination of a trusted execution environment for code and data security and isolation, the implemented system seams viable and offers a balanced and moderate trade-off between security and performance.

6.2 Open Issues and Limitations

During the design and implementation of the system, some issues and limitations appeared. The SGX performance and memory limitations are still an issue and, to some use cases where performance is the main factor and cannot be diminished by adding additional security guarantees, this solution might not be the best suitable to adopt. Also, the work with homomorphic encryption is still an ongoing study case, and it means that this is not a full-fledge solution, and may not support all operations over an encrypted dataset.

As any work implemented for a specific platform, this solution will only be effective on Intel machines with a processor that supports the latest software guard extensions technology and although the code is easily deployable using containers, the processor is yet not available massively and globally across multiple cloud providers or even personal machines thus, reducing the portability factor.

One important issue that was found in the implementation of the system is that Redis storage data persistency mechanism, is exposing data in plain text when trying to persist itself and backing up data into the unprotected persistent disk storage even though running on an isolated secure enclave that should not ever expose data. This is a big vulnerability, that was patched temporarily by disabling data persistency on disk, however, data backups are an important feature, and the *SCONE* technology may have a way to resolve the issue with their encrypted file system support, but the problem did not have further investigation on this dissertation.

6.3 Future Work Directions

After all the work performed and all some limitations and issues catalogued the following are future work that can be performed to improve a trusted and privacy-enhanced in-memory datastore:

The use of the still in progress but already available second version of the secure processor, the **SGX2** will help to overcome some memory limitations and problems by providing dynamic memory swapping and allocation and dynamic enclave resizing at runtime, something that was not possible to test in this dissertation, but will improve not only in enclave memory but also startup times and the whole system resources consumptions.

The **proxy server** is identified as a single point of failure, and although no tested in the current project, the replication of the proxy server can help not only with performance but also with system availability.

Also, although the proxy and the system can support multiple users with multiple roles, it does not offer **multi-tenant support**, meaning that all data is secured with the same encryption and decryption keys. The usage of different keys per user/tenant, proxy generated or user-supplied keys and also a key rotation system can help privacy and improve the overall security of the system.

The implemented system allows for the user to request software and hardware stack information on demand, but the trusted SGX attestation procedure, only happens at enclave startup. Some investigation is necessary but, the implementation of an **on-demand SGX enabled attestation** mechanism is very interesting and an important asset to the system.

Mentioning the limitations described in section 6.2, fixing the **backup issue** would be a really important improvement for the system, to allow for data backup and system recovery in the event of a crash of terminal failure.

Finally, the system was implemented with the help of the *SCONE* framework, but it would be very important and informative to try and test the same system on another SGX or enclave enabled platform and compare the performances, security and resource usage between the different frameworks.

There are always ways to **generally improve performance** and resource consumption of the infrastructure, and the improvement of a secure and trusted system is a plus for all users that wish to hold control of their data, keep it private and authentic and never relinquish control to anyone, regardless of the environment on which their software runs on.

Bibliography

[1] *80% Of Enterprise IT Will Move To The Cloud By 2025*. Accessed: 2020-08-12. URL: https://www.forbes.com/sites/oracle/2019/02/07/prediction-80-of-enterprise-it-will-move-to-the-cloud-by-2025/.

[2] *A brief history of cloud computing*. Accessed: 2020-01-11. URL: https://www.ibm.com/blogs/cloud-computing/2014/03/18/a-brief-history-of-cloud-computing-3/.

[3] *A history of cloud computing*. Accessed: 2020-01-11. URL: https://www.computerweekly.com/feature/A-history-of-cloud-computing.

[4] *Aerospike*. Accessed: 2019-06-16. URL: https://www.aerospike.com.

[5] *Aerospike*. Accessed: 2019-06-16. URL: https://www.aerospike.com/docs/guide/security/index.html.

[6] *Amazon Dynamo DB*. Accessed: 2019-06-16. URL: aws.amazon.com/dynamodb.

