ESCOLA
SUPERIOR
DE TECNOLOGIA
E GESTÃO
POLITÉCNICO
DO PORTO

M
MESTRADO
ENGENHARIA INFORMÁTICA

Secure Remote Storage of Logs with Search Capabilities

Rui Araújo

12/2019

Secure Remote Storage of Logs with Search Capabilities. Rui Araújo
12/2019
Secure Remote Storage of Logs
with Search Capabilities
Rui Araújo
M
MESTRