Miguel Afonso Madeira

Licenciado em Engenharia Informática

Towards more Secure and Efficient Password Databases

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática

Orientador: Bernardo Ferreira, Researcher, NOVA University of Lisbon

Co-orientador: João Leitão, Assistant Professor, NOVA University of Lisbon

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Novembro, 2018

Towards more Secure and Efficient Password Databases

Este documento foi gerado utilizando o processador (pdf)LaTeX, com base no template "novathesis" [1] desenvolvido no Dep. Informática da FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

Acknowledgements

I would like to thank my advisors for the help given during my work in this thesis. I am also thankful for the support from FCT/MCTES, through the strategic project NOVA LINCS (UID/CEC/04516/2013) and project HADES (PTDC/CCI-INF/31698/2017), and from the LightKone project (H2020 grant agreement ID 732505).

Abstract

Password databases form one of the backbones of nowadays web applications. Every web application needs to store its users' credentials (email and password) in an efficient way, and in popular applications (Google, Facebook, Twitter, etc.) these databases can grow to store millions of user credentials simultaneously. However, despite their critical nature and susceptibility to targeted attacks, the techniques used for securing password databases are still very rudimentary, opening the way to devastating attacks. Just in the year of 2016, and as far as publicly disclosed, there were more than 500 million passwords stolen in internet hacking attacks.

To solve this problem we commit to study several schemes like property-preserving encryption schemes (e.g. deterministic encryption), encrypted data-structures that support operations (e.g. searchable encryption), partially homomorphic encryption schemes, and commodity trusted hardware (e.g. TPM and Intel SGX).

In this thesis we propose to make a summary of the most efficient and secure techniques for password database management systems that exist today and recreating them to accommodate a new and simple universal API.

We also propose SSPM(Simple Secure Password Management), a new password database scheme that simultaneously improves efficiency and security of current solutions existing in literature. SSPM is based on Searchable Symmetric Encryption techniques, more specifically ciphered data structures, that allow efficient queries with the minimum leak of access patterns. SSPM adapts these structures to work with the necessary operation of password database schemes preserving the security guarantees.

Furthermore, SSPM explores the use of trusted hardware to minimize the revelation of access patterns during the execution of operations and protecting the storage of cryptographic keys. Experimental results with real password databases shows us that SSPM has a similar performance compared with the solutions used today in the industry, while simultaneous increasing the offered security conditions.

Keywords: Password Databases, Searchable Encryption, Trusted Hardware, Web Applications.

Resumo

Base de dados de passwords formam o esqueleto de aplicações web atuais. Todas as aplicações web necessitam de armazenar as credenciais do utilizador (email e password) de uma maneira eficiente, e em aplicações populares (Google, Facebook, Twitter, etc.) estas bases de dados podem crescer e ter que armazenar milhões de credenciais simultaneamente. No entanto, apesar da sua natureza crítica e susceptibilidade a ataques, as técnicas usadas para armazenar passwords continuam a ser muito rudimentares, abrindo o caminho para ataques devastadores. Só no ano de 2016, e apenas os números partilhados, houve mais de 500 milhões de passwords roubadas devido a ataques de hackers.

Para resolver este problema estamos determinados en estudar os principais esquemas de cifra existentes, tais como esquemas criptográficos que preservam propriedades (por ex. Cifra Determinista), estruturas de dados cifradas que suportam operações (por ex. Cifra Pesquisável), Encriptação Homomórfica Completa e Parcial, e Hardware Confiável (como TPM e Intel SGX).

Nesta tese fazemos um resumo resumo das técnicas mais eficientes e seguras para base de dados de passwords que existem atualmente, e recriá-las para acomodar uma nova e simples API universal.

Também propomos SSPM (Simple Secure Password Management), um novo protocolo criptográfico para proteção de BDs depasswords que simultaneamente melhora a eficiência e segurança de soluções atuais da literatura. SSPM é baseado em técnicas de Cifra Simétrica Pesquisável, mais especificamente estruturas de dados cifradas que permitem operações de consulta eficientes com revelação de mínima informação.SSPM adapta estas estruturas de forma a suportar as operações necessárias numa BD de passwords e preservando as suas condições de segurança.

SSPM explora também o uso de hardware confiável moderno baseado em atestação (TPM e Intel SGX) como forma de minimizar ainda mais a revelação de informações durante a execução de operações e como forma de proteção de chaves criptográficas. Resultados experimentais com BDs de passwords reais demonstram que SSPM possui um desempenho semelhante a soluções usadas hoje na industria,

simultaneamente melhorando em muito as condições de segurança oferecidas.

Palavras-chave: Aplicações Web, Base de Dados de Passwords, Cifra Simétrica Pesquisável, Hardware Confiável.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

1.1 Context and Motivation

Nowadays, data in cloud computing is usually available from everywhere in the world. Password databases is by far the most important piece of data that an average user can have stored in an outside storage system. As the times passes, cloud computing is getting more cheaper, more popular and with more hardware capabilities, so it's only natural that its usage has increased exponentially in recent years and more companies tend to store important data in the cloud, which includes password databases.

As the access to the Internet continues to grow everyday, more and more people around the globe tend to join it. More people means increased storage capacity needed to store user's personal data in popular applications like Google, Twitter, Facebook, Instagram, just to name a few. Each of these websites contains sensible information about their users, which is usually protected by a username and a password. That's what always separated ones personal information from the rest of the world, their password, just like your house key protects your house from everybody coming in uninvited, the users password supposedly is required and mandatory for them to access their account in web applications.

1.2 Problem Statement

Despite their critical nature and susceptibility to targeted attacks, the techniques used today for securing password databases are still very rudimentary, opening the

way to devastating attacks. Adversaries, including internet hackers, tend to explore new and innovative ways to attack and access the data, and unfortunately they're still very successful.

Just in the year of 2016, and as far as publicly disclosed, there were more than 500 million passwords stolen in Internet hacking attacks [49]. Last year, on 2017 all 3 billion of Yahoo accounts were breached [30]. This accounted for the users full name, usernames and password.

Password databases are particularly susceptible to snapshot attackers, which is a hacker that may get a small time window of access to the database and makes a snapshot copy of all available information. If the data in the database storage is not encrypted, and is just in plain text, the adversary can just look at it and learn all the user's credential data. If the users credentials in the snapshot copy are hashed, the adversary can just run a Brute Force or a Dictionary Attack to create a collision, among others ways to discover the original plaintext that was hashed.

A common solution to this problem is to encrypt the users credential data, and every time the user wishes to enter the system, he provides its username and password. The system then decrypts its stored data with the corresponding key and compares the decrypted plaintext user's password with the user's entered password. One problem with this solution is that there's a time frame in which the original password is in its plaintext format, for the system to make the comparison between the password values, making the system vulnerable to an attack in this time frame, as the adversary can create a snapshot at this moment to gain the original plaintext password.

Another solution from the state of the art is to encrypt the data in a way that allows computations over the encrypted data. However this method is very expensive and somewhat slow, causing it to be inefficient for the users to make operations like logging in or registering.

These are the reasons some websites still store the users passwords with just the hashed values of them, or just in plaintext, to remain fast and simple for the system when a user wants to complete a login operation.

So in this thesis we will try to answer the following question:

Can we build a secure password database system that once attacked by an adversary can not be compromised and still be efficient enough for users to login and register under the "one second" usability mark?

1.3 Objectives

SSPM (Simple Secure Password Management) is a new password database system that protects users credentials while allowing them to efficiently perform login and register operations. SSPM uses a not so usual way of storing the users credential so it can better evade attacks of adversaries aimed to a specific user. Usually the user's credential are stored in a Map (where the key is a the username of the users, and the value is the users password, or a table (in where there's a column for each of the user's credential). SSPM hashes the result of the concatenation of the username, the password and a fixed salt, and stores it in a set. When the users logins to the system, SSPM makes the same hashing of the concatenation process, and verifies if the result of the hash is inside the set. In an affirmative state, the user is granted access.

Our system, with the help of cryptographic techniques for computing on encrypted data and the help of Intel SGX, TPM and other secure hardware, can securely defend against attacks made by adversaries and can avoid the attack of leaking access patterns, even to rogue servers and malicious OS and HyperVisors.

We implemented the most used password database mechanisms used in the industry today, and designed an universal API with the most common password database operations. The objective of this is to easily use and test the encryption and hashing schemes that are most used today. We also used stress tests to all implemented systems to compare the performance of the systems. The implemented API makes the process of testing much easier, and it can also be used by other users to easily test the schemes.

Following the objectives we propose as main objective to build more secure and efficient methods of password database management.

1.4 Contributions

In the list below we enumerate the main contributions provided in this thesis:

- We design and implement an universal API with the most used password database management methods are used today in the industry and in the literature. The API includes login, register, change username, change password and delete user operations. For the implementation of existing password database schemes (like B-Crypt, and PolyPasswordHasher) we used open-source repositories (that will be later mentioned) and make some changes to the code, while keeping the copyright authors, to adapt its implementation to our API.

- We designed and implemented new cryptographic mechanisms to protect password databases. We call our scheme SSPM (Simple Secure Password Management). Several versions of this scheme were implemented, each one adding more security measures but trading off performance.
- Conception of a scheme for password database management that we called SSPM, based on trusted hardware, such as Intel's technology Intel SGX (Software Guard Extensions). We keep the same basic mechanisms of SSPM, but make all cryptographic operations inside the Enclave created by SGX, as well as store the cryptographic keys inside the enclave.
- Experimentally evaluate the performance of our schemes and compare them previous works from the literature, using the API that we mentioned above. To achieve a more realistic performance test we acquired a password database data set based on various password leaks, that were revealed through the years.
- The design of a GUI (Graphical User Interface) so we can better demonstrate the work done by us, and give the reader a better understanding of the implemented password database schemes and show how they store user's credentials.

By the end of the work in this thesis, we explained 2 different solutions to be presented, that are proofed to be as (if not more) secure and efficient then current and past methods used in the industry. For efficiency we measured the time complexity of how fast the schemes can perform the operations of the built API.

1.5 Report Organization

The remaining of this report is organized by the following structure:

- **Chapter 2:** In this chapter we present the important background and related work that includes fundamental concepts necessary to the understanding of the concept of this thesis.
- **Chapter 3:** In this chapter called "Designed Solutions", we present the methods involved in the design and ideas of the password database schemed used in the thesis, like the initial view of the system model and its architecture, the adversary model, and the detailed algorithms of our schemes.
- **Chapter 4:** In this chapter called "Implementation and Evaluation", we show in more detail on how we implemented various common hashing schemes and other used password database scheme, and implemented it to our universal API.

We also show how we implemented the versions of SSPM that are only based in cryptography, and how we implemented the versions that depend on Hardware Modules, like TPM and Intel SGX. In the second part of the chapter we show the performance evaluation that we made for the implemented schemes, using stress tests to the Login and Register operations through our API, using up to ten million users entries for the performance tests.

- **Chapter 5:** In this chapter, simply entitled "Conclusion", we show our conclusions on the making of this thesis and also leave a section dedicated to the future work that can be later picked up by another researcher to further progress the work made in this thesis.

CHAPTER 2

Background and Related Work

In this chapter, we discuss the relevant background and related work with importance for this thesis. The chapter is divided into three sections. In Section 2.1 we present several ways to make operations on encrypted data on a cloud server. In Section 2.2 we present Searchable Encryption and Searchable Symmetric Encryption. In section 2.3 we present the state of the art in password databases and also present the most common attacks made by adversaries on those.

2.1 Computation On Encrypted Data

Everyday cloud computing gets faster, more available and cheaper, so it's only natural that extra users are joining this types of services to store their data. Cloud computing enable network on demand and convenient access to a shared pool of configurable computing resources like networks, servers, storage, applications and services [37] and can be provisioned and released with minimal user effort or interaction with the service provider. But the main advantage of cloud computing is that is accessible from almost anywhere in the world, at any time. As long as there is Internet access, the user can access the services of the cloud to outsource his data and later can request to access it and make computations over it. Cloud users include private parties and also companies, that use this system as the primary focus to their business models, so they don't have to expand their resources to servers infrastructures and have the extra cost of maintaining them.

The problem with cloud computing is that an unwanted party, like the manufacturer, a software engineer or even an Internet attacker, can see the users data if it

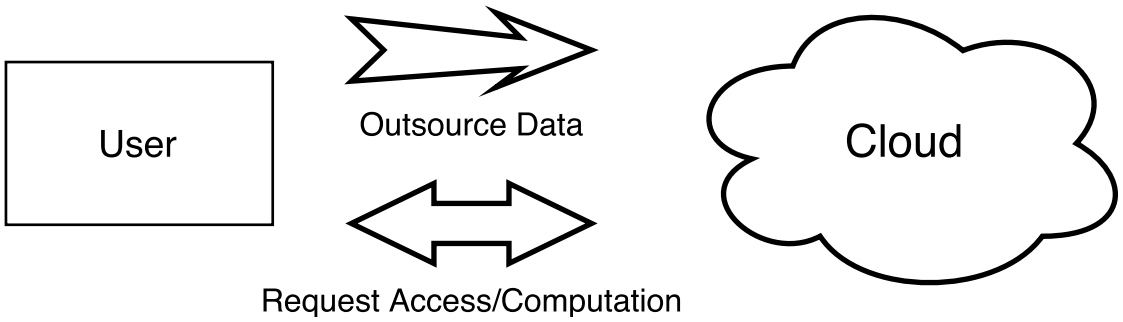


Figure 2.1: User-Cloud Interaction

is not encrypted. That's why the data stored in the storage cloud should always be encrypted, but this raises the problem that computations can't be simply done in an encrypted form.
Computation over encrypted data introduces several cryptographic methods to securely perform computations without revealing private and sensitive information to the cloud, and without requiring the user's secret key. This means that the user or the cloud server doesn't have to decrypt the whole encrypted data to perform a desired function [22], as the computations are done on the encrypted form of the data.
In the following sub-sections we are going to discuss Property-Preserving Encryption (Deterministic Encryption and Order Preserving Encryption), Homomorphic Encryption (Fully Homomorphic Encryption and Partial Homomorphic Encryption) and Computations on Trusted Hardware (TPM, Intel SGX and ARM TrustZone).

2.1.1 Property-preserving Encryption

Property-preserving Encryption enables some properties to be preserved on encrypted data. For this the algorithm has to preserve some properties of the encrypted data, such as its order. These associated properties allow for some pre-determined computation on the encrypted data such as range queries, even if the data was generated independently and individually [42]. In the following sections we present two types of Property-Preserving Encryption.

2.1.1.1 Deterministic Encryption

Deterministic Encryption is a cryptosystem, that given a certain plaintext and key, always produces the same exact ciphertext, even over separate executions.

We can see an example of Deterministic Encryption in **Figure 2.2**. In this example there are two different runs of Deterministic Encryption, in the first run we input

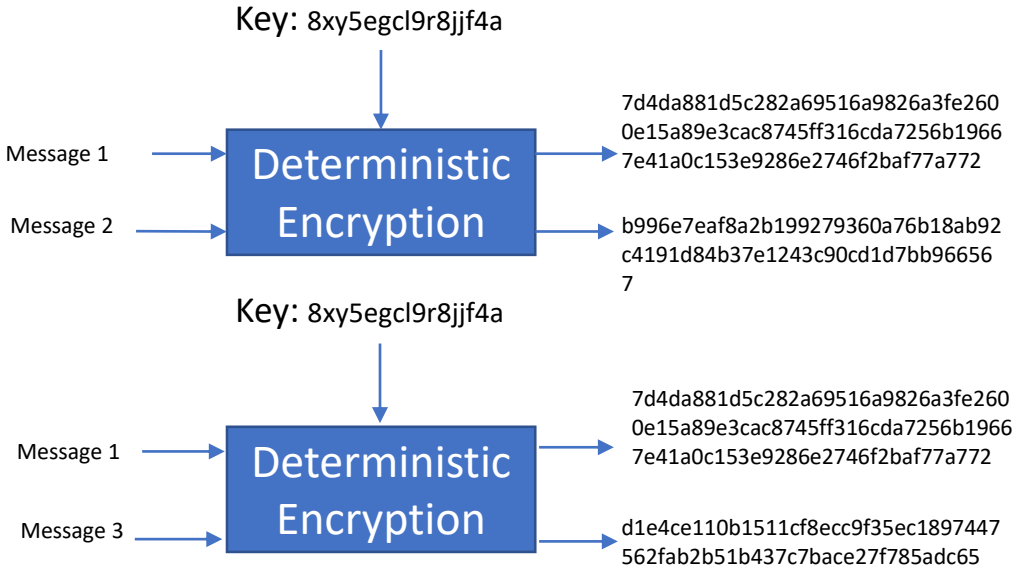


Figure 2.2: Example of Deterministic Encryption

Message1 and Message2 along with a key, and get a different result for each one. In the second run we input the same Message1, and Message3, with the same key as the first run and also get a pair of results. As we can note, the output for the same input (Message1 + key) is the same for each run and when the input is different, the output will also be different (see the output of Message2 and Message3).

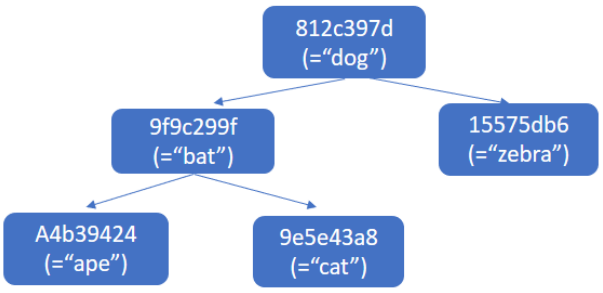
2.1.1.2 Order-Preserving Encryption

Order preserving encryption is a deterministic encryption scheme, that preservers the order of the numerical plaintext [5]. This is useful because it allows efficient range queries on encrypted data. This way an untrusted database server can index sensitive data in encrypted form, inside a data structure (such as a b-tree) permitting range queries (for example, from a to b), to locate the desired function in logarithm time. Basically, given two plaintexts x and y the definition of OPE can be given as follows:

$$x > y -> encryption(x) > encryption(y).$$

In figure **Figure 2.3** we can see a example of how to store the encrypted data with OPE in a database storage. The figure contains a reference table and a b-tree. The reference table contains the hexadecimal value of every plaintext and also the ciphertext, that is the result of deterministic encryption on the plaintext, so the algorithm can calculate where on the b-tree should look for a value to insert or

remove, and also allowing queries. An operation on this structured data should have on average time complexity of log(n), with n being the number of elements of data in the database [44].



CipherText	Plain Text	Hex Value
A4b39424	ape	61 70 65
9f9c299f	bat	62 61 74
9e5e43a8	cat	63 61 74
812c397d	dog	64 6f 67
15575db6	zebra	7a 65 62 72 61

Figure 2.3: B-Tree of the result of a OPE with reference table

2.1.2 Homomorphic Encryption

Homomorphic Encryption comes from homomorphic abstract algebra, that preserves a structure map between the algebraic structures. Basically, Homomorphic Encryption allows some computations to be made in the ciphered text as if they were in plaintext and outputs the result in ciphered text that when deciphered matches the result as if the same operation was made in the original plaintext [59]. There are several drawbacks to Homomorphic Encryption, specially on Fully Homomorphic Encryption, causing it to not be very practical due to its efficiency. Partially Homomorphic Encryption cuts some of the costs of Fully Homomorphic Encryption.
In the following sections we present two types of Homomorphic Encryption.

2.1.2.1 Fully Homomorphic Encryption

Fully Homomorphic Encryption [17] allows for arbitrary computations on encrypted data, namely multiplication and addition operations. Despite the extensive work done in this subject, Fully Homomorphic Encryption continues to not be considered efficient enough for everyday use. One example of this, is the work made in [18], a Fully Homomorphic Encryption work was accomplished with an AES Circuit, but it takes 7 hours to make the first successful operation (despite having the algorithm become faster as more rounds are made) and it also requires a machine with 256 GB of RAM to run it.