[7] *Amazon S3 - Cloud Storage*. Accessed: 2019-07-09. URL: https://aws.amazon.com/s3/.

[8] *Amazon Web Services*. Accessed: 2019-06-16. URL: https://aws.amazon.com.

[9] *Apache JMeter™*. URL: https://jmeter.apache.org.

[10] *Apple's iCloud service suffers cyber-attack in China, putting passwords in peril*. Accessed: 2020-01-11. URL: https://www.washingtonpost.com/news/the-switch/wp/2014/10/21/apples-icloud-service-suffers-cyber-attack-in-china-putting-passwords-in-peril/.

[11] *ARM TrustZone*. Accessed: 2020-01-08. URL: https://developer.arm.com/ip-products/security-ip/trustzone.

[12] *ARM TrustZone Security Whitepaper*. Tech. rep. PRD29-GENC-009492C. ARM Limited, Dec. 2008. URL: http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf.

[13] *ARM TrustZone Stack Image*. Accessed: 2020-01-08. URL: https://malware.news/t/introduction-to-trusted-execution-environment-arms-trustzone/20823.

[14] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. 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)*. Savannah, GA: USENIX Association, 2016, pp. 689–703. ISBN: 978-1-931971-33-1. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov.

[15] *Around 75% of Open Redis Servers Are Infected With Malware*. Accessed: 2020-02-18. URL: https://www.bleepingcomputer.com/news/security/around-75-percent-of-open-redis-servers-are-infected-with-malware/.

[16] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. 1.00. Arpaci-Dusseau Books, 2018.

[17] *asylo*. Accessed: 2020-09-20. URL: https://asylo.dev.

[18] *asylo*. Accessed: 2020-09-20. URL: https://github.com/google/asylo.

[19] *Attestation and Trusted Computing - CSEP 590: Practical Aspects of Modern Cryptography*. Accessed: 2019-11-26. URL: https://courses.cs.washington.edu/courses/csep590/06wi/finalprojects/bare.pdf.

[20] *Attestation Identity Key (AIK) Certificate Enrollment Specification*. Accessed: 2019-11-26. URL: https://www.trustedcomputinggroup.org/wp-content/uploads/IWG-AIK-CMC-enrollment-FAQ.pdf.

[21] *AWS hit by major DDoS attack*. Accessed: 2020-01-11. URL: https://www.techradar.com/news/aws-hit-by-major-ddos-attack.

[22] *Azure Blob Storage*. Accessed: 2019-07-09. URL: https://azure.microsoft.com/en-us/services/storage/blobs/.

[23] 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)*. Boston, MA: USENIX Association, 2019, pp. 173–190. ISBN: 978-1-931971-48-5. URL: https://www.usenix.org/conference/fast19/presentation/bailleu.

[24] S. Banescu. *Cache Timing Attacks*. July 2011. URL: https://www.academia.edu/3224323/Cache_Timing_Attacks.

[25] *Benefits of Hardware Trusted Modules*. Accessed: 2019-11-30. URL: https://www.hardware-security-module.com/benefits/.

[26] S. Christopherson. *Intel SGX Virtualization - KVM Forum 2018*. Intel Open Source Technology Center | 01.org Accessed: 2020-10-16. URL: https://www.linux-kvm.org/images/e/e8/KVM_Forum_2018_-_Intel_SGX.pdf.

[27] V. Costan and S. Devadas. *Intel SGX Explained*. Cryptology ePrint Archive, Report 2016/086. 'https://eprint.iacr.org/2016/086'. 2016.

[28] V. Costan, I. Lebedev, and S. Devadas. *Sanctum: Minimal Hardware Extensions for Strong Software Isolation*. Cryptology ePrint Archive, Report 2015/564. ' https://eprint.iacr.org/201 2015.

[29] *Data leaks: The most common sources*. Accessed: 2020-02-18. URL: https://www.zdnet.com/pictures/data-leaks-the-most-common-sources/13/.

