Miguel Gomes Rosa

Master of Science

Virtual HSM: Building a Hardware-backed Dependable Cryptographic Store

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

Master of Science in
Computer Science and Informatics Engineering

Adviser: Bernardo Luís da Silva Ferreira,
Researcher,
Faculdade de Ciências e Tecnologia,
Universidade Nova de Lisboa

Examination Committee

Chairperson: Vítor Manuel Alves Duarte
Raporteur: Bernardo Luís Fernandes Portela

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

September, 2019

Virtual HSM: Building a Hardware-backed Dependable Cryptographic Store

Copyright © Miguel Gomes Rosa, Faculty of Sciences and Technology, NOVA University Lisbon.
The Faculty of Sciences and Technology and the NOVA University Lisbon have the right, perpetual and without geographical boundaries, to file and publish this dissertation through printed copies reproduced on paper or on digital form, or by any other means known or that may be invented, and to disseminate through scientific repositories and admit its copying and distribution for non-commercial, educational or research purposes, as long as credit is given to the author and editor.

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

To my loved ones.

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my advisor, Bernardo Ferreira, for his valuable guidance. His knowledge, availability, and support were fundamental throughout the whole process of this thesis.

I would like to thank my closest friends and colleagues who helped me throughout this Master in Computer Science with study sessions, course assignments, and many other wonderful moments.

I am also thankful for the financial support from FCT/MCTES, through the strategic project NOVA LINCS (UID/CEC/04516/2013) and project HADES (PTDC/CCI-INF/31698/2017).

Finally, I must express my very profound gratitude to my parents, my sister and my girlfriend for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them. Thank you.

ABSTRACT

Cloud computing is being used by almost everyone, from regular consumer to IT specialists, as it is a way to have high availability, geo-replication, and resource elasticity with pay-as-you-go charging models. Another benefit is the minimal management effort and maintenance expenses for its users.

However, security is still pointed out as the main reason hindering the full adoption of cloud services. Consumers lose ownership of their data as soon as it goes to the cloud; therefore, they have to rely on cloud provider's security assumptions and Service Level Agreements regarding privacy and integrity guarantees for their data.

Hardware Security Modules (HSMs) are dedicated cryptographic processors, typically used in secure cloud applications, that are designed specifically for the protection of cryptographic keys in all steps of their life cycles. They are physical devices with tamper-proof resistance, but rather expensive. There have been some attempts to virtualize HSMs. Virtual solutions can reduce its costs but without much success as performance is incomparable and security guarantees are hard to achieve in software implementations.

In this dissertation, we aim at developing a virtualized HSM supported by modern attestation-based trusted hardware in commodity CPUs to ensure privacy and reliability, which are the main requirements of an HSM. High availability will also be achieved through techniques such as cloud-of-clouds replication on top of those nodes. Therefore virtual HSMs, on the cloud, backed with trusted hardware, seem increasingly promising as security, attestation, and high availability will be guaranteed by our solution, and it would be much cheaper and as reliable as having physical HSMs.

Keywords: Hardware Security Module; Trusted Hardware; Intel SGX; Cloud Computing; Cloud-of-Clouds

RESUMO

A computação na nuvem é hoje utilizada por quase todos, desde utilizadores comuns a especialistas na área da Informática, pois é uma maneira de garantir alta disponibilidade, geo-replicação e elasticidade dos recursos com o modelo em que se paga consoante a utilização desses mesmos recursos. Tem ainda outro benefício relativo à facilidade de gestão e manutenção da infra-estrutura, passando estas tarefas a ser responsabilidade do provedor da nuvem.

Contudo, preocupações de segurança continuam a ser apontadas como a razão principal que impede uma adoção completa da nuvem. Ao enviarem os seus dados para a nuvem, os utilizadores perdem o controlo sobre os mesmos e são obrigados a confiar nas políticas de segurança dos provedores de serviços na nuvem sobre a privacidade e integridade dos seus dados.

Módulos de Hardware Seguros (HSMs) são processadores criptográficos dedicados, tipicamente utilizados em aplicações seguras na cloud, que foram desenhados de modo a manterem seguras as chaves criptográficas em todo o seu ciclo de utilização. Estes HSMs são dispositivos físicos, resistentes a atacantes que possam adulterar os dados, mas são bastante caros. Houve algumas tentativas de virtualizar HSMs de modo a reduzir os seus custos financeiros mas sem grande sucesso porque a performance é incomparável e é difícil alcançar as garantias de segurança necessárias apenas em implementações de software.

Nesta tese temos como objetivo desenvolver um HSM virtual suportado por hardware confiável presente em CPUs normais que consiga garantir privacidade e confiabilidade, que são os principais requisitos de um HSM. Adicionalmente vamos também oferecer alta disponibilidade através de técnicas como replicação de nuvem de nuvens. Assim os HSMs virtuais na cloud, suportados por hardware confiável parecem ser cada vez mais interessantes e promissores pois a segurança, a atestação e a alta disponibilidade estão garantidos e serão muito mais baratos e tão confiáveis como HSMs físicos.

Palavras-chave: Módulo de Hardware Seguro; Hardware Confiável; Intel SGX; Computação na Nuvem; Nuvem de Nuvens

CONTENTS

List of Figures **xvii**

List of Tables **xix**

Listings **xxi**

1 Introduction **1**
- 1.1 Context and Motivation 1
- 1.2 Problem 2
- 1.3 Objectives 3
- 1.4 Main Contributions 3
- 1.5 Document Structure 4

2 Background **5**
- 2.1 Hardware Security Module 5
  - 2.1.1 Public-Key Cryptographic Standards 6
  - 2.1.2 Virtual HSM 7
  - 2.1.3 Discussion 9
- 2.2 Hardware-backed Trusted Computing 10
  - 2.2.1 Isolated Execution Environment 11
  - 2.2.2 Technologies 11
  - 2.2.3 Discussion 17
- 2.3 Dependable Cloud Storage 18
  - 2.3.1 Cryptographic Cloud Storage 18
  - 2.3.2 DepSky 20
  - 2.3.3 CryptDB 23
  - 2.3.4 Cipherbase 24
  - 2.3.5 Discussion 24

3 Virtual HSM **27**
- 3.1 System Model and Architecture 27
- 3.2 Adversary Model 29
- 3.3 Supported Algorithms 29

3.3.1 Hash Functions . . . 30
3.3.2 Hash Message Authentication Codes . . . 31
3.3.3 Block-ciphers . . . 31
3.3.4 Asymmetric-key Ciphers . . . 33
3.3.5 Elliptic Curve Ciphers . . . 33
3.4 HSM API . . . 34
3.4.1 Digest . . . 34
3.4.2 Key Generation . . . 34
3.4.3 Sign and Verify . . . 35
3.4.4 Encrypt and Decrypt . . . 35
3.4.5 Other Functions . . . 36
3.4.6 Critical Analysis . . . 36
3.5 Implementation Details . . . 37
3.5.1 Setup . . . 37
3.5.2 Secure Channel . . . 38
3.5.3 HSM Internal State . . . 39
3.5.4 External Storage . . . 40
3.5.5 Client Requests . . . 40
3.5.6 Enclave Configurations . . . 41
3.6 Security Analysis . . . 41
3.7 Summary . . . 43

4 Distributed VirtualHSM **45**
4.1 System Model and Architecture . . . 45
4.1.1 Certificate Authority . . . 46
4.2 Adversary Model . . . 47
4.3 Use Cases . . . 48
4.4 Implementation Details . . . 48
4.4.1 Setup . . . 48
4.4.2 Secure Channel . . . 49
4.4.3 Replication . . . 49
4.5 Security Analysis . . . 50
4.6 Discussion . . . 51

5 Experimental Evaluation **53**
5.1 Experimental Test Bench . . . 53
5.2 VirtualHSM Evaluation . . . 54
5.2.1 HASH . . . 54
5.2.2 HMAC . . . 55
5.2.3 AES . . . 55
5.2.4 RSA . . . 56

5.2.5 EC 58
5.2.6 Impact of Distributed Version 59
5.2.7 Comparison with Related Work 60
5.3 Discussion 61

6 Conclusion **63**
6.1 Future Work 64

Bibliography **65**

List of Figures

2.1 Poor Man's HSM architecture (taken from [16]). 9
2.2 Time (in milliseconds) used by SoftHSM and pmHSM in the signing procedure (taken from [16]). 10
2.3 Components of a Trusted Platform Module (taken from [7]). 12
2.4 Intel SGX mechanisms for local and remote attestations (taken from [2]). . . 15
2.5 ARM TrustZone architecture (taken from [12]). 16
2.6 Cryptographic Cloud Storage architecture (taken from [29]). 20
2.7 DepSky architecture (taken from [10]). 21
2.8 DepSky-CA (taken from [10]). 23
2.9 CryptDB architecture (taken from [40]). 23
2.10 Cipherbase architecture (taken from [3]). 24

3.1 System architecture. Server acts as a proxy to securely connect Client with IEE. Persistent Storage is loaded by the Server and sent to IEE, still encrypted. 28

4.1 Distributed system architecture. Clients communicate with HSMs using a secure channel. Each HSM is connect to every other HSM using another secure channel. Certificate Authority is apart from the system and issues certificates for every trustworthy Client and HSM. 46
4.2 CA structure. Each intermediate CA only issues certificates to Clients or to Enclaves. 47

5.1 Digest. 54
5.2 HMAC key generation. 55
5.3 HMAC sign and verify. 55
5.4 AES key generation. 56
5.5 AES mode comparison. 56
5.6 AES encrypt. 56
5.7 AES decrypt. 56
5.8 RSA key generation. 57
5.9 RSA sign. 57
5.10 RSA verify. 57
5.11 RSA seal. 58

5.12 RSA unseal. 58
5.13 EC key generation. 58
5.14 Adapted from: NIST Recommendations(2016) [23]. 58
5.15 EC sign. 59
5.16 EC verify. 59
5.17 Replication process: serialization and deserialization impact. 60

LIST OF TABLES

2.1 Subset of PKCS#11 functions (taken and adapted from [24]). 7
2.2 Subset of Intel SGX instructions (taken from [12, 34]) 14

5.1 Comparison between VirtualHSM and SoftHSM, in Ops/s. 61
5.2 Comparison between VirtualHSM and AWS CloudHSM. 61

LISTINGS

3.1 Virtual HSM API: digest. 34
3.2 Virtual HSM API: key generation. 35
3.3 Virtual HSM API: sign and verify. 35
3.4 Virtual HSM API: encrypt and decrypt. 36
3.5 Virtual HSM API: utils. 36
3.6 Keys structure. 39
3.7 Enclave configuration file. 41

Chapter 1

Introduction

1.1 Context and Motivation

Cloud computing is a hot topic nowadays. Almost two billion people were using the cloud at the end of 2018 [39], and it has become so widely popular that, already in 2009, 95% of IT professionals assume using it dail.

The pay-as-you-go model is not the only advantage that cloud computing offers. One of the main advantages is resource elasticity, where the user begins with few resources and increase them as demand increases. When using the cloud, there are no concerns related to maintenance: the cloud provider is now maintaining everything. The user gets higher availability and geo-replication for a reasonable price (we could opt for the same availability on a private cloud, but the costs would increase drastically, and for geo-replication, we would need to have data warehouses around the globe).

Despite the increase in its popularity, there are some concerns regarding security and privacy that come with cloud computing. From the consumer's point of view, they give away the ownership of their data to the cloud providers and rely on their assumptions of data privacy and security, having no longer control over their data and the infrastructure involved. Despite the rapid growth of cloud computing, many users and companies still do not rely on it to store sensitive data. We are talking about private information, health records, and many others that if leaked, can cause severe issues (not only financial but also judicial).

One of the most used techniques to overcome these security concerns is to encrypt data before sending it to the cloud, ensuring that no one but the owner can decrypt the data. As everything is encrypted, it is impossible to perform computations and queries over data. Thus, whenever it is needed, the owner has to download the data, decrypt it, make all the necessary computations, encrypt it again, and upload it back in the cloud.

This approach is called encryption at rest and not only increases client-side overhead but also wastes cloud resources. It is necessary to find a better solution that allows us to store data securely in the cloud and to provide a cryptographic key management solution.

1.2 Problem

Hardware Security Modules (HSMs) [25, 53] are dedicated cryptographic processors that safeguard and manage cryptographic keys for operations such as authentication, encryption, and signing. They are widely used by security companies and other companies who value security because they have internationally recognized certifications that can vouch for their security guarantees. Tampering resistance and responsiveness are also another of their features as the first makes it very hard for attackers to tamper physically with an HSM without making it inoperable and the second that could, for example, erase all keys if tampering is detected.

Highly dedicated hardware with such specific components and certifications imposes high financial costs, making it impracticable for the average user or company. As an example, a well known physical HSM like Sun Crypto Acelerator costs nearly 1.300$. That is why there have been some attempts to virtualize HSMs [16, 37]. The objective was to provide the same guarantees and performance as physical HSMs. The problem is that performance is incomparable as one is a highly dedicated hardware explicitly designed for that purpose, and the other is just a software running on commodity hardware. Strong security guarantees were also not provided by existing solutions because it is challenging to assure that neither hardware or software can be compromised.

The appearance of hardware-backed trusted computing, i.e., safe hardware (CPUs) that can provide strong security guarantees, such as confidentiality, integrity, and attestation of their computations seemed very promising. In some cases they can even vouch for some hardware increasing the trust base, thus allowing to provide those strong guarantees to software running on this kind of hardware. Therefore virtual HSMs backed with trusted hardware seem increasingly promising because security and attestation are now guaranteed, and it would be much cheaper than purchasing physical HSMs.

In cloud computing, availability is one of the most important features offered [35]. However, some times, even 99% availability is not enough. Critical services or infrastructures, e.g., hospitals, banks, stock exchange, and many more, can not go down for a few seconds with the risk of losing lives and money [17]. Models such as the cloud-of-clouds [10] have been proposed to counter this issue, improving the availability of cloud services even higher. This model gathers multiple independent public cloud providers to replicate our data and perform computations. This model's primary goal is to increase availability; however, it can also be leveraged to improve reliability as individual clouds can be tampered or corrupted.

1.3 Objectives

In this dissertation, we aim to explore new hardware technologies for trusted computation and to develop a secure distributed cryptographic store. More specifically, we intended to create a virtualized and distributed HSM on the cloud, using Intel SGX-enabled nodes, for private key management in which availability, privacy, and reliability are crucial requirements. Regarding availability, we explored the cloud-of-clouds model, leveraging nodes equipped with trusted hardware. Secure and attested communication between those trusted components and data replication protocols were studied to ensure the desired behavior of such a system. HSMs store sensitive information, such as private keys that must always have the latest version available. That is why consistency management protocols were analyzed and, afterward, decided that we should implement a strong consistency protocol so that our solution can give guarantees that an outdated key will never be used.

Our objective was to develop a virtual HSM that could provide the same security guarantees as physical ones, that could be validated experimentally, with increased availability. We evaluated other contributions and made comparisons regarding the performance and security of our solution. Our final objective was that our virtual HSM could be used in testing and production environments.

Furthermore, with this dissertation, we aim at answering the following scientific question:

How can we provide the security guarantees of a physical HSM through commodity hardware while increasing availability and reducing financial costs?

1.4 Main Contributions

Our main contribution consists of a fully developed virtualized HSM supported by modern attestation-based trusted hardware in commodity CPUs to ensure privacy and reliability. We then extended this implementation to a distributed cloud version whereas every node has trusted hardware, and together, they form a trusted cloud-of-clouds. Finally, we executed performance benchmarks as well as extensive security evaluations comparing our solutions with previous virtualized solutions and a cloud-based HSM.

Our contributions can be summarized as follows:

1. A virtualized HSM supported by a node with Intel SGX. We use an encrypted persistent data storage outside the Enclave to preserve the HSM state even if it shuts down.

2. A distributed version of the virtualized HSM so that high availability requirements can be met with the support of multiple nodes with Intel SGX.

3. Extensive evaluation of the developed contributions. We analyzed performance and scalability benchmarks of both solutions as well as ensuring the security requirements of an HSM.

1.5 Document Structure

This document is divided into six chapters. Chapter 2 discusses the background related to this dissertation, including hardware security modules, hardware-backed trusted computing, and dependable cloud storage. At the end of each section, we will have a subsection dedicated to discussing its relevance for our work.

In Chapter 3, we talk about our first solution, a single Virtual HSM. We explain the system model and its architecture, the adversary model, the supported algorithms in our HSM and how to use it, i.e., its API. Then we get into implementation details, further explaining the most important parts and components, followed by an informal security analysis ending with a discussion. Chapter 4 describes the distributed version of Virtual HSM: differences in the system model, architecture, adversary model, and implementation. Ending with another informal security analysis and a final discussion.

Chapter 5 contains the test bench and the measurements done to evaluate our system performance. The chapter ends with a comparison against the related work and highlights the most important conclusions regarding our system performance. This thesis concludes in Chapter 6 with a conclusion and future work indications.

Chapter 2

Background

This chapter is organized as follows: the first section explains what is a Hardware Security Module and what it does (2.1), presents the standardized API for interacting with one (2.1.1) and analyses a couple examples of Virtual HSMs (2.1.2). On Section (2.2) we will explain what is Hardware-backed Trusted Computing, the differences between TEE and IEE (2.2.1) and what kind of technologies make use of them (2.2.2). Section (2.3) will explore the concept of Dependable Cloud Storage and discuss some approaches that have been made on that field: Cryptographic Cloud Storage (2.3.1), DepSky (2.3.2), CryptDB (2.3.3) and Cipherbase (2.3.4).

2.1 Hardware Security Module

A hardware security module (HSM)[25, 53] is a dedicated cryptographic processor that is designed specifically for the protection of cryptographic keys in all steps of their life cycles. Hardware security modules excel at securing cryptographic keys and provisioning encryption, decryption, authentication, and digital signing services for a wide range of applications. As this kind of hardware manages highly sensitive data, it is heavily regulated and certified to internationally recognized standards such as Common Criteria [18] or FIPS 140-2 [36]. It also has mechanisms to prevent, detect, and react against possible intrusions or tampering (e.g., it can wipe its storage when it detects tampering).

There are four levels of certifications that an HSM can achieve, according to FIPS 140-2:

- **Level 1:** It is the lowest security level; it only needs to use an externally tested algorithm or security function. It does not need to provide any physical security mechanism to prevent tampering.

- **Level 2:** In this level, there are physical security mechanisms that ensure physical tamper-evidence (e.g., coatings or seals that must be broken to attain physical access to the cryptographic keys) and role-based authentication.

- **Level 3:** There is now a requirement for tamper-resistance (e.g., it can destroy its content if tampering is detected) and identity-based authentication. There is also the need to separate between physical and logical interfaces by which critical security parameters enter and leave the module (e.g., private keys can only enter or leave it in an encrypted way).

- **Level 4:** The highest security level requires the module to be tamper-active. It means that it has even more extensive physical security guarantees with the intent of detecting and responding to all unauthorized attempts at physical access and any sign of tampering attempt has a very high probability of being detected. This level is the most relevant in situations the modules are in physically unprotected environments.

2.1.1 Public-Key Cryptographic Standards

PKCS comes from Public-Key Cryptography Standards [43] that gathers a group of cryptographic standards that provide guidelines and application programming interfaces (APIs) for the usage of cryptographic methods, emphasizing the usage of public-key cryptography. Specifically, PKCS#11 is a cryptographic token interface called Cryptoki [28] in which applications can address cryptographic devices as tokens, and it is used to communicate or access the cryptographic devices such as HSMs.

In PKCS#11 terminology, a token is a device that stores objects (e.g., Keys, Data and Certificates) and performs cryptographic operations. A device may have more than one tokens as they are just logical components, not physical ones.

Every object has its properties; for example, a key object has a "Key Type"property that differentiates from public, private, or secret key.

2.1.1.1 Functions

PKCS#11 has a wide range of functions [24] to allow applications to manage and enforce cryptographic operations. Some functions are related to Cryptoki in general, like initializing Cryptoki, get a list of slots and tokens available, get information related to a specific token, initialize or modify the regular user's PIN, open or close session and login or logout from a token.

Specific cryptographic functions are also available divided into some useful categories. Encryption and decryption functions (regardless if it is a single-part or multiple-part data), message digesting functions, signing functions (with signatures or Message Authentication Codes) and functions to verify those digests, signs, and MACs.