2.1.2.2 Partially Homomorphic Encryption

Partially Homomorphic Encryption is a form of Homomorphic Encryption, only certain mathematical homomorphic operations can be made, like additive or multiplicative homomorphism, but not both operations [40]. A Partially Homorphomic Encryption supports adding Homomorphism if and only if: $\xi(x)+\xi(y)=\xi(x+y)$. A Partially Homorphomic Encryption supports Multiplicative Homomorphism if and only if: $\xi(x)*\xi(y)=\xi(x*y)$. Several algorithms are presented below that preserve one of the two mathematical operations available on Partially Homorphomic Encryption:

- **RSA**

  RSA, or more specifically unpadded RSA, is used to exhibit **multiplicative homomorphism**. It multiplies two RSA ciphertexts values, with the decrypted result being the same as if the multiplication was made with the two values in plaintext citeGentry09.
  Being the modulus m and exponent e constant integers, the homomorphic property is given by:

$$\xi(x)*\xi(y)=x^e*y^e \mod m=(x*y)^e \mod m=\xi(x*y) \tag{2.1}$$

- **ElGamal**

  Similarly to RSA, ElGamal is also used to exhibit **multiplicative homomorphism**, which means multiplying various elements of a ciphered text and decrypting the result equals to multiplying the values in their original plaintext in unencrypted form [36]. The ElGamal algorithm exhibits multiplicative homomorphism working this way:
  Given a plain text x1, private key a, public keys k1, k2 and k3 and a nonce no1 where $\xi(x1,no1)=(y1,y2)$, where y1 and y2 mean:

$$\begin{aligned} y1 &\equiv k1^{no1} \mod k3 \\ y2 &\equiv x1*k2^{no1} \mod k3 \end{aligned} \tag{2.2}$$

  Multiplying the plaintexts x1 and x2:

$$\begin{aligned} x1*x2 &= (y1,y2)*(y3,y4) \\ &= (y1*y3, y2*y4) \\ &= (k1^{no1}*k1^{no2}, (x1*k2^{no1})*(x2*k2^{no2})) \\ &= (k1^{no1+no2}, x1*x2*k2^{no1+no2}) \end{aligned} \tag{2.3}$$

Decrypting will give us:

$$decrypt(k1^{no1+no2}, x1 * x2 * k2^{no1+no2}) = x1 * x2 \tag{2.4}$$

- **Paillier**

  Paillier is used to exhibit **additive homomorphism**. Each component of multiple ciphertext is multiplied with its respective component. The result when decrypted is equal to the sum of the values in plaintext [41]. As the original paper states, the Encryption and Decryption in this algorithm is made as it is shown bellow.

  Encryption:

  plaintext $m < n$

  select a random $r < n$

  ciphertext $c = g^m * r^n \mod n^2$

  Decryption:

  ciphertext $c < n^2$

  plaintext m $= \frac{L(c^\gamma \mod n^2)}{L(g^\gamma \mod n^2)} \mod n$

  As stated previously the result of multiplying two ciphertexts values will decrypt to the sum of those two values in plaintext, given that x is the first plaintext and y is the second one:

$$decrypt(encrypted(x, r1) * encrypted(y, r2) \mod n^2) = x + y \mod n \tag{2.5}$$

  Pailler encryptions also allows the use of homomorphic multiplication of ciphertext with plaintext.

2.1.3 Computations on Trusted Hardware

Computations on Trusted Hardware is about developing trusted technologies that guarantee the user safety while running the software on theirs devices. A device can be considered trusted if it always behaves in an expected manner even if an adversary tries to attack it [34]. In this section we present several technologies that attempt to execute software on a remote computer owned and maintained by an untrusted party, with some integrity and confidentiality guarantees [12]. In **Figure 2.4** the user relies on a Remote Computer, owned by an untrusted party, to perform some computation on. The user trusts the manufacturer that has a piece of hardware in the remote computer, so the user has assurance of the computation and confidentiality of its data.

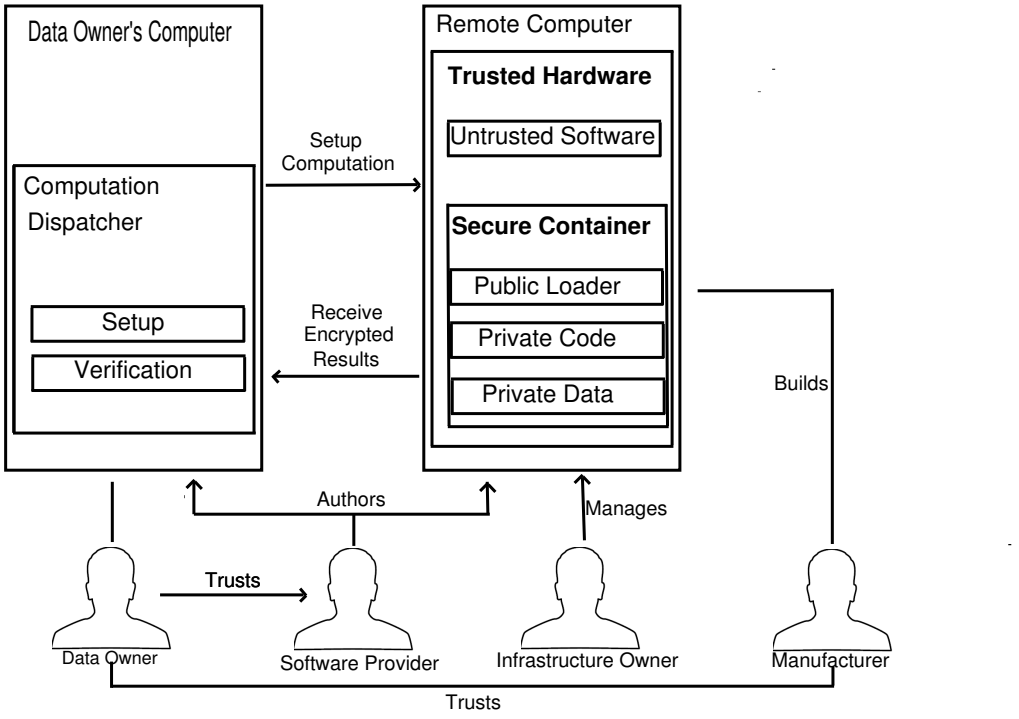


Figure 2.4: Trust Computing on Secure Remote Computation [34]

2.1.3.1 TPM

TPM refers to Trusted Platform Module and is a hardware module incorporated and deployed in a motherboard, smart card or processor that provides the resources needed for trusted computing [51].

Through secret sharing and cooperation with other hardware and software components, the TPM provisions three services: authenticated boot, certification, and encryption.

The authenticated boot service allows to verify that the boot of the entire operating system is made in well defined stages at which only approved versions of the modules of the OS are loaded. This could be done by verifying a digital signature associated with each module. Additionally, a tamper-proof log of the loading process is kept by the TPM that later can be consulted. Tampering is detected using cryptographic hash functions. It is also possible to configure the TPM to include additional hardware and application and utility software in its TCB given some restrictions to prevent threats. This service can be used to guarantee that the machine which hosts the TPM is in a well defined and trusted state after booting.

Despite the advances in TPM, this technology seems to have been compromised in 2010 by Christopher Tarnovsky, but this was disregarded by the manufacturing

company of the chips, Infineon, as the company stated that it was necessary a high skill level of hardware experience, as it required removal of the chip's case top layer, then tapping into a data bus to get the unencrypted data [53]. In October 2017 a new code library by Infineon, exposed some vulnerabilities with the system, as it allowed some private keys to be guessed from the public keys, subsequently a lot of data was compromised [21].

2.1.3.2 Intel® SGX

Intel®SGX (Intel's Software Guard Extensions) is a set of CPU instructions that can be used by applications to set aside private regions of code and data called enclaves [24]. SGX was introduced with the Skylake architecture and is considered the successor to TPM. In SGX, the hardware establishes a secure container and then the user uploads its data and desired computation to the secure container. SGX protects the data's confidentiality and integrity while the computation on the data is being executed [34], even from a potentially malicious OS/Hypervisor.

To be more specific, SGX reserves a space in memory called PRM (Processor Reserved Memory). The PRM contains an EPC (Enclave Page Cache). The EPC contains memory for all enclaves and consist of 4KB pages that store the enclaves data and code, with the maximum of 128MB (32768 pages). The EPC is always encrypted.

The CPU then protects this part of the memory from all non-enclaved attempts to access the memory blocks, this including the kernel, hypervisors and SMM accesses. The initial code and data is uploaded by an the untrusted system software, relying in software attestation, so it can be proved that it is the user that is communicating and the the user is interacting with the right enclave. The communication is proofed by a cryptographic signature that certifies the hash of the secure container contents matches with the ones of the expected. After this, the CPU uploads the unprotected memory to EPC pages and commits them to the enclave area. The CPU now marks the enclave to initialized and its contents are hashed. Operations on the enclave part can only undergo under specific CPU call instructions. After all the computation is done, the CPU hashes the results and returns it to the system software.

With Intel SGX version 2 it will be added the ability to dynamically add pages and threads to a running enclave and the feature to increase the enclave's heap during runtime [12].

2.1.3.3 ARM TrustZone

ARM's Trustzone approach to secure computing is to separate secure and non-secure hardware, so that it can stop non-secure software from accessing secure resources directly. For the processor there are 2 types of software: secure and non-secure. It separates the two by having a System-on-Chip (SoC) that virtualizes 2 different CPU's. This way the important assets can be protected from adversary attacks such as software attacks and common hardware attacks [33].
This technology can be implemented in ARM Cortex-A, Cortex-M23 and Cortex-M33 based systems. An overview of the ARM TrustZone can be seen in **Figure 2.5**, where there 2 are distinct hardware CPU virtualizations, one is non-trusted and the other trusted.

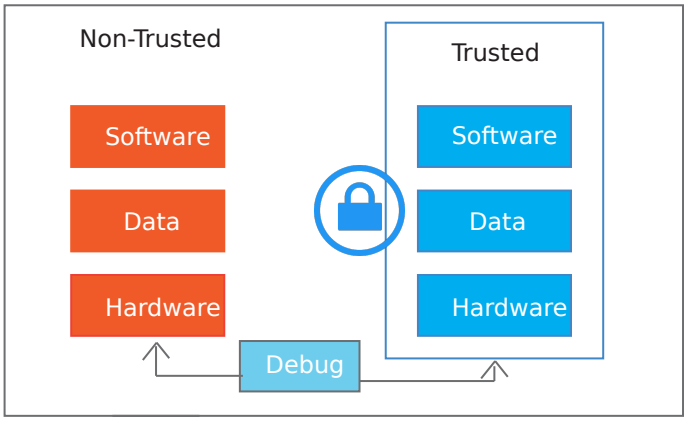


Figure 2.5: Overview of ARM Trustzone [33]

2.2 Searchable Encryption

Searchable Encryption allows the user to outsource a collection of encrypted data on a remote server in a private manner, while still maintaining the ability to make keywords searches over it. These search functionalities are supported without decrypting the data, with the smallest possible loss of confidentiality.

More specifically, Searchable Encryption allows a user to create a search token from the keywords that he wants to search for, in a way that given the token, the server can retrieve the encrypted contents relevant to those keywords. The search token represents the encrypted query and can only be generated by the users with the appropriate secret key [22].

Nonetheless, it includes some type of Deterministic Encryption as the use of deterministic primitives. A disadvantage of this method is the leakage of search and access patterns when some operations are made that adversaries can recognize. For

this reason the security of query keywords and search results should be guaranteed [41].

Searchable Encryption includes 4 parties: the data owner, the semi-trusted server, a collection of users that will read the data and the key generator.

- **The Data Owner** - The data owner outsources all of his data together with appropriate keywords. The user is responsible for encrypting the data with a particular cryptographic scheme like creating an index that enables searching over his encrypted data, see figure 2.6.
- **The Cloud Server** - It's used to outsource the encrypted data. Upon receiving a trapdoor from a user with keywords, the server searches over the encrypted data using the Index created by the data owner. After finishing the search for the relevant content, the server returns it in an encrypted form.
- **The Data User** - The data user that wants to send a query to the server, so he gathers his secret key and creates an encrypted trapdoor containing his query keywords and sends the trapdoor to the server. After completed, the server sends the encrypted results to the user, see figure 2.6. This user can be the same as the data owner.
- **Key Generator scheme** - This party is considered trusted and is responsible for the generation and management of secret keys, like the encryption and decryption keys [46] [56]. In some application, this part can be considered to be the data owner.

In Figure 2.6 we show the execution of an upload operation, where the user builds an index I, based on the Database DB and the collection of Messages W, and sends to the Server the index I and the collection of the encrypted messages, from M1 to Mn. To make a query operation, the user creates a Trapdoor T, containing his secret key and the keywords f, and sends to the server the Trapdoor T. The server then searches over with the index I and the trapdoor T, the relevant messages M and returns them to the user.

There are two main types of Searchable Encryption, those types are Symmetric Searchable Encryption (SSE) and Public-Key Searchable Encryption (PKAS). In the following section we are only going to present Searchable Symmetric Encryption.

2.2.1 Searchable Symmetric Encryption

Searchable Symmetric Encryption (SSE) is one of the two main types of Searchable Encryption, with the other being Public key encryption with keyword search (PEKS).

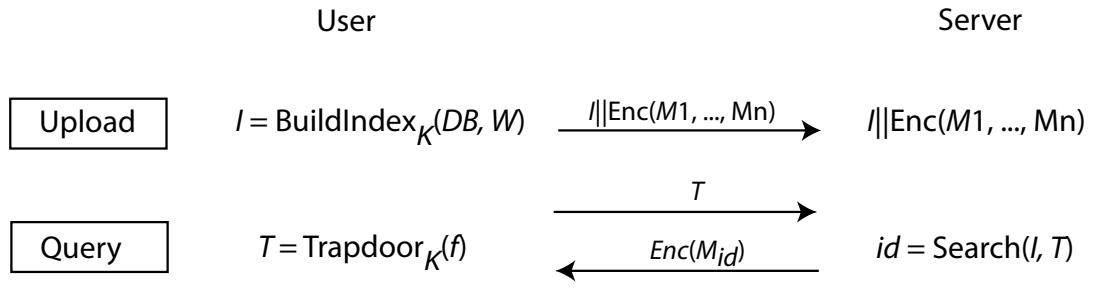


Figure 2.6: Upload and Query scheme in Searchable Encryption [22]

As the name suggests Searchable Symmetric Encryption is associated to symmetric key primitives, as it allows only the private key holder to upload encrypted data and to create trapdoors for queries. On the contrary, Public key encryption with keyword search, is associated to public key primitives. It enables all the users who have the public key to upload encrypted data but only allows the private key holder to create trapdoors for queries [56]. For these disputed reasons SSE is considered more efficient than PEKS [13].

In Figure 2.7 [46] we can see the system model of a Searchable Symmetric Encryption Scheme. The user encrypts a set of data and creates an encrypted index file witch contains a set of encrypted keywords, extracted from the data. The user outsources the index and the encrypted data to the server. During a search the user creates the encrypted query and sends the query to the server. The Cloud server takes this information and retrieves pointers to the document that contain the search keywords in the query and returns it to the client [16].

2.2.1.1 SSE Schemes

In this subsection we are going to present a number of SSE Schemes.

- **Song**
  This scheme was the first Searchable Symmetric Encryption scheme [50], and it was presented in the year 2000. It consists in encrypting the document's keyword and XORing the encryption with a HMAC of a random generated value. This scheme didn't contain an index, so for searching for a query keyword consisted on verifying the HMAC signatures, meaning that this method consists on having linear search complexity for the total number of the query's keywords [15].

- **Goh**
  Goh's implementation of a SSE scheme [19] consists of having an index for every document, as it was the first scheme containing one. This results on every

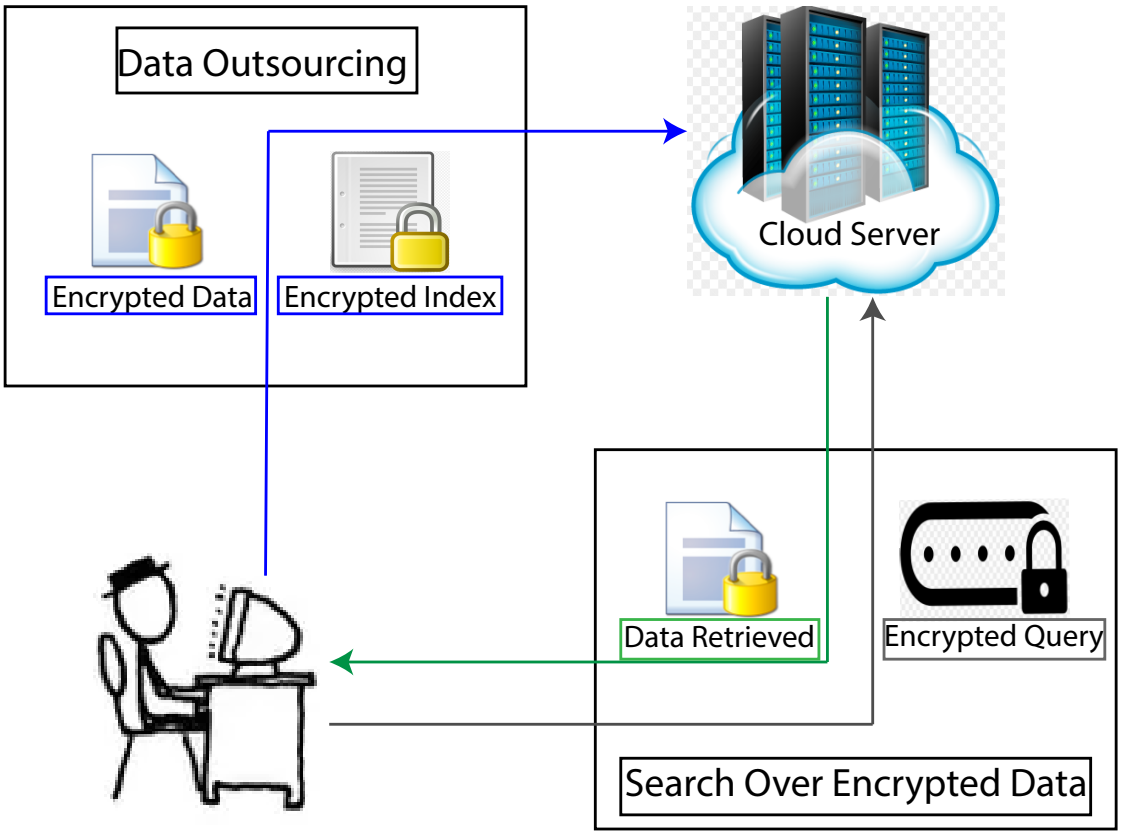


Figure 2.7: System Model of a SSE scheme [46]

time a query is made the server has to search every index for the keywords present in the query, and the amount of computation work that has to be done for a query is proportional to the number of documents in the collection.

- **Oblivious RAMs**
  With Oblivious RAMs, SSE can be fully achieved. Using this technique any type of search query can be done, including disjunctions and conjunctions of words, without showing its access patterns, that means the information is not leaked to the server. But there's a main disadvantage in this method, as it requires multiple rounds for write and read operations and there is a logarithmic cost in the average operation. O. Goldreich and R. Ostrovsky wrote on [20] and a two round solution to this problem, but causing an extreme large square root overhead in the process.

- **SSE-1**
  SSE-1 was presented in [13] and is called a non-adaptively secure construction. Like most SSE schemes all the documents are encrypted and an index I is built during the process of setup.

The index I consists of two different data structures, an array A that for all the distinct words in all the documents, contains a stored encryption of the documents, and a look-up table T, that for all the distinct words in all the documents, contains the information to locate and decrypt the appropriate elements inside data structure A. Despite the efficiency of this algorithm, SSE-1 is not considered secured to adaptive adversaries.