[30] *Data leaks: The most common sources*. Accessed: 2020-02-18. URL: https://www.zdnet.com/pictures/data-leaks-the-most-common-sources/12/.

[31] *Data leaks: The most common sources*. Accessed: 2020-02-18. URL: https://www.zdnet.com/pictures/data-leaks-the-most-common-sources/14/.

[32] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. "Dynamo: Amazon's Highly Available Key-value Store." In: *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*. SOSP '07. Stevenson, Washington, USA: ACM, 2007, pp. 205–220. ISBN: 978-1-59593-591-5. DOI: 10.1145/1294261.1294281. URL: http://doi.acm.org/10.1145/1294261.1294281.

[33] *Docker*. Accessed: 2020-02-18. URL: https://www.docker.com.

[34] *Docker Compose*. Accessed: 2020-02-18. URL: https://docs.docker.com/compose/.

[35] *Docker Hub Redis Official Repository*. Accessed: 2020-08-15. URL: https://hub.docker.com/_/redis/.

[36] *Document Store*. Accessed: 2019-06-16. URL: https://aws.amazon.com/nosql/document/.

[37] *FINDING SECRETS...* URL: https://sconedocs.github.io/memory_dump/.

[38] *Gatling - Load test as code*. URL: https://gatling.io.

[39] C. Gentry, S. Halevi, and N. P. Smart. "Homomorphic Evaluation of the AES Circuit." In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2012, pp. 850–867. DOI: 10.1007/978-3-642-32009-5_49. URL: https://doi.org/10.1007/978-3-642-32009-5_49.

[40] *Google Cloud Storage*. Accessed: 2019-07-09. URL: https://cloud.google.com/storage/.

[41] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller. "Cache Attacks on Intel 'SGX'." In: *Proceedings of the 10th European Workshop on Systems Security - 'EuroSec' '17*. 'ACM' Press, 2017. DOI: 10.1145/3065913.3065915. URL: https://doi.org/10.1145/3065913.3065915.

[42] *Graph DBMS*. Accessed: 2019-06-16. URL: https://aws.amazon.com/nosql/graph/.

[43] *Graphene Library OS with Intel SGX Support*. Accessed: 2020-02-16. URL: https://github.com/oscarlab/graphene.

[44] *Graphene-SGX Secure Container (GSC): Automatic Protection of Containerized Applications with Intel SGX*. Accessed: 2020-02-18. URL: https://github.com/rainfld/gsc.

[45] S. Gueron. *A Memory Encryption Engine Suitable for General Purpose Processors*. Cryptology ePrint Archive, Report 2016/204. 2016. URL: https://eprint.iacr.org/2016/204.

[46] S. K. Haider, H. Omar, M. Ahmad, C. Jin, and M. van Dijk. *Intel's SGX In-depth Architecture*. URL: https://scl.engr.uconn.edu/courses/ece6095/lectures/sgx_architecture.pdf.

[47] *Hardware Trusted Modules*. Accessed: 2019-11-30. URL: https://resources.infosecinstitute.com/tpms-or-hsms-and-their-role-in-full-disk-encryption-fde/.

[48] *How fast is Redis?* Accessed: 2020-02-16. URL: https://redis.io/topics/benchmarks.

[49] M. S. Inci, B. Gulmezoglu, G. Irazoqui, T. Eisenbarth, and B. Sunar. *Seriously, get off my cloud! Cross-VM RSA Key Recovery in a Public Cloud*. Cryptology ePrint Archive, Report 2015/898. 2015. URL: https://eprint.iacr.org/2015/898.

[50] *Intel SGX*. Accessed: 2020-01-04. URL: https://software.intel.com/en-us/sgx.

[51] *intel/kvm-sgx*. Accessed: 2020-10-16. URL: https://github.com/intel/kvm-sgx.

[52] *Intel® Software Guard Extensions (Intel® SGX) - Developer Guide*. Accessed: 2020-01-25. URL: https://download.01.org/intel-sgx/linux-2.2/docs/Intel_SGX_Developer_Guide.pdf.