Table 2.1: Subset of PKCS#11 functions (taken and adapted from [24]).

Category	Function	Description
General Purpose	Initialize	Initializes Cryptoki
	Finalize	Clean up Cryptoki-associated resources
	GetFunctionList	Obtains Cryptoki functions
Slot and Token Management	GetSlotList	Obtains a list of existing slots
	GetSlotInfo	Obtains information about a particular slot
	GetTokenInfo	Obtains information about a particular token
	InitToken	Initializes a token
	InitPIN	Initializes the normal user's PIN
Session Management	OpenSession	Opens a connection between an application and a particular token
	CloseSession	Closes a session
	Login	Logs into a token
	Logout	Logs out from a token
Encryption and Decryption	Encrypt	Encrypts data
	Decrypt	Decrypts encrypted data
Message Digesting	Digest	Digests data
	DigestKey	Digests a key
Signing and MAC	Sign	Sign data
	Verify	Verifies a signature on data
Dual-purpose Cryptographic	DigestEncryptUpdate	Continues simultaneous digesting and encryption operations
	SignEncryptUpdate	Continues simultaneous signature and encryption operations
Key Management	GenerateKey	Generates a secret key
	GenerateKeyPair	Generates a public-key/private-key pair
	WrapKey	Encrypts a key
	UnwrapKey	Decrypts a key
	DeriveKey	Derives a key from a base key

There are also key management functions that allow us to generate a secret key, create a public/private key pair, encrypt and decrypt those generated keys, and to derive a key from a base key.

Cryptoki function calls are, in general, "atomic"operations (i.e., a function call accomplishes either its entire goal or nothing at all). If it executes successfully it returns an OK and if it does not execute successfully, it returns an error related value.

In Table 2.1, we have some of the essential functions of PKCS#11 API.

2.1.2 Virtual HSM

Physical Hardware Security Modules are costly solutions. Depending on the certification and security levels required, they can cost up to fifty thousand dollars [16]. That is why there have been some attempts to create virtualized HSMs to drastically reduce those costs while maintaining performance and security guarantees as high as possible. Virtual HSMs can not be used to meet the requirements of services in production level, but are nonetheless typically used for testing purposes.

2.1.2.1 SoftHSM

SoftHSM [37] is an open-source project developed by DNSSEC (DNS Security Extensions) in which they emulate an HSM for their clients who were incapable of buying such an expensive and specific hardware, resulting in a software implementation of a cryptographic store accessible through a PKCS#11 interface.

This project is important because it was one of the first attempts to virtualize an HSM, and it is still used today in testing environments.

2.1.2.2 Poor Man's HSM

Afterward, DNSSEC created Poor Man's Hardware Security Module (pmHSM) [16] to provide the same functionalities as SoftHSM (2.1.2.1) without being so expensive and prone to hardware fails as a physical one. pmHSM is based on threshold cryptography which improves security and availability of the system as it is a distributed solution.

pmHSM is extremely easy to adopt by already in use software as it presents the standard PKCS#11 interface to the outside, which is the API used to access the majority of physical HSMs.

Threshold cryptography [20] means that we need to gather a number of shares/parties to be able to use/sign something. For instance: K out of N threshold signature scheme is a protocol that allows any subset of K players out of N to generate a signature, but disallows the creation of a valid signature if fewer than K players participate in the protocol. This scheme means that we need to divide the secret key into K shares, and each share can be attributed to a different party to request to sign a document. Thus, we need to merge at least K signature shares to sign the document. There is also a public key associated with the N key shares that allows end-users to verify the document signature regardless of the set of signatures used.

This kind of cryptographic scheme improves security as the secret key has many shares, and each one of them is stored in a different node. It also enhances availability because it does not need all N shares of the key, having some margin for nodes that can fail in a byzantine way.

pmHSM is divided in three layers: application, manager and node (Figure 2.1). The application layer translates the signing function calls to the PKCS#11 API into messages sent to the manager. The manager is the one that deals with the requests that have been done by the application (e.g., signing, generate key-pair and a few more) and the one who waits until K of N nodes reply with their signature share (when it is a signing request).

Nodes are different parties that are subscribed to the manager to receive the documents to be signed; thus, having multiple nodes ensures reliability.

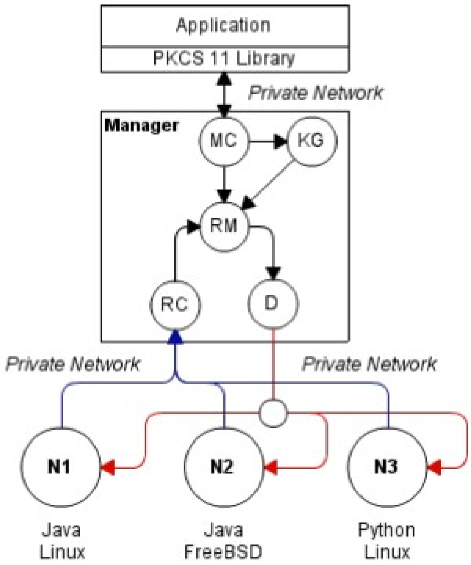


Figure 2.1: Poor Man's HSM architecture (taken from [16]).

2.1.2.3 AWS CloudHSM

AWS CloudHSM [6] is a cloud-based HSM that allows users to easily add secure key storage and high-performance cryptographic operations to their AWS applications. It uses the Amazon Virtual Private Cloud (VPC) which provides a logically isolated section where users can launch AWS resources in a virtual network defined by the end-user.

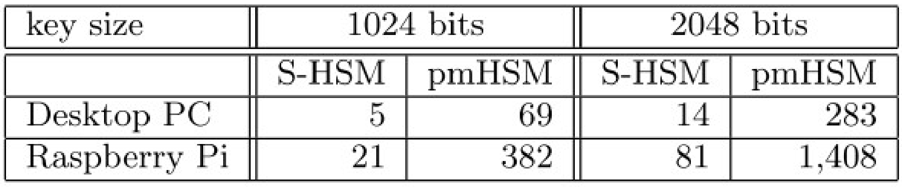
Amazon offers a scalable HSM solution where users can add and remove HSM nodes, on-demand, depending on their needs. It uses industry-standard APIs as PKCS#11 and JCE, and it is a FIPS 140-2 level 3 compliant HSM.

2.1.3 Discussion

Hardware Security Modules, as said above, are specialized cryptographic processors capable of performing thousands of signatures and cryptographic operations per second with a very high financial cost. For instance, Sun Crypto Accelerator 6000 [44, 47] is a well known HSM that cost around 1 300 dollars with a certification of FIPS 140-2 level 3 and can perform 13.000 signatures per second with 1024-bits keys. We are paying for performance and security certifications from single hardware prone to fails as it has been already documented with some incidents resulting in a 2-day lockdown and unreachable HSMs caused by some hardware fault [33, 51].

There are virtual HSMs like SoftHSM [37] and pmHSM [16] that try to face those giants but fail in some aspects. SoftHSM is faster at signing 1024-bits keys than pmHSM

(Figure 2.2) but it lacks availability. It runs on a single computer, without taking into account the availability and integrity of the hardware, software and every cryptographic material it produces and stores. pmHSM, as reported by the authors, can sign fifteen 1024-bits keys per second and, has a bit more robustness as it has multiple nodes that improve both availability and integrity. pmHSM ensures availability by working with threshold signatures, i.e., only K out of N nodes have to be available, and integrity is ensured because at least K nodes have to sign so that the process can go on.



key size	1024 bits		2048 bits	
	S-HSM	pmHSM	S-HSM	pmHSM
Desktop PC	5	69	14	283
Raspberry Pi	21	382	81	1,408

Figure 2.2: Time (in milliseconds) used by SoftHSM and pmHSM in the signing procedure (taken from [16]).

At the moment, there are no better virtual solutions for HSMs (as far as we know). Software implementations can not compete with the performance of highly dedicated processors, and there is still the need to have HSMs certified to internationally recognized standards (to use them in production environments) or at least to provide guarantees regarding security aspects ensured by those certifications.

Amazon offers a cloud-based HSM with auspicious features: tamper-resistant hardware with FIPS 140-2 level 3 certification (same used by physical HSMs); possibility to add and remove nodes when needed; uses industry-standard APIs like PKCS#11. Even though performance does not match that of physical HSMs, it is much higher than software implementations. Regarding financial costs, there are no upfront costs; users only pay an hourly fee for each HSM they launch. Although their solution seems very promising we have to ask ourselves a question: do we trust AWS? They have full control of the infrastructure and the HSM platform, which could be a problem because we have no way of knowing what happens inside their infrastructure and who has access to our sensitive information.

DNSSEC already discovered the key to success on pmHSM's article [16], the objective is not to beat the performance, but to achieve the same security guarantees given by physical HSMs using commodity hardware (or another type) lowering substantially the costs. Nonetheless, with this dissertation, we aim at providing the best performance and security guarantees possible, by leveraging trusted hardware and cryptographic acceleration implemented on commodity CPUs.

2.2 Hardware-backed Trusted Computing

Trusted computing refers to technologies and proposals for resolving computer security problems through hardware enhancements and associated software modifications. Those hardware enhancements, thus called Trusted Hardware [50], is hardware that is certified to meet a given set of security properties, according to Radu Sion [26]. Trusted hardware can ensure privacy, integrity, or attestation guarantees over the instructions it executes. Trusted computing had the objective of offering a way to outsource computations to the cloud with confidential guarantees [45].

Trusted Execution Environment (TEE) [21] is an isolated area on the processor of a device that is separated from the operating system. It ensures that data is stored, processed, and protected in a trusted environment. TEE gives confidentiality and integrity guarantees for code and data loaded inside it. This alongside-system is intended to be more secure than the classic system (called REE or Rich Execution Environment [27]) by using both hardware and software to protect data and code. TEEs are based on dedicated security chips and processor modes (e.g., TPM [52] and ARM TrustZone [4]) or emerging processor architectures, like Intel SGX [19], that allow the normal execution environment to create or activate a TEE instance when needed.

2.2.1 Isolated Execution Environment

Isolated Execution Environment (IEE) is an abstraction of a TEE provided by the authors [8]. Execution of a program in such an isolated way that its initial state and a set of fully known and well-defined inputs given into that isolated environment determine the program's output. IEE ensures full isolation from other applications running on the same machine, i.e., provides privacy against untrusted programs on that machine, protecting both data and computations.

Attested computing is needed and introduced to allow outsourcing of programs running on IEEs to assure that a program is running without being tampered on a remote IEE. The authors [8] define attestation as a set of three algorithms: Compile, Attest, and Verify. Compile is the first verification and attestation of the program; Attest runs on the remote machine and interacts with the program on IEE; Verify is run by the user to check the validity of the output from Attest.

Every communication between the user and the program running on the remote IEE is done by a secure communication channel to guarantee integrity and confidentiality.

2.2.2 Technologies

In the subsections below, we will explore and discuss the core model of trusted hardware (Trusted Platform Module (TPM)) and the most relevant solutions for TEE (Intel SGX and ARM TrustZone).

2.2.2.1 Trusted Platform Module

A TPM [52] is a specification for a hardware module incorporated and deployed in a motherboard aimed at providing a Trusted Computing Base (TCB). These chips can provide boot authentication, software attestation, and encryption. TPMs are considered to be trustworthy and tamper-proof, thus extending the trust base into software attested by them.

An authenticated boot service allows verifying that a machine's operating system has not been tampered with, i.e., that the boot of the system is made in well known and defined stages. It produces hashes of the software modules being loaded, storing those, and comparing them with predetermined ones. If hashes do not match, it will abort the boot process.

Figure 2.3 shows us the different components of a TPM: cryptographic processor, persistent memory, and versatile memory. Persistent memory is used to store encryption keys, predetermined hashes, and other records that need to be maintained and untouchable. Versatile memory is used to store information that keeps changing, e.g., those hashes calculated at boot authentication service, stored into internal registers - Platform Configuration Registers (PCRs).

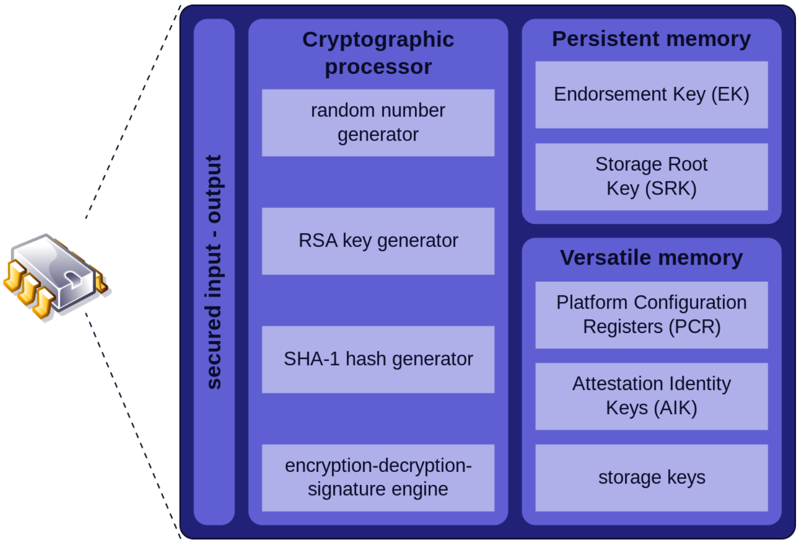

secured input - output
Cryptographic processor
random number generator
RSA key generator
SHA-1 hash generator
encryption-decryption-signature engine
Persistent memory
Endorsement Key (EK)
Storage Root Key (SRK)
Versatile memory
Platform Configuration Registers (PCR)
Attestation Identity Keys (AIK)
storage keys

Figure 2.3: Components of a Trusted Platform Module (taken from [7]).

A software attestation service allows to verify that the contents and interaction of some software, executed by trusted hardware, is as expected, i.e., TPM can verify the loading sequence of a program by producing a hash of the program's code (before execution) and storing it in a PCR. The user has the expected PCR values and, when needed, can request attestation from the TPM by sending a nonce and required PCR values. Then the TPM produces signed attestation proof that the user can verify against the expected values. If they do not match, it means that something has changed in the code or the software behavior and it will abort all computations.

Encryption service is ensured by a secure co-processor (cryptographic processor) that can perform cryptographic operations (e.g., random number generation, key generation, hash generation, encryption, decryption and signing [Figure 2.3]). Every key is stored inside the TPM, which guarantees that they can not be stolen or exported to the outside and that data can only be unsealed (decrypted) on that machine.

As TPMs are capable of boot authentication and attestation of the software stack that runs on the TPM enabled machine, they can ensure that a TEE with processing, memory, and storage capabilities is secure.

2.2.2.2 Intel SGX

Intel Software Guard Extension (SGX) is a new set of instructions available on recent Intel CPUs. These instructions allow the programmer to define enclaves (IEEs) that are like containers where code is secured and can run with confidentiality and integrity guarantees provided by the hardware. Those enclaves are physical memory regions isolated from the rest of the system, including the operating system and hypervisor.

A programmer that is designing an application under the SGX software model must know that there are two different components: an untrusted and a trusted one, the latter being the enclave. Whenever the untrusted part requests the creation of an enclave, the CPU allocates a region of the memory that is only accessible to that enclave. That region is completely inaccessible from any other application or component in that machine as memory is kept encrypted and it is only decrypted by the CPU when requested by the enclave. Those enclaves have a limitation of 128 MB of memory.

Intel SGX has some special instructions:

- ECALL: a call from an untrusted component or application into an enclave
- OCALL: a call from inside an enclave to an untrusted function

The second call is needed because ECALLs have no access to the exterior of the enclave, e.g., system libraries, networking, disk I/O and many more. Whenever OCALLs are performed the enclave is temporarily exited, the untrusted function is performed, and execution goes back to the enclave. These calls are defined through a specification file (Enclave Definition Language (EDL)) where it is clear what functions can be issued from trusted components (OCALLs) and untrusted components (ECALLs).

Processor Reserved Memory (PRM) is an isolated region of memory where all enclaves' data structures, code, and data are and can not be accessed by other software [34]. It also contains the Enclave Protected Cache (EPC) where some protected memory pages can be assigned to enclaves. In Table 2.2 we present thirteen of the eighteen most relevant SGX's instructions.

Data integrity checks, attestation, sealing, debugging, and initialization can be done with those enclave operations. For example, to create an enclave, we start by issuing EECREATE instruction, which allocates the needed EPC pages and initializes SGX's enclave data structures. Then, if we need to add pages to the enclave, we can call the EADD instruction. When everything is ready, we can call EINIT instruction to end the initialization process and start the enclave. Afterward calls into an enclave function (ECALLs) are performed by the EENTER instruction, exiting enclave with EEXIT instruction.

Table 2.2: Subset of Intel SGX instructions (taken from [12, 34])

Category	Instruction	Description
Initialization	EECREATE	Declare an enclave, its size and initialize data structures
	EADD	Add a 4KB EPC page to the enclave
	EREMOVE	Remove an EPC page from the enclave
	EINIT	Finalize the enclave initialization, enabling its execution
Entry and Exit	EENTER	Enter an enclave via an ECALL
	EEXIT	Exit the enclave
	AEX	Asynchronous enclave exit, saving the CPU state
	ERESUME	Resume enclave execution, restoring previous CPU state
Paging Management	EBLOCK	Block an enclave page, prepare for eviction from protected memory
	EWB	Evict a page from protected memory
	ELDB/U	Reload an evicted page into protected memory
Enclave Security	EREPORT	Produce a report with information about the enclave and the hardware it is running on
	EGETKEY	Generate a unique key

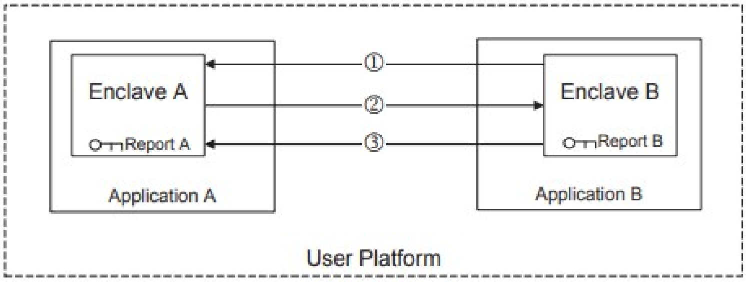
As we have seen, memory inside PRM is limited, so it has a mechanism to swap pages from PRM to untrusted memory and back. Those mechanisms consist of "Entry and Exit"category of Table 2.2: EBLOCK, EWB, and ELDB/U instructions. The first prepares an enclave page for eviction, i.e., blocking writes and encrypting the page. EWB evicts the page, now encrypted, from PRM to untrusted memory. Lastly, to reload a page, an ELDB/U call copies the evicted page content into the PRM, decrypts it and runs security checks. Evicted pages are stored with integrity proofs (MACs and version number) to guarantee it was not tampered and ensuring page's version is the last one, thus preventing replaying-like attacks.

Local attestation (Figure 2.4a):

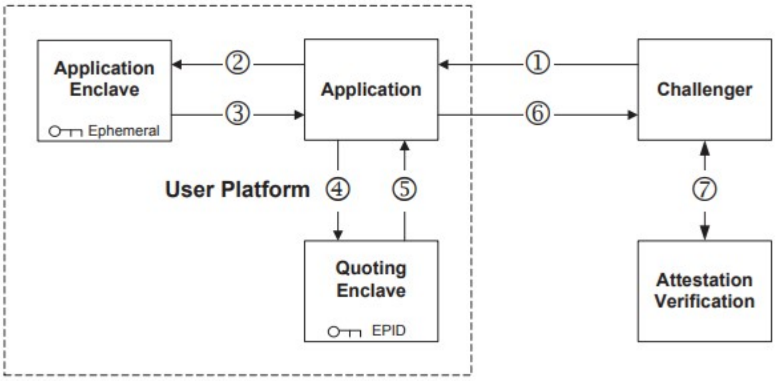
1. Enclave B requests attestation to enclave A.

2. Enclave A produces a report (calling EREPORT instruction) containing its identity, attributes and information about the hardware it is running on. Then it calculates a MAC for that report using a symmetric key (owned by the CPU and given to the enclave by the EGETKEY instruction). Finally sends the report to B.