- **SSE-2**
  SSE-2, on the contrary of SSE-1, can achieve security against adaptive adversaries, although trading security for performance, as this scheme takes more space to perform a query and size in the server to store data. However, the computation costs for SSE-2 are the same as in SSE-1.

  SSE-2 is constructed similar to SSE-1, but instead a index I is constructed using the pair values with <Key, Value>, where key is a distinct word from the all the words in all the documents and the value is a pointer to the document where that word appears. Given a word W, that word must be concatenated with a integer J that represent how many times that word has appeared in the set of documents until that point, for example, if the word coin appears 3 times in the set of documents, we are going to have a set of pairs in the Index like <coin1, address1>, <coin2, address2> and <coin3, address3>.

  It's stated in [13] that this method of Indexing will allow the simulator to construct an index for the adversary that is indistinguishable from a real index, therefore, secure to adaptability from the adversary.

- **Cash**
  Cash's implementation of a SSE scheme [9] consists in having a Dynamic Searchable Encryption in Very-Large Databases, where the entries scale to millions of records. It is the fastest of presented schemes if there are a large amount of entries stored in the system. It subsists in associating with each record/keyword a pseudorandom label, and then for each pair storing the encrypted record identifier with that label in a non secure dictionary. The labels are then derived, so when the user makes a keyword query, the client computes some keyword specific short key that allows the server to search by computing the label, and then retrieve the identifiers from the dictionary, and also retrieves the encrypted matching record identifiers. A disadvantage of this method is that the size of the dictionary is compromised to the server, so it can know the number of records, keywords and pairs in the data.

The labels are derived so that the client, on the user inputting the keywords to a query, can compute a keyword-specific short key, allowing the server to search by first recomputing the labels, then retrieving the encrypted identifiers from the dictionary, and finally decrypting the matching encrypted record identifiers.

Table 2.1: Different SSE Schemes [46] [13]

Scheme	Time Complexity	Index Size	Server Storage	#Rounds
Song [50]	$O(n)$	N/A	$O(n)$	1
Goh [19]	$O(n)$	$O(n)$	$O(n)$	1
Oblivious RAMs [20]	$O(log^3(n))$	N/A	$O(n * log^2(n))$	$O(log(n))$
SSE-1 [13]	$O(r)$	$O(m+n)$	$O(n)$	1
SSE-2 [13]	$O(r)$	$O(m*n)$	$O(n)$	1
Cash [9]	$O(r/p)$	$O(n)$	$O(n)$	1

In **table 2.1** we show several computations costs of five different SSE schemes. The letter n denotes the size of the documents set, the letter r denotes the number of documents, the letter p denotes the number of processors and the letter m denotes the size of the keywords. To make the comparison easier and more fair we assume each document has the same size.

2.2.2 Range Queries

The result of a range query operation is all the documents between between a certain range in a database. Using an ordinary Searchable Encryption if we'd like all the results on the interval of [a,b], the user would have to make b - a + 1 operations, causing it to be a very inefficient and insecure method [26].

In [48] it's built a Multi-dimensional range query over encrypted data, where it was adapted a tree structure and developed a multi-dimensional range query system, which included working with an authority holding a master key that can issue searches to an authorized party, allowing it to decrypt data entries whose attributes fall within specific ranges, while still preserving privacy of other data entries.

In [6] it was presented a solution which involved bilinear maps defined over elliptic curves, permitting Conjunctive, Subset, and Range Queries on Encrypted Data, but still depending on a public key scheme technique.

In [26] a method was presented which consisted in utilizing only symmetric key encryption systems, with the search time depending with the number of documents

corresponding to the retrieved documents corresponding to the keywords, and not the entire database. This method involved doing the search query on a linked chain structure, and not in a tree structure. For every entry it is created an external link containing the following node. This way when the user creates a trapdoor for a query in the interval [a,b] it must contain the decryption key for the chain a and the decryption key for chain b By searching two linked chains and all documents related to the interval [a,b] are searched.

2.3 Password Databases

Password Databases are storage engines where the user's credential, including the username and its password are stored. Some examples of how the passwords are stored in modern days system include Plain Text Passwords, Basic Password Hashing, Slow Hashes, Encrypted Passwords and Encrypted Passwords with a Dash of Salt.

This section is divided in two subsections, the first one focuses on how passwords can be stored in a database, stating how each process works and its advantages and disadvantages. In the second subsection we focus on attacks made by an adversary on these password databases.

2.3.1 Password Database Storage in The Industry

Information in password databases should be considered more sensitive, because it contains all the information necessary for an adversary to enter with users stolen password credentials. Most people recycle passwords between websites, so the adversary can easily enter any website with the stolen password.

2.3.1.1 Plain Text Passwords

Plain Text Passwords is the most simple way to store a password and also the most unsafe. The password is stored as it is, and everyone taking a quick glance at the database can see it. This is a very bad idea for companies to store a password, as the password can be checked by an employee, social engineering methods, a backdoor created by a missing Operating System patch or a virus... Hackers can use a simple SQL Injection method to see everyone passwords. This is the equivalent of someone breaking into one's house and finding the key for the safe just on top of it.

2.3.1.2 Password Hashing

Password Hashing consists of generating a String of characters from a given password. The generated String is of a fixed length and based on the possible variations from the inputted password. The algorithms that use hashing are one way functions and are made in a manner that is impossible to obtain the original password. Some examples of algorithms that use Basic Password Hashing are:

- **MD5**
  Originally designed to be a hash function, MD5 has been proven to be a not so secure method of doing so after some vulnerabilities about it being exposed. It is nowadays more often used has a checksum to check data validity and integrity [55] .

- **SHA-3**
  SHA-3 refers to Secure Hash Algorithm 3 is the latest iteration of the former SHA-0, SHA-1 and SHA-2. It was published in just August of 2015 and is based on the KECCAK algorithm [14]. It consists of a set of hash functions and it can only be implemented in a 64-bit processor. It has been proven that it is more secure then its predecessor(SHA-2), but it takes as twice as much time.

In figure 2.8 we see an illustration of hashing a password. Once the password is hashed there's no way to get the password back.

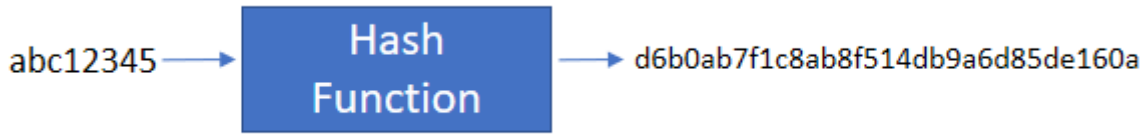


Figure 2.8: Hashing the Password 'abc12345'

2.3.1.3 Password Hashing with salt

Password Hashing with Salt refers to not just hashing the password, but hashing it with the combination of the password + salt. A salt is randomized hash, and it is appended to the password in the beginning of it or the end [14]. This way lookup table and rainbow table attacks will not work, as those tables are only successful if the hashed passwords are the same. We can see an example of password hashing with salt below:

MD5Hash('abc123') = a90acb735b18a3314f2db44104bea194
MD5Hash('abc123' + '9oGQSS1cTd') = 2e8cfc279e0f90e392d0068af2a01f6f
MD5Hash('abc123' + 'K89xywW5PE') = 3a6f3a484e0d39f09760ef03cd70bf2d

The salt used shouldn't be the same for each hash, as the adversaries can still run a dictionary attack to discover the sites 'secret' salt. Instead a salt should be generated every time a new password is inserted into storage. Also, the salt shouldn't be short in terms of length, as it can be easily discovered by adversaries using a lookup table.

- **Bcrypt**
  Bcrypt is a password hashing scheme [35], based on the Blowfish block cipher, that makes use of the salt, making it very useful against rainbow table attacks. Bcrypt takes as input the cost, the salt and the password to be hashed. The input cost determines how expensive this function is going to be, as this is an adaptive function, the number of iterations in bcrypt are directly related to the value of the input cost, making the algorithm slower for every new iteration, although, theoretically meaning the final hash will become more secure. The more processing power this algorithm takes to hash, the more secure it will be. This algorithm is considered future-proof, because as the processors become more powerful and cheap over time, brute force attacks will become easier, but so will the bcrypt become more secure as its security depends on the processing power involved in the process of hashing. For every iteration of this algorithm the generated key gets expanded, with the use of the original password and the original salt, making it more difficult to crack, like this. The returned result of this algorithm is a bcrypt hash [35].

2.3.1.4 Encrypted Passwords

Encryption of passwords is the process of transforming the password to a series of characters, that are unreadable to the human eye. As already stated in this work, symmetric encryption allows a party with access to the key to decrypt the encrypted to obtain the original password. Some examples of some algorithms that use Basic Password Encryption are:

- **RSA**
  RSA refers to Rivest–Shamir–Adleman and is on the first forms of public-key encryption and the standard for transmitting information over the web. This method is relatively slow, so it just usually passes the public encryption key around, so the encryption/decryption operations can be executed faster at a

much higher speed[45]. As there is no random factor in the process of the RSA algorithm, it is fully deterministic, meaning if an adversary knows the linear relation from the plaintext to the cypertext, can easily crack it [11].

- **AES**
  AES refers to Advanced Encryption Standard and is a symmetric key algorithm used the by USA government to cypher encrypt any sensible data and information. This method has been proven over the time to not be so secure, as various methods have been proofed to break the AES algorithm, more notably, brute force methods [4]. The AES algorithm is considered of very high speed as it doesn't use a lot of computer resources as RAM and CPU, as it only requires 16 cycles per byte of encrypted data [7].

- **PBKDF2**
  PBKDF2 refers to Password-Based Key Derivation Function 2 and applies a pseudo-random function to derive keys. This derived key has no limits in terms of length, although it can be limited to the size of the structure of the pseudo-random function. It is remarkable how hard it is to brute force this algorithm, due to its huge number of layers of encryption (minimum recommended is 4096). The PBKDF2 method takes as input the password, a salt, the number of layers and the desired length of th e resulting key, it outputs the ciphered text. Although it is considered very hard to brute force, it is mentioned that this method is very slow [39].

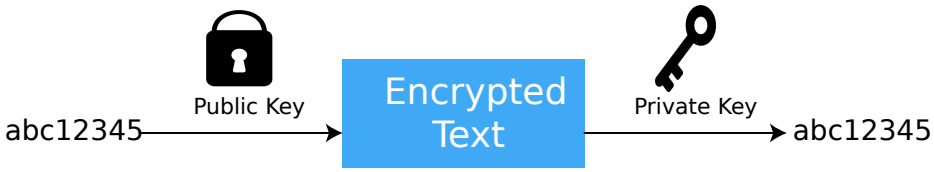


Figure 2.9: Encryption and Decryption of the Password 'abc12345'

In figure 2.9 we see an illustration of encrypting a password. A public key is requires to encrypt the text and the private key is necessary to decrypt it.

2.3.1.5 Encryption vs Hashing

Encryption and Hashing are the main ways of storing passwords in a database, and both operate on the original plaintext password. The main difference between them is that in hashing once you hash a password there's no way to get back to the original plaintext, on the opposite, encrypting a password permits getting the

original password as long as the private key is given. It's disputed that the best way to store a password is to hash it, because, supposedly, the hashed password can't be hashed back, although this can be avoided by adversaries with a dictionary or a rainbow attack. On the other hand if the adversary gains access to the private key, he can decrypt the ciphertext and gain access to all the plaintext passwords.

2.3.2 Attacks on Password Databases

According to a recent study made in the research work [38] this is how the attacks on main websites were made:

1. In the largest percentage of cases (35.3%), the entity that was victim of the security breach chose not to disclose the attack mechanism.
2. **SQL injection** — accounts for 29.4% of the attacks, the most prevalent mechanism for disclosed attacks.
3. **Account hijacking** — (14.7%), wherein access was obtained via stolen account credentials, is the second most prevalent attack methodology.
4. **Spear-phishing and 3rd party software flaws** — were tied at 5.9% each

There's close to nothing we can do about Account Hijacking and Spear-Pishing, the only way to efficiently solve this problem is to force a two-way step login verification. In this situation the users logins using the knowledge factor, something the user knows like his password, and a possession factor, something only the user has, like his mobile phone, per example, if the user wishes to login in a website he has to insert his password, and then use some form of interaction with his phone, like inputting a generated code.
In the case of SQL Injection, or other methods that an adversary takes a snapshot of the database storage, the data comes in plaintext, hashed or encrypted. If the data is in plaintext, the adversary can already see the data he wishes and sensitive information is already considered compromised. If it is the hashed, the adversary has to hash it back to get the original text and if it is encrypted only with the key can he decypher the ciphertext back to its original plaintext.
In this section we are going to present the most common and efficient methods of attacking private information, used by an adversary.

2.3.2.1 Brute force Attack

A brute force attack, also referred as Exhaustive Key Search, consists of an adversary trying to find a password or pass-phrase by trying every single possible combination

of characters available. This method may be efficient against short passwords, but not long passwords, as the possible password combination will be much larger [28]. As we can see in **Table 2.2** there are 97 possible combinations (26 + 26 + 10 + 35) for a digit to make up a password character.
A 3 digit password would take at worst 97^3 (912 673) attempts for an adversary to guess the password by using brute force, with the current CPU and GPU available in the market it is cheap and easy to brute force a password with only 3 digits. Currently, the standard minimum size for a password is 8 digits and that would take 97^8 (7 837 433 594 376 961) attempts at worst, it is not a viable option for an adversary to brute force its way with so many possible combinations.

Table 2.2: Possible digits that make up a password character.

Type	Digits	Number
Lowercase letters	*'a' to 'z'*	26
Uppercase letters	*'A' to 'Z'*	26
Numbers	*0-9*	10
Special Characters	*"^'!'#$%&'()*+,-./:;<=>?@_ ´'{|}«»~*	35
Total		97

2.3.2.2 Dictionary attack

A Dictionary Attack is very similar to a Brute Force Attack, but instead of trying to force its way with every possible combination, it uses a technique, that only tries words contained in a dictionary. The dictionary is created by an adversary, and it contains all the most likely words to be contained inside a password, therefore reducing the number of attempts on cracking a password. In every attempt, if a word from the dictionary fail, the algorithm tries another word. The appending of a salt in the hash of passwords, or even on the plaintext ones, can totally render the use of a Dictionary attack useless [10], as the salt appends a random string of characters to the passwords that most likely won't exist in the dictionary.

The dictionary can contain real plaintext words or hashed text from the real plaintext words. More advanced dictionary attacks can also attempt to use common special characters, numbers, words and a mixing combination of all of them.

The adversary can use the most relevant keywords available on the user's profile on an attempt to fill the dictionary in a more efficient way. The user can also avoid an dictionary attack by misspelling words on his password, to words that are less probable to don't exist in the dictionary created by the adversary.

2.3.2.3 Birthday Attack

The Birthday Attack uses the same approach as the Birthday Problem from Probabilistic Theory. The Birthday Problem consists on having a population of n random people, and considering some pair of two of those n people will share the same birthday [58].

Using the same logic an adversary, that has gained information about one password, can try to guess that there are at least two passwords equal to one another in the database storage, given that are enough stored passwords.

2.3.2.4 Rainbow table

Considered to be the most efficient way of cracking a password, a Rainbow Table consists on an adversary building a pre-computed table with the same hash algorithm of the hashed password its trying to crack, and a reduction algorithm that's used to reverse a password hash to a plaintext. The chains which make up rainbow tables are of a hash and reduction functions starting at a certain plaintext, and ending at a certain hash. The Rainbow table algorithm starts with a hash and works like this [23]:

1. If the Hash you have is the same as one in the list of final Hashes, go to step 5.
2. Insert the Hash in the list of final Hashes
3. Reduce the hash to another plaintext
4. Hash the reduced plaintext
5. If the hash obtained is the same as the original hash, the intended plaintext is discovered. If not, choose a random plaintext, hash it and go to step 1

This method also takes a long time to run from scratch, but with the difference that the adversary doesn't have to start from zero every time he wants to crack a hashed password, and it can index and search the previously completed hash results [54].

A Rainbow Table does a space-memory trade-off, which means, it trades the long computing time to discover a password, for memory, meaning the longer the method runs, the more disk space will occupate in the hard drive or RAM. A Rainbow Table size will vary, from 52 GB up to 864 GB,[32] although on the long run it will probably take much less time to be successfully for the adversary, than a Dictionary Attack or a Brute Force Attack [27].

2.3.3 Mitigating Attacks on Password Databases

2.3.3.1 Password Reuse

In [25] it's stated that most password theft happens because of the Domino Affect on Password Reuse, meaning that the users use the same 4 or 5 password for every website. The problem with this is that if an adversary gains access to a password database of just one website, that could lead to failure on other systems.

One way to prevent attacks is when the system detects an adversary repeatedly attempting to use incorrect password to access one's account, the system would shut down in defence.

The paper also mentions public-key encryption (PKE), Public-key infrastructure (PKI) and Biometrics to serve as the authentication process, instead of the usual user-password combination.

Biometrics involve some form of data obtained from ther user's body such as, the iris of his eyes, the fingertip, the whole face, or some patterns in the user's voice. While convenient, unlike a password, once compromised the biometric authentication can't be changed.

2.3.3.2 Password Managers

Password Managers helps users manage multiples password accounts by turning a single memorized password into a different password for every of is accounts. Typical password managers are integrated to web browsers. When the password managers detects that the user has return to a site for which he has a stored password, it fills the login form with the appropriate username and password. Most password managers do a trade off in their design, trading the process of logging in more conveniently, reducing the user's memory burden for every website, but introduces an external dependency [57].

Some usability goals should be guaranteed in password managers like, let the user only have to remember one password, allow the user to change said password and security goals like, the use of an unique password for each site, resist offline attacks and automatically detect pishing sites. This should relief the user of having to create and to memorize a new password for every website that the user is signed up to.

Although the advancements on password managers over the years some studies made recently like in [31] showed some critical vulnerabilities, and even stating how the attackers could steal arbitrary credentials of the user from the most popular password managers around the world.

2.3.3.3 Reverse Turing Tests

Reverse Turing Tests, also referred as just RTT, and better known by some people for the name CAPTCHAs, are a series of tests that distinguish between a computer program and a human. These RTT should be easy for the real user to solve, hard for automated programs to resolve and have a very low probability of guessing the answers correctly [43]. This is useful because it can limit the number of automated attacks on a database password, so a real human has to pass the test every time he wants to try a user password combination. Some examples of RTT could be asking the user to input the words contained in a picture, a math operation, picking some images, etc...

In figure 2.10 we can see two examples of RTT. In the first one the user is required to select all the images containing a bus, and in the second example the user is required to type the words in that picture.

Figure 2.10: Example 1 [1] and Example 2 [2] of Reverse Turing Tests

2.3.4 Scientific Solutions to Secure Password Databases

2.3.4.1 PolyPasswordHasher

PolyPasswordHasher is a software password storage mechanism deployed on the the server side. The technique used prevents the adversaries to crack an individual password, because PolyPasswordHasher uses cryptographic hashing and threshold cryptography to combine password hash data with shares so that users unknowingly protect each other's password data [8]. With the amount of increased work on the server by a magnitude order, the adversary needs more time to crack a password.

Basically this scheme additionally protects hash password with salt with an additional secret shared used with the Shamir's Secret Sharing method [47]. The secret shares are stored in memory and are used to verify the hashed passwords, although it's guaranteed that an adversary can only read what's on is persisted

on disk, including the password database, but can't read from the memory. A disadvantage of this, is that if an adversary gains a access to a single space of memory it can steal the secret share.

2.3.4.2 ErsatzPasswords

In [3] a scheme is presented similar to traditional password database schemes, when an adversary tries to access the database, the only passwords that he will get, will be fake passwords, regarded in this paper as ErsatzPasswords. Then, once the adversary tries to use these fake passwords to authenticate to a user's account, the system detects it and raises an alarm. The fake passwords should maintain the same format and appearance of typical passwords to succeed in the process of deception on the adversary. The authors of the paper claim it is impossible for an attacker to recover user passwords from the hashed format, without physical access to the systems database where the passwords are stored.