[53] *Intel® Software Guard Extensions SDK for Linux*. Accessed: 2020-02-18. URL: https://01.org/intel-softwareguard-extensions.

[54] S. IT. *Key-Value Stores*. Accessed: 2019-06-16. URL: https://db-engines.com/en/article/Key-value+Stores.

[55] S. IT. *Key-Value Stores Ranking*. Accessed: 2019-06-16. URL: https://db-engines.com/en/ranking/key-value+store.

[56] *Jedis - A blazingly small and sane redis java client*. URL: https://github.com/redis/jedis.

[57] *Kernel Virtual Machine*. Accessed: 2020-09-20. URL: https://www.linux-kvm.org/page/Main_Page.

[58] *Keycloak - Open Source Identity and Access Management*. URL: https://www.keycloak.org.

[59] A. K. Khan and H. J. Mahanta. "Side channel attacks and their mitigation techniques." In: *2014 First International Conference on Automation, Control, Energy and Systems ('ACES')*. IEEE, Feb. 2014. DOI: 10.1109/aces.2014.6807983. URL: https://doi.org/10.1109/aces.2014.6807983.

[60] 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*. EuroSys '19. Dresden, Germany: ACM, 2019, 14:1–14:15. ISBN: 978-1-4503-6281-8. DOI: 10.1145/3302424.3303951. URL: http://doi.acm.org/10.1145/3302424.3303951.

[61] *Kinetic Object Storage*. Accessed: 2019-07-09. URL: https://storageioblog.com/seagate-kinetic-cloud-object-storage-io-platform/.

[62] R. Krahn, B. Trach, A. Vahldiek-Oberwagner, T. Knauth, P. Bhatotia, and C. Fetzer. "Pesos." In: *Proceedings of the Thirteenth EuroSys Conference on - EuroSys 18*. ACM Press, 2018. DOI: 10.1145/3190508.3190518. URL: https://doi.org/10.1145/3190508.3190518.

[63] *linux-sgx-driver*. Accessed: 2020-02-18. URL: https://github.com/intel/linux-sgx-driver.

[64] *Local Attestation*. URL: http://www.sgx101.com/portfolio/local_attestation/.

[65] N. Lord. *What is the Principle of Least Privilege (POLP)? A Best Practice for Information Security and Compliance*. Accessed: 2020-10-24. URL: https://digitalguardian.com/blog/what-principle-least-privilege-polp-best-practice-information-security-and-compliance.

[66] F. McKeen, I. Alexandrovich, A. Berenzon, C. Rozas, H. Shafi, V. Shanbhogue, and U. Savagaonkar. *Innovative Instructions and Software Model for Isolated Execution*. Tech. rep. Intel Corporation, Aug. 2013. URL: https://software.intel.com/sites/default/files/article/413936/hasp-2013-innovative-instructions-and-software-model-for-isolated-execution.pdf.

[67] *Memcached*. Accessed: 2019-06-16. URL: http://www.memcached.org.

[68] *Memcached Github*. Accessed: 2019-06-16. URL: https://github.com/memcached/memcached/wiki/Overview.

[69] *Microsoft Azure Cache For Redis*. Accessed: 2019-06-16. URL: https://azure.microsoft.com/en-us/services/cache/.

[70] *Microsoft Azure Cosmos DB*. Accessed: 2019-06-16. URL: https://azure.microsoft.com/en-us/services/cosmos-db/.

[71] *Microsoft Azure Services*. Accessed: 2019-06-16. URL: https://azure.microsoft.com/en-us.

[72] *Microsoft Security Intelligence Report Volume 22*. Accessed: 2020-01-11. URL: https://www.microsoft.com/security/blog/2017/08/17/microsoft-security-intelligence-report-volume-22-is-now-available/.

[73] M. Minkin. "Improving Performance and Security of Intel SGX." Master's thesis. Israel Institute of Technology, Dec. 2018.