3. Enclave B can use the same symmetric key (local attestation, thus both enclaves share the CPU) to generate a report of A and calculate its MAC comparing with the


Enclave A
Report A
Application A
①
②
③
Enclave B
Report B
Application B
User Platform

a Local attestation


Application Enclave
Ephemeral
②
③
Application
①
⑥
Challenger
User Platform
④
⑤
⑦
Quoting Enclave
EPID
Attestation Verification

b Remote attestation (e.g. enclave running on the cloud)

Figure 2.4: Intel SGX mechanisms for local and remote attestations (taken from [2]).

one received. If they are running on the same machine, MACs should check, and B reciprocates the attestation process and sends it to A its report.

Remote attestation (Figure 2.4b):

1. A challenger requests attestation to the untrusted application A.

2,3. Application A requests a report from its enclave, which creates it with the same information and the same way as in the local attestation. Sending it back to the application.

4. The application now sends the report to a special enclave (Quoting Enclave).

5. This special enclave signs the report with an EPID key (asymmetric key) producing a *Quote* and returning it to the application.

6. The application acts as an intermediary for communication and sends the *Quote* to the challenger.

7. The remote user (challenger) must have a public key certificate of that EPID that he can use to verify the *Quote* from SGX.

As the enclave size is limited, it has another feature called data sealing, i.e., the enclave encrypts data (with all the security guarantees it provides) and then stores it outside the enclave boundaries. This key (sealing key) is requested via the EGETKEY instruction and is unique to that enclave, meaning that no one else could ever use that key.

2.2.2.3 ARM TrustZone

ARM TrustZone is the security extension made to their System-on-Chip (SoC) that provide a hardware-level isolation between two execution domains defined by ARM: *Normal World* and *Secure World* [4, 46]. As these domains are isolated they have independent memory addresses and privileges: from the *Normal World* it is impossible to access the *Secure World*, while programs running in *Secure World* can access *Normal World*.

Transitions between these two worlds are assured using the secure-monitor call (SMC) instruction (Figure 2.5) that saves all registers and protects sensitive memory before switching from one world to another. Both worlds exist in the same physical core, but they are seen as independent virtual cores. As in Intel SGX, ARM TrustZone *Secure World* provides a full IEE for untrusted applications.

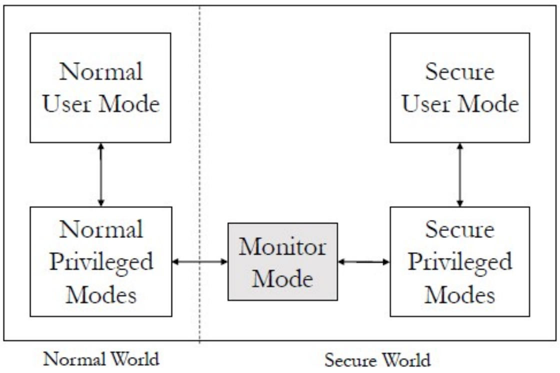


Figure 2.5: ARM TrustZone architecture (taken from [12]).

This new CPU privileged mode supports SMC, called monitor mode, and can be compared to the SGX's instructions EEXIT and AEX. Contrary to Intel's SGX there is no programming model imposed by ARM, and there are a few ways to handle *Secure World* software stack. The simplest one is, like Intel's enclaves, that applications running from *Normal World* that need to isolate some computations to make synchronous calls into the *Secure World* returning to *Normal World* after completing them. Another, more complex, way is to run the whole OS Kernel inside the *Secure World* relying on multiple applications running in parallel inside it.

As Figure 2.5 shows, TrustZone's architecture is divided into two virtual cores, one for each world, and several duplicated hardware modules to ensure physical isolation. Regarding memory management, there are two physical Memory Management Units (MMUs) that allow a completely independent memory mapping and management. On the other hand, cache shares the same physical housing, but an additional bit is indicating the security state (whether it is being used in a secure or normal world). TrustZone also has a separate bus to secure peripheral access, i.e., to I/O and network devices, guaranteeing that a secure peripheral can not be accessed from the *Normal World*.

ARM's TrustZone is being more used and exploited in mobile devices than for cloud applications because it has been present in mobile architectures for over a decade, and it is providing trusted execution of applications and preventing mobile OS security breaches [21, 46].

2.2.3 Discussion

These technologies are neither perfect nor bulletproof. TPM's, for instance, can suffer from a few well-known attacks. One of them is called "software attack"and targets the attestation service, i.e., given an application already configured in the TPM adversaries, it can exploit the difference between the instant when an application is loaded and when it is used to introduce run time vulnerabilities. This attack is called Time Of Check to Time Of Use (TOCTUO), and a solution was proposed [14]: dynamically updating PCR values when the memory it measures is changed.

TPMs have some additional vulnerabilities that are worth mentioning. It is possible to conduct a passive attack where we analyze the unencrypted data bus between the TPM and the CPU [30], and an active attack which saves the record of the trusted boot process forces the reset of the TPM and utilize a previous trusted record in the malicious system. The last is called TPM reset attack [49].

As authenticated boot service attests to the integrity of the operating system, it means that no update can be done to that operating system after its initialization without resetting all boot attestation records. Also, cryptographic keys are heavily secured inside the TPM, but there is no solution to recover sealed documents if something happens to the TPM and those keys become inaccessible.

Intel SGX considers, aside from the CPU, both software and hardware to be untrusted, which eliminates many attacks from its threat model but it remains vulnerable to some attacks: privileged software and side-channel attacks. The first is related to the ability to support hyper-threading by the CPU, i.e., logical processors (LP) share the same physical processor and its resources. It means that a malicious system can execute code in a thread running on an enclave and execute, in another thread that shares the same physical processor, some snooping program that allows this thread to learn the instructions executed by the enclave and its memory access patterns.

Side-channel is another type of attack that SGX does not provide any protection. It is related to SGX's handling mechanism and page faults that allow the malicious operating system to take full control of the enclave's execution by stopping the program, unmapping the target's memory pages and resuming execution [13]. There is already a Transaction Synchronization Extension (TSX) provided by Intel to mitigate this attack [48]. Intel SGX is also vulnerable to speculative execution like spectre and meltdown, and specific for SGX, foreshadow [54], where the processor predicts computations to achieve better performance, but attackers can explore this and get enclave's secrets.

ARM's TrustZone is a more general notion to the programmer and can be configured to provide countermeasures for different threats. Regardless of those configurations it is vulnerable to an attack resulting from the unauthenticated access to resources in TrustZone, which makes the communication channel between *Secure World* and *Normal World* susceptible to anyone who has kernel privileges in the *Normal World* mode. This specific attack is made by continuously sending requests with specific arguments to try to find vulnerabilities on the resources in TrustZone. Like this, a man-in-the-middle attack is also possible by exploring the communication channel. Some authors proposed the creation of a secure channel for cross-world communication [27].

TrustZone increases the broader utilization of an IEE and secure communication between TrustZone's components with the cost of programming complexity and the availability of trusted firmware. With the inclusion of hardware other than CPU into TrustZone's trust base, we are putting at risk every other component if a vulnerability is found in one of them. Contrary to what happens with SGX's policy that is much more restrictive but offers a more secure trust base.

2.3 Dependable Cloud Storage

Cloud Storage is a cloud computing model in which data is stored on remote servers accessed from the internet. It is maintained, operated, and managed by a cloud storage provider. The benefits of using such storage are well known: high availability and reliability at a relatively low cost. The problem is that there are many security and privacy concerns as data stored on those cloud providers go out of our control which inevitably poses new security risks toward the correctness of our data in the cloud.

To address this new problem and further achieve secure and dependable cloud storage, we need to understand how existing solutions try to solve those concerns over the confidentiality and integrity of data.

2.3.1 Cryptographic Cloud Storage

Cryptographic Cloud Storage [29] tries to leverage the benefits of public cloud services while introducing both confidentiality and integrity, resulting in a virtual private storage service.

The architecture described is divided into four main components:

- Data Processor (DP) which is responsible for processing data before it is sent to the cloud, i.e., it creates metadata to that specific data, joins it with the original data and finally encrypts both data and metadata.

- Data Verifier (DV) which can be used at any moment and verifies the integrity of the data stored in the cloud.

- Token Generator (TG) which is responsible for the creation of temporary tokens that are related to the segment of customer data we want to retrieve from cloud.
- Credential Generator (CG) that implements an access control policy issuing credentials to other parties to be able to decrypt encrypted data.

If it is needed, DP can add a decrypt policy that only allows certain data to be decrypted by an authorized party or team.

There are concerns related to public cloud services that can be avoided when using this kind of virtual private storage service as it is the client who controls data and has certain security properties (e.g., confidentiality). Those concerns can be:

- Regulatory compliance[41]: certain public cloud providers can be forced to deliver data to law enforcement agencies, which is not a problem as all data is encrypted.
- The same applies if the public cloud provider must follow certain rules against the data it stores and its geographical location [56]. As all data is encrypted, it is impossible to distinguish it.
- Clients must be notified and cooperate if data has to be analyzed (again, if law enforcement agencies force it, they need the client to decrypt data) [57].
- In a public provider it is nearly impossible to guarantee that data has been fully eliminated but, in this case, it is only needed to eliminate the key and then it is impossible to access that data again.

Whenever DP prepares data to sent to the cloud, it starts by indexing and encrypting it with a symmetric encryption scheme, like AES, with a unique key. Then it will encrypt again but with a searchable encryption scheme [9] and encrypt that unique key with an attribute-based encryption scheme [11]. At last, it encodes the encrypted data and indexes it in such a way that the DV can, at any moment, verify the data integrity with proof of storage scheme [5].

The searchable encryption scheme provides a way to encrypt a search index so that contents are hidden except to a party that has the appropriate tokens and can search without the need to decrypt the data. This technique has two important properties: first is that given a token for a keyword it can only retrieve pointers to the encrypted files that contain the keyword; second is that without a token the contents of the index are hidden.

An attribute-based encryption scheme allows associating a decrypt policy with the ciphertext. All decrypt keys existing in the system have a set of attributes, and so, it is only possible to decrypt specific files if the key used has some attributes that match the decryption policy of that file.

Proofs of storage protocols are executed by the client to request the server to prove that it did not tamper the files it is storing. They can be used an arbitrary number of times and data resulting from that proof is extremely small and independent from the size of the data.

Whenever Alice wants to upload data to the cloud her DP attaches some metadata and encrypts data and metadata as previously explained (Figure 2.6 [1]). If Bob wants to access data from the cloud he has to ask Alice for permission to search for a keyword (Figure 2.6 [2]) and Alice's token and credential generators send back a token for the keyword and a credential to Bob (Figure 2.6 [3]). Now Bob can send the token he received to the cloud (Figure 2.6 [4]) and the cloud provider can use the token to find the right encrypted documents and return them to Bob (Figure 2.6 [5]). At any time Alice's DV can be invoked and ascertain the integrity of the data (Figure 2.6 [?]).

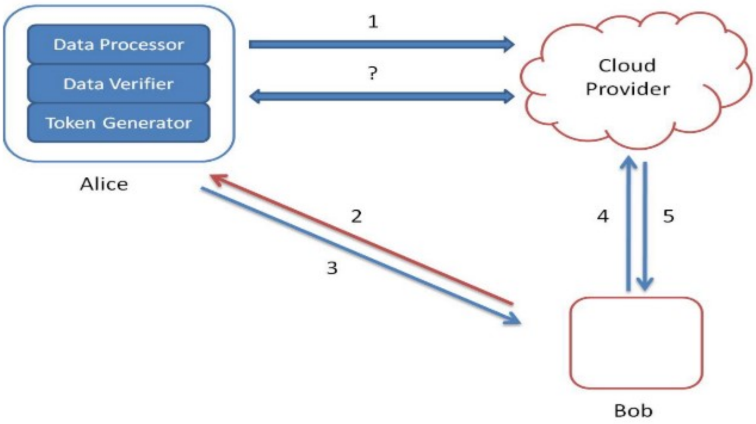


Figure 2.6: Cryptographic Cloud Storage architecture (taken from [29]).

2.3.2 DepSky

DepSky [10] is a system that improves the availability, integrity and confidentiality [38] of information stored in the cloud through the encryption, encoding, and replication of data on several clouds that form a cloud-of-clouds.

There are four important limitations of using a single, commercial cloud computing for data storage:

- Loss of availability is an inevitable problem that a user or company can suffer when moving to cloud storage. DepSky solves this problem by replicating data and storing it on different cloud providers, thus the above name: cloud-of-clouds. All the data will be available if a subset of them are reachable [1].

- DepSky addresses loss and corruption of data by using Byzantine fault-tolerant replication protocols [32] for storing data on those different cloud providers, being possible to get the data correctly even if some of the clouds corrupt or lose data.

- Loss of privacy is another massive concern as the cloud provider has access to all the stored data, even if the provider is 100% trustworthy, it can be infected by some malicious software that exposes everything we stored. One of the possibilities is to encrypt everything before storing it in the cloud, which involves protocols for key distribution as different distributed applications may need access. DepSky uses a secret sharing scheme [15] to ensure that all the data is stored after being encrypted and that it can be decrypted with one or a set of keys from that secret sharing scheme.

- As it uses several cloud providers, we will not be "locked-in"with a particular cloud provider. Also, as DepSky uses erasure codes [42] to store only a fraction (usually half) of the data in each cloud, it would only cost half to exchange from a cloud provider to another.

As pointed by the authors, DepSky would be very beneficial to services like critical data storage as the cloud-of-clouds would guarantee very high availability and integrity, besides, DepSky-CA protocol ensures the confidentiality of stored data. Those guarantees are some of the most important to build, e.g., a cryptographic cloud store.

To deal with the heterogeneity of using different clouds, DepSky assumes there is a generic Data Unit or container that contains metadata and files (real data). Metadata files contain a version number, a verification number and some information that may be application dependent. The files are named as "value-<version>".

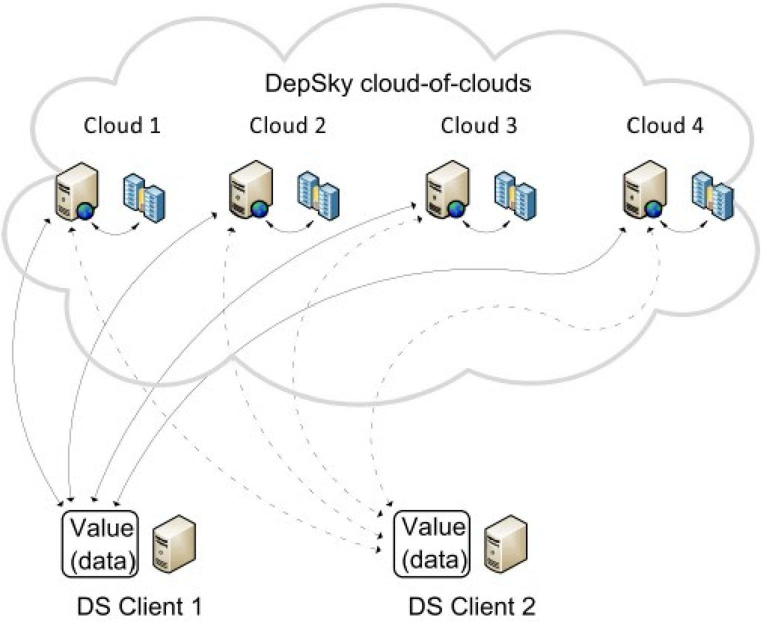


Figure 2.7: DepSky architecture (taken from [10]).

The system proposed by DepSky (Figure 2.7) models three parties: writers, readers, and cloud storage providers. All writers of a Data Unit have a shared private key used to sign some of the data written on the data unit, while readers of a Data Unit know the corresponding public key to verify these signatures.

Cloud storage providers only support five operations: list, get, create, put, and remove. DepSky assumes that it is a cloud provider's job only to allow readers to invoke list and get operations (i.e., access control). As clouds can fail in a Byzantine way, DepSky requires a set of N=3F+1 storage clouds where at most F clouds can fail. Thus, we need any subset of N-F storage clouds to gather a quorum. DepSky has implemented a single-writer multi-reader register to overcome the problem of having multiple writers and the inability to verify different version numbers being written concurrently.

DepSky protocols always need at least two round-trips: to read or write from metadata and to read or write the actual files, having to acknowledge whenever one of those reads or writes are done correctly. It is also consistency-proportional, i.e., its consistency is as strong as the underlying clouds allow, from eventual [55] to regular consistency [31].

As pointed early, to ensure confidentiality, data is encrypted using symmetric cryptography with a secret sharing scheme for that symmetric key. In this case, the writer splits the symmetric key into N (number of clouds) different shares of that key and give one share to each of the clouds. The main advantage of this scheme is that at least F+1 <= N-F, which means that even if F shares are corrupted, the secret can never be disclosed. It also means that a client that is authorized to access data needs at least F+1 shares of the key to be able to rebuild it and decrypt the data.

DepSky also uses a protocol named informational-optimal erasure code to avoid the monetary costs of replicating all the data on every cloud provider, which would increase by a factor of N.

How does DepSky-A work? DepSky-A only ensures availability and integrity by replicating data on several clouds using specific quorum protocols. The idea is that whenever a write must occur, it writes the value on a quorum of clouds and then writes its corresponding metadata to ensure that when a read operation is issued, it can only read metadata from data already existing on that cloud. If it is the first write of that Data Unit, the writer must first contact a quorum of clouds to obtain the metadata with the highest version number. Whenever a read is issued it starts by gathering metadata from a quorum of clouds and choosing the one with the highest version number, then it searches for a value file containing the value matching the digest on the metadata.

Availability is guaranteed because the read operation has to retrieve the value from only one of the clouds, which is always available because of (N-F)-F > 1. Integrity is also guaranteed because metadata contains a cryptographic hash being easily verifiable if data was corrupted or tampered.

As DepSky-A only ensures availability and integrity, so the authors also developed DepSky-CA (Figure 2.8) that joins confidentiality to the previous protocol. The differences between these two protocols are that whenever DepSky-CA requests a write, it encrypts the data, generates the key shares, divides information, and those key shares for each cloud. On reading requests, it is the reverse process: joins data and shares of the key, rebuild the key and decrypt data. As this protocol uses a key share scheme, it needs that at least F+1 clouds reply with their key share whenever a read request is made to read the Data Unit's current value properly.

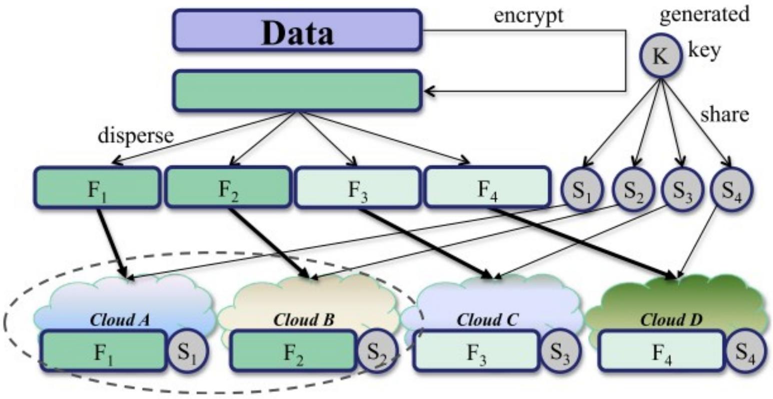

Data
encrypt
generated
K
key
share
disperse
F_1
F_2
F_3
F_4
S_1
S_2
S_3
S_4
Cloud A
Cloud B
Cloud C
Cloud D

Figure 2.8: DepSky-CA (taken from [10]).

2.3.3 CryptDB

CryptDB [40] ensures confidentiality for applications that use database management systems (DBMS) by exploring its intermediate design point as middleware and by enabling the execution of SQL queries over encrypted data.

There are two threats solved by CryptDB:

1. A curious database administrator (DBA) who tries to learn private data;

2. An adversary who gains complete control of application and DBMS servers.

CryptDB acts like a proxy, where all the queries go through, and encrypts, decrypts and rewrite queries so they can be executed on encrypted data. Therefore, the DBMS server never receives decryption keys, and so DBAs could never see private data.

Regarding threat number two CryptDB divides SQL schema between different principals (users) whose keys allow decrypting different parts of the database. This process makes it impossible for attackers that compromise the whole system (application or proxy or DBMS) to be able to decrypt or read data from users that are not logged-in, without providing the same guarantees for those who are.