2.3.4.3 SafeKeeper

In [29] a new scheme called SafeKeeper, is presented thats ensure secrecy of passwords in web authentication systems. This scheme assures that the users credential are secure ever since the user inputs his credentials on to the web browser. SafeKeeper ensures this by requiring the installation of a addon on to the users browser and makes the communication to the server (where the password database is) in secure channels. Inside the server it computes a CMAC (cipher-based message authentication code) on the password before they are stored in the database. It uses Intel SGX, generating and storing the cryptographic key inside an Enclave created by SGX, and also makes the cryptographic operations inside the enclave. Inside the Enclave some other properties are stored, like how many attempts has that particular user used, before successfully entering the right credentials.

2.4 Summary of the Related Work

As seen in the previous sections of this chapter, the outsourcing of data to an untrusted hardware server poses great danger to the user because of the unknown parties responsible for maintaining them. Methods like Searchable Symmetric Encryption have proven to be an efficient way to make searches over encrypted data showing potential for securing password databases, but despite this operations can still leak patterns that adversaries can exploit. Oblivious RAMs avoids this problem

with a dynamic solution, frequently switching the position of indexes, and although it is considered a secure method it comes with the great cost in efficiency.

Recent technologies, like Intel SGX, allow creating trusted execution environments where computations can be performed in isolation from other (possibly priviliged) processes, although there is little work done in password database management on this technology.

CHAPTER 3

Designed Solutions

In this chapter we present the details of our new solution for password database management. We start by presenting our system and adversary models. Then we present two schemes: one based solely on cryptographic techniques and one that additionally uses trusted hardware.

3.1 Architecture

Traditional password database systems work with a key-value approach. When the user provides the login information, the system searches for the username and checks if the password (or hash of the password) matches. In this approach the login information can be stored in a variety of data structures, like SQL-Table or an Hash-Map.

3.1.1 System Model

In this work we propose a new model for secure password database management, based on cryptographic primitives. In our model a simple API is provided for password management. This API can be applied in a non-trustable cloud infrastructure, without interfering with the security premises of our scheme. The proposed system model considers several writers and readers, referred as **Users**, which uses the password database stored in the **Cloud Server**, through a **Web Application**, assuming that all the data is sent with a secure TLS connection.

In Figure 3.1 we show a high level view of the proposed system. The user sends his credentials through a WEB Application, which id redirected to the database

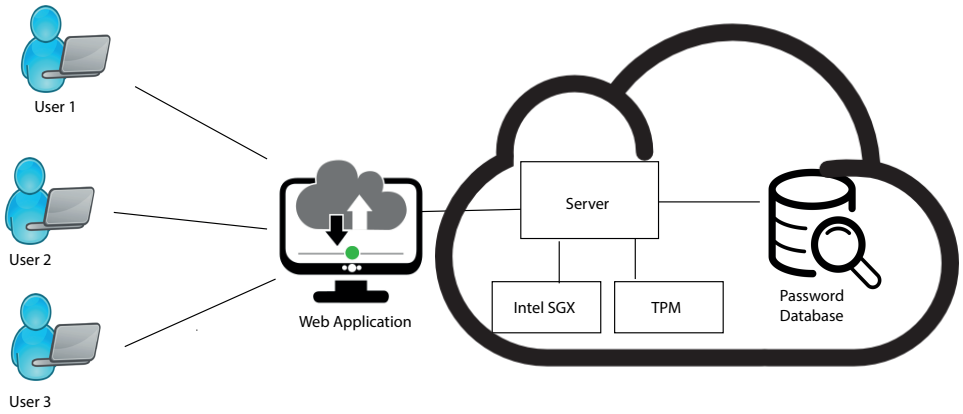


Figure 3.1: System Model

server. The server saves the database of passwords in persistent storage, and can optionally use trustable hardware modules(i.e. TPM and Intel SGX) to perform cryptographic operations in isolation and securely store the respective cryptographic keys.

To interact with our system, we resort to the previously mentioned API. The API is composed of database operations as shown below:

- Register (String username, String password)
- Login (String username, String password)
- ChangePassword (String username, String oldPassword, String newPassword)
- ChangeUsername (String oldUsername, String password, String newUsername)
- DeleteUser (String username, String password)

The server sends a boolean answer to the user indicating whether or not the authentication process was successful. The WEB Application can use this variable to transform it into an access token so the users can authenticate more easily.

The main objective of this model is to secure password databases without sacrificing performance, on the most common operations of password database systems. In Figure 3.2 it is demonstrated a more detailed vision of what happens in our system when the user makes a login operation and the interaction that is made with the password database, represented as an encrypted data structure.

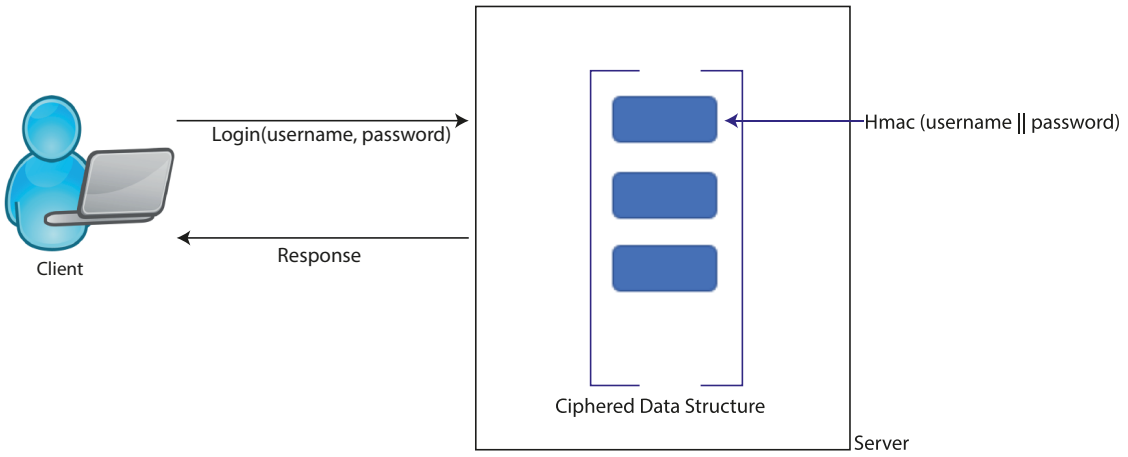


Figure 3.2: Top level view of the execution of a Login operation and interaction with a password database

3.1.2 Adversary Model

In this thesis we aim to protect the user's passwords from passive attacks. The main adversary we consider, is a *snapshot* adversary, that obtains a complete copy of the password database, even if he only has a very small time frame of access to the database. This adversary usually corresponds to the Internet hacker, trying to find vulnerabilities in the cloud infrastructure. We also consider as an adversary a cloud administrator that is trustable, but also curious, i.e. that operates in the cloud infra-structure and its servers. The basic principle of our scheme is to ensure that even when adversaries gain access to protected passwords, they can not attack (e.g. brute force, dictionary attack, rainbow table) and decipher those passwords, without the corresponding cryptographic key.

A problem with the commonly used hashing password schemes (like B-Crypt, SHA-3, etc..), is that when there is a database password leak, a big list of usernames and their passwords are published online and most of the times the username of the user is their e-mail. An adversary can decide to target a password hash of a known username, and it can know the username of the target just by knowing the user's email. After the adversary has gained access to the targeted users password hash, it is just a matter of time before the hash of the password can be reverted to the original text. Nowadays it can take months to get the original password from its hashed value, but with the rising increase of processor performance it is also a matter of time for more powerful and cost-efficiency processors to be available to the public market. Once the adversary gains access to the targeted users original decrypted password text, and the adversary usually has access to all the online information

of where that user is registered, because most people use the same password for all their accounts [52].

3.2 Design of SSPM

In this section we explain the design of SSPM (Simple Secure Password Management).

SSPM receives the username and the password of the user (i.e. through secure TLS channels). The system takes the result of the concatenation of the username and its password, and hashes with a secret key using a HMAC-SHA1 algorithm, then storing the result of it in an Hash-Set. With this method it is very difficult to target a certain user, even if the adversary has made a snapshot copy of the password database, it is almost impossible to guess what hash corresponds to each user. Additionally, the HMAC function will act as an encryption operation, being more secure than a hash + salt scheme, which can be attacked through advanced rainbow table techniques. In Figure 3.2, we show a top level view of how SSPM stores the users credentials in n hash set.

In Figure 3.3 we demonstrate how the values are stored in a plaintext form and how it is stored in our system. If an adversary has a snapshot copy of the database it can easily try to gain knowledge about a known users password by searching its user name in the data structure and attacking the password. In our system the adversary has no idea where to attack the targeted user, because just by having the database, the adversary has no idea where the username information is stored.

Problems with this approach

A problem with this approach, is that two different users can have the same concatenation of their username and password combination. The probability of that happening is very small, but it can happen. For example, if user 1 has the username 'Michael' and the password '123456', and user 2 has the username 'Michael1' and the password '23456', the resulting concatenation of the two combinations would be 'Michael123456', and the result of the HMAC-SHA1 algorithm would be the same, even if the usernames are unique.

To solve this problem we have come with the solution of inputing a fixed salt value to work as a separator for the username and the password. For example, if we have the salt '$2a$10$' the result of the concatenation of user 1 would be 'Michael$2a$10$123456' and user 2 would be 'Michael1$2a$10$23456', therefore resulting in a different output by the encryption HMAC-sha1 algorithm.

Algorithms 1 and 2 represent in detail the most simple SSPM operations. In the

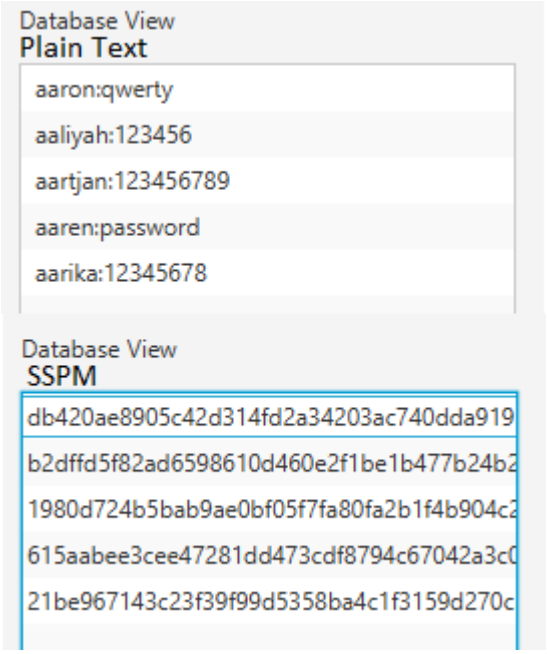


Figure 3.3: Small sample size of the stored login information in the new system

algorithms, the index *Users* works like a encrypted data structure. The deterministic encryption algorithm that we use is HMAC-SHA1 (because we don't need to decipher the values). To guarantee that this index never possesses the two equal entries, we define that each entry saves the result of the concatenation of the users username and respective password (with a random set of characters in the middle). Using this method, even if two different users have the same password, their entries will be distinct, because their username must be unique.

Additionally, to guarantee that there are no users with the same username, we use another ciphered data structure that only saves the usernames registered in the system (referred as Usernames in the algorithms 1 and 2). It is possible to simplify these data structures in just one, in which the key of the data structure would be the username and the value would be the password, although the proposed separation of the data structures, allows to make login operations faster (sacrificing some performance in operations like Register).

There is always the possibility of the adversary studying the access pattern of a users login. A hash set works like a hash table (or a hash map), when inserting is concerned. A hash function is used to compute an index into an array of buckets or slots, and this hash function decides this index by taking the characters of the given string (already hashed) and its size, thereby, the index of the users data in the hash set stays the same. An adversary can take advantage of this by studying the access patterns of a user.

To avoid what was above stated, in our system we periodically change the HMAC-SHA1 cryptographic key that we make use of. We do this by saving the users username and password in an has map, and encrypt it using AES with a key with the length of 256 bits. Then, we decrypt the AES hash map and we generate a new HMAC-SHA1 key. After that we make a new hash for each user using HMAC-SHA1 with the new key, we create a new hash set containing the new users concatenation hash of its username and password, and we replace this hash-set with the old one. By doing this, we guarantee that an adversary can't explore the access patterns of a users activity so frequently.

A problem with this approach, is that it leaves the data exposed temporarily when the data in the hash map encrypted in AES is decrypted, leaving the users credentials in plaintext inside the main memory for several moments. For this reason we leave this approach as a setting for the administrator of the password database to choose if he so desires. This was the reason we thought of working with Intel SGXs Enclave, and we explain in the next subsection 3.3.2 how we solved this issue.

Here, we detail the operations of our API:

- Login (String username, String password) Every time SSPM receives an operation, it also receives two or three String arguments, depending on what type of operation it was requested. For the login operation we receive two arguments, the username and its password. We gather the received username and password from the client, apply a concatenation that consists of the username + a fixed salt + password and apply the HMAC-SHA1 function with a designated key. Then the scheme verifies if the resulting hash is contained inside the Users hash set. The scheme then sends the boolean result of this verification.

- Register(String username, String password) The scheme first hashes the username and checks if it exists in the Usernames hash set. We can't have two users with the same username, so if the hashed username already exists in this hash set the register operation is canceled. The scheme also applies the HMAC-SHA1 function to the result of the concatenation between the usernames, the fixed salt and the password. Then we insert the resulting hash to the Users hash set, and send a confirmation that the register operation was successful.

- ChangePassword (String username, String oldPassword, String newPassword) The scheme hashes the concatenation of the username and the oldPassword. If it exists in the Users hashset, it is removed. Then it is applied the concatenation

Algorithm 1 SSPM v1

1: $Users$: HashSet with all the hashes of the concatenations of the usernames and their passwords
2: $Usernames$:HashSet with all the hashes of just the usernames
3: sep : Random String to seperate the username of their password, during the concatenation
4: key : Cryptographic key to be used during the HMAC-SHA1 function call
5: λ : Security parameter
6: **procedure** SSPMv1 (λ)
7: $Usernames \leftarrow \{\}$
8: $Users \leftarrow \{\}$
9: $sep \leftarrow randomString()$
10: $key \leftarrow generateKey(\lambda)$
11: **procedure** Login (username, password)
12: Client:
13: $Response \leftarrow sspm.Login(Username, Password)$
14: **if** $Response == true$ **then return** $createToken()$
15: **else return** $null$
16: Server:
17: $hashedUser \leftarrow applyHMAC(key, username + sep + password)$
18: **if** $hashedUser$ in $Users$ **then return** $true$
19: **else return** $false$
20: **procedure** Register (username, password)
21: Client:
22: **return** $sspm.Register(Username, Password)$
23: Server:
24: $hashedUsername \leftarrow applyHMAC(key, username)$
25: **if** $hashedUsername$ in $Usernames$ **then**
26: **return** $false$
27: **else**
28: $Usernames \leftarrow Users \cup hashedUsername$
29: $hashedUser \leftarrow applyHMAC(key, username + sep + password)$
30: $Users \leftarrow Users \cup hashedUser$
31: **return** $true$
32: **procedure** ChangePassword (username, oldPassword, NewPassword)
33: Client:
34: **return** $sspm.ChangePassword(username, oldPassword, NewPassword)$
35: Server:
36: $hashedUser \leftarrow applyHMAC(key, username + sep + oldPassword)$
37: **if** $hashedUser$ not in $Users$ **then**
38: **return** $false$
39: **else**
40: $Users \leftarrow Users \setminus hashedUser$
41: $newHashedUser \leftarrow applyHMAC(key, username + sep + newPassword)$
42: $Users \leftarrow Users \cup newHashedUser$
43: **return** $true$

Algorithm 2 SSPM v1

1: **procedure** ChangeUsername (OldUsername, password, NewUsername)
2: Client:
3: **return** $sspm.ChangeUsername(OldUsername, password, NewUsername)$
4: Server:
5: $hashedUser \leftarrow applyHMAC(key, OldUsername + sep + password)$
6: **if** $hashedUser$ not in $Users$ **then**
7: **return** $false$
8: **else**
9: $hashedUserName \leftarrow applyHMAC(key, OldUsername)$
10: $Usernames \leftarrow Usernames \setminus hashedUserName$
11: $Users \leftarrow Users \setminus hashedUser$
12: $newHashedUserName \leftarrow applyHMAC(key, NewUsername)$
13: $newHashedUser \leftarrow applyHMAC(key, NewUsername + sep + password)$
14: $Usernames \leftarrow Usernames \cup newHashedUserName$
15: $Users \leftarrow Users \cup newHashedUser$
16: **return** $true$
17: **procedure** deleteUser (username, password)
18: Client:
19: **return** $sspm.deleteUser(username, password)$
20: Server:
21: $hashedUser \leftarrow applyHMAC(key, username + sep + password)$
22: **if** $hashedUser$ not in $Users$ **then**
23: **return** $false$
24: **else**
25: $Users \leftarrow Users \setminus hashedUser$
26: $hashedUsername \leftarrow applyHMAC(key, username)$
27: $Users \leftarrow Users \setminus hashedUser$
28: $Usernames \leftarrow Usernames \setminus hashedUsername$
29: **return** $true$

of the username and the oldPassword and inserts the resulting hash in the Users list.

- ChangeUsername (String oldUsername, String password, String newUsername) The scheme first hashes the resulting concatenation of the oldUsername with the password. It checks if the resulting hash exists in the Users hash set, and deletes it if so. It changes the old username with the new username in the Usernames hash set. Then it inserts the hash of the result of the concatenation of the newUsername with the password and inserts it into the Users hash set.

- DeleteUser (String username, String password) Same as the Login and Register operations, the scheme first gathers the received username and password from

the client, applies a concatenation to them and then checks if its result exists in the Users hash set, so it can be deleted. The username from the Usernames hash set is also deleted.

3.2.1 Scheme Versions

In the work done for the scheme presented in this thesis we present several versions of it. Each new version iteration of the scheme guarantees more security properties sacrificing performance. Below is a summarize of what each version does and what it differs from one another:

- **SSPM V0**

  It is the most basic implementation of the new scheme, while we consider it more secure than ordinary schemes, it has some compromises, like different Users can have the same username. As long as they do not have the same password, confidentiality is guaranteed. Data is stored in one data structure that only remains in memory. Users can have repeated usernames. This version is theoretically the fastest. All data is stored in memory and persistence in disk is non existing. It uses HMAC-SHA1 to encrypt the data, using the same fixed key that is randomly generated when the process of the program is initialized. As the data is only stored in memory, persistence of the data is non-existing.

- **SSPM V1**

  Same as V0, but with the difference that also has an additional data structure in memory, that stores all usernames using a simple hash function, just to make sure that two different users don't have the same username. A simple verification is made when a user is trying to register in the system, just to make sure that username is not already in use. Delete operations and change username operations have to also include the changes in this new data structure. Persistence does not exist here also.

  Several alternative versions of this scheme, also included in the work done, consist of using different hashing encryption algorithms. Including **HMAC-sha256**, **HMAC-md5**, **sha-224** and **sha-3**. The last of the two don't require a generated key.

  Please note, that despite the fact that solely using md5 is not considered a secure method of hashing anymore (due to easily proven collisions, being easily generated nowadays), HMAC-md5 is still considered a secure method due to its generated random key.

- **SSPM V2**

  In addition to the features of V1, it introduces two new data structures. In one of them, all the data is stored (using key-> value, for usernames and passwords) in an encrypted way using AES with a 256 bit key. The other data structure, called *log*, that records all operations so that periodically the ciphered database using AES can be updated. Also, these functionalities are added:

  - Every X seconds (this can be changed as a parameter) the *log* is read, so the ciphered database of values can be updated. This requires that the database is deciphered so it can be updated, leaving it temporarily exposed, vulnerable to adversary attacks.
  - Every Y seconds, a new key is randomly generated to be worked with the HMAC-sha1 algorithm. All the hashes of the concatenation of the users and passwords are again generated using the new key with the HMAC-sha1 algorithm. This step uses the ciphered database to obtain the original values of the users usernames and passwords, having to also decipher the database, leaving the users data temporarily exposed.