[74] *Open Redis Servers Infected with Malware*. Accessed: 2020-02-18. URL: https://www.infosecurity-magazine.com/news/open-redis-servers-infected-with/.

[75] *OVHcloud - Intel Software Guard Extensions (SGX)*. Accessed: 2020-02-16. URL: https://www.ovh.ie/dedicated_servers/software-guard-extensions/.

[76] C. Priebe, K. Vaswani, and M. Costa. "EnclaveDB – A Secure Database using SGX." In: *To appear in the Proceedings of the IEEE Symposium on Security & Privacy, May 2018*. IEEE, May 2018. URL: https://www.microsoft.com/en-us/research/publication/enclavedb-a-secure-database-using-sgx/.

[77] *Professor John McCarthy*. Accessed: 2020-01-11. URL: https://cs.stanford.edu/memoriam/professor-john-mccarthy.

[78] *Redis*. Accessed: 2019-06-16. URL: https://redis.io.

[79] *Redis 6.0.06*. Accessed: 2020-08-15. URL: https://raw.githubusercontent.com/redis/redis/6.0/00-RELEASENOTES.

[80] *Redis Multi Model Store*. Accessed: 2019-06-16. URL: https://db-engines.com/en/system/Redis.

[81] *Redis Performance Benchmark*. Accessed: 2019-06-16. URL: https://redislabs.com/docs/nosql-performance-benchmark/.

[82] *Redis Security*. Accessed: 2019-06-16. URL: https://redis.io/topics/security.

[83] *SASL Rfc*. Accessed: 2019-06-16. URL: https://tools.ietf.org/html/rfc2222.

[84] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard. "Malware Guard Extension: Using SGX to Conceal Cache Attacks." In: *arXiv e-prints*, arXiv:1702.08719 (Feb. 2017), arXiv:1702.08719. arXiv: 1702.08719 [cs.CR].

[85] *Search Engine Database*. Accessed: 2019-06-16. URL: https://aws.amazon.com/nosql/search/.

[86] *secure-redis-client*. URL: https://github.com/aanciaes/secure-redis-client.

[87] *secure-redis-container*. URL: https://github.com/aanciaes/secure-redis-container.

[88] *secure-redis-proxy*. URL: https://github.com/aanciaes/secure-redis-proxy.

[89] *Sgx Memory Limits*. Accessed: 2019-06-16. URL: https://software.intel.com/en-us/forums/intel-software-guard-extensions-intel-sgx/topic/670322.

[90] *SGX Secure Enclaves in Practice: Security and Crypto Review*. Black Hat USA 2016 Security Conference. URL: https://www.youtube.com/watch?v=0ZVFy4Qsryc.

[91] *SGX Virtualization*. Accessed: 2020-10-16. URL: https://01.org/intel-software-guard-extensions/sgx-virtualization.

[92] *SJHOMOLIB*. Accessed: 2020-02-18. URL: http://nova-lincs.di.fct.unl.pt/prototype/233.

[93] M. Taassori, A. Shafiee, and R. Balasubramonian. "VAULT: Reducing Paging Overheads in SGX with Efficient Integrity Verification Structures." In: *ASPLOS '18*. 2018.

[94] *Tesla cloud systems exploited by hackers to mine cryptocurrency*. Accessed: 2020-02-18. URL: https://www.zdnet.com/article/tesla-systems-used-by-hackers-to-mine-cryptocurrency/.

[95] *The Benefits Of Moving To The Cloud*. Accessed: 2020-01-11. URL: https://www.forbes.com/sites/forbestechcouncil/2017/05/19/the-benefits-of-moving-to-the-cloud.

[96] *The Cambridge Analytica scandal changed the world – but it didn't change Facebook*. Accessed: 2020-02-18. URL: https://www.theguardian.com/technology/2019/mar/17/the-cambridge-analytica-scandal-changed-the-world-but-it-didnt-change-facebook.

[97] *Time Series Databases*. Accessed: 2019-06-16. URL: https://www.forbes.com/sites/metabrown/2018/03/31/get-the-basics-on-nosql-databases-time-series-databases/.