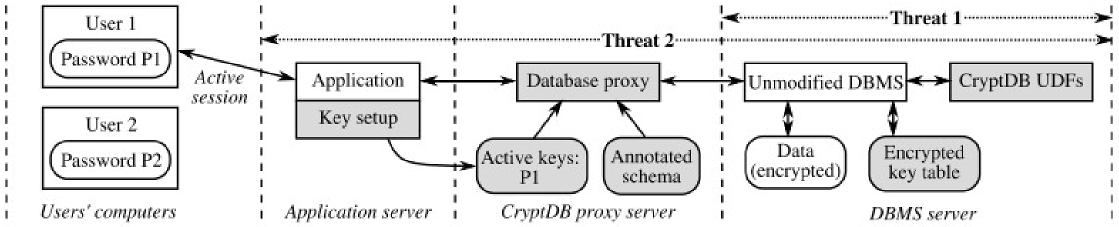

User 1
Password P1
User 2
Password P2
Users' computers
Active session
Application
Key setup
Application server
Threat 2
Threat 1
Database proxy
Active keys: P1
Annotated schema
CryptDB proxy server
Unmodified DBMS
CryptDB UDFs
Data (encrypted)
Encrypted key table
DBMS server

Figure 2.9: CryptDB architecture (taken from [40]).

2.3.4 Cipherbase

Cipherbase [3] is a full-fledged SQL database system that achieves high performance on cloud computations nevertheless ensures confidentiality by storing and processing strongly encrypted data on specially designated secure nodes. Those nodes can be some secure co-processors (SCPs) available at the server-side, instead of creating some middleware between client and server, to manage computations between a Trusted Machine (TM) and an Untrusted Machine (UM) that are side by side.

On the original paper, the authors leveraged a Field Programmable Gate Array (FPGA) and placed it inside the UM connected to the TM, to reproduce that fully-fledged database system.

Whenever Cipherbase receives a query plan, it sends the encrypted tuples from the UM to the TM where they are decrypted, processed, and re-encrypted before sending the results back from the TM to the UM.

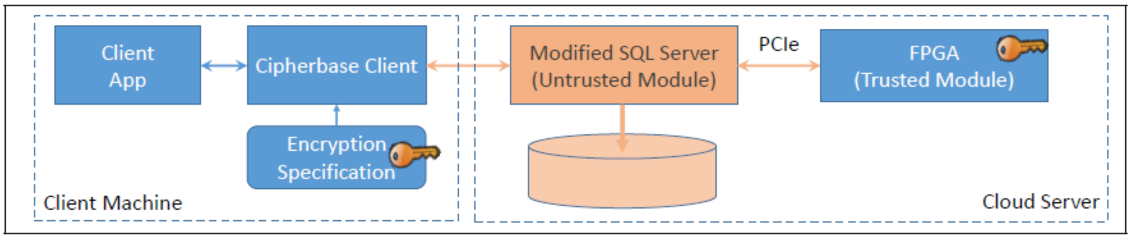


Figure 2.10: Cipherbase architecture (taken from [3]).

2.3.5 Discussion

CryptDB and Cipherbase are compelling solutions that achieve high performance on encrypted computations, ensuring confidentiality. However, we are aiming at storage solutions that do not need to perform computations nor searches over encrypted data.

Cryptographic Cloud Storage is a solution that offers a virtual private storage service, i.e., providing the security of private clouds and the functionality and costs of public clouds. It means that all the perks of using a public cloud provider are maintained: availability, reliability, and data sharing, and also, there are confidentiality and integrity guarantees.

To ensure confidentiality, it encrypts data with symmetric encryption, encrypts it again with searchable symmetric encryption and finally encrypts the key with attribute-based encryption. Even though it is a highly confidential and secure storage solution, it introduces a considerable overhead in the system, and we do not need to search over encrypted data or to have a decrypt policy associated with the ciphertext stored.

DepSky puts in practice the notion of cloud-of-clouds: using multiple independent public cloud providers in the same storage system to increase availability. It uses byzantine fault-tolerant replication protocols on those different cloud providers to allow them to get data correctly even if there is some corrupted cloud or data. DepSky-CA protocol ensures the confidentiality of stored data by encrypting data and divide encrypted data and symmetric key into N shares (being N the number of clouds in use). Then each cloud provider receives their share of the data and the key itself. As it uses quorum protocols, it means that when a read is issued, DepSky does not need to get data or key shares from every cloud, just from F+1 clouds (being F the number of faulty clouds) to be able to rebuild the key and decrypt data.

With the cloud-of-clouds concept, a new concern arises regarding consistency. DepSky offers a consistency-proportional solution where the system provides the client with the same level of consistency of the underlying clouds. Integrity guarantees are provided by metadata which is stored together with the data and contains a cryptographic hash that can be verified to guarantee that data has not been tampered with.

A critical aspect that could be raised is related to costs that this system can add to cloud storage. Therefore, DepSky employs erasure codes that allow storing only a fraction of the data in each cloud. It also optimized read and write quorums achieving an impressive increase cost of only 23% compared with the average cost of storing data on a single cloud (experimental evaluation performed by DepSky authors with four different clouds [29]).

Despite all the advantages that each one of these solutions has, no solution has all the security guarantees with high availability required. Regarding security guarantees, the isolation and remote attestation required is only possible by using trusted hardware (IEE) that, apart from Cipherbase, none of the solutions use it. However, Cipherbase does not provide other necessary guarantees such as high availability given by the replication in multiple clouds.

CHAPTER 3

VIRTUAL HSM

In this chapter, we introduce our Virtual HSM that combines traditional cryptographic schemes and key generation with trusted hardware, particularly an Isolated Execution Environment (IEE) component. This component allows us to securely execute computations on untrusted machines, like a Cloud environment, thus leveraging its advantages, such as high computational power and large persistent storage.

State-of-the-art Virtual HSMs usually find a trade-off between compromising some security guarantees, or even not having any security guarantees, (e.g., SoftHSM) to ensure a better performance, or choose to implement some security guarantees (e.g., pmHSM) incurring on a significant decrease of performance while not being as secure as a physical HSM should be. Using an IEE allows us to efficiently outsource computations to otherwise untrusted Cloud servers, while still preserving strong security guarantees.

This chapter is organized as following: firstly we introduce the system model, it's architecture (Section 3.1) and the adversary model (Section 3.2), we go through every algorithm implemented (Section 3.3), then we explain our API and how to use it (Section 3.4) and discuss the implementation details (Section 3.5). Finally we provide a security analysis about our solution (Section 3.6) and finish this chapter with conclusions regarding our Virtual HSM (Section 3.7).

3.1 System Model and Architecture

In this chapter, we consider a system with a single client interacting with a single IEE-enabled cloud server. Inside of this cloud server, there are both trusted and untrusted components that, combined, are what we called our Virtual HSM:

- **Client** is a simple application where it creates the requests made by the end-user and pre-processes them so that they can be sent to Server.

- **Server** is just used for initializing the IEE, mediating communications between Client and IEE and run every I/O operations that can not be issued by the IEE.
- **IEE** executes every computation needed by an HSM with strong guarantees regarding confidentiality, integrity, and authentication.
- **Persistent Storage** is one of the needed untrusted components because of the well-known memory limitations (128 MB) of SGX Enclaves. It is used to store the state of our Virtual HSM with a strong encryption scheme as it will be explained later.

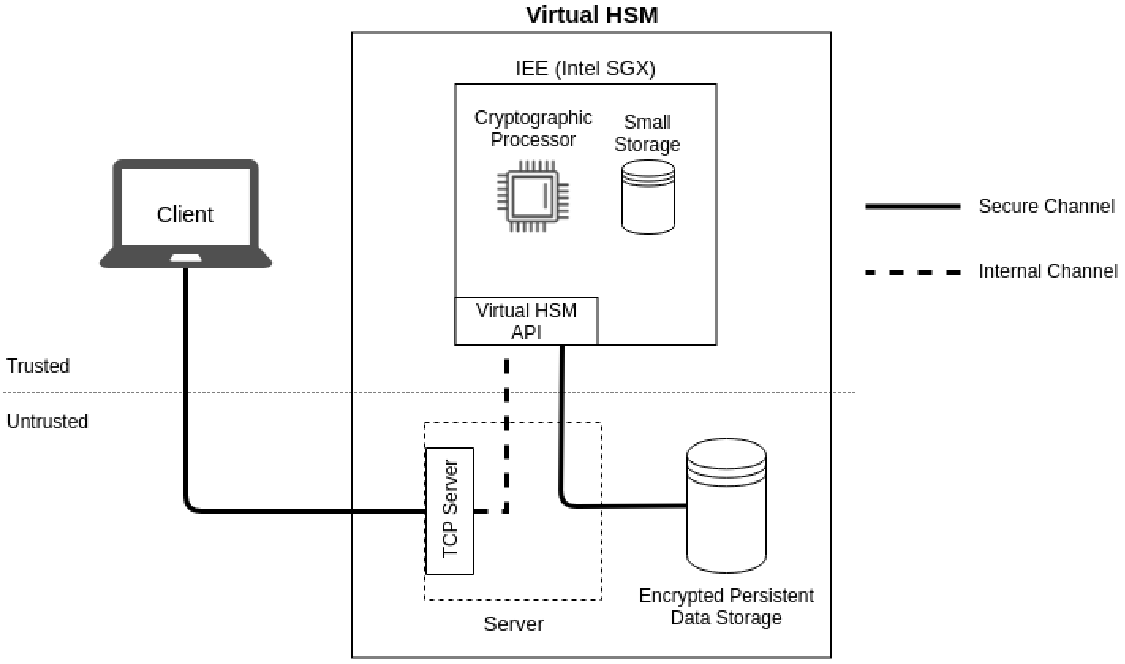

Virtual HSM
IEE (Intel SGX)
Cryptographic Processor
Small Storage
Client
Secure Channel
Internal Channel
Virtual HSM API
Trusted
Untrusted
TCP Server
Server
Encrypted Persistent Data Storage

Figure 3.1: System architecture. Server acts as a proxy to securely connect Client with IEE. Persistent Storage is loaded by the Server and sent to IEE, still encrypted.

On Figure 3.1 we present the architecture of our system. Our Client is considered one of the trusted components of the system, and any device can play this role as it is a thin application that only uses a couple of cryptographic operations for secure communications.

On the Server side, we have the other three components. The Server itself does not add much to the system. It is an untrusted component where its primary function is to run I/O operations and mediating communications between Client and IEE (as explained above). It is also the entry point of our cloud server as it is deployed in the cloud and has a network interface exposed to receive outside connections.

IEE, also called Enclave, is also running in our cloud server, which is an SGX-enabled node, but it is not accessible from the outside. Only the Server can communicate directly with the Enclave and vice versa. Lastly, our untrusted but persistent storage is part of our cloud server and stores the state of our HSM (every key generated) outside of the

Enclave, meaning that whenever the Enclave wants to restore or save the HSM's state, it needs to ask the Server to read/write operations. Everything written by the Enclave on persistent storage is fully encrypted with a scheme that allows verifying data integrity anytime needed and act accordingly with the policy used whenever tamper is detected.

In summary, our Server, Enclave, and Storage are on the cloud, deployed in a single machine, and the Client can either be on the same machine or a different one, with our only requirement being that they have network communication. Communication between client devices and the Enclave is performed through a secure channel abstraction with privacy, integrity, and authentication guarantees, instantiated through an attestation-based key exchange protocol described in Section 3.5.2. It is where one of the main functions of the Server appears: mediating communications. Even being an untrusted component, the underlying security guarantees that are given by the key exchange protocol allow us to create a secure channel from the Client to the Enclave, using the Server as a proxy.

3.2 Adversary Model

In our solution, we will address a fully-malicious attacker who can perform both passive and active attacks. By passive attacks, it means that it can get information about data that goes to and from our trusted component or data that is stored in untrusted persistent storage, which means snooping data both in transit and at rest while active attacks involve tampering with data and computations.

This kind of adversary model is primarily threatening the confidentiality and integrity assumptions of users running critical computations such as processing, data access, and key management. Our solution will preserve full privacy and integrity guarantees so that we can prevent those passive and active attacks. Regarding availability, we assume that the node is always available, meaning that we are not focused on attacks against the availability of computing or storage services, or any related Denial-of-Service attacks. In the next chapter (Chapter 4) we will discuss some system modifications that will significantly increase availability, thus acting somewhat as a counter-measure of this last attack.

The whole system is based on the premise that both the Client and the Enclave are secure and trusted. Everything else is assumed to be insecure; this includes network communications and data storage.

3.3 Supported Algorithms

In this Section, we describe every algorithm implemented on our Virtual HSM. First we started by analyzing which library we would use on our HSM, we compared *Libsodium*[1],

---

[1] Libsodium library: `https://libsodium.gitbook.io/doc/`

Mbed TLS[2] and *OpenSSL*[3]. *Libsodium*, despite its simple API, did not have much to offer regarding commonly used cryptographic algorithms because it only uses specific algorithms like Salsa20 or ChaCha20 stream cipher for symmetric-key encryption, Poly1305 MAC for authenticated encryption, Ed25519 for public-key signatures and Curve25519 used with Diffie-Hellman for key agreement scheme. Although they are good algorithms, we would rather have a wider variety of algorithms, including some well known and widely used algorithms, like AES and RSA.

Mbed TLS offers a wide range of cryptographic algorithms for many architectures, being ARM and embedded their primary focus. They have every algorithm that we would like to implement combined with an intuitive API and readable source code.

We ended up using *OpenSSL* cryptographic library as it is widely used, is a full-featured toolkit that provides every functionality we needed to implement our system and leaves room for future updates and increased supported features. It is worth to mention that *OpenSSL* supports every algorithm used by *Libsodium* and *Mbed TLS*. Since **Libsodium** is the only library that can protect our solution against side-channel attacks, it can be easily adopted in our Virtual HSM.

3.3.1 Hash Functions

A cryptographic hash function is a hash function that is suitable for use in cryptography, thus can be used to map data of arbitrary size to data of fixed size. The values returned by a hash function are called hash values, hash codes, digests, or only hashes. The same hash function will always reproduce the same digest for the same message, but from the digest, we can not obtain the message as it is a one-way function. This is why this kind of functions are widely used in secure operations, because they can get a message from an arbitrary size and get a unique output (for that message) with a fixed size, that output can be used to verify if the message was not tampered (if it were, the newly calculated hash would give a different digest than the calculated first).

In summary, the ideal cryptographic hash function has the following properties: it is deterministic, it is quick to compute the hash value, it is infeasible to generate a message that yields a given hash value, it is infeasible to find two different messages with the same hash value and a small change to a message should change the hash value so extensively that the new hash value appears uncorrelated with the old hash value.

In our system, we support the following hash functions:

- **MD5** or Message-Digest algorithm 5, is a hash function producing a 128-bit digest but it has been found to suffer from extensive vulnerabilities, so it is now being discontinued or used only to non-cryptographic purposes.

---

[2] Mbed TLS library: `https://tls.mbed.org/`

[3] OpenSSL library: `https://www.openssl.org/`

- **SHA-1** or Secure Hash Algorithm 1, is a hash function that produces a 160-bit digest, but in 2017 it was dropped by the SSL Certificates, and there has been some success with collision attacks, which means that they can reproduce the same output with similar but not equal inputs.
- **SHA-2** or Secure Hash Algorithm 2, includes significant changes from its predecessor, SHA-1, and is now one of the mos used hash functions with a subset of functions that are very similar but varying on output length and, therefore, in security guarantees given:
  - **SHA-224** with a 224-bit message digest
  - **SHA-256** with a 256-bit message digest
  - **SHA-384** with a 384-bit message digest
  - **SHA-512** with a 512-bit message digest

3.3.2 Hash Message Authentication Codes

Hash Message Authentication Code, or HMAC, is a specific type of message authentication code (MAC) involving a cryptographic hash function and a secret cryptographic key. It may be used to simultaneously verify both the data integrity and the authentication of a message, as with any MAC. Any cryptographic hash function may be used in the calculation of an HMAC.

HMAC does not encrypt the message. Instead, the message (encrypted or not) must be sent alongside the HMAC hash. The difference between using Hash or HMAC is that HMAC is impossible to create the same digest without having the secret key. Only parties with the secret key will hash the message again themselves, and if it is authentic, the received and computed hashes will match.

In our system we offer the possibility to use **HMAC-SHA256** or **HMAC-SHA512**.

3.3.3 Block-ciphers

Advanced Encryption Standard, or AES, is a symmetric block cipher that encrypts data in blocks of 128 bits using cryptographic keys of 128, 192 or 254 bits long. Symmetric ciphers use the same key for both encrypting and decrypting, so the sender and the receiver must both know, and use the same secret key. Despite having known vulnerabilities (on particular use cases with specific modes), it is still the most widely used symmetric encryption scheme.

In our solution, we decided to support seven different modes of the same algorithm so that despite only having one algorithm, it could be suitable for countless situations. The modes are the following:

- **ECB** or Electronic Codebook: the message is divided into blocks, and each block is encrypted separately. ECB's colossal disadvantage ist that there is no diffusion, which means identical plaintexts will originate identical ciphertexts. It is not recommended for use in cryptographic protocols, especially if communications will be the same or have a pattern.

- **CBC** or Cipher Block Chaining: each block of the plaintext is *XORed* (the process of applying an Exclusive OR) with the previous ciphertext block before being encrypted. In this way, each ciphertext block depends on all plaintext blocks processed up to that point. So that each message is unique, a different initialization vector must be used with the first block.

  CBC has been the most commonly used mode of operation. Its main drawback is that encryption is sequential (i.e., it cannot be parallelized).

  In our system, we also offer two specific CBC modes that, combined with HMAC function, provide sound encryption plus integrity mode:

  - CBC with HMAC SHA-1
  - CBC with HMAC SHA-256

- **OFB** or Output Feedback: make a block cipher into a synchronous stream cipher. It generates keystream blocks, which are then *XORed* with the plaintext blocks to get the ciphertext. Just as with other stream ciphers, flipping a bit in the ciphertext produces a flipped bit in the plaintext at the same location. Each operation depends on all previous ones, and so cannot be performed in parallel. However, because the plaintext or ciphertext is only used for the final XOR, the block cipher operations may be performed in advance, allowing the last step to be completed in parallel once the plaintext or ciphertext is available.

- **CTR** or Counter: like OFB, Counter mode turns a block cipher into a stream cipher. It generates the next keystream block by encrypting the successive values of a "counter". The counter can be any function that produces a sequence that is guaranteed not to repeat for a long time, although an actual increment-by-one counter is the simplest and most popular. Along with CBC, CTR mode is one of two block cipher modes recommended by Niels Ferguson and Bruce Schneier [22].

- **GCM** or Galois/Counter Mode: is an authenticated encryption mode designed to provide both data authenticity (integrity) and confidentiality. GCM can take full advantage of parallel processing, being widely adopted because of its efficiency and performance. Like all counter modes, this is essentially a stream cipher, but the result is encrypted, producing an authentication tag that can be used to verify the integrity of the data. The encrypted text then contains the IV, ciphertext, and authentication tag.

For the sake of having a general-purpose HSM, we have only implemented AES as our symmetric encryption scheme, but it is left for future work to support other well known symmetric algorithms, such as Blowfish, Twofish, DES (Data Encryption Standard), 3DES, and many more.

3.3.4 Asymmetric-key Ciphers

The acronym, RSA, stands for Rivest, Shamir, and Adelman, the inventors of the technique. It is one of the first public-key cryptosystems and is widely used for secure data transmission. In a public-key cryptosystem, there are two different keys: the public and the private key. The private key is used to ensure authentication; the public key is used to ensure confidentiality, as only who owns the private key can decrypt the message.

This asymmetric system is based on the practical difficulty of the factorization of the product of two large prime numbers.

As RSA is a relatively slow algorithm, it is less commonly used to encrypt user data directly. Although it can be used for encryption/decryption purposes it has a substantial disadvantage; it can only encrypt data to a maximum amount of the key size used, i.e., a 2048-bit key can only encrypt less than 256 bytes of data. More often, RSA encrypts shared keys for symmetric key cryptography which in turn can perform bulk encryption-decryption operations at a much higher speed.