  This version also adds persistence. All the data is stored in disk periodically, and when the process starts it checks if there is any saved data in disk, so the process can read it.

- **SSPM V3**

  Same as V2, except every Y seconds, when the data structure (where the encryption of the concatenation of the usernames and password are stored) is updated, it randomly selects a different scheme to encrypt the data, using the different versions of the V1 scheme that include the schemes **HMAC-sha256**, **HMAC-md5**, **sha-224** and **sha-3**. The last of the two do not require a generated key.

- **SSPM V4**

  This version is similar to V3, but with the difference that all data is periodically stored in a Redis Database. Persistence is guaranteed in the process.

  There is also an alternative version to V4, that stores all data using Redis instead of storing the data normally in memory. Although there is Persistence in this method, Performance is affectedly compromised by this.

- **SSPM with different settings** By the end of our work, we have made a SSPM version that includes all the features described in the different version above and we leave this features as settings that can be left to the administrator to select which one he desires to work with, when he wants to start the system. The desired settings can be defined when initializing the scheme, by sending boolean values as parameters that represent the settings. These settings are:

  - **usernames** - Don't allow replicate usernames.
  - **redis** - Use a Redis database to achive persistency (a Redis service is required for this).
  - **new key** - Generate new cryptographic keys periodically.
  - **tpm key** - Use TPM to store the secret cryptographic key to the HMAC and AES functions (described in detail in Section 3.3.1).
  - **tpm op** - Use TPM for all the hashing operations (may be very slow).

3.2.1.1 Security Analysis of SSPM

Here we make a security analysis of the SSPM scheme without the incorporation of Trusted Hardware Platforms (that are explained in the next sections).

SSPM keeps two data structures in memory. One, that we refer as Users is a Hash-Set. If an adversary gains access to this data structure it can see all the results of the HMAC-SHA1 function, of the concatenation of the username, the password and a fixed salt. However, even tough HMAC-SHA1 is a deterministic function, all these entries will be different, as long as there are no repetitions in the usernames and passwords. Furthermore, all entries will be of equal length (20 bytes for HMAC-SHA1), independently of the underlying passwords. If the adversary also gains access to the HMAC-SHA1 key, it can try to find a collision for the password, so we assume the data in the Users Hash-Set is as secure as a SHA1 algorithm. Even then, as all information is inside a Hash-Set, the adversary can not know where a specific users entry is. However, the adversary can study the access patterns of a specific user, revealing where the entry of that user is.

The other data structure that we keep saved is where the HMAC-SHA1 hash of the usernames of the users is kept, that we refer as Usernames. This data structure is only used during the register and delete operations, to make sure that there are not users with the same username. If the adversary gains access to this data structure and is able to create collisions, it can only have access to the hash of the usernames of the users and not the hash of their passwords.

We keep the cryptographic key for the HMAC-SHA1 in memory. As the data inside the Users data structure is hashed using an one-way function, the adversary can potentially revert back the original plain text of users credentials if it gains access to the secret key.

In later versions of SSPM we use another data structure to prevent the adversary to study the access patterns of an user. This data structure consists of a Hash-Map, in which the key is the username and the value is the password, and it is encrypted in AES (Advanced Encryption Scheme) with a key of 256 bit length. This data structure is referred as DBPW. Periodically, SSPM decrypts DBPW, generates a new cryptographic key, and uses the entries in DBPW to create a new Users data structures. As the key used in the HMAC-SHA1 is different, the hashed entries inside the Users will have different addresses in memory, and the study of the access patterns of the Adversary will become difficult. There is a major flaw in this approach, as if the adversary gains access of DBPW and the AES key, it can easily decrypt and have the plaintexts of the users. For these reasons, we keep this solution as an optional setting. This flaw is what inspired us to study and make use of Trusted Hardware on SSPM.

3.3 Design with Trusted Hardware

In this Section we are going to explain how we used Trusted Hardware for the design of our thesis. First, we talk about TPM (Trusted Platform Modules) and secondly. we are going to explain how we incorporated Intel SGX (Intel's Software Guard Extensions) in our scheme. Lastly, we are going to make a security analysis of SSPM with SGX.

3.3.1 SSPM TPM

This version works with the use of TPM (Trusted Platform Module). TPM is a cryptographic chip installed in the motherboard of the computer. Most modern computers have an available socket for easy installation of a TPM chip, although this version first detects if the hosting computer has an existing TPM module installed in the motherboard, and if not it launches a simulator.

This version is divided into two optional implementations. In one of them all cryptographic operations are executed inside the TPM, and SSPM connects to the TPM every time it has to make any kind of cryptographic function. To mention that all keys are kept inside the TPM. In the another implementation of this version, when the HMAC-SHA1 key is created, half the key is kept in memory (and stored

in disk) and the other half is generated and kept inside the TPM. Every time that SSPM has to make a cryptographic operation it makes a request to the TPM for its half of the key. The second implementation takes less resources and is faster than the first implementation.

3.3.2 SSPM SGX

In this version we use the technology created by Intel, named SGX (Software Guard Extensions) in a new version of SSPM that we call SSPM SGX. We can see a visual interface of SSPM SGX in Figure 3.4, and we also made a sequence diagram for a better understanding of how SGX was used in our scheme, available in Figure 3.5, that displays our scheme SSPM using SGX.

For the SGX implementation of our scheme we do organize the data structures a little differently than the other versions. The only data structure that is in unprotected memory is the Users hash set. All hashing and encryption functions are made inside the Enclave, which is created as we are starting up the process. In this case the HMAC function key is kept inside the Enclave and never leaves it. Also a log is kept inside the Enclave, containing the latest modifications to the password database, so we can periodically update the encrypted map data structure that we named DBPW. DBPW is encrypted using the CPU unique key, and can only be decrypted using the same CPU. All calls to SSPM SGX pass through the Enclave.

We work with 2 active threads in this version:

- One of the threads is always listening for operation calls by an outside asset, like a client. Once it receives an operation request and the respective string arguments, it sends the arguments to the Enclave where it applies the HMAC function with the key that is stored and saves the changes to a log. After that the Enclave returns the string result of the HMAC function to memory and in memory the User hash set is searched over. After all this has happened the thread returns a boolean answer.

- The second thread enters an infinite loop and waits X seconds before getting DBPW from disk and sending it to the Enclave. Inside the Enclave, DBPW is decrypted, using the processors unique key, and changes that were inserted in the log are applied to DBPW. A new Users hash set is also created in this step. Both Users and DBPW (after encryption) data structures are sent to memory after this. The encrypted DBPW is stored in the disk and the new Users hash set replaces the existing one.

In the case the computer that is running SSPM SGX has to restart or turn off, the program will try to wait for the second thread to initiate its execution before shutting down, and store DBPW in disk. After a reboot SSPM SGX will try to read from disk a DBPW data structure, so it can restore all the data structures that compose SSPM SGX.

In the Algorithms 3 and 4 we can see the pseudo-code that we use for the application that interacts with the Intel SGX. In the Algorithm 5 we can see the pseudo-code that shows how the operations inside the Enclave that we designed are made.

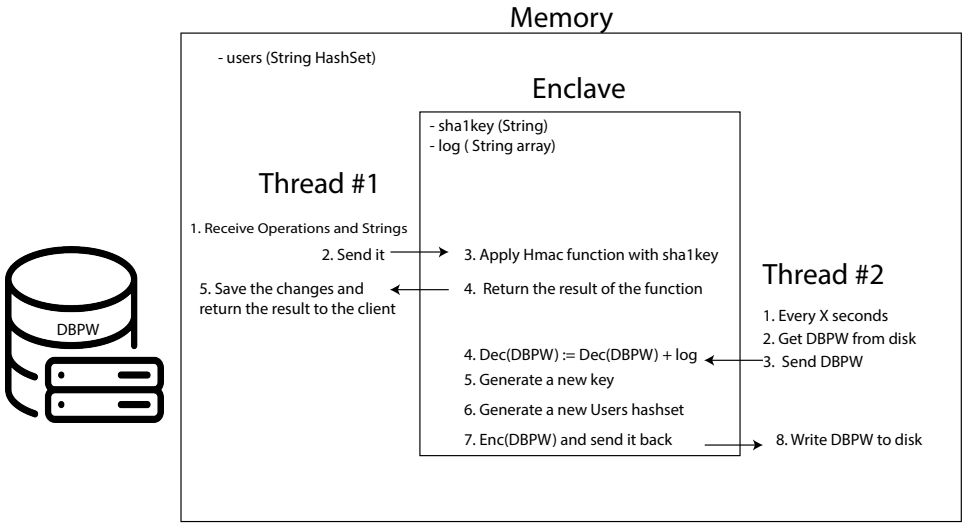


Figure 3.4: Visual representation of how the SSPM SGX

3.3.2.1 Security Analysis of SSPM SGX

Here we make the security analysis of the SSPM scheme with the incorporation of our implementation of SGX.

SSPM SGX also keeps the Users and the Usernames data structures in memory, as we already described in subsection 3.2.1.1. Inside the Enclave created by Intel SGX, SSPM generates the cryptographic key to be used for the HMAC-SHA1 function. In this version, all cryptographic functions are made inside the Enclave, as the key never leaves the Enclave. The program in main memory sends the strings that require to be hashed (the concatenation of the user and their password) to the Enclave. We consider the generated cryptographic key to be as secure, as long as the data inside the Enclave will not be breached.

Algorithm 3 SSPM SGX - Main Memory

1: $Users$: HashSet with all the hashes of the concatenations of the usernames and their passwords
2: $Usernames$:HashSet with all the hashes of just the usernames
3: λ : Security parameter
4: **procedure** SSPM_SGX (λ)
5: $DBPW \leftarrow getDBPWfromDisk()$
6: $ecall\_init(DBPW, length)$
7: $new\ Thread()\{$
8: $Sleep(216000)$ $//every\ hour$
9: $DBPW \leftarrow getDBPWfromDisk()$
10: $DBPW \leftarrow ecall_newHmac(DBPW, length)\}$
11: **procedure** LOGIN (USERNAME, PASSWORD)
12: $hashedUser \leftarrow ecall\_hmac\_this(0, username + sep + password, length)$
13: **if** $hashedUser$ in $Users$ **then**
14: **return** $true$
15: **else**
16: **return** $false$
17: **procedure** REGISTER (USERNAME, PASSWORD)
18: $hashedUsername \leftarrow applyHMAC(key, username)$
19: **if** $hashedUsername$ in $Usernames$ **then**
20: **return** $false$
21: **else**
22: $Usernames \leftarrow Usernames \cup hashedUsername$
23: $hashedUser \leftarrow ecall\_hmac\_this(1, username + sep + password, length)$
24: $Users \leftarrow Users \cup hashedUser$
25: **return** $true$
26: **procedure** CHANGEPASSWORD (USERNAME, OLDPASSWORD, NEWPASSWORD)
27: $concatenatedUser \leftarrow username + sep + oldPassword$
28: $hashedUser \leftarrow ecall\_hmac\_this(2, username + sep + oldPassword, length)$
29: **if** $hashedUser$ not in $Users$ **then**
30: **return** $false$
31: **else**
32: $Users \leftarrow Users \setminus hashedUser$
33: $hashedUser \leftarrow ecall\_hmac\_this(2, username + sep + newPassword, length)$
34: $Users \leftarrow Users \cup hashedUser$
35: **return** $true$

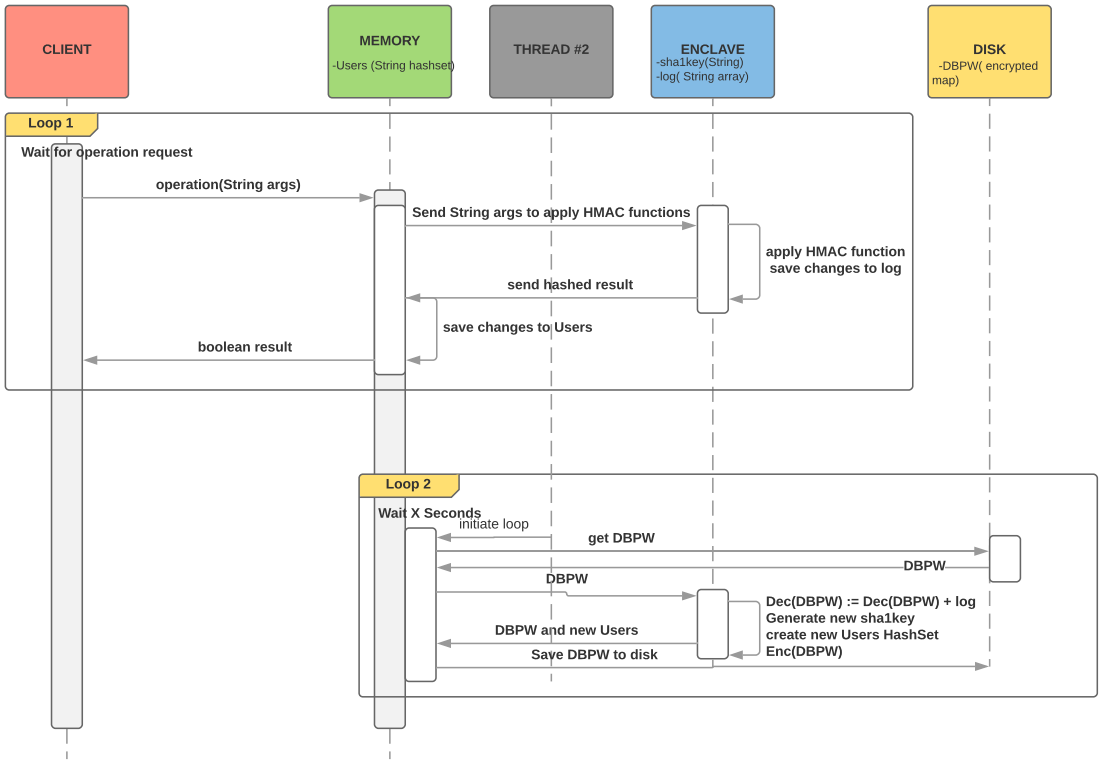


Figure 3.5: Sequence diagram of SSPM SGX

Algorithm 4 SSPM SGX - Main Memory 2

1: **procedure** CHANGEUSERNAME (OLDUSERNAME, NEWUSERNAME, PASSWORD)
2: $hashedUser \leftarrow ecall\_hmac\_this(3, oldUsername + sep + password, length)$
3: **if** $hashedUser$ not in $Users$ **then return** $false$
4: **else**
5: $Users \leftarrow Users \setminus hashedUser$
6: $hashedUser \leftarrow ecall\_hmac\_this(3, newUsername + sep + password, length)$
7: $Users \leftarrow Users \cup hashedUser$
8: **return** $true$
9: **procedure** DELETEUSER (USERNAME, PASSWORD)
10: $hashedUser \leftarrow ecall\_hmac\_this(0, Username + sep + password, length)$
11: **if** $hashedUser$ not in $Users$ **then return** $false$
12: **else**
13: $Users \leftarrow Users \setminus hashedUser$
14: $hashedUsername \leftarrow applyHMAC(key, username)$
15: $Users \leftarrow Users \setminus hashedUser$
16: $Usernames \leftarrow Usernames \setminus hashedUsername$
17: **return** $true$

Algorithm 5 SSPM SGX - Enclave Operations

sep: Random String to seperate the username of their password, during the concatenation
log: Array of Array of String to save the latest changes made to the database
hmac_key: Random String key for the HMAC hashing operations

procedure ECALL_INIT (DBPW, LENGTH)
    $sep \leftarrow generateRandomString()$
    $hmac\_key \leftarrow generateRandomKey()$
    if $DBPW$!= $null$ **then**
        $newUsers \leftarrow generateNewUsersHashset(DBPW, hmac\_key)$
        return $newUsers$
    else
        return $null$

procedure ECALL_HMAC_THIS (CODE, USER, LENGTH)
    $log \leftarrow log \cup code, user$
    $hashedUser \leftarrow applyHmacFunction(user, hmac\_key)$
    return $hashedUser$

procedure ECALL_NEWHMAC (DBPW, LENGTH)
    $decDBPW \leftarrow decrypt(DBPW)$
    $decDBPW \leftarrow decDBPW \cup log$
    for all $entry \in decDBPW$ **do**
        $hashedEntry \leftarrow applyHmacFunction(entry, hmac\_key)$
        $Users \leftarrow Users \cup hashedEntry$
    $DBPW \leftarrow encrypt(decDBPW)$
    return $\{DBPW, Users\}$

To obfuscate leaks of access patterns to the adversary, SSPM periodically generates new Users hashes. To achieve this we use the same approach as in subsection 3.2.1.1, we save every usernames and passwords in plaintext, in an encrypted map. In this version, we encrypt the hash map with the unique CPU id using Intel SGX. We refer to this data structure as DBPW. DBPW is kept in disk. Periodically, the program in main memory gets the DBPW from disk and sends it to the Enclave for it to decipher, and make a new Users hash-set. This way SSPM doesn't leak access patterns to the adversaries and assures the security of the cryptographic keys. If the adversary gains access to DBPW, it must gain physical access to the CPU to be able to decipher it.

CHAPTER 4

Implementation and Evaluation

In this chapter we show in more detail the design and implementation of our scheme and we also show performance evaluations that we made, comparing it to other Database Password Schemes.

All the work is currently implemented in JAVA in a Windows environment, except the implementation with SGX that was made in C and C++, due to the fact that Intel released the SGX Software Development Kit (SDK) only being available in this programming language and it's available in a Linux environment.

4.1 Implementing existing Hashing schemes

To evaluate our scheme in the most accurate way, we imported the most used password database schemes used nowadays, majorly from open source projects and other means of literature, like papers. We modified some part of the open source code so we can best incorporate it in our designed API, always keeping the copyright left by the original authors.

The following hashing and database schemes are available in our universal API that we built:

- Diverse traditional hashing schemes :
  - **Plain Text**
    Although it is not secure, we considered it for evaluation.
  - **MD5**
    It is also not considered secure. For the use of the MD5 function we use

the Security library included in Java and call the MD5 function from there.

  - **SHA-1**
    Also use the security library included in JAVA.
  - **SHA-224**
    Also use the security library included in JAVA.
  - **SHA-3**
    To implement SHA-3 we used the open source library available in https://www.bouncycastle.org/. We also adapted some of its code that we used.
  - **HMAC-SHA1**
    To implement HMAC-SHA1 (and all other HMAC) algorithms we use the Javax Crypto libraries.

- **B-Crypt**
  Adapted from the open source repository available online in https://github.com/jeremyh/jBCrypt.

- **PolyPasswordHasher**
  Adapted from the open source repository available online in https://github.com/PolyPasswordHasher/PolyPasswordHasher-Java.

All the schemes mentioned above were imported from some already existing sources and were modified making use of online documentation, literature and online forums. All of these schemes were modified by us, so they can accommodate the universal API that was previously mentioned in Subsection 3.1.1.

4.2 Implementation of SSPM

The intuition behind the SSPM is that we can use Searchable Symmetric Encryption techniques, namely encrypted data structures, in a way that it can support the most common operation of a database password management scheme with the most viable security and efficiency. A structure of ciphered data is a structure like a dictionary, where every entry is encrypted with a deterministic encryption scheme. The use of deterministic encryption allows to easily make edit and remote queries over the data, sending to the server a ciphered *token* with a deterministic encryption scheme.

To compensate the evident security fault introduced by the determinism of the encryption scheme, the structure of encrypted data obliges, by definition, that each

entry saves a different value. This way we can obtain the same level of security as in a probabilistic encryption scheme, even when the data structure is not in memory when the process is running, and only showing information and access patterns when it is time for the execution of operations (e.g. login operations).

We use data structures from the default JAVA Library. In SSPM we use HMAC-SHA1 and AES encryption, both of theses cryptographic methods are imported from the Javax crypto library.