[98] *Trusted Platform Modules*. Accessed: 2019-07-09. URL: https://docs.microsoft.com/en-us/windows/security/information-protection/tpm/trusted-platform-module-overview.

[99] 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)*. URL: https://www.usenix.org/system/files/conference/atc17/atc17-tsai.pdf.

[100] C.-C. Tsai, D. E. Porter, K. S. Arora, N. Bandi, B. Jain, W. Jannen, J. John, H. A. Kalodner, V. Kulkarni, and D. Oliveira. "Cooperation and security isolation of library OSes for multi-process applications." In: *Proceedings of the Ninth European Conference on Computer Systems - EuroSys '14*. ACM Press, 2014. DOI: 10.1145/2592798.2592812. URL: https://doi.org/10.1145/2592798.2592812.

[101] A. Waterman, Y. Lee, D. A. Patterson, K. Asanovic, V. I. U. level Isa, A. Waterman, Y. Lee, and D. Patterson. *The RISC-V Instruction Set Manual*. 2014.

[102] *What is a Trusted Execution Environment (TEE)?* Accessed: 2019-12-03. URL: https://www.trustonic.com/news/technology/what-is-a-trusted-execution-environment-tee/.

[103] B. C. Xing, M. Shanahan, and R. Leslie-Hurd. "Intel® Software Guard Extensions (Intel® SGX) Software Support for Dynamic Memory Allocation inside an Enclave." In: *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016 on - HASP 2016*. ACM Press, 2016. DOI: 10.1145/2948618.2954330. URL: https://doi.org/10.1145/2948618.2954330.

ANNEX I

Technologies and Versions

Table I.1 lists and summarises the version of all technologies used in this thesis.

Table I.1: Versions of Used Technologies

Technology	Version
CloudLinux (OVH Cloud OS)	Ubuntu 18.04 LTS Server 64 bits
Linux Kernel	4.15.0-101-generic
Intel SGX Linux	Linux 2.11 Open Source Gold Release
Intel SGX Linux Driver	2.11
Docker	19.03.6
Docker Compose	1.17.1
Redis	6.0.8
SCONE	4.2.1
Java OpenJDK 64-Bit	1.8.0_222
Kotlin	1.4.10
Jedis	3.3.0
HLib	1.2r2
C++	C++14
Linux Musl g++ (GCC)	10.2.0
Spring Boot Starter Web	2.3.0.RELEASE
OpenSSL	1.1.1h
OpenAPI	3.0.0
Keycloak	10.0.2
Scala	2.12.3
Gatling	3.2.1
Jmeter	5.3
Secure-Redis-Container (this)	**1.1.5**
Secure-Redis-Proxy (this)	**1.3.1**
Secure-Redis-Client (this)	**1.0.0**

ANNEX II

SGX Local Attestation

This annex contains the information that describes the Intel's SGX local attestation process and the text was directly taken from *sgx101.com* [64].

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

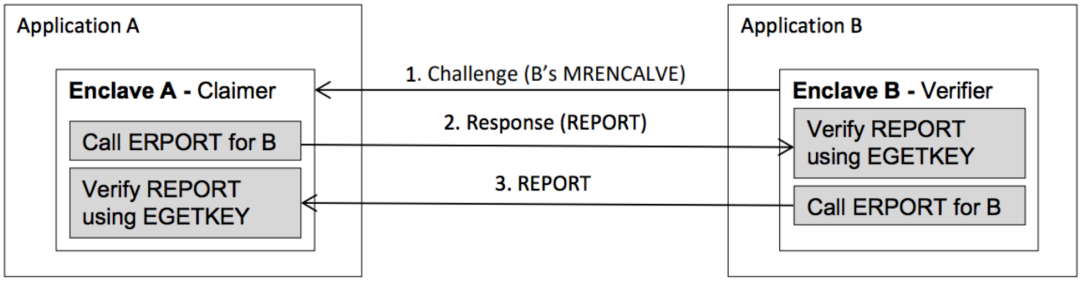


Figure II.1: SGX Local Attestation

There are two enclaves on the same platform, referred to as Enclave A and Enclave B. We assume they have established a communication path between each other, and the path doesn't need to be trusted. W.l.o.g we assume B is asking A to prove it's running on the same platform as B.