In our system, we support both RSA operations: Signing and Encrypting. The first ensures authentication and produces a signature that can only be reproduced by the party that has the private key, and the second uses a public key to encrypt data that can only be decrypted by the private correspondent key. This last operation, in our HSM, is done by generating a symmetric key, encrypting it with a public key, encrypting data with that symmetric key and then send it all together. The receiver first decrypts the symmetric key with its asymmetric private key, and then it can decrypt the actual message with the symmetric key.

We support RSA keys of 1024, 2048, 3072 and 4096-bits length.

3.3.5 Elliptic Curve Ciphers

In mathematics, an Elliptic Curve, or EC, is a plane algebraic curve defined by an equation of the form: $y^2 = x^3 + ax + b$.

In cryptography, EC is an approach to public-key cryptography based on the algebraic structure of elliptic curves over finite fields. ECC requires smaller keys compared to RSA to provide equivalent security, i.e., a 256-bit elliptic curve public key should provide comparable security to a 3072-bit RSA public key.

The security of elliptic curve cryptography depends on the ability to compute a point multiplication and the inability to compute the multiplicand given the original and product points. The size of the elliptic curve determines the difficulty of the problem.

In our system, we support the use of Elliptic Curve Cryptography to sign and verify documents but not to encrypt and decrypt documents, as supported by RSA. Nevertheless, our system is fully prepared to support that feature whenever OpenSSL supports it.

Regarding EC keys, we support every curve that is already supported by OpenSSL.

3.4 HSM API

As explained in Section 2.1.1, our objective was to implement PKCS#11 API on top of our system. It is an industry-standard cryptographic API that allows us to communicate and issue operations on cryptographic devices such as HSMs.

The main advantage of using PKCS#11 API is that the majority of physical HSMs use it; therefore, companies are used to issue PKCS#11 commands on their clients. Meaning that it would be effortless for companies to change to our solution, as instructions used would be the same as before.

Unfortunately using that API proved to be an impossible task due to its complexity and lack of documentation on how to implement it. From this point, we left aside the implementation of PKCS#11 and created our own Virtual HSM API. We do not deny the ease and compatibility that PKCS#11 would bring to our system but our API has almost all of the PKCS#11 functions, which means that in the future we would only need to add another layer on top of our API.

3.4.1 Digest

To use the digest function (Listing 3.1), we need to send the data we want to hash and its length, along with the hash function we want to use. *mode* is the argument that indicates which hash function will be used. We use numbers to distinguish between hash functions:

1. MD5
2. SHA-1
3. SHA-224
4. SHA-256
5. SHA-384
6. SHA-512

Lastly, *md* and *md_len* are the response array and its length, respectively. Digest returns an integer depending on its success or failure.

```
int digest(unsigned char* plaintext, int plaintext_len, int mode,
    unsigned char** md, int* md_len);
```

Listing 3.1: Virtual HSM API: digest.

3.4.2 Key Generation

In Listing 3.2, we have only one function implemented to generate new cryptographic keys of four different algorithms: HMAC, AES, RSA, and EC.

To generate a new HMAC key, we need to send a password as an argument, its length, and an id to store the key. On AES and RSA, the arguments used to generate new keys are quite similar; we need to choose a key length and, again, an id. Key length can vary

```
int key_gen(int algorithm, int key_len, int key_id, unsigned char* password,
    const char* curve);
```

Listing 3.2: Virtual HSM API: key generation.

from 128, 192 or 256 on AES and 1024, 2048, 3072 or 4096 on RSA. Regarding EC keys, we need to send a *string* with the name of the curve we want to generate and, of course, an id to store it.

Key generation functions return an integer depending on its success or failure.

3.4.3 Sign and Verify

There are three types of signatures supported by our system: HMAC, RSA, and EC. Depending on the algorithm used, the functions on Listing 3.3 will produce and verify HMAC, RSA or EC signatures.

To sign a document, we only need to provide the plaintext, its size, and the key that will be used (this key must be of the same type as the algorithm used). If successful, the array *signature* will contain the produced signature.

To verify a signature, we need to provide the same plaintext and its size, the signature and its size, and the id of the key used to produce the signature.

Signing and verifying functions return an integer depending on its success or failure.

```
int sign(int algorithm, unsigned char* plaintext, int plaintext_len, int key_id,
    unsigned char** signature, int* signature_len);

int verify(int algorithm, unsigned char* plaintext, int plaintext_len,
    int key_id, unsigned char* signature, int signature_len);
```

Listing 3.3: Virtual HSM API: sign and verify.

3.4.4 Encrypt and Decrypt

Listing 3.4 shows only one function to encrypt and another to decrypt, either using the AES algorithm and encryption mode or using RSA keys to seal documents. Our solution is prepared to seal with EC, although OpenSSL does not yet support it.

These functions work as follows: to encrypt, it receives the plaintext and its length, the key id to encrypt with, and an array (*encrypted*) where the encrypted text will be stored. To decrypt, we need to provide the ciphertext and its length, the same key id, and an array (*decrypted*) where the decrypted text will be saved.

The only difference between using AES or RSA algorithms is that on AES, we need to provide one of the encrypt/decrypt modes supported (explained in more detail in Section 3.3.3).

```
int encrypt(int algorithm, unsigned char* plaintext, int plaintext_len,
    int key_id, int mode, unsigned char** encrypted, int* encrypted_len);

int decrypt(int algorithm, unsigned char* ciphertext, int ciphertext_len,
    int key_id, int mode, unsigned char** decrypted, int* decrypted_len);
```

Listing 3.4: Virtual HSM API: encrypt and decrypt.

3.4.5 Other Functions

In Listing 3.5 we have the last functions available in our system. It is possible to export the Server's public key into an array to be stored somewhere else, and it is also possible to import a public key to the system, however it is necessary to select if we are importing an RSA or an EC key, the latter still need to know what type of curve we are importing.

init_server_keys runs when Server initializes, and it simply initializes Server's public and private keys and stores them inside the Enclave. After initializing server keys, if there is a previously stored HSM state on disk, we will load it to the Enclave and recover to the previous state.

As it will be explained in Section 3.5.2, whenever the Client establishes a secure channel with the Server, the latter stores the session key agreed between them with the client id. When the Client is done communicating with Server, it closes the secure channel and Server erases the session key previously stored.

```
int export_pubkey(int key_id, unsigned char** response,
    int* response_len);
int import_pubkey(int rsa, unsigned char *pubkey, int pubkey_len,
    const char* curve_name, int key_id);

void init_server_keys(unsigned char* privkey, int privkey_len,
    unsigned char* cert, int cert_len);
void reconstruct_map(unsigned char *sealed_data, int sealed_len, int sealed);
void erase_client(int client_id);
void erase_key(int key_id);
```

Listing 3.5: Virtual HSM API: utils.

3.4.6 Critical Analysis

An HSM has two main requirements: first is to generate and store cryptographic keys safely, second is to issue cryptographic operations in a fully secure and isolated environment. The first requirement is met on our solution by using our *key_gen* function, described in Section 3.4.2, where keys from 4 different algorithms can be safely generated and stored inside the Enclave.

Regarding cryptographic operations, we decided to implement five different operations. The *digest* function that uses cryptographic hash functions is an essential component in the cryptography world because it is very light (regarding computational resources) and can provide simpler integrity proofs.

Signatures are one of the most important techniques regarding authentication and integrity. We have implemented two functions to deal with signatures: *sign* and *verify*. The first can produce digital signatures using three different algorithms. *verify* is needed to verify those signatures and confirm if documents were signed by whoever should have signed them.

The last two cryptographic functions implemented on our solution are *encrypt* and *decrypt*. They can be used to encrypt and decrypt documents using both symmetric-key and asymmetric-key algorithms.

We decided to implement these functions because we believe they are the most used cryptographic functions; therefore, we can reach the majority of HSM users.

3.5 Implementation Details

In this section, we describe the implementation details of our Virtual HSM. A system with robust security guarantees, such as confidentiality, integrity, and authentication, mainly given by the IEE instantiated through Intel SGX.

Our system was implemented in C/C++ using Intel SGX SDK version 2.6[4]. The last version has around 6 200 lines of code, including both versions (this and the distributed one, on Chapter 4), Server, Enclave and Client, as well as auxiliary tools.

Virtual HSM is available on `https://github.com/mgrosa/virtualHSM` as open-source. Our Server, IEE and its storage are implemented as one application, and Client is a completely separated application.

3.5.1 Setup

After compiling the code, we start our system by issuing: **./hsm <port>**. The Server starts, and the very first thing it tries is to initialize the Intel SGX Enclave. If the Enclave's successfully initialized, we can proceed to our HSM initialization.

Upon Enclave initialization, it will generate a fresh public and private key pair that will never leave the Enclave. These are 2048-bit RSA keys, and its only purpose is to provide a secure channel between the Enclave and Client, as it will be explained in the next Section.

Whenever the system is shut down, or new cryptographic keys are generated, we store a new encrypted version of the HSM state on disk. Therefore, upon initialization, the system will always search for previous HSM state on disk and, if it exists, it will load it and restore its former state as the actual state.

---

[4] `https://software.intel.com/en-us/sgx/sdk`

The port used to initialize the system will now be used to start a TCP server. The main program is waiting in a loop for new connections to happen. When a new client connects with the Server, a new thread is created to process the Client's request.

3.5.2 Secure Channel

Since the beginning of this chapter, we assume the Client to be trusted, but we can not accept communications from every Client that tries to connect with our system as they can have malicious code on them and ruin the whole purpose of having a virtual but secure HSM.

In this first version, we are only supporting a single client. The easier and most secure way to establish a connection with a trusted client was to include the Client's public key in the Enclave code. Which we know is not ideal but serves for demonstration purposes. In the next chapter, as multiple clients will be supported, there was the need to change from hard-coded keys to another effective way that can still guarantee we are communicating with a trusted client.

After setup protocol, the Server is waiting for clients to connect, which means that the first step for establishing a secure channel must come on the Client side. The Client starts by generating a public key encryption key pair (pk_C, sk_C), of which pk_C is hard-coded into the IEE program code, and will be used to bind the IEE with its Client.

The Server asks the IEE to compare the pk_C received with the hard-coded one, which if it matches the Server accepts the Client connection and will proceed to create a secure channel. After that, the Server will only act as a proxy for all Client-IEE communications.

The Client then starts the authentication procedure with the IEE, following the IEE algorithm of [8], which provides the following properties: that an IEE is bound to a specific client, and can not answer to illegitimate clients; that a client must be able to attest it is communicating with a legitimate IEE, and the code being run in it is the one the Client expects. To ensure these properties, an authenticated key-exchange scheme is executed; at the end of which an authenticated symmetric key is shared between Client and IEE. The algorithm works as follows:

1. The Enclave generates a fresh public key encryption key pair (pk_E,sk_E) and sends pk_E to the Client, accompanied by a cryptographic proof of attestation (in SGX Enclave nomenclature known as a quote).

2. The Client attests the received proof and generates a symmetric key k, encrypts it with pk_E and signs it with sk_C, sending it to the Enclave.

3. The Enclave can then decrypt the received key k and verify its signature. If no errors occurred during signature verification, the Enclave produces a confirmation message to the Client.

4. The Client receives and attests the confirmation message.

```
struct Keys {
  //if AES
  unsigned char* aes_key;
  int aes_key_len;
  //if RSA
  EVP_PKEY* rsa_pubkey;
  EVP_PKEY* rsa_privkey;
  //if EC
  EVP_PKEY* ec_pubkey;
  EVP_PKEY* ec_privkey;
  int eccgroup;
  //if HMAC
  EVP_PKEY* hmac_key;
  unsigned char* hmac_password;
  int hmac_password_len;
};
```

Listing 3.6: Keys structure.

From here onwards, the Client-Enclave channel is ready for secure point-to-point communications. The key-exchange algorithm provides two-way authentication and confidentiality guarantees. The use of an authenticated cipher in further communication can be used to provide integrity guarantees, and techniques like the use of nonces hinder attacks such as replaying.

There is a symmetric key assigned for every Client communication, which means that when the Client closes this channel, the assigned key will be erased. Even if the same Client wants to open another secure channel, it needs to go through the whole process again.

3.5.3 HSM Internal State

Our HSM internal state is an abstraction of a data structure where every cryptographic key generated by a client request is stored. We are using a Map as we can easily access the value of the Map with a key identifier, meaning key identifiers must be unique. If a client issues a new key with an already in use key id, the previously stored key will be erased and a new key will take its place.

The value of the Map is a C/C++ structure (Listing 3.6) with every information needed to use that keys.

There are some attributes that may seem odd to store, e.g., *aes_key_len*, *eccgroup*, *hmac_password* and *hmac_password_len*. Those attributes are stored to facilitate the process of serializing and reconstructing our Map whenever it needs to be saved or loaded from disk, respectively. It may be seen as a counter-productive measure because the more information we keep could lead to more information that is vulnerable, but cryptographic keys and other attributes only exist, in clear, inside the Enclave, which on our adversary model we assume it can not be compromised. Whenever they leave the Enclave, they are encrypted with Enclave's public key, meaning that only the same Enclave can decrypt them.

3.5.4 External Storage

The main reason to use external storage is that Enclave's memory is not persistent, i.e., if the Enclave is shutdown, everything stored inside it will be lost. We do not want that kind of behavior in our system, so we decided to save the HSM state outside the Enclave. With that solution, even if the Enclave is shut down and then restarted, there is the possibility to load the last HSM state and restore it.

Storing highly sensitive data outside the Enclave requires an adequate way of encrypting and checking the integrity of data to prevent access of untrustworthy parties and detect tampering, respectively.

Intel SGX provides a capability called "data sealing"which encrypts data inside the Enclave using an encryption key that is derived from the CPU. This encrypted data block, also called the sealed data, can only be decrypted, or unsealed, on the same system (and, typically, in the same Enclave) where it was created. The encryption itself provides assurances of confidentiality, integrity, and authenticity on the data.[5]

There are two signing policies: *MRENCLAVE* and *MRSIGNER*. The first uses a cryptographic key that is derived from the Enclave Identity, which uniquely identifies any particular enclave and so it will restrict access to sealed data only to instances of that Enclave. Different builds/versions will result in different Enclave Identities; therefore, later versions will not be able to unseal early versions sealed data.

When using *MRSIGNER* policy, the cryptographic key is derived from a Signing Identity which can sign different enclave keys, thus allowing the same key to be used on different versions of the same Enclave. We use this policy in our system so that, in the future, we could have a system composed of various Enclaves that use the same key, and that can share sensitive sealed data between them.

As explained earlier, whenever a new cryptographic key is generated inside the Enclave, the Map is serialized, sealed and stored on disk so that the latest version of the HSM's state is always saved.

Upon Enclave initialization, the Server loads the sealed data and sends it to the Enclave to unseal it. It is up to the Enclave to make an integrity check before unsealing data and, if tampering is detected, the Enclave is shut down and the sealed HSM state is erased from the disk. This is a counter-measure upon tampering detection as we assume that cryptographic keys could be highly sensitive data, and it is better to lose everything than risk falling into the wrong hands.

3.5.5 Client Requests

Requests made from Client to Enclave are pretty straightforward: Client sends the request, Server acts as a proxy and forward the request to the Enclave that processes it.

---

[5] `https://software.intel.com/en-us/blogs/2016/05/04/introduction-to-intel-sgx-sealing`

```xml
<EnclaveConfiguration>
  <ProdID>0</ProdID>
  <ISVSVN>0</ISVSVN>
  <StackMaxSize>0x400000</StackMaxSize>
  <HeapMaxSize>0x40000000</HeapMaxSize>
  <TCSNum>8</TCSNum>
  <TCSPolicy>1</TCSPolicy>
  <DisableDebug>1</DisableDebug>
</EnclaveConfiguration>
```

Listing 3.7: Enclave configuration file.

Finally, a response is sent to the Client. After the key exchange protocol, every communication between Client and Server is encrypted with the established key.

The user issues a request by passing arguments through the command line, such as algorithms to be used, key lengths and input/output files, then the Client processes those arguments to join them in a single array that can be processed by the Enclave.

The Enclave always sends the response to the Client, even if it is an error response. Functions like key generation or signature verification will always return a success or failure message. If it is a function like signing or encrypting, they return the actual content requested (signature or ciphertext) or a failure message.

3.5.6 Enclave Configurations

It is worth to mention that there is an Enclave configuration file: **Enclave.config.xml**, where we can configure some Enclave definitions such as Heap and Stack maximum size, number of threads to be used by the Enclave and other policies.

The most relevant details on our configuration file (Listing 3.7) are *StackMaxSize* and *HeapMaxSize*, because it is their configuration that determines how many RAM our system will use and what is the maximum file size that the Enclave supports. We choose those values because they allow us to handle files up to 300 MB, which we consider to be more than enough to a system like this, using slightly more than 1GB of RAM.

As explained in Section 2.2.2.2 enclaves have a limitation of 128 MB of memory, i.e., 128 MB is the size of an Enclave's page. If more than 128 MB is needed, it requires the Enclave to use pagination, which leads to performance losses. Although our system is capable of encrypting (or any other operation) files up to 300 MB, we recommend using files with less than 100 MB, preventing pagination from occurring, thus having the desired performance.

3.6 Security Analysis

In this Section, we will informally analyze the security of our system. We consider both the Client and Enclave to be part of our trusted computing base, and as such, providing full confidentiality, integrity, and authentication guarantees.

We will start by analyzing each untrusted component and the data it has access to, then the Client-Enclave channel, and finally analyze leakage and privacy guarantees given by the system.

Our **Server** acts like a proxy for the Client-Enclave channel. Since every communication between them is encrypted, the only possible attack via our Server against our system is the denial-of-service. Since we do not include it in our adversary model, we can disregard this kind of attack. Malicious code could also try to tamper data on Server, but it will not be successful as everything is authenticated and verified both at the Enclave and Client, therefore tampering attempts will be discovered.

Regarding **Storage**, it is a very straightforward component, and it only read/writes from/to disk. Despite being an untrusted component, we have strong security guarantees, given by Intel SGX, that data will not be vulnerable at rest. Data only comes out of the Enclave sealed by the Enclave itself, where the sealing key is derived from the CPU and unique to that Enclave. The sealing and unsealing process takes into consideration integrity checks that allow us to guarantee not only confidentiality but also the integrity of the HSM state stored on disk.

The channel is established between Client-Server-Enclave, but as Server only acts as a proxy, we can name it **Client-Enclave channel**. The Enclave is hard-coded with the Client's public key so that only that specific Client speaks with the Enclave, discarding the possibility of an attacker to suddenly interact with the Enclave, forcing it to answer malicious operations. To leverage Enclave guarantees, the Client must be sure that it is communicating with a legitimate Enclave, and that the Enclave is running the expected program, of which the Client has a digest. The Client does so by verifying that the attestation proof received during the initialization procedure is signed with an IEE-specific key (in practice provided by its vendor) and that it also contains the correct digest of the program expected to be running. These verifications are performed during the key-exchange protocol (Section 3.5.2), and thus the Client can be sure of confidentiality, integrity, and authentication guarantees at the start of communications. Given that both endpoints of the channel are considered secure, it follows that the channel remains secure throughout system operation.

Communication between Client and Enclave is encrypted. Although there is always information leaks that could give a hint on what is happening to the attacker, this is related to the size of the request and the response, e.g., when the user requests to generate a new key, the request size is rather small comparing when the user requests to encrypt a 100MB file. It also happens with responses because if the latter request succeeds the Enclave will send encrypted data (with more than the file size), but if it fails, the response will be a minimal error message. We do not consider this to be a problem because only an attacker with in-depth knowledge of the system could have hints on what kind of operations were issued.

The key exchanged between Client and Enclave is unique and different for every connection, this means that, if the Client wants to make two separate requests, it will exchange two different keys, one for each request. If an attacker got one secret key, only the request and response associated with that secret key could be decrypted. It is still impossible for the attacker to decrypt other communications with that key.

Confidentiality is achieved, outside the Enclave, by using secure encryption schemes to protect data in transit and at rest. Data in transit is protected by a secure channel abstraction where a symmetric encryption scheme with CBC HMAC SHA-1 mode and 256-bit keys are used. Data at rest is protected by Intel SGX sealing capability, where data is encrypted using the Enclave key, and only the same Enclave can decrypt it.