We have several versions of SSPM and each one is organized in a different Java class. To run these schemes, a Main class was created. All of the schemes can be invoked by simply creating an object of the desired scheme, and simply run the desired operations (login, register...).

4.2.1 Extending functionalities

A problem with the design of SSPM is that it depends on a cryptographic key and it has to be saved near the database password. To prevent this security fail, we used two different approaches: The use of TPMs (Trusted Platform Modules) as secure specialized hardware devices that can save the cryptographic keys and make light cryptographic operations. Also useful, is the use of new technologies of trustable hardware in common CPUs, like Intel SGX, for the creation of reserved and secure memory zones where the keys can be stored and operated securely. In the following two sections we are going to explain in detail how we implemented our scheme using the integration of TPM and Intel SGX, respectively.

In some schemes we use a Redis database to store the data and achieve persistence, so the person that uses the SSPM can store all the information relative to the data structures to be stored in another computer (through setting another host computer to store the data), or in the same computer that SSPM is running (through local host).

4.3 Using Trusted Platform Module

For the use of Trusted Platform Modules (TPM), we adapted the open source code in Microsoft's public repository at github, available in `https://github.com/Microsoft/TSS.MSR`. We adapted this project to incorporate it in our API. We made some changes to the code available, and added some operations to easily store and retrieve a cryptographic key. The system that we made checks on start up, if a TPM module exists in the users computer, so our scheme can connect to it. If it does not exist a running TPM Simulator service is required to be running in the

users computer, which we include in the folder of the SSPM versions that feature TPM.

We implemented the SSPM variants that feature TPM, in JAVA, and also made performance evaluations using the same programming language. The TPM can be used using one or both of the following ways:

- All cryptographic operations, like hashing and encryption, are made inside the TPM. This way, all keys are generated and stored in the TPM, and don't leave the module. Using this method, we do not reveal any access pattern to an adversary, unless it has physical access to the computer where the password database is stored. Although, we found out, that using this method is very slow in both simulator mode and in hardware mode.

- This solution, which is faster, consists of only storing the cryptographic keys inside the TPM, while all cryptographic operations are run in memory, as normal. The adversary can also only obtain the keys if it has physical access to the users computer. Although, this method is considerably faster it is less secure than running all cryptographic operations inside the TPM, because it may reveal some access patterns that an Adversary can take advantage of.

4.4 Using Intel SGX

To make use of Intel SGX we used the Software Development Kit that was officially released by Intel in github, available in https://github.com/intel/linux-sgx. Our scheme is only compatible with a Linux environment. We used C and C++ programming languages to produce the SSPM schemes that feature Intel SGX. We implemented our system from the ground up in C++, for us to make us of this SDK (Software Development Kit).

Inside the SGXs Enclave most normal data structures are not supported, for security reasons. Some hashing and encryption schemes like HMAC-SHA1 and AES encryption are not easily importable. To make use of hashing and encryption operations we used the libsodium and libsodium-sgx libraries. Libsodium is an easy-to-use software library for encryption, decryption, signatures, password hashing and more, which is publicly available in https://github.com/jedisct1/libsodium. For another user to make use of our scheme, it must first install the Intel SGXs SDK and libsodium libraries. More detailed information about installation can be found in the repository of our scheme, which can be found in https://github.com/Mgj645/sspm_c.

In the final implementation of SGX in SSPM we use only a total of 3 ecalls. During the calls of the ecall functions, at all times the length of the sent argument must be specified, unless we send a value of fixed length, like an integer or a single char. Here are the refereed ecalls:

```
1 ecall_init(DBPW, length);
2 ecall_hmac_this(code, userConcatenation, length);
3 ecall_newHMAC(DBPW, length)
```

The first ecall is part of the initialization process of the Enclave. The intent to have a function to start, is for some variables to be initialized and assigned in case it is the first time that this Enclave is ran. If this is the first time that the Enclave is ran, the variable to be sent over an argument will be a null value, with the length of 0. If it is not the first time, an encrypted database will be attempted to be read from disk containing a hash map with the Usernames and the respective mapped Passwords, and it will be sent over, for it to be unencrypted by the Enclave using the unique key inside the CPU. Then the Enclave will generate a new HMAC-SHA1 key and create a new Users hash set (using the decrypted DBPW) and return it to main memory.

The second ecall is for the program running in main memory to call when it has to make a hashing operation, since the HMAC-SHA1 key does not leave the Enclave. The function receives a code by the form of an Integer, that represents the type of operation that the program in main memory is making, the Enclave saves this information in a data structure called Log, so it can later update DBPW. The code 0 means it is a Register operation, 1 is Login, 2 is Change Username, 3 is Change Password and 4 is Delete. The function divides the received userConcatenation (that is a char array) based on the received code, there can be two or three keywords inside this char array, so the division is important. After this the function returns to the program in main memory the result of the HMAC-SHA1 operation.

The third ecall is only called periodically once every X seconds (the user that is running can change theses values), by the program in main memory. One of the two intuitions is to send the encrypted Database to the Enclave so it can decipher it and update it using the Log that is stored inside the Enclave. This ecall also triggers the Enclave to generate a new cryptographic key for the HMAC-SHA1 function and uses the data of the deciphered DBPW to make a new Users hash set. After this, SSPM encrypts the DBPW with the unique identifier of the CPU and returns it, as well as the Users hash set.

4.5 Evaluation Performance

In this section we are going to show the stress tests that we made. To run these tests we used a computer with an Intel Core i5 6500 3.2 ghz com 4 cores CPU, 16 GB of Dual-Channel Ram with 2133 MHZ frequency, running on Windows 10 with CPU Priority set to Very High, except for the tests made using SSPM SGX, that were made in a Linux environment using the distro Linux Mint 19. We ran the same tests at least five times, and the values that we present in the Figures below are the the resulting average of these tests. We ran the same tests several times to make sure that the obtained the most reliable resulting values, although the did not vary much.

For a most realistic test of our schemes we obtained a big data set, consisting of the most common Usernames and Passwords. For the Passwords, we used the most common used Passwords on the Internet, based on a study of Internet security and available online in the repository `https://github.com/danielmiessler/SecLists/tree/master/Passwords`, that consists in analysing password database *leaks*, that have been happening through the last years. For the usernames we used the most common names in the English Language.

We compare every scheme that we implemented with the most basic version of SSPM (SSPM V0 and SSPM V1) using Login and Register operations. The other versions of SSPM include other security properties that could interfere in the performance tests, so we only used these 2 versions.

4.5.1 PolyPasswordHasher

In this subsection we can see the performance comparison of our SSPM V1 scheme with PolyPasswordHasher. For the evaluation in this example we used one and ten thousand entries for the login and register operation and one, ten and hundred thousand entries for the change username, change password and delete operations, because these numbers are enough to obtain conclusive results, using our constructed API and our big data set of usernames and passwords to make Register operations followed by Login operations.

By our several tests, we can conclude that our scheme is considerably faster in both Register and Login operation (as we can see in Figures 4.1 and 4.2]), as we only use conventional cryptographic encryption based on HMACs, instead of threshold cryptography. We can see that PolyPasswordHasher is fast with a small amount of entries, but as the database grows the time to make an operation increases greatly. To note that we see similar results for PolyPasswordHasher in Login and Register

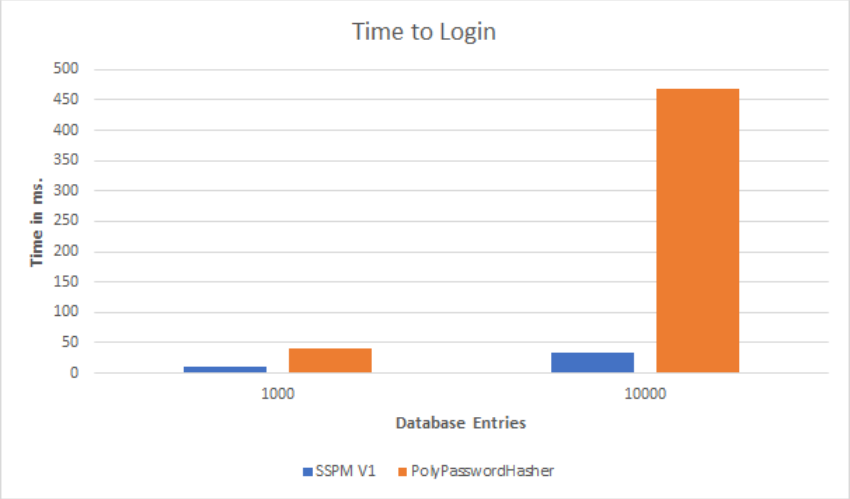


Figure 4.1: Database Performance with up to 10 000 entries, using SSPM and PolyPasswordHasher - Login Operations

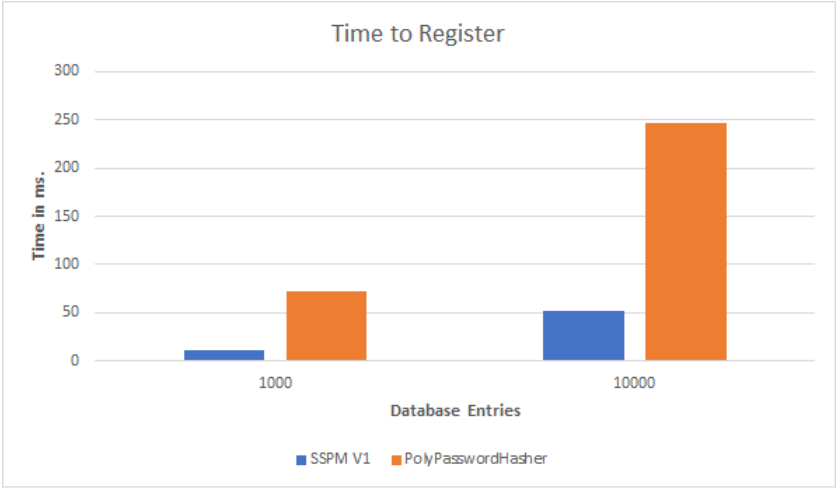


Figure 4.2: Database Performance with up to 100 000 entries, using SSPM and PolyPasswordHasher - Register Operations

operations.

In Figures 4.3, 4.4 and 4.5 we can see the results of the performance comparison between SSPM V1 and PolyPasswordHasher for the change username, change password and delete operations, respectively. In these operations PolyPasswordHasher is shown to be a lot faster than its login and register operations, and is actually faster than SSPM V1 up to ten thousand database entries, but as we already mentioned, PolyPasswordHasher tends to have slow operations as the database grows, and is about four times slower than SSPM V1, to make one hundred thousand operations.

With these results we conclude that PolyPasswordHasher is a good scheme for a Password Database Scheme with a small amount of entries.

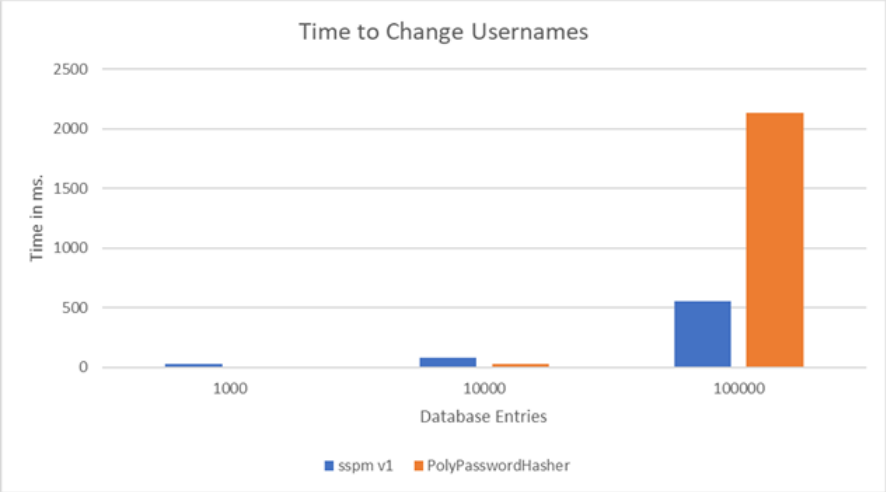


Figure 4.3: Database Performance with up to 100 000 entries, using SSPM and PolyPasswordHasher - Change Username Operations

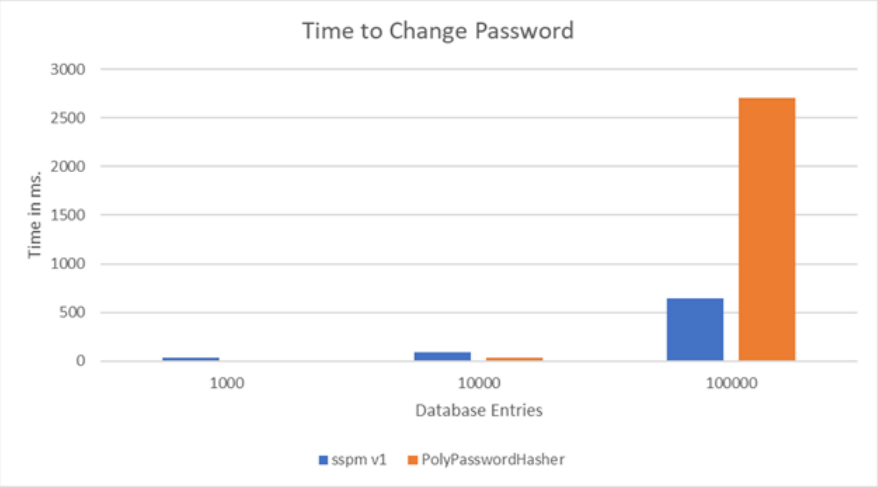


Figure 4.4: Database Performance with up to 100 000 entries, using SSPM and PolyPasswordHasher - Change Password Operations

4.5.2 B-Crypt

In Figure 4.6 and Figure 4.7 we can see the performance comparison between our scheme and B-Crypt, using login and register operation. We used just 100 and 500 entries in this test, because it was enough entries to reveal that B-Crypt is so much slower than our scheme. We think this major difference in performance is due to the fact that B-Crypt uses multiple rounds of symmetric encryption while our scheme only uses HMAC functions.

To better demonstrate the performance results of the Change Username, Change Password and Delete operations we made the tables 4.1, 4.2 4.3, that respectively show the results of the mentioned operations. We also used just 100 and 500 entries

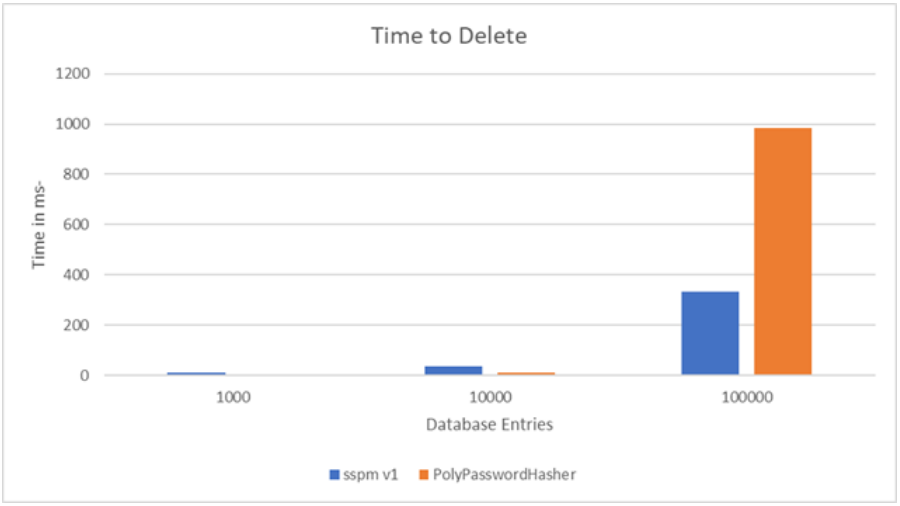


Figure 4.5: Database Performance with up to 100 000 entries, using SSPM and PolyPasswordHasher - Delete Operations

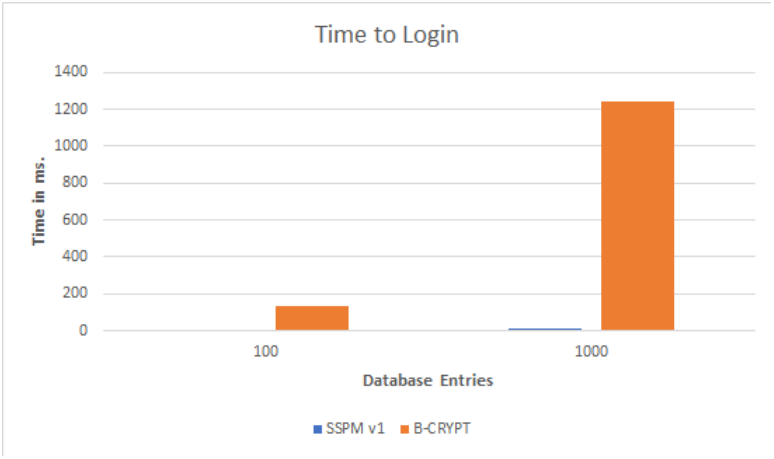


Figure 4.6: Database Performance with up to 500 entries, using B-CRYPT and SSPM V1 - Login Operations

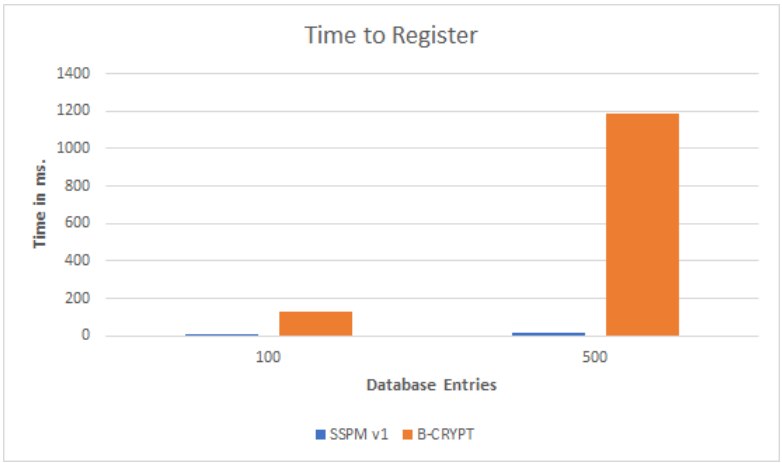


Figure 4.7: Database Performance with up to 500 entries, using B-CRYPT and SSPM V1 - Register Operations

Table 4.1: Database Performance with up to 500 entries, using B-CRYPT and SSPM V1 - Change Username Operations

Database Entries	SSPM V1	B-CRYPT
100	16 ms.	15944 ms.
500	28 ms.	79449 ms.

Table 4.2: Database Performance with up to 500 entries, using B-CRYPT and SSPM V1 - Change Password Operations

Database Entries	SSPM V1	B-CRYPT
100	8 ms.	16002 ms.
500	10 ms.	79584 ms.

Table 4.3: Database Performance with up to 500 entries, using B-CRYPT and SSPM V1 - Delete Operations

Database Entries	SSPM V1	B-CRYPT
100	3 ms.	8017 ms.
500	6 ms.	39427 ms.

in these tests, because it was enough to show that B-CRYPT is much slower than SSPM V1. We used tables to show the result of these operations, instead of the usual graphics because the difference between SSPM V1 and B-CRYPT was just too much to understand it through graph bars.

4.5.3 Intel SGX

In Figure 4.8 and Figure 4.9 we can show the performance test on login and register operations that we made using our SGX integration in SSPM with SSPM V0 and SSPM V1 tests also made here. We can see that the time it takes to complete the amount of logins and sign ups is linear, and on average if it takes X time to make 1000 operations it will take X*10 time to make 10000 operations. Despite the fact that SSPM SGX is a bit slower than the most conventional versions of our scheme, its performance is very good with little overhead performance. We can see that the difference between the version of SSPM that depends on SGX and those that do not, do not show much of a difference in terms of time to make the login and register operations.