1. First, B retrieves its MRENCLAVE value and sends it to A via the untrusted channel.

2. A uses EREPORT instruction to produce a report for B using B's MRENCLAVE. Then A sends this report back to B. A can also include Diffie-Hellman Key Exchange data in the REPORT as user data for trusted channel creation in the future.

3. After B receives the REPORT from A, B calls EGETKEY instruction to get REPORT KEY to verify the REPORT. If the REPORT can be verified with the REPORT KEY, then B assures that A is on the same platform as B because the REPORT KEY is specific to the platform.

4. Then B use the MRENCLAVE received from A's REPORT to create another REPORT for A and sends the REPORT to A. A then also can do the same as step 4 to verity B is on the same platform as A.

5. A then also can do the same as step 4 to verity B is on the same platform as A.

6. By utilising the user data field of the REPORT, A and B can create a secure channel using Diffie-Hellman Key Exchange. Information exchange can be encrypted by the shared symmetric key.

ANNEX III

Software Stack Attestation

Listing III.1 presents an eaxmple of an software and stack attestation response where various measurements about binaries, configuration files and hardware information are provided along side a signature that can be verified for each component instance.

Listing III.1: Software Attestation Response Example


```
{
 "redis": [
  {
   "node": "ns31249243.ip-51-210-0.eu:8541",
   "quote": {
    "challenges": [
     {
      "filename": "/usr/local/bin/redis-server",
      "hash": "b6a07d069d17a2ab5b36869271e266e9ee94823908c9cd9747c33b293207196a"
     },
     {
      "filename": "/usr/local/etc/redis/redis.conf",
      "hash": "65bf49826aca20825206aa3679201eb08516604977efce699d42fca2c8712a03"
     },
     {
      "filename": "/home/redis/mrenclave",
      "hash": "792233b7ef44a0e7a985f561f596a1ce449bf9150e4863e2cc29dc0653829450"
     },
     {
      "filename": "/home/attestation_server/attestation-server",
      "hash": "caab16d965d8c4924a7d8a59558e5d761c8986901a3762cde93ea30c9b30701a"
     },
     {
      "filename": "/home/attestation_server/mrenclave",
      "hash": "b75392ab9d7dad6f42dbd0be0d0584951e8c95dad6ae9af60ef4ae921c712c23"
     }
```

```
    ],
    "nonce": 2,
    "system": {
     "processorCount": 16,
     "processorModel": "Intel(R) Xeon(R) E-2288G CPU @ 3.70GHz",
     "operaringSystem": "Alpine Linux v3.8"
    },
    "quoteSignature": "Dr5afOli3vCI/vvtU5aEol2VdjVc4wW(...)"
   }
  }
 ],

  "proxy": {
   "quote": {
    "challenges": [
     {
      "filename": "/home/secure-proxy-redis/secure-redis-proxy-1.3.0.jar",
      "hash": "6f9731f7bc6a325caa314c724d4d9beea8621dac139c7ecc7deae1ce5368b65f"
     },
     {
      "filename": "/home/secure-proxy-redis/mrenclave",
      "hash": "ed5ae8a95e54392ff9e28f20bb7539c639b1a800fe6d75443330a33abc893896"
     }
    ],
    "nonce": 2,
    "system": {
     "processorCount": 4,
     "processorModel": "Intel(R) Xeon(R) E-2288G CPU @ 3.70GHz",
     "operaringSystem": "Alpine Linux v3.7"
    },
    "quoteSignature": "lz9bRJ75WT70Wabek02YWwummvqhE+8cD32AuhDxg4KP5u(...)"
   }
  }
}
```