Data **Integrity** is always verified, whether in transit or at rest. The first is accomplished with the HMAC used in the encryption scheme, which is verified every time the Enclave receives secure communication. The second is provided, again, by the sealing capability of Intel SGX, where data integrity is verified before unsealing it.

Authentication is achieved by having the Client's public key hard-coded into the Enclave's code. This simple, but effective, measure allows us to only accept communications from a trusted Client, thus ensuring authentication.

3.7 Summary

The primary purpose of building a system like Virtual HSM is to avoid resort to specialized hardware that physical HSMs do, reducing the minimum financial costs that a company or an individual has to bear. The only requirement to use our solution is that the processor being used on Virtual HSM is SGX-enabled.

Virtual HSM is composed of four different components: two of them are trusted (IEE and Client), and the other two are untrusted (Server and Storage). The Server only acts as a proxy between Client and IEE, because the IEE can not have access to networking and so, the Server handles the connections and passes its content to the IEE.

The HSM's state is regularly stored on disk because the Enclave's memory is not persistent, which means that whenever it is shut down, we would lose everything stored inside it. On top of that is the fact that the Enclave's memory has a limitation of 128 MB. Before leaving the Enclave, the HSM's state is encrypted with the Enclave's key, providing confidentiality and integrity guarantees to stored data.

The IEE is the central part of the system. It is where the processing of requests is made, where every cryptographic operation is executed, from key generation to signature production or encrypting data. The whole system is supported by the robust security guarantees given by Intel SGX Enclaves (confidentiality, integrity, and authentication).

As we are using *OpenSSL* library, our API has some of the well known and most used cryptographic operations. For this prototype, we choose to implement operations like production of message digests using MD5, SHA-1, and SHA-2; key generation of AES, HMAC, RSA, and ECC; signing and verifying documents using HMAC, RSA, and ECC; encrypting and decrypting documents using AES and RSA. If a specific algorithm exists on *OpenSSL* and is needed on Virtual HSM, it can be easily added to the system.

Upon system initialization, the Enclave is also initialized and ready to process Client requests. When a Client wants to connect with the Enclave, they start a key-exchange protocol where the Enclave generates a symmetric key that will be used by both parties to secure their communications. From now on, the Client can send requests to the Enclave without being worried if their content is vulnerable and could be compromised.

CHAPTER 4

DISTRIBUTED VIRTUALHSM

In this chapter we explain how we transform our prototype of a single Virtual HSM to a distributed version of one, supporting multiple clients, ensuring the same security guarantees as to the first version and adding high-availability to the whole system.

This chapter is organized as following: we will first introduce this version's system model and architecture (Section 4.1), its adversary model slightly differences (Section 4.2) and the possible use cases(Section 4.3). Then we explain the implementation details of the distributed version (Section 4.4), following with a security analysis (Section 4.5) ending with conclusions regarding Distributed Virtual HSM (Section 4.6).

4.1 System Model and Architecture

As explained earlier, in this chapter, we consider a system with multiple clients interacting with multiple IEE-enabled cloud servers. It means that not only we support numerous clients using the system at the same time as it is also possible to start a Virtual HSM cluster that will increase the system's availability drastically.

In this version we have the same four components as before: **Client** and **IEE** (trusted), **Server** and **Persistent Storage** (untrusted). The **Client** is the same as before; it only needs to know to which Virtual HSM server it should connect, and every request will be sent to that Server.

The other three components (**Server**, **IEE**, and **Persistent Storage**) exist in the same SGX-enabled node, being that the key difference from the first version is having multiple SGX-enabled nodes connected to every other node, as Figure 4.1 indicates.

This kind of architecture allows us to have dedicated secure channels between every node, creating a fully connected mesh topology. Having those dedicated links to and from every node ensures that even if a **Server**/**Enclave** is shutdown, for example *HSM #A*, the

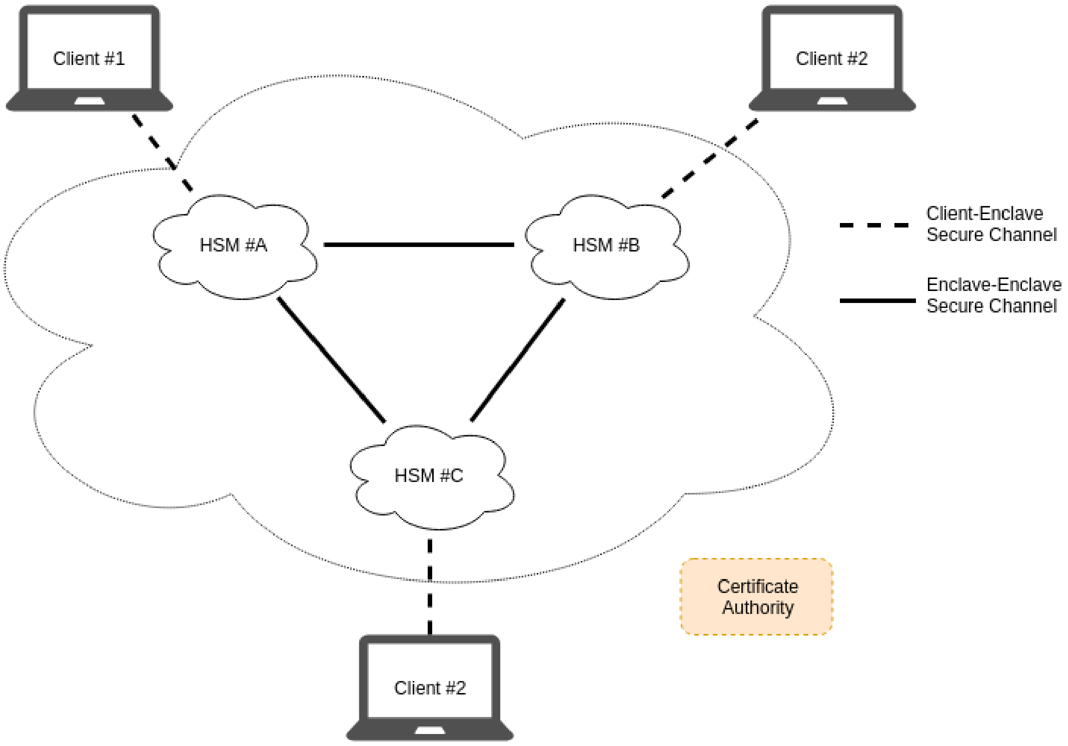


Figure 4.1: Distributed system architecture. Clients communicate with HSMs using a secure channel. Each HSM is connect to every other HSM using another secure channel. Certificate Authority is apart from the system and issues certificates for every trustworthy Client and HSM.

whole system can still fully work with just *HSM #B* and *HSM #C*, the only difference is that *Client #1* should reconnect to one of the available HSMs.

Communication between different IEEs is performed through a secure channel abstraction (details will be explained in Section 4.4.2) with privacy, integrity and authentication guarantees, very similar to the channel used between Client and Enclave. The Server still acts as a proxy on this channel, meaning that the Enclaves will communicate only with each other, not with their Servers.

Only trustworthy HSM nodes can connect with other HSM nodes. To ensure that nodes are trustworthy, we added a new component to our system: a Certificate Authority. So that new Enclaves can be added to our trust computing base they need to have certificates issued and signed by this new component. After a successful verification regarding the trustworthiness of the new node's certificate, it will be able to communicate with other HSM nodes in order to initiate the key exchange protocol that will allow them to communicate using an encrypted channel.

4.1.1 Certificate Authority

A Certificate Authority, or CA, is an entity that issues digital certificates and acts as a trusted third party, trusted both by the owner of the certificate and by the party relying upon the certificate. The authority itself issue and signs certificates, and its chain can be verified, i.e., an independent party can ascertain if a trustworthy CA issued a certificate, thereby trust in the certificate's owner.

Having a CA in our system is useful because we trust in our CA; therefore, we trust in certificates issued and signed by our CA. This component allows us to add and remove new Enclaves or Clients to our trust base without having their keys hard-coded, and to distinguish between Client machines and HSM machines so that Clients can not act as HSMs nor HSMs act as Clients.

To distinguish between Client and Enclave certificates, we created two intermediary CAs: *Client's CA* and *Enclave's CA*. To verify if certificates were signed by one of these two intermediate CAs, we need to have their keys hard-coded in the Enclave. Independently of the number of Clients and Enclaves that are trustworthy, we will only have those two CA keys in the Enclave's code.

Our CA's structure is as shown in Figure 4.2, having a *Root CA* on top of everything, which is our system's mainly trusted CA. *Root CA* then creates two intermediate CAs: *Client's CA* and *Enclave's CA*. The first one is responsible for issuing and signing certificates for every Client allowed to issue requests to our Virtual HSM. The latter is the opposite, and it is responsible for issuing and signing certificates for every Enclave, allowing other Enclaves to trust in the first, thus allowing it to connect to them.

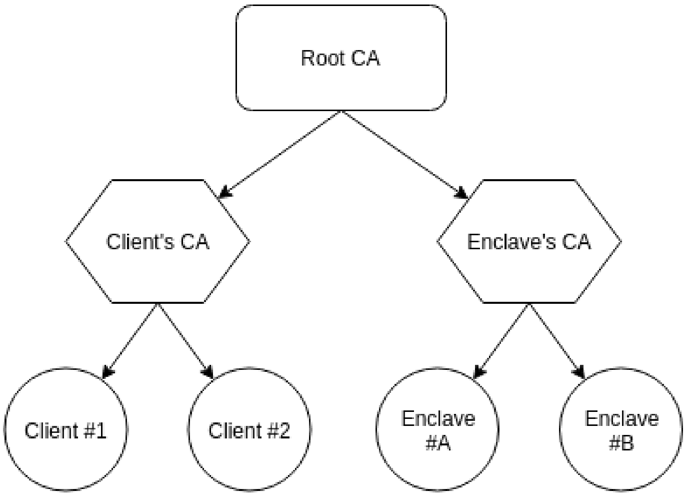


Figure 4.2: CA structure. Each intermediate CA only issues certificates to Clients or to Enclaves.

4.2 Adversary Model

The adversary model is pretty much the same as in the previous chapter (Section 3.2). We consider a fully-malicious attacker who can perform both active and passive attacks on any node. The key difference is that in this distributed version of Virtual HSM, we do consider an attacker capable of compromising the system's availability.

Counter-measures against availability attacks were not implemented as Denial-of-Service kind of attacks are very hard to counter, and it was not the focus of our implementation. Nonetheless, we do assume that public cloud providers used have high availability, and at least one node is always available.

4.3 Use Cases

Testing Companies that own physical HSMs won't use them for testing purposes. The risk of using such dedicated and expensive hardware that owns highly sensitive information, with software that is being tested is too high. That is why those companies choose to use software implementations of HSMs to test their applications before releasing them. Our solution is far more adequate than simple software implementations as higher security, availability, and performance are guaranteed. Thus, keeping testing operations completely independent from production environments.

Production Small companies and users that do not have funds or do not feel the need to have a physical HSM can also resort to our solution. We offer great performances along with high security and availability guarantees using only commodity hardware.

4.4 Implementation Details

In this section, we will describe the implementation details that have changed from the first version of Virtual HSM. Every node of this version is the same as in the previous version, it has a **Server**, an **Enclave** and **Storage**. The key difference is that we can have as many nodes as we want, in contrast with the first version where only one node exists.

4.4.1 Setup

The first step in the setup process is to start the Server. It will try to initialize the Enclave and, if successfully, it will load both its private and public keys to the Enclave. These keys were, in this version, issued and signed by an intermediate CA, *Enclave's CA*.

The initialization of the Enclave is now more robust and secure because the Enclave will verify if its keys, previously loaded by the Server, were issued and signed by the system's CA and are therefore reliable. We do not trust in our Server. Therefore it could have changed the keys before sending them to the Enclave. If that occurs, the Enclave will detect that the keys received were not issued by the *Enclave's CA*, it will shut down and delete the HSM state stored on disk.

After compilation, we start a new node by issuing: **./hsm <port> -multi**, where the argument **-multi** indicates that the Server should start the setup for multiple server configuration. There is a configuration file in the root of the project named *servers.config*.

This configuration file has, on every line, the Server's IP, followed by two different ports. The first port is where the Server is listening for new Client connections, the same used as an argument; the latter is where the Server is listening for new Server connections. After reading the configuration file, the Server will open a socket for new Servers and will try to connect to every other Server on file. Upon a successful connection, it will initiate a key exchange protocol so that Servers communicate using a secure channel.

4.4.2 Secure Channel

Both Client and Enclave CAs have their keys hard-coded on Enclave. It is needed so that the Enclave can securely verify if received certificates are trustworthy. However, they are the only hard-coded keys in this version because every other key used is issued by one of the CAs.

Regarding the explanation on Section 3.5.2, we now accept every communication from Clients that have had their key issued by the *Client's CA*, and only by that CA. Certificates signed by other CAs will not be accepted. The key-exchange protocol for creating a secure channel between Client-IEE will work as described in the previous chapter with the only difference being that now we do not compare the received key with the hard-coded one; instead, we verify the certificate chain.

A secure channel between Enclave-Enclave is now needed; therefore, a key-exchange protocol between two Enclaves will occur. This protocol starts when a Server tries to establish a connection with every other Server in the configuration file. If the second Server is up and running, it will accept the connection from the first Server after verifying the chain of the Server's certificate. A Server certificate is considered trustworthy if it was issued and signed by the *Enclave's CA*, and only by that CA. From then on, the key-exchange protocol will occur, resulting in a symmetric key exchanged between both Enclaves. Until the channel is closed, those two Enclaves will always communicate using that symmetric key.

4.4.3 Replication

The main objective of creating a Distributed Virtual HSM was to increase system availability and, to accomplish that, we introduced a full mesh topology with multiple Virtual HSM nodes. Every node has a secure channel with every other node so that communications between them are encrypted and authenticated.

The approach used to replicate data between HSM nodes is very straightforward. Whenever a node updates its state, i.e., a new key has been generated or deleted, a flag will be raised. This flag indicates that there have been changes and they need to be replicated to every other node. This process involves the serialization of the updated HSM state, its encryption with the symmetric key associated with the Enclave that will receive it and, lastly, the reconstruction of the received HSM state on the new node.

To prevent network flooding, we decided to implement a batching technique, i.e., the Enclave will wait a period before starting the replication process, replicating only the last HSM state rather than replicating every previous state. To do it we have a thread running that has only one job: verify periodically if the flag was raised. If it was not raised the thread will sleep for another period until another verification is made. If the flag was raised, the Enclave will initiate the replication process and send the last HSM state, encrypted, to every other node in the system. The time used for testing purposes was 5 seconds, but it can be easily changed to what better suits the end-user necessities.

The node that receives the updated HSM state will merge it with its outdated HSM state. It is worth to mention that we do not replace the previous state with the new one, we will merge the previous HSM entries with the new ones. If, for some reason, the HSM that is receiving the updated state notice that it has keys that the first HSM does not have, after merging it will send the second HSM state to the first HSM, having the same state in both HSMs.

The use of a consistency protocol was planned in the preparation of this dissertation, but, unfortunately, it was not implemented. We are aware that a system such as an HSM should have strong consistency guarantees so that its highly sensitive information is always consistent on every node. The lack of time and the fact that other difficulties arose made it impossible to find the right strong consistency protocol that would not drastically decrease the performance of the whole system. Even though it was not implemented, we do recognize its importance and leave it for future work.

For instance, using Figure 4.1 as a reference, the normal flow of the replication process would be:

1. **Client #1** will connect to **HSM #A** and issue a new key;
2. After generating the new key, the **HSM #A** will send the updated HSM state to **HSM #B** and **HSM #C**;
3. Even if **HSM #A** fails, the new key already exists on every other node, being that **Client #1** only needs to connect to **HSM #B** or **HSM #C** to continue to use that key.

4.5 Security Analysis

Since each component did not change from the first version, there is no need to go through their security analysis again. Only the Server has had a slight change as now it has two channels: **Client-Enclave** and **Enclave-Enclave**, both secure channels.

The question that arises in this version is related to the replication process. It is where the most sensitive data (HSM state) leave the Enclave and is sent to another machine. Even though the channel is secured by a symmetric key, that only those two Enclaves know, which ensures confidentiality and authentication, the process inevitably leaks some information regarding the size of the sent message. If an attacker has an inside understanding of how the system works, it may give him a hint on how big is the structure storing all cryptographic keys.

Using a Certificate Authority ensures that Enclaves only communicate with Clients or other Enclaves that are trustworthy, thus ensures the authentication of this version of Virtual HSM. Therefore we consider our CA to be a secure component which, in a real scenario, should be protected and disconnected from the Internet so that it can not be compromised, because the security of **Client-Enclave** and **Enclave-Enclave** channels are based on it. If the CA is compromised, an attacker could issue certificates for untrustworthy parties which could lead to severe problems, such as access to sensitive information.

4.6 Discussion

Distributed Virtual HSM is an enhanced version of the system described in the previous chapter. It now accepts multiple Clients, has multiple interconnected nodes, and uses a Certificate Authority to issue certificates to trustworthy parties.

One of the main points of this version is having a CA. It allows us to accept multiple Client and Enclave communications without having their keys hard-coded, ensuring that we only accept communications from trustworthy parties.

The new secure channel used for Enclave-Enclave communications is quite similar to the one used for Client-Enclave communications. It keeps confidential the information that is passed from one Enclave to another, as long as their key remains secret.

Although we do not have a strong consistency protocol, if no problem happens, the HSM's state will be replicated from one node to every other node.

In this distributed version, the performance will not be affected as every node is still independent. Even though every node is connected, they are running as usually having a dedicated thread that sends or receives, processes and updates the HSM state — not affecting, in theory, the performance of the whole system. In the next chapter (Section 5.2.6) we will analyze this question and compare the performance of both versions.

Chapter 5

Experimental Evaluation

We start this chapter by presenting our test bench (Section 5.1) and then we will evaluate the whole system (Section 5.2).

Our primarly objective is to evaluate the performance of using different algorithms supported by our solution: HASH (Section 5.2.1), HMAC (Section 5.2.2), AES (Section 5.2.3), RSA (Section 5.2.4) and EC (Section 5.2.5). We also discuss the impact of some implementation decisions (Section 5.2.6). Furthermore, we compare our Virtual HSM with the some of the related work described (Section 5.2.7) and end it with a discussion regarding our results (Section 5.3).

5.1 Experimental Test Bench

Our experimental test bench was composed of an SGX-enabled machine with the **Client**, **Server** and **IEE**.

Our SGX-enabled machine was a 4-core Intel NUC i7-8559U, with 2.7GHz of CPU frequency, 16GB of RAM and 512GB of SSD storage, running *Linux Mint 19.2*. When using multiple HSM instances we deployed them in the same machine — nevertheless, to discard network noise, we consider local latencies of the main participants individually (**Client** and **IEE**), but not the network latency (we still consider the time of buffer allocation for messages, and their respective writing and reading). We omit **Server** performance, as its work consists solely of message forwarding, and so we considered it as network time.

5.2 VirtualHSM Evaluation

To evaluate the performance of the algorithms implemented in our Virtual HSM, we measure the execution time of each algorithm 1 000 times, calculate the average time each operation takes and the standard deviation of each evaluation. Using the average for each operation and total time for 1 000 executions, we calculate a handy performance measure: the number of operations our system can perform per second.

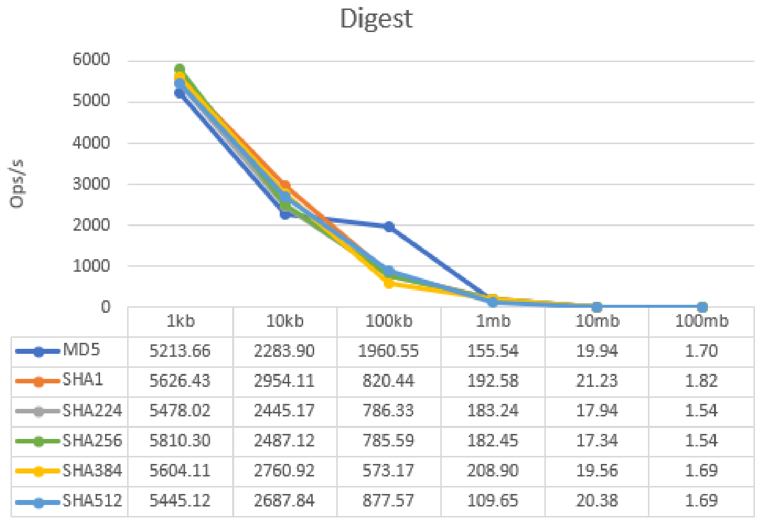
We have two kinds of operations: key generation and file operations. The first operations will be benchmarked by issuing 1 000 new keys of a specific algorithm for every key length or key type.