In Figure 4.10, Figure 4.11 and Figure 4.12 we can see the results of the performance comparison between SSPM V1, SSPM V0 and SSPM SGX for the change username, change password and delete operations, respectively. We can note that

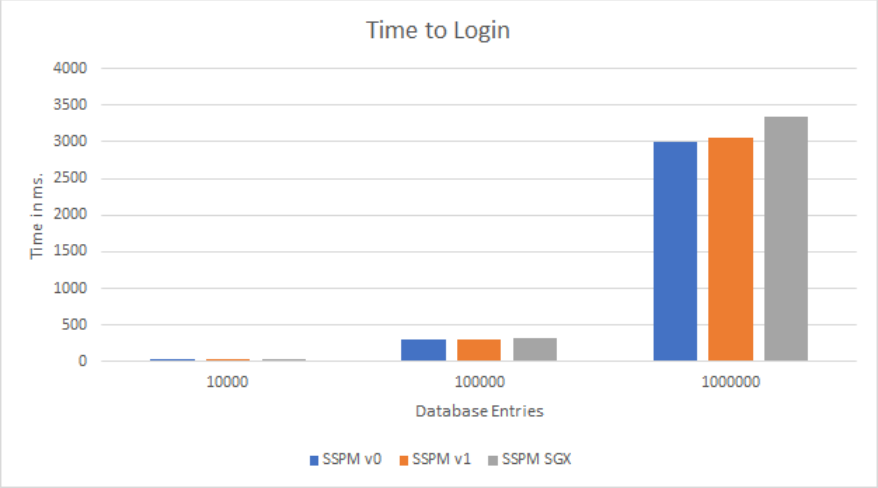


Figure 4.8: Database Performance with up to 1M entries, using SSPM and SSPM SGX - Login Operations

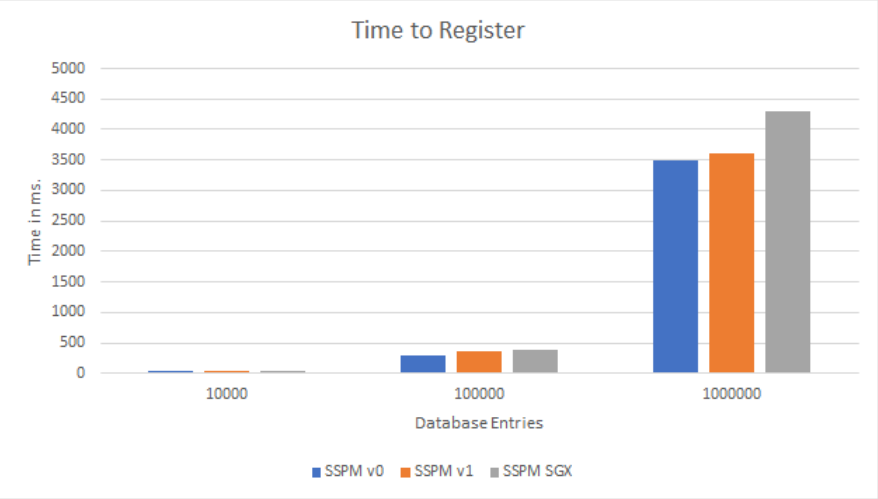


Figure 4.9: Database Performance with up to 1M entries, using SSPM and SSPM SGX - Register Operations

SSPM SGX takes a bit longer during these operations than SSPM V0 and SSPM V1, than in the login and register operations. This is mostly due to the fact, that the change username and change password operations make more operations to accomplish this result.

4.5.4 Other hashing Schemes

In this subsection we can see the performance comparison between our scheme (version V0 and V1) and the most common *hashing* plus *salt* used today. We used ten million entries, running the test three times to obtain the most possible conclusive and fair results.

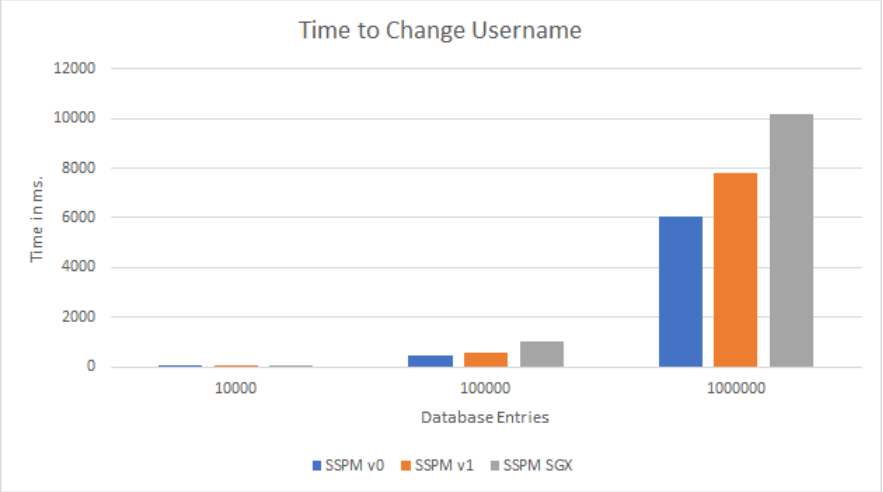


Figure 4.10: Database Performance with up to 1M entries, using SSPM and SSPM SGX - Change Username Operations

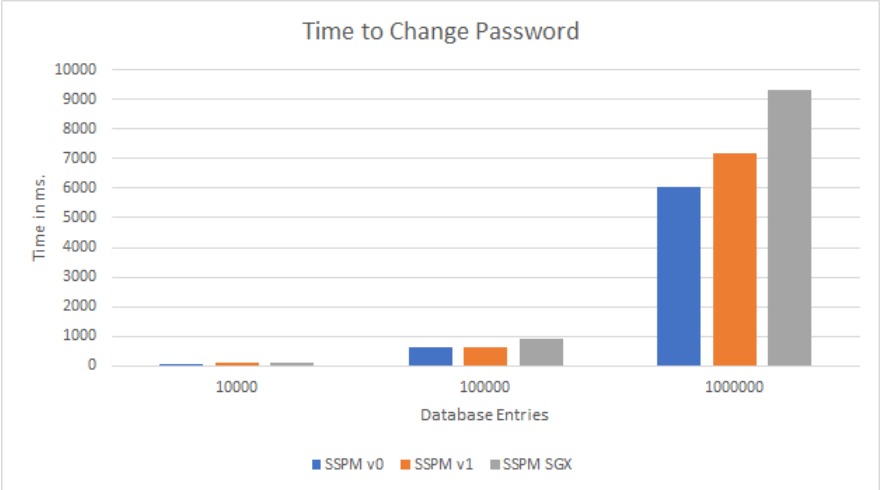


Figure 4.11: Database Performance with up to 1M entries, using SSPM and SSPM SGX - Change Password Operations

In Figure 4.13 and Figure 4.14 we show the results of the login and register operations respectively. Here we can see, as expected, the version V1 of our scheme is somewhat slower than version V0. Also, as expected the plain text is faster than all our secure versions of our schemes, and other hashing functions. The schemes SHA-224 and SHA-1, present slightly better results than our scheme, while the scheme SHA-3 is slightly slower in the login and register operations (which is normal, because our scheme uses HMAC-SHA1). Also, in Figure 4.17 we can see the result of the delete operations, which have similar results to the login and register operations, although SSPM V1 was not as fast as expected.

In Figure 4.15 and Figure 4.16 we can see the results of the change usernames and change passwords operations respectively. As we expected in the original design

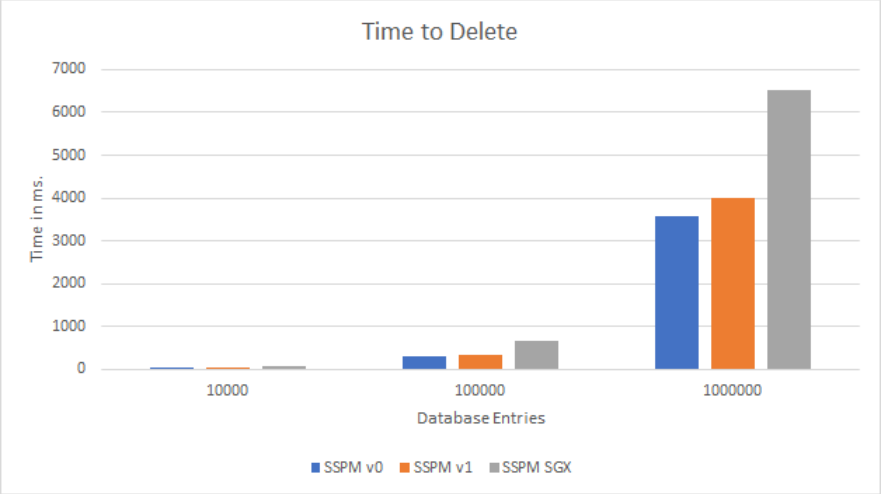


Figure 4.12: Database Performance with up to 1M entries, using SSPM and SSPM SGX - Delete Operations

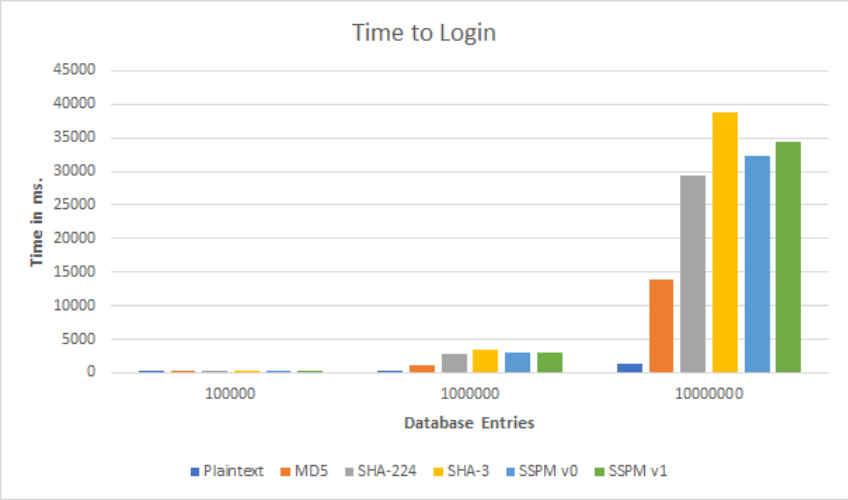


Figure 4.13: Database Performance with up to 10M entries, using various schemes - Login Operations

of SSPM, both schemes are a lot slower in these operations, due to the higher amount of operations needed to update the data structures inside SSPM.

We can view that simple hashing schemes are faster than our versions of SSPM, although they are not as a secure way to store use's passwords. As we were predicting, SSPM shows its best results in the Login, Register and Delete operations and is only a little slower than simple hashing schemes in these operations.

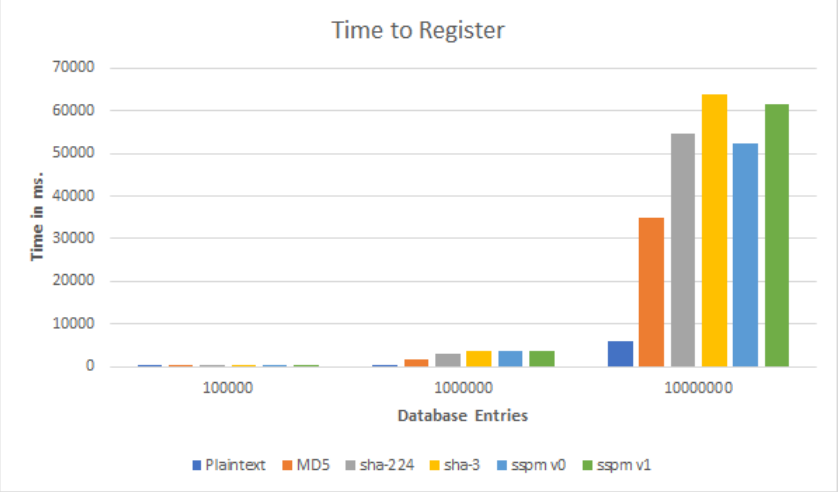


Figure 4.14: Database Performance with up to 10M entries, using various schemes - Register Operations

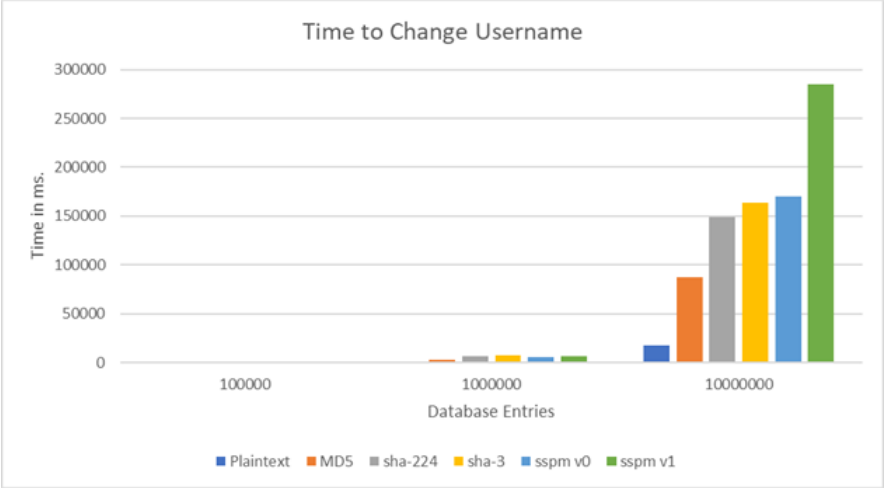


Figure 4.15: Database Performance with up to 10M entries, using various schemes - Change Usernames Operations

4.6 Graphical User Interface

To better display the work we made during this thesis, we made a graphical user interface showcasing the hashing schemes we implemented. The API we constructed (Subsection 3.1.1) is exposed in this graphical interface. The layout of the application can be seen in Figure 4.18. We used the library JAVA FX to design and implement the application.

On the left side of the program we can select the scheme that we intend to test, the schemes available are all those we implemented from existing literature or from open source material. We don nott have PolyPasswordHasher in the interface, because it has some restrictions in this scheme, namely the way this scheme stores

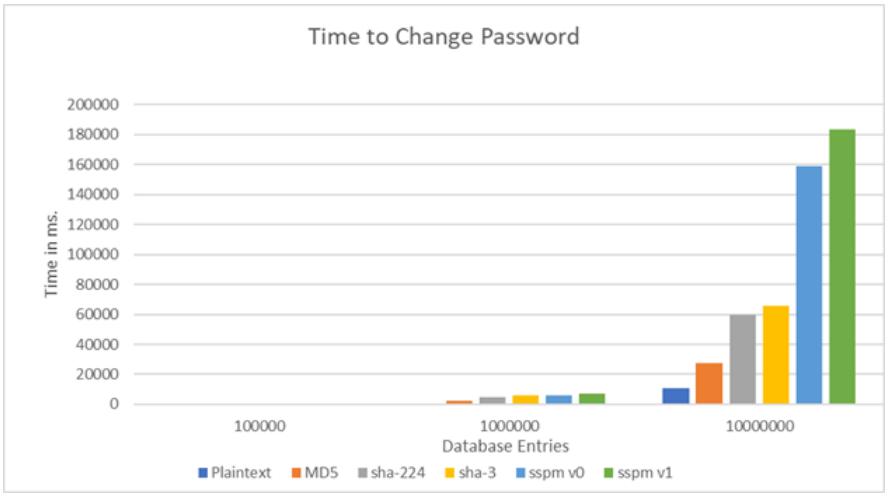


Figure 4.16: Database Performance with up to 10M entries, using various schemes - Change Passwords Operations

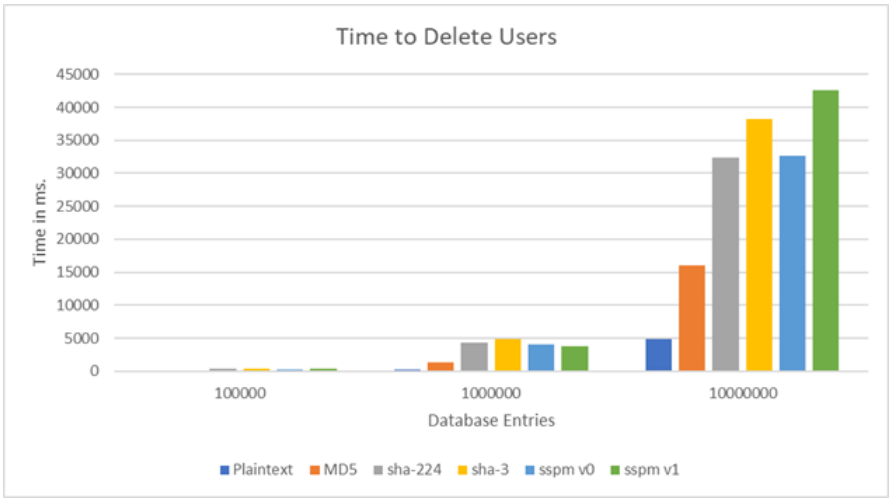


Figure 4.17: Database Performance with up to 10M entries, using various schemes - Delete Operations

the users password, as it concatenates some users passwords together, so we choose to not display PolyPasswordHasher in the interface as of right now. We can choose to simulate some users to fill the database, using the form in the bottom left, that makes use of our big data collection of usernames and passwords. Please note that the data collection (the Usernames.txt and Passwors.txt) must be inside the right directory for this option to work.

The operations form makes use of our universal API, and works the same way for all schemes. We can choose any of the operations available. The console below shows if the instruction was successful. We also have a console showing how much time in ms it took to made the operations when users are simulated using the bottom left form.

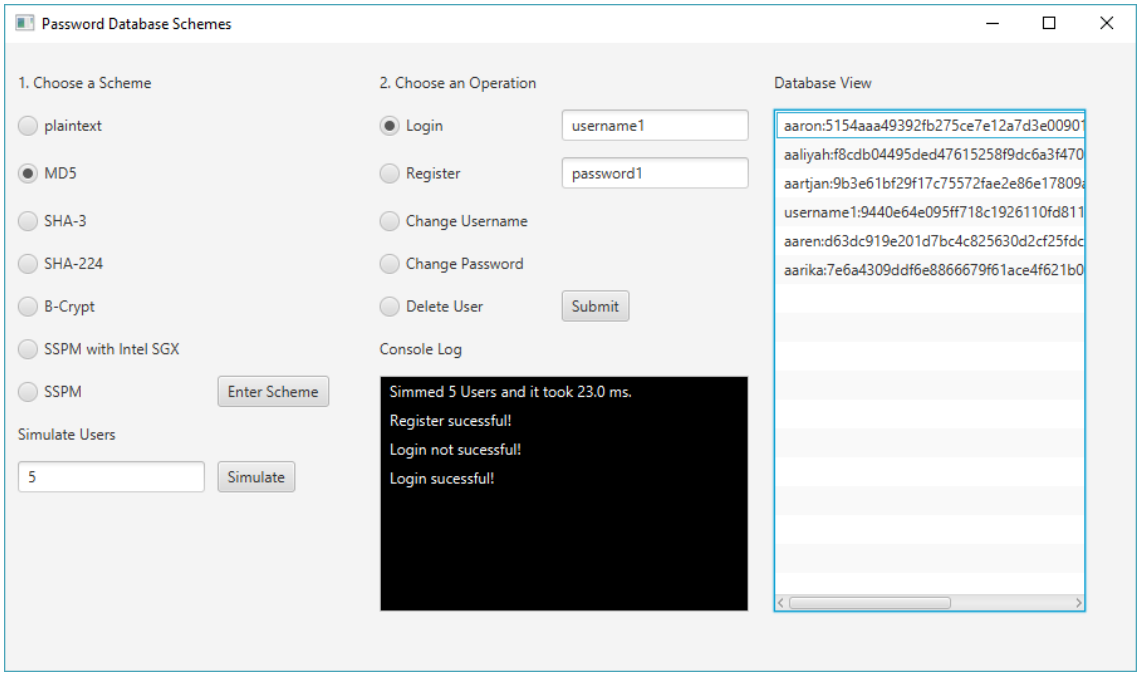


Figure 4.18: Graphical Interface

On the right side of the application we have a view of how the entries on the database are stored.

In Figure 4.18 we can see an example of how the execution of application can turn out to be. In this example we have chosen the MD5 scheme and simulated 5 users. Furthermore we registered one user manually, through the form in the middle, and the result of its database is on the right column of the application.

If the user chooses our SSPM scheme in the left column, it will be prompted with a pop up window as shown in Figure 4.19. This will ask the user what properties of the SSPM scheme will it want to use, before launching the scheme. This uses the version of SSPM that we consider final, that is described in subsection 3.2.1

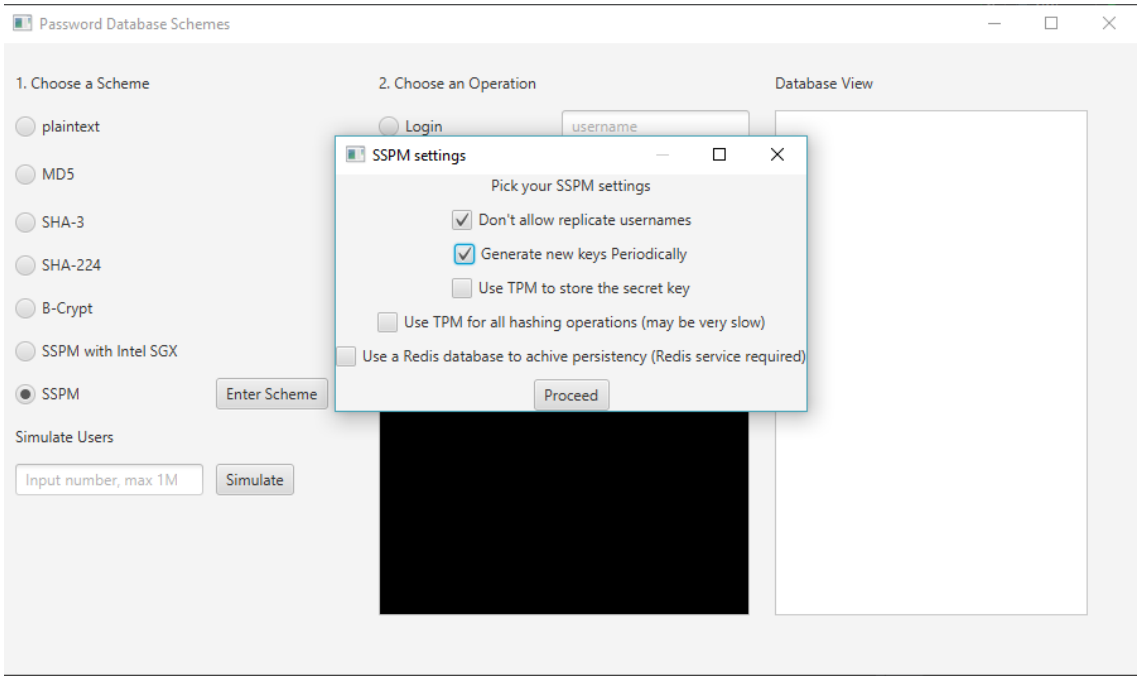


Figure 4.19: Graphical Interface - SSPM Options

Chapter 5

Conclusion

In this chapter we give our final thoughts on what was accomplished during the work developed in this thesis and how this work can be later resumed by another researcher.

5.1 Final Remarks

In this thesis we proposed a new password database management scheme called SSPM (Simple Secure Password Management). SSPM allows for the administrators of websites to store the users credentials in a simple, safe and secure manner. The methodology of SSPM is based on Searchable Symmetric Encryption techniques and encrypted data structures that does not allow adversaries that get a snapshot copy of the database, to decipher it, even when using offline attacks, suck as brute force attacks, dictionary attacks or using rainbow tables.

In the first part of this thesis we implemented SSPM in Java, along with the most common hashing schemes today and other used schemes, such as B-Crypt and PolyPasswordHasher, constructing an universal API for the most common password database operations. Several performance tests were performed using different scenarios, like making tests that involved the sign up of up to 10 million users, using the implemented API, for each scheme. By doing this we were able to compare the performance and conditions of the database password management schemes. We also created a version of SSPM that stores the data in a Redis server.

We also used TPM in an alternative version of SSPM, to store and keep the cryptographic keys. We also tested a way to use the TPM to run all hashing and

encryption functions, but we found this method to be very slow. We found that only storing the keys in the TPM is the best way to use it. We kept both versions of our implementation of SSPM using TPM, so that others can test it if they desire. Our results of TPM were not very satisfactory as TPM showed to be very slow to make cryptographic operations, although it was a good, secure way of storing the needed cryptographic keys.

For us to use Intel SGX (Intel's Software Guards Extensions) we had to implement our scheme in C and C++ language, in a Linux environment, because we used the official Linux SDK available in Intel's online repository of GitHub. This SDK only works in an Linux environment. We concluded with our benchmarks, that Intel SGX has very little overhead performance, even when required to make a lot of operations at the same time, making it perfect to use on our password database schemes, and we have also proven that this technology made by Intel has the capabilities to also work in other areas, due to its security measures and good performance.

The results that we obtained, show that SSPM is a bit slower than the conventional hashing plus salt schemes, although it offers better security measurements against attacks made by adversaries. The inclusion of Hardware Secure Platforms, such as TPM and Intel SGX, only reinforced these security measures. On the contrary, SSPM revealed to be much faster then other password database, such as B-Crypt and PolyPasswordHasher.

The API that we implemented works for all mentioned schemes, and offers the basic operations of a password database management scheme and it is simple to use, what allows to any person to download and test our work, making it to any other fitting project. We also designed a graphical interface that makes use of our universal API, so that the we can better demonstrate ours and other schemes. Our work is available online in the following github repositories: `https://github.com/Mgj645/Thesis_crypts` and `https://github.com/Mgj645/sspm_c` , and the work will continue to be updated in the following months.

5.2 Future Work

In this section we state the work that we believe that can be improved to the work already done during the course of this thesis:

- With Intel SGX, the encrypted database that can be used to recover the User's credentials requires the CPU unique key that initially encrypted it. This poses a problem if the CPU malfunctions as the data will be gone forever. Work in

Intel SGX can be done to make the migration of the original CPU to a new one.

- Implementation of a Redis integrated system in the SGX variant of our scheme. Right now the encrypted data is stored in the disk where the scheme is running.
- Repartition of the Users database, so the most recently accessed users credentials are kept in a different data structure accessed in memory, and other later accessed users credentials are kept in disk. The objective of this is to not occupy the memory too much, specially when the users database gets increasingly bigger with the passing of time.
- Remove some bugs that are left by Intel's SDK of SGX, and keep it updated as new versions of SGX are released by Intel.
- Integration of SSPM SGX in the GUI application. We did not include the SGX variant in the application because it was programmed in a different language and a way to pass the String formed in Java to C++ would require a way to encapsulate the data, like a JSON document.

Bibliography

[1] URL: https://www.google.com/recaptcha/api2/demo.

[2] URL: http://www.captcha.net/.

[3] M. H. Almeshekah, C. N. Gutierrez, M. J. Atallah, and E. H. Spafford. "Ersatzpasswords: Ending password cracking and detecting password leakage." In: *Proceedings of the 31st Annual Computer Security Applications Conference*. ACM. 2015, pp. 311–320.

[4] A. Bogdanov, D. Khovratovich, and C. Rechberger. "Biclique Cryptanalysis of the Full AES." In: *Advances in Cryptology - ASIACRYPT 2011 - 17th International Conference on the Theory and Application of Cryptology and Information Security, Seoul, South Korea, December 4-8, 2011. Proceedings*. 2011, pp. 344–371. DOI: 10.1007/978-3-642-25385-0_19. URL: https://doi.org/10.1007/978-3-642-25385-0_19.

[5] A. Boldyreva, N. Chenette, Y. Lee, and A. O'Neill. "Order-Preserving Symmetric Encryption." In: *Advances in Cryptology - EUROCRYPT 2009, 28th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cologne, Germany, April 26-30, 2009. Proceedings*. 2009, pp. 224–241. DOI: 10.1007/978-3-642-01001-9_13. URL: https://doi.org/10.1007/978-3-642-01001-9_13.

[6] D. Boneh and B. Waters. "Conjunctive, subset, and range queries on encrypted data." In: *Theory of Cryptography Conference*. Springer. 2007, pp. 535–554.

[7] D. W. D. W. C. H. Bruce Schneier John Kelsey and N. Ferguson. "Performance Comparison of the AES Submissions." In: 1999.

[8] J. Cappos and S. Torres. *PolyPasswordHasher: Protecting Passwords In The Event Of A Password File Disclosure*. Tech. rep. 2014.

[9] D. Cash, J. Jaeger, S. Jarecki, C. S. Jutla, H. Krawczyk, M.-C. Rosu, and M. Steiner. "Dynamic Searchable Encryption in Very-Large Databases: Data Structures and Implementation." In: *NDSS*. Vol. 14. Citeseer. 2014, pp. 23–26.

[10] S. Chakrabarti and M. Singhal. "Password-Based Authentication: Preventing Dictionary Attacks." In: *IEEE Computer* 40.6 (2007), pp. 68–74. DOI: 10.1109/MC.2007.216. URL: https://doi.org/10.1109/MC.2007.216.

[11] D. Coppersmith. "Small Solutions to Polynomial Equations, and Low Exponent RSA Vulnerabilities." In: *J. Cryptology* 10.4 (1997), pp. 233–260. DOI: 10.1007/s001459900030. URL: https://doi.org/10.1007/s001459900030.

[12] V. Costan and S. Devadas. "Intel SGX Explained." In: *IACR Cryptology ePrint Archive* 2016 (2016), p. 86.

[13] R. Curtmola, J. A. Garay, S. Kamara, and R. Ostrovsky. "Searchable symmetric encryption: Improved definitions and efficient constructions." In: *Journal of Computer Security* 19.5 (2011), pp. 895–934. DOI: 10.3233/JCS-2011-0426. URL: https://doi.org/10.3233/JCS-2011-0426.

[14] M. J. Dworkin. "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions." In: *Journal of Object Technology*. 2015.

[15] B. Ferreira. "Privacy-preserving efficient searchable encryption." Doctoral dissertation. Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa, 2016.

[16] B. Ferreira, J. Leitao, and H. Domingos. "MuSE: Multimodal Searchable Encryption for Cloud Applications." In: *2018 IEEE 37th Symposium on Reliable Distributed Systems (SRDS)*. IEEE. 2018, pp. 181–190.

[17] C. Gentry. "Fully homomorphic encryption using ideal lattices." In: *Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009, Bethesda, MD, USA, May 31 - June 2, 2009*. 200, pp. 169–178. DOI: 10.1145/1536414.1536440. URL: http://doi.acm.org/10.1145/1536414.1536440.

[18] C. Gentry, S. Halevi, and N. P. Smart. "Homomorphic evaluation of the AES circuit." In: *Advances in Cryptology–CRYPTO 2012*. Springer, 2012, pp. 850–867.

[19] E.-J. Goh et al. "Secure indexes." In: *IACR Cryptology ePrint Archive* 2003 (2003), p. 216.

[20] O. Goldreich and R. Ostrovsky. "Software Protection and Simulation on Oblivious RAMs." In: *J. ACM* 43.3 (1996), pp. 431–473. DOI: 10.1145/233551.233553. URL: http://doi.acm.org/10.1145/233551.233553.

[21] D. Goodin. *Millions of high-security crypto keys crippled by newly discovered flaw*. 2017. URL: https://arstechnica.com/information-technology/2017/10/crypto-failure-cripples-millions-of-high-security-keys-750k-estonian-ids/.

[22] F. Hao, X. Yi, and E. Bertino. "Editorial of special issue on security and privacy in cloud computing." In: *J. Inf. Sec. Appl.* 27-28 (2016), pp. 1–2. DOI: 10.1016/j.jisa.2016.04.003. URL: https://doi.org/10.1016/j.jisa.2016.04.003.

[23] *How Rainbow Tables work*. URL: http://kestas.kuliukas.com/RainbowTables/ (visited on 01/16/2018).

[24] *intel sgx for dummies*. 2015. URL: https://software.intel.com/en-us/blogs/2013/09/26/protecting-application-secrets-with-intel-sgx.

[25] B. Ives, K. R. Walsh, and H. Schneider. "The domino effect of password reuse." In: *Communications of the ACM* 47.4 (2004), pp. 75–78.

[26] N.-S. Jho, K.-Y. Chang, D. Hong, and C. Seo. "Symmetric searchable encryption with efficient range query using multi-layered linked chains." In: *The Journal of Supercomputing* 72.11 (2016), pp. 4233–4246.

[27] M. Kalenderi, D. N. Pnevmatikatos, I. Papaefstathiou, and C. Manifavas. "Breaking the GSM A5/1 cryptography algorithm with rainbow tables and high-end FPGAS." In: *22nd International Conference on Field Programmable Logic and Applications (FPL), Oslo, Norway, August 29-31, 2012*. 2012, pp. 747–753. DOI: 10.1109/FPL.2012.6339146. URL: https://doi.org/10.1109/FPL.2012.6339146.

[28] L. R. Knudsen and M. Robshaw. *The Block Cipher Companion*. Information Security and Cryptography. Springer, 2011. ISBN: 978-3-642-17341-7. DOI: 10.1007/978-3-642-17342-4. URL: https://doi.org/10.1007/978-3-642-17342-4.

[29] K. Krawiecka, A. Kurnikov, A. Paverd, M. Mannan, and N Asokan. "SafeKeeper: Protecting Web Passwords using Trusted Execution Environments." In: *The Web Conference (WWW). https://doi. org/10.1145/3178876.3186101*. 2018.

[30] S. Larson. *Every single Yahoo account was hacked - 3 billion in all*. 2017. URL: http://money.cnn.com/2017/10/03/technology/business/yahoo-breach-3-billion-accounts/index.html.

[31] Z. Li, W. He, D. Akhawe, and D. Song. "The Emperor's New Password Manager: Security Analysis of Web-based Password Managers." In: *USENIX Security Symposium*. 2014, pp. 465–479.

[32] *List of Rainbow Tables*. 2007. URL: http://project-rainbowcrack.com/table.htm (visited on 01/15/2018).

[33] A. Ltd. *TrustZone – Arm*. URL: https://www.arm.com/products/security-on-arm/trustzone.

[34] P. Maene, J. Gotzfried, R. De Clercq, T. Muller, F. Freiling, and I. Verbauwhede. "Hardware-Based Trusted Computing Architectures for Isolation and Attestation." In: *IEEE Transactions on Computers* (2017).

[35] K. Malvoni and J. Knezovi. "Are your passwords safe: Energy-efficient bcrypt cracking with low-cost parallel hardware." In: *WOOT'14 8th Usenix Workshop on Offensive Technologies Proceedings 23rd USENIX Security Symposium*.

[36] A. V. Meier. "The ElGamal Cryptosystem." In: 2005.

[37] P. Mell, T. Grance, et al. "The NIST definition of cloud computing." In: (2011).

[38] D. Mirante and J. Cappos. "Understanding Password Database Compromises." In: 2013.

[39] K. M. Moriarty, B. Kaliski, and A. Rusch. "PKCS 5: Password-Based Cryptography Specification Version 2.1." In: *RFC* 8018 (2017), pp. 1–40. DOI: 10.17487/RFC8018. URL: https://doi.org/10.17487/RFC8018.

[40] L. Morris. "Analysis of Partially and Fully Homomorphic Encryption." In: Rochester Institute of Technology, Rochester, New York, 2013.

[41] P. Paillier. "Public-Key Cryptosystems Based on Composite Degree Residuosity Classes." In: *Advances in Cryptology - EUROCRYPT '99, International Conference on the Theory and Application of Cryptographic Techniques, Prague, Czech Republic, May 2-6, 1999, Proceeding*. 1999, pp. 223–238. DOI: 10.1007/3-540-48910-X_16. URL: https://doi.org/10.1007/3-540-48910-X_16.

[42] O. Pandey and Y. Rouselakis. "Property preserving symmetric encryption." In: *Proceedings of the 31st Annual international conference on Theory and Applications of Cryptographic Techniques*. Springer-Verlag. 2012, pp. 375–391.

[43] B. Pinkas and T. Sander. "Securing passwords against dictionary attacks." In: *Proceedings of the 9th ACM conference on Computer and communications security*. ACM. 2002, pp. 161–170.

[44] R. S. Y. X. Rakesh Agrawal Jerry Kiernan. "Order Preserving Encryption for Numeric Data." In: SIGMOD '04, 2013, pp. 563–574.

[45] R. L. Rivest, A. Shamir, and L. M. Adleman. "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems (Reprint)." In: *Commun. ACM* 26.1 (1983), pp. 96–99. DOI: 10.1145/357980.358017. URL: http://doi.acm.org/10.1145/357980.358017.

[46] M. I. Salam, W.-C. Yau, J.-J. Chin, S.-H. Heng, H.-C. Ling, R. C. Phan, G. S. Poh, S.-Y. Tan, and W.-S. Yap. "Implementation of searchable symmetric encryption for privacy-preserving keyword search on cloud storage." In: *Human-centric Computing and Information Sciences* 5.1 (2015), pp. 1–16.

[47] A. Shamir. "How to share a secret." In: *Communications of the ACM* 22.11 (1979), pp. 612–613.

[48] E. Shi, J. Bethencourt, T. H. Chan, D. Song, and A. Perrig. "Multi-dimensional range query over encrypted data." In: *Security and Privacy, 2007. SP'07. IEEE Symposium on*. IEEE. 2007, pp. 350–364.

[49] M. Snider and E. Weise. *500 million Yahoo accounts breached*. USA TODAY. URL: https://www.usatoday.com/story/tech/2016/09/22/report-yahoo-may-confirm-massive-data-breach/90824934/.

[50] D. X. Song, D. Wagner, and A. Perrig. "Practical techniques for searches on encrypted data." In: *Security and Privacy, 2000. S&P 2000. Proceedings. 2000 IEEE Symposium on*. IEEE. 2000, pp. 44–55.

[51] W. Stallings, L. Brown, M. D. Bauer, and A. K. Bhattacharjee. *Computer security: principles and practice*. Pearson Education, 2012.

[52] *Survey: Majority of Americans Reuse Passwords and Millennials Are the Biggest Culprits*. 2017. URL: https://www.secureauth.com/company/newsroom/secureauth-survey-majority-reuse-passwords.

[53] M. Szczys. *TPM crytography cracked*. 2010. URL: https://hackaday.com/2010/02/09/tpm-crytography-cracked/.

[54] *UNDERSTANDING RAINBOW TABLES*. 2016. URL: https://www.drchaos.com/understanding-rainbow-tables/ (visited on 01/15/2018).

[55] X. Wang and H. Yu. "How to Break." In:

[56] Y. Wang, J. Wang, and X. Chen. "Secure searchable encryption: a survey." In: *Journal of communications and information networks* 1.4 (2016), pp. 52–65.

[57] K.-P. Yee and K. Sitaker. "Passpet: convenient password management and phishing protection." In: *Proceedings of the second symposium on Usable privacy and security*. ACM. 2006, pp. 32–43.

[58] M. Yung, ed. *Advances in Cryptology - CRYPTO 2002, 22nd Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 2002, Proceedings*. Vol. 2442. Lecture Notes in Computer Science. Springer, 2002. ISBN: 3-540-44050-X. DOI: 10.1007/3-540-45708-9. URL: https://doi.org/10.1007/3-540-45708-9.

[59] L. Zhang, Y. Zheng, and R. Kantola. "A Review of Homomorphic Encryption and its Applications." In: *Proceedings of the 9th EAI International Conference on Mobile Multimedia Communications, MobiMedia 2016, Xi'an, China, June 18-20, 2016*. 2016, pp. 97–106. URL: http://dl.acm.org/citation.cfm?id=3021405.