We consider every operation that needs an input file (Hash, Sign, Verify, Encrypt, and Decrypt) as file operations. For these kinds of operations, we will evaluate the performance of 1 000 executions of the same operation for each one of our six test files. The test files used vary in a magnitude of 10: 1kb, 10kb, 100kb, 1mb, 10mb, and 100mb file.

The objective of using keys and files of different sizes is to evaluate how the system behaves with larger keys and files and to measure its scalability.

5.2.1 HASH

As explained earlier, we have used six files with different sizes for our performance analyses. The objective here was to understand how does our system behaves using diverse hash functions to digest smaller (1kb) to bigger files (100mb).



	1kb	10kb	100kb	1mb	10mb	100mb
MD5	5213.66	2283.90	1960.55	155.54	19.94	1.70
SHA1	5626.43	2954.11	820.44	192.58	21.23	1.82
SHA224	5478.02	2445.17	786.33	183.24	17.94	1.54
SHA256	5810.30	2487.12	785.59	182.45	17.34	1.54
SHA384	5604.11	2760.92	573.17	208.90	19.56	1.69
SHA512	5445.12	2687.84	877.57	109.65	20.38	1.69

Figure 5.1: Digest.

In Figure 5.1 we have the number of operations, per second, our system can make. This translates to the number of Hashes it can produce per second, for a file with the size on the X-axis. The first thing we can conclude is that the six different Hash functions supported in Virtual HSM have nearly the same performance, regardless of the size of the file. Even

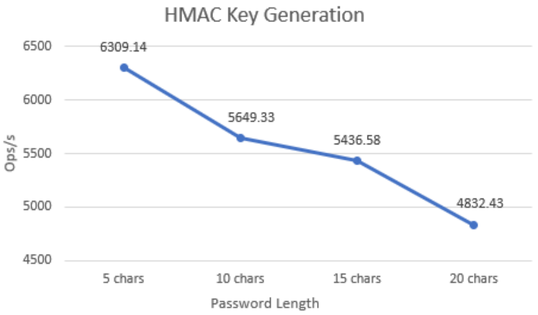


Figure 5.2: HMAC key generation.

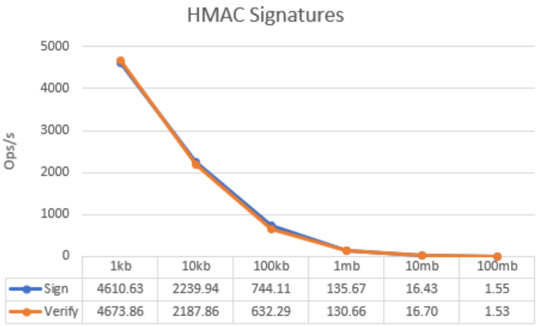


Figure 5.3: HMAC sign and verify.

when comparing MD5 (produces a 128-bit digest) with SHA-512 (produces a much more secure 512-bit digest) we can claim that they have almost the same performance. Being that performance is almost exclusively influenced by the file size, not by the algorithm used.

5.2.2 HMAC

HMAC keys are generated using a password (sequence of chars) so that the key can only be built by a party who knows the password used. The first test we have done with HMAC algorithms was to generate four keys with different password lengths to evaluate if longer passwords influence the performance of HMAC key generation. In Figure 5.2 we can observe that performance is inversely proportional to the size of the password, i.e., bigger passwords will take longer to generate.

Figure 5.3 shows the number of signing and verifying HMAC operations our system can issue per second. In this chart, we only display data that was generated using a five char key. Although bigger keys take long to generate, it does not influence signing and verifying processes, i.e., different keys will take approximately the same time per operation.

5.2.3 AES

Generating AES keys is generating cryptographically secure pseudo-random bytes with 128, 192 or 256-bits long. Although Figure 5.4 has an apparent decrease in performance when generating bigger keys, it is worth to mention that generating a 256-bit key is only 3% slower than generating a 128-bit key.

We thought that it would be interesting to evaluate the performance of using the seven different AES modes to encrypt a 100kb file. *OFB* and *CBC_SHA256* are, by far, the slowest AES modes we have implemented in our system. The other five modes have almost the same performance, as Figure 5.5 demonstrates.

Regarding AES encryption and decryption schemes (Figure 5.6 and 5.7, respectively), we can state they have almost the same performance, even when using different key sizes

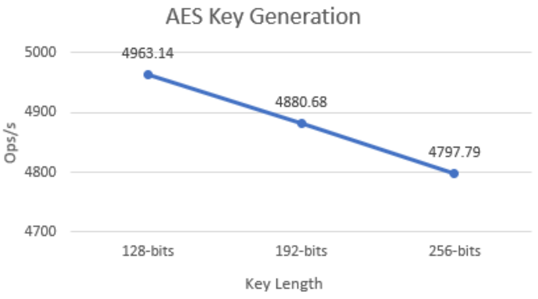


Figure 5.4: AES key generation.

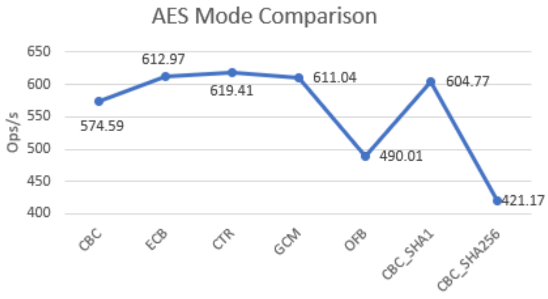
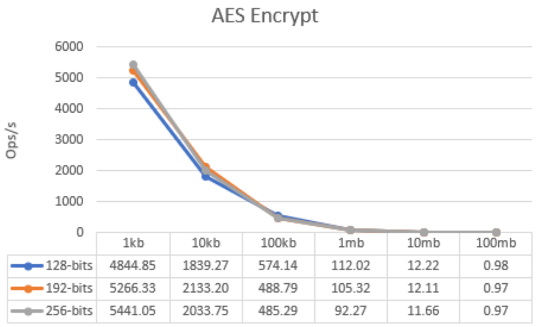
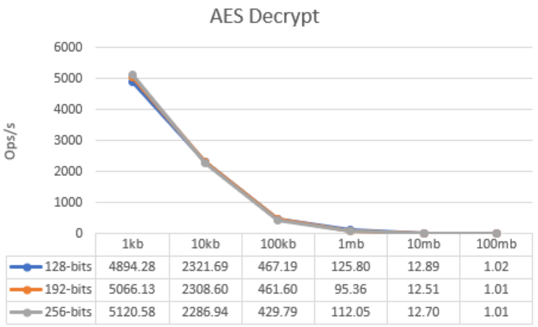


Figure 5.5: AES mode comparison.



	1kb	10kb	100kb	1mb	10mb	100mb
128-bits	4844.85	1839.27	574.14	112.02	12.22	0.98
192-bits	5266.33	2133.20	488.79	105.32	12.11	0.97
256-bits	5441.05	2033.75	485.29	92.27	11.66	0.97

Figure 5.6: AES encrypt.



	1kb	10kb	100kb	1mb	10mb	100mb
128-bits	4894.28	2321.69	467.19	125.80	12.89	1.02
192-bits	5066.13	2308.60	461.60	95.36	12.51	1.01
256-bits	5120.58	2286.94	429.79	112.05	12.70	1.01

Figure 5.7: AES decrypt.

to encrypt the same file (we have used ECB mode for the testings). This information leads us, once again, to the conclusion that the performance of our system is almost exclusively influenced by the file size, primarily with bigger files. Files up to 1mb still have slight performance differences, which is no longer visible in larger files.

5.2.4 RSA

The RSA key generation process is much more complex than merely generating random bytes. It works around prime number generation and is based on the practical difficulty of the factorization of two large prime numbers. It means that for generating bigger keys, larger exponents are used, thus being more complex to find bigger prime numbers. Figure 5.8 shows how many keys of different sizes (from 1024 to 4096-bits length) our system can issue per second. It takes a massive drop in performance from 1024 to 2048-bit length keys, but it is perfectly acceptable with the amount of complexity that a larger key entails.

Figure 5.9 and 5.10 show the behavior of our system against RSA signing and verifying schemes. In the signing chart, we can observe that smaller keys, namely 1024-bit keys, have much higher performance than bigger keys. It is possible to issue 3 times more signing operations with 1024-bit keys than with 2048-bit keys. From 2048 to 3072, or 3072 to 4096, the difference is slightly smaller, but it always remains above two times. This ratio only occurs on files up to 100kb, larger files will take approximately the same

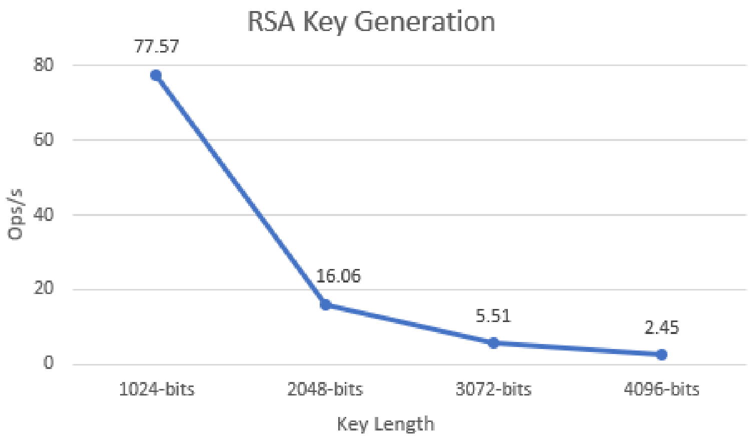
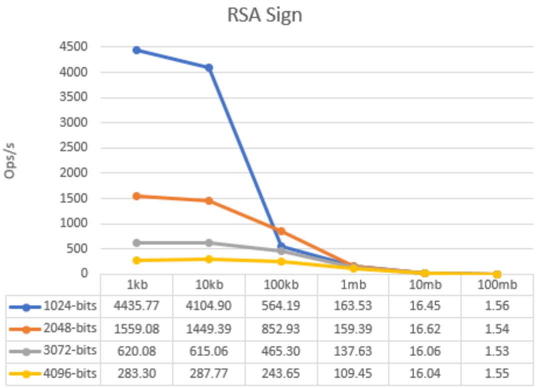
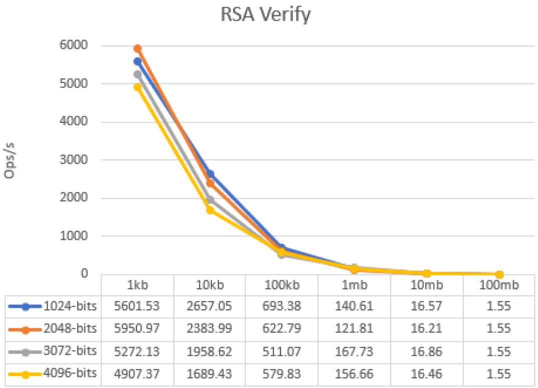


Figure 5.8: RSA key generation.



	1kb	10kb	100kb	1mb	10mb	100mb
1024-bits	4435.77	4104.90	564.19	163.53	16.45	1.56
2048-bits	1559.08	1449.39	852.93	159.39	16.62	1.54
3072-bits	620.08	615.06	465.30	137.63	16.06	1.53
4096-bits	283.30	287.77	243.65	109.45	16.04	1.55

Figure 5.9: RSA sign.



	1kb	10kb	100kb	1mb	10mb	100mb
1024-bits	5601.53	2657.05	693.38	140.61	16.57	1.55
2048-bits	5950.97	2383.99	622.79	121.81	16.21	1.55
3072-bits	5272.13	1958.62	511.07	167.73	16.86	1.55
4096-bits	4907.37	1689.43	579.83	156.66	16.46	1.55

Figure 5.10: RSA verify.

time, even when using bigger keys, as it is the time the file takes to go through every channel.

Regarding the verification procedure, it has a much higher performance than the actual signing operation, having little to no differences regarding key sizes.

As we explained in Section 3.3.4, the sealing process using RSA is as follows: we generate a 256-bit symmetric key, encrypt the file with that symmetric key (using AES CBC mode), generate a 1024-bit asymmetric key and encrypt the symmetric key using the asymmetric key. We then append the encrypted file with the encrypted symmetric key, sending it to the client. The unsealing process is the inverse: decrypting the symmetric key using the respective asymmetric key, decrypting the file with the decrypted symmetric key.

This process takes advantage of the RSA, providing a secure symmetric key exchange and benefit of the higher performance that an AES encryption/decryption scheme can achieve.

In Figure 5.11 we achieve some odd results with 10kb files as 2048-bit keys have slightly higher performance than 1024-bit keys. In the unsealing process, the results

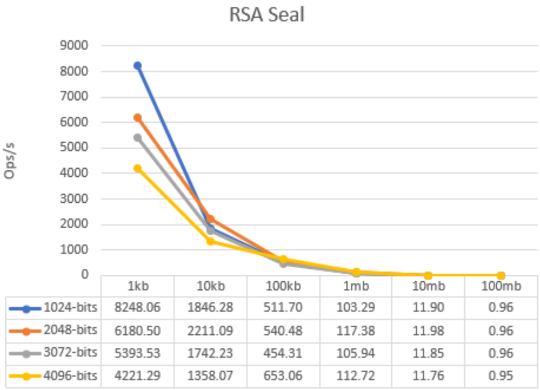


Figure 5.11: RSA seal.

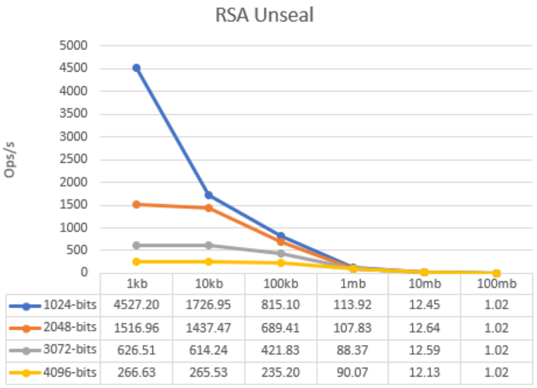


Figure 5.12: RSA unseal.

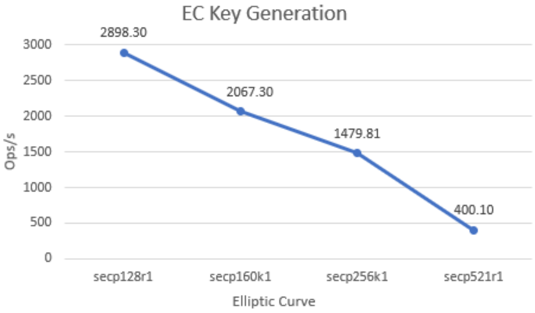
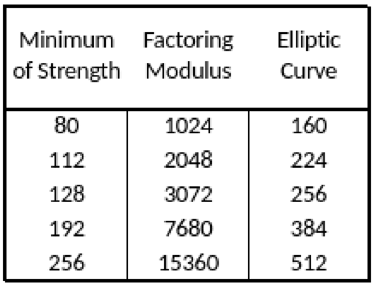


Figure 5.13: EC key generation.



Minimum of Strength	Factoring Modulus	Elliptic Curve
80	1024	160
112	2048	224
128	3072	256
192	7680	384
256	15360	512

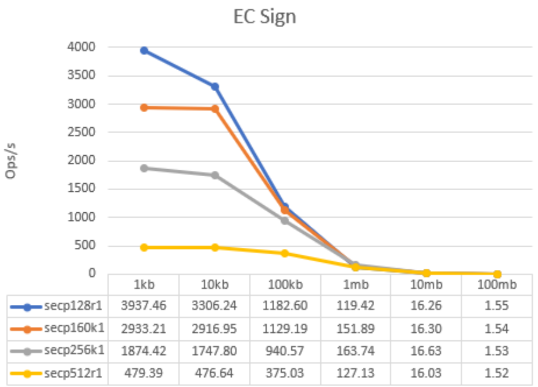
Figure 5.14: Adapted from: NIST Recommendations(2016) [23].

were as expected, where bigger keys have lower performance than smaller ones.

5.2.5 EC

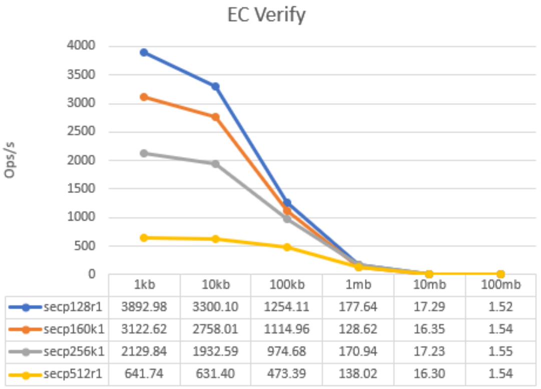
Figure 5.13 shows our system's performance on the elliptic curve key generation, using different key sizes from the same family: *Secp*. The keys used have 128, 160, 256, and 512-bits length. In Figure 5.14, we adapted a table from NIST Recomendations(2016), where it shows how secure are keys from different algorithms. For example, an 80-bit symmetric key provides the same security as a 1024-bit RSA key and a 160 EC key. This comparison shows us how promising are Elliptic Curves and their performance is reflected in our evaluation as we can generate more than 2 000 EC keys of 160-bits but only 77 RSA keys of 1024-bits per second.

Regarding signing and verifying operations using EC keys (Figure 5.15 and 5.16, respectively), we can identify a clear decrease in performance when using bigger keys. The fact that signing or verifying a document with a 160-bit EC key is much slower than using a 1024-bit RSA key may be seen as a disadvantage against Elliptic Curve Cryptography. The main point of EC is that we can provide the same security as RSA keys with much smaller keys.



	1kb	10kb	100kb	1mb	10mb	100mb
secp128r1	3937.46	3306.24	1182.60	119.42	16.26	1.55
secp160k1	2933.21	2916.95	1129.19	151.89	16.30	1.54
secp256k1	1874.42	1747.80	940.57	163.74	16.63	1.53
secp512r1	479.39	476.64	375.03	127.13	16.03	1.52

Figure 5.15: EC sign.



	1kb	10kb	100kb	1mb	10mb	100mb
secp128r1	3892.98	3300.10	1254.11	177.64	17.29	1.52
secp160k1	3122.62	2758.01	1114.96	128.62	16.35	1.54
secp256k1	2129.84	1932.59	974.68	170.94	17.23	1.55
secp512r1	641.74	631.40	473.39	138.02	16.30	1.54

Figure 5.16: EC verify.

5.2.6 Impact of Distributed Version

Three things have changed from the single Virtual HSM version to the Distributed Virtual HSM: we added a second secure channel abstraction that connects one node to every other HSM node, we added a Certificate Authority which issues certificates to trustworthy parties, and we can now run multiple instances of Virtual HSM.

Having a CA has no impact on system performance. It is a static component that its only objective is to issue certificates before the system is running; thus, generating new certificates does not influence performance. When the system is running the IEE verifies if the certificate received was issued and signed by a CA, which takes the same amount of time as verifying a hard-coded key.

Having multiple instances of Virtual HSM running will have little to none impact on system performance as they are independent nodes, with their Server, IEE, and Storage. The only thing that can slightly decrease system performance is the replication process. In Figure 5.17, we demonstrate how much impact it causes to send, and receive, the HSM state. The figure indicates that as the database increases, serialization and deserialization time also increases, taking less than 100 ms to serialize or deserialize 3 000 1024-bit RSA keys. Serialization and deserialization times also include encrypting and decrypting data, receptively.

Although we do have another secure channel abstraction, which forces the system to lose more time opening sockets, running key exchange protocol and encrypting/decrypting every communication, those can not be seen as a performance impact because they are strictly necessary to create and use a secure distributed system.

If we have had time to implement a consistency protocol we would surely notice a drop in performance, on key generation operations, because ensuring strong consistency between nodes is ensuring that the system will only generate the next key after ensuring that the previous key exists in every node.

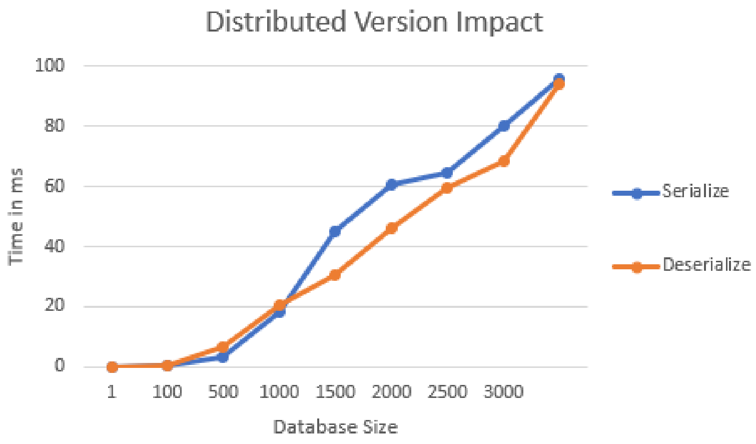


Figure 5.17: Replication process: serialization and deserialization impact.

5.2.7 Comparison with Related Work

Initially, our objective was to compare our solution, Virtual HSM, with a physical HSM, a software HSM, and a Cloud HSM. Although we are fully aware that we can not compete against the performance of a physical HSM, it would be interesting to compare the results. Unfortunately, it was not possible to have access to a physical HSM; therefore, we could not include it in our comparison. The only data we can have in mind is that, for instance, Sun Crypto Accelerator 6000, which costs around 1 300 dollars can perform up to 13 000 signatures per second with a 1024-bit key.

We have two distinct Software HSMs for comparison: SoftHSM (Section 2.1.2.1) and pmHSM (Section 2.1.2.2). To run pmHSM, we would need to configure and use several machines/nodes because it uses threshold signatures as a security measure. As SoftHSM is just a software running on a single machine, we decided to run our comparison against this software HSM.

The only information we could get about the performance of those software HSMs was on pmHSM paper. The authors state that it could perform up to 15 signatures per second using 1024-bits keys. The question that arises is: 15 signatures per second with files of what size? It is a fundamental question as we already have seen that the file size influences, a lot, on the system performance. The same authors also state (Figure 2.2) that pmHSM is 15 to 20 times slower than SoftHSM, which leads us to assume that, on their evaluation, SoftHSM could perform from 225 to 300 signatures per second.

Our SoftHSM evaluation was done on the same machine described in Section 5.1 and results are on Table 5.1. It is worth to mention that we are not using a low-tier processor (as they were using on pmHSM paper). The results that we got on SoftHSM evaluation are surprisingly poor. The best result we could obtain from signing a 1kb file with a 1024-bits key did not even reach five operations per second.

The only requirement of our system is that every node used is an SGX-enabled one. We can provide robust security guarantees, closer to physical HSMs, without compromising

Key Generation	**Virtual HSM**	**Soft HSM**
AES 128-bits	4963.14	5.79
AES 256-bits	4797.79	5.72
EC secp256r1	1479.81	5.60
RSA 1024-bits	77.57	5.63
RSA 2048-bits	16.06	4.36
RSA 3072-bits	5.51	3.44
RSA 4096-bits	2.45	1.83

RSA Sign 1024-bits	**Virtual HSM**	**Soft HSM**
1kb	4435.77	4.46
10kb	4104.90	4.47
100kb	564.19	4.39
1mb	163.53	4.32
10mb	16.45	3.81
100mb	1.56	1.77

Table 5.1: Comparison between VirtualHSM and SoftHSM, in Ops/s.

	Virtual HSM	**AWS Cloud HSM**	
RSA 2048-bits Key Generation	16	0.5	Ops/s
RSA 2048-bits Signing	1559	1100	Ops/s
AES 256-bits Encryption	100	300	mb/s

Table 5.2: Comparison between VirtualHSM and AWS CloudHSM.

performance. Even if we compare our solution with Sun Crypto Accelerator 6000, our solution is only three times slower. It shows how promising is our system, which does not incur high financial costs because it uses commodity hardware.

Regarding Cloud HSMs we did not have time to evaluate AWS CloudHSM but, from what we could investigate trough their page[1], they state that their solution can perform 1 100 RSA signatures using 2048-bits keys, taking 2 seconds to generate a 2048-bits key and encrypt 300mb of data, using AES, with 256-bits key per second. With that information, we made a quick comparison between our solution and AWS CloudHSM (Table 5.2). On RSA key generation and signing operations, we have better results but, on AES encryption, we can only encrypt 1/3 of the data they can encrypt in one second. We have poor performance on encryption because of our SGX limitation, which takes too much time allocating memory inside the Enclave when pagination is required.

5.3 Discussion

With our performance evaluation, we can conclude a couple of things. The first is that the bottleneck of our system is the size of the files used. The allocation of memory inside the Enclave and transitioning big chunks of data to and from the Enclave is relatively slow. It can be seen as an Intel SGX limitation, and there is nothing we can do to improve performance with bigger files. This bottleneck can be seen in every chart we have presented earlier, starting with 1mb files but it is even more evident with 10mb and 100mb files. For instance, with 100mb files, there is almost no difference with every operation our system performs; it always has values from 0.97 to 1.82 operations per second.

---

[1] AWS CloudHSM FAQ: `https://aws.amazon.com/cloudhsm/faqs/`

The other thing we can conclude is that our system behaves very well regarding scalability, i.e., in almost every scenario we increase the file size by a factor of 10, but the system performance does not decrease by a factor of 10.

Chapter 6

Conclusion

The cloud computing paradigm is prevalent and is an attractive solution to deploy services that scale, are highly available and fault-tolerant. However, with the rise of its popularity, the number of security-related incidents also increased. Most of these incidents are associated with privacy, confidentiality, and integrity of the data stored in the cloud.

Hardware Security Modules are dedicated cryptographic processors that safeguard and manage cryptographic keys for operations such as authentication, encryption, and signing. The problem is that with physical HSMs, we are paying a very high cost for the sake of security and performance guarantees, and these costs may be unbearable for some users or companies.

Current solutions for software HSMs make a trade-off between security, usability, and performance, usually sacrificing at least one of these dimensions in favor of others. We have studied different approaches and the state-of-the-art on Software and Cloud HSMs, combined with an analysis of new trusted hardware solutions. By developing a new Virtual HSM, we provide an answer to the question posed in this thesis: *How can we provide the security guarantees of a physical HSM through commodity hardware while increasing availability and reducing financial costs?*

Our Virtual HSM provides an entirely understandable API that allows issuing cryptographic operations inside our IEE, creating a fully secure environment where every sensitive information is managed. The use of SGX-enabled nodes allows us to ensure robust security guarantees such as confidentiality, integrity, and authentication without the need for dedicated hardware; thus, maintaining financial costs very low.

Those nodes are fully prepared to be deployed on the cloud, preferably on different Cloud providers, making use of the abstraction of Cloud-of-Clouds increasing, even more, the system availability. Whether nodes are on different Cloud providers, or not, Client-Enclave and Enclave-Enclave connections are fully secured by secure channel abstractions, making it impossible for attackers and even to the Cloud provider themselves to access and tamper the communication content.

Although we are relinquishing highly sensitive data and delivering it to Cloud providers, every chunk of data is either protected inside the Enclave or, if stored outside the Enclave, it is encrypted in such a way that only the Enclave can decrypt it.

In addition to the guarantees of security and availability that our solution offers, it has an astonishing performance that is much closer to physical HSMs than to state-of-the-art software implementations. Every one of these features makes our solution perfect for testing purposes and even to production environments where small/medium companies prefer to save money and do not recognize the need for the extra performance or international certifications.

6.1 Future Work

The most crucial part that is missing on our system, and we intend to extend as future work is the absence of a strong consistency protocol and a byzantine fault-tolerance protocol. It is fundamental to the proper functioning of our solution in the real world scenario. The first would ensure that we would never have access to an outdated key because if it happens, it will ruin the whole purpose of having an HSM. Some form of byzantine fault-tolerance protocol would also be a massive improvement because the system would be able to handle situations where nodes fail in a byzantine way and even recovering those nodes.

PKCS#11 is considered the API used for the majority of HSMs, thus the need to implement that API. It would increase the ease and likelihood of users and companies that are using other solutions to change to our solution because the API used by them would be the same.

Another extension of our work would be an internationally recognized certification, such as Common Criteria or FIPS 140-2. As explained in Section 2.1, according to FIPS 140-2, there are four levels of certifications. We understand the difficulty of having a real certification of that kind of a pure software implementation, but we would like to achieve at least the level 3 of FIPS, or similar certification.

Lastly, we would like to increase the number of algorithms supported by our solution. From already in use algorithms such as *Twofish* and *Blowfish*, to the most recent and secure ones that may be discovered. The only requirement is that they are supported by *OpenSSL* library.

BIBLIOGRAPHY

[1] M. A. Alzain, E. Pardede, B. Soh, and J. A. Thom. "Cloud Computing Security: From Single to Multi-Clouds." In: *45th Hawaii International Conference on System Sciences* (2012). DOI: 10.1109/HICSS.2012.153. URL: https://www.computer.org/csdl/proceedings/hicss/2012/4525/00/4525f490.pdf.

[2] I. Anati, S. Gueron, S. Johnson, and V. Scarlata. "Innovative technology for CPU based attestation and sealing." In: *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*. Vol. 13. ACM New York, NY, USA. 2013.

[3] A. Arasu, K. Eguro, M. Joglekar, R. Kaushik, D. Kossmann, and R. Ramamurthy. *Transaction Processing on Confidential Data using Cipherbase*. Tech. rep. IEEE, 2015. URL: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/cipherbase.pdf.

[4] A Arm. "Security technology-building a secure system using TrustZone technology." In: *ARM Technical White Paper* (2009).

[5] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song. *Provable Data Possession at Untrusted Stores*. ACM, 2007. ISBN: 9781595937032. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.478.5509{\&}rep=rep1{\&}type=pdf.

[6] *AWS CloudHSM*. URL: https://aws.amazon.com/cloudhsm/ (visited on 08/02/2019).

[7] S. Bajaj. *TPM: Trusted Platform Module*. Tech. rep. 2011. URL: https://zxr.io/teaching/stonybrook/CSE509.2012.F/slides/class13.tpm.pdf.

[8] M. Barbosa, B. Portela, G. Scerri, and B. Warinschi. "Foundations of Hardware-Based Attested Computation and Application to SGX." In: *2016 IEEE European Symposium on Security and Privacy* (2016). DOI: 10.1109/EuroS&P.34. URL: https://repositorio.inesctec.pt/bitstream/123456789/4212/1/P-00K-H7Z.pdf.

[9] M. Bellare, A. Boldyreva, and A. O'neill. *Deterministic and Efficiently Searchable Encryption*. Springer, 2007. URL: http://www-cse.ucsd.edu/users/mihirhttp://www.cc.gatech.edu/{\%}7BâĹijaboldyre,amoneill{\%}7D.

[10] A Bessani, M Correia, B Quaresma, F André, and P Sousa. "DEPSKY: Dependable and secure storage in a cloud-of-clouds." In: *ACM Trans. Storage* 9 (2013), p. 33. DOI: 10.1145/2535929. URL: http://dx.doi.org/10.1145/2535929.

[11] J. Bethencourt, A. Sahai, and B. Waters. "Ciphertext-policy attribute-based encryption." In: *2007 IEEE symposium on security and privacy (SP'07)*. IEEE. 2007, pp. 321–334.

[12] G. R. Borges. "Practical Isolated Searchable Encryption in a Trusted Computing Environment." Master's thesis. 2018.

[13] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi. *Software Grand Exposure: SGX Cache Attacks Are Practical*. Tech. rep. 11th Workshop on Offensive Technologies, 2017. URL: https://www.usenix.org/system/files/conference/woot17/woot17-paper-brasser.pdf.

[14] S. Bratus, N. D'Cunha, E. Sparks, and S. W. Smith. "TOCTOU, traps, and trusted computing." In: *International Conference on Trusted Computing*. Springer. 2008, pp. 14–32.

[15] E. F. Brickell. "Some ideal secret sharing schemes." In: *Workshop on the Theory and Application of of Cryptographic Techniques*. Springer. 1989, pp. 468–475.

[16] F. Cifuentes, A. Hevia, F. Montoto, T. Barros, V. Ramiro, and J. Bustos-Jiménez. "Poor Man's Hardware Security Module (pmHSM)." In: *Proceedings of the 9th Latin America Networking Conference on - LANC '16*. New York, New York, USA: ACM Press, 2016, pp. 59–64. ISBN: 9781450345910. DOI: 10.1145/2998373.2998452. URL: https://dl.acm.org/citation.cfm?id=2998452.

[17] *Cloud Computing: Here is how much a huge outage could cost you*. (Visited on 02/05/2019).

[18] *Common Criteria : New CC Portal*. URL: https://www.commoncriteriaportal.org/ (visited on 02/08/2019).

[19] V. Costan and S. Devadas. "Intel SGX Explained." In: *IACR Cryptology ePrint Archive* 2016.086 (2016), pp. 1–118.

[20] Y. G. Desmedt. "Threshold cryptography." In: *European Transactions on Telecommunications* 5.4 (1994), pp. 449–458. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/ett.4460050407.

[21] J.-E. Ekberg, K. Kostiainen, and N Asokan. "The untapped potential of trusted execution environments on mobile devices." In: *IEEE Security & Privacy* 12.4 (2014), pp. 29–37.

[22] N. Ferguson, B. Schneier, and T. Kohno. "Cryptography engineering." In: *Design Princi* (2010).

[23] D Giry. "Keylength-NIST report on cryptographic key length and cryptoperiod (2016)." In: *URL https://www. keylength. com/en/4/.(accessed: 2017-08-01)* (2017).

[24] R. Griffin. *PKCS #11 Cryptographic Token Interface Base Specification Version 2.40*. Tech. rep. 2015. URL: http://docs.oasis-open.org/pkcs11/pkcs11-base/v2.40/os/pkcs11-base-v2.40-os.html.

[25] *Hardware Security Module (HSM) vs. Key Management Service (KMS) | InterConnections - The Equinix Blog*. URL: https://blog.equinix.com/blog/2018/06/19/hardware-security-module-hsm-vs-key-management-service-kms/ (visited on 02/03/2019).

[26] H. Hinterberger, J. Domingo-Ferrer, and Kashyap. "Trusted Hardware." In: *Encyclopedia of Database Systems*. Boston, MA: Springer US, 2009, pp. 3191–3192. DOI: 10.1007/978-0-387-39940-9_1491. URL: http://www.springerlink.com/index/10.1007/978-0-387-39940-9{_}1491.

[27] J. S. Jang, S. Kong, M. Kim, D. Kim, and B. B. Kang. "SeCReT: Secure Channel between Rich Execution Environment and Trusted Execution Environment." In: *NDSS*. 2015.

[28] B Kaliski. "Cryptoki: A Cryptographic Token Interface, Version 1.0." In: *RSA Laboratories, April* 28 (1995).

[29] S. Kamara and K. Lauter. *Cryptographic Cloud Storage*. Tech. rep. 2010. URL: http://voip.csie.org/shihfan/cloud2013/papers/crypto-cloud09.pdf.

[30] K. Kursawe, D. Schellekens, and B. Preneel. "Analyzing trusted platform communication." In: *ECRYPT Workshop, CRASH-Cryptographic Advances in Secure Hardware*. 2005, p. 8.

[31] L. Lamport. "On interprocess communication." In: *Distributed computing* 1.2 (1986), pp. 86–101.

[32] L. Lamport, R. Shostak, and M. Pease. *The Byzantine Generals Problem*. Tech. rep. 1982. URL: http://people.cs.uchicago.edu/{\%}7B{~}{\%}7Dshanlu/teaching/33100{_}wi15/papers/byz.pdf.

[33] V. Levigneron. "Key deletion issues and other dnssec stories." In: *Proc. Int'l Conf. Artificial Neural Networks (ICANN'41)*. 2011.

[34] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. "Innovative instructions and software model for isolated execution." In: *HASP@ ISCA* 10 (2013).

[35] M. R. Mesbahi, A. M. Rahmani, and M. Hosseinzadeh. "Reliability and high availability in cloud computing environments: a reference roadmap." In: *Human-centric Computing and Information Sciences* 8.1 (2018), p. 20.

[36] NIST. *FIPS PUB 140-2*. 2002. URL: https://csrc.nist.gov/publications/detail/fips/140/2/final.

[37] *OpenDNSSEC » SoftHSM*. URL: https://www.opendnssec.org/softhsm/ (visited on 02/08/2019).

[38] C. Perrin. "The CIA triad." In: *Dostopno na: http://www.techrepublic.com/blog/security/the-cia-triad/488* (2008).

[39] *Personal cloud storage user numbers worldwide 2014-2020 | Statistic.* URL: https://www.statista.com/statistics/499558/worldwide-personal-cloud-storage-users/ (visited on 02/12/2019).

[40] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. "CryptDB." In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles - SOSP '11*. New York, New York, USA: ACM Press, 2011, p. 85. ISBN: 9781450309776. DOI: 10.1145/2043556.2043566. URL: http://dl.acm.org/citation.cfm?doid=2043556.2043566.

[41] *Regulatory Compliance in the Cloud.* URL: https://www.tripwire.com/state-of-security/regulatory-compliance/regulatory-compliance-cloud/ (visited on 02/08/2019).

[42] L. Rizzo. "Effective erasure codes for reliable computer communication protocols." In: *ACM SIGCOMM computer communication review* 27.2 (1997), pp. 24–36.

[43] RSA Laboratories. *PKCS # 11 : Cryptographic Token Interface Standard*. 1997. URL: ftp://193.174.13.99/pub/pca/docs/PKCS/ftp.rsa.com/pkcs-11/v2drft2.pdf.

[44] *SafeNet PCIe HSM - Cryptographic Acceleration and Key Security | Gemalto.* URL: https://safenet.gemalto.com/data-encryption/hardware-security-modules-hsms/pci-hsm/ (visited on 02/11/2019).

[45] N. Santos, K. P. Gummadi, and R. Rodrigues. "Towards Trusted Cloud Computing." In: *HotCloud* 9.9 (2009), p. 3.

[46] N. Santos, H. Raj, S. Saroiu, and A. Wolman. "Using ARM TrustZone to build a trusted language runtime for mobile applications." In: *ACM SIGARCH Computer Architecture News* 42.1 (2014), pp. 67–80.

[47] J. Schlyter -jakob. *HSM Hardware Security Modules*. Tech. rep. URL: https://www.iis.se/docs/hsm-20090529.pdf.

[48] M.-W. Shih, S. Lee, T. Kim, and M. Peinado. "T-SGX: Eradicating controlled-channel attacks against enclave programs." In: *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA*. 2017.

[49] E. Sparks and S. W. Smith. "TPM reset attack." In: *Web page* (). URL: http://www.cs.dartmouth.edu/{~}pkilab/sparks.

[50] W. Stallings and L. Brown. *Computer security: Principles and Practice*. 3rd. Upper Saddle River, NJ, USA: Prentice Hall Press, 2014. ISBN: 9780133773927.

[51] R. D. D. Team and Others. "Dnssec test plan for the root zone (draft)." In: *ICANN and VeriSign, Tech. Rep.* (2010).

[52] *TPM 2.0 Library Specification - Trusted Computing Group*. URL: https://trustedcomputinggroup.org/resource/tpm-library-specification/ (visited on 02/10/2019).

[53] *Understanding Hardware Security Modules (HSMs)*. URL: https://www.cryptomathic.com/news-events/blog/understanding-hardware-security-modules-hsms (visited on 02/03/2019).

[54] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. “Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution.” In: *27th Security Symposium (Security 18)*. 2018, pp. 991–1008.

[55] W. Vogels. “Eventually consistent.” In: *Communications of the ACM* 52.1 (2009), pp. 40–44.

[56] P. K. Yu. “Towards the seamless global distribution of cloud content.” In: *PRIVACY AND LEGAL ISSUES IN CLOUD COMPUTING, Anne SY Cheung and Rolf H. Weber, eds., Edward Elgar Publishing* (2015), pp. 180–213.

[57] K. Zetter. “Compay caught in texas data center raid loses suit against FBI.” In: *Wired Magazine, April* (2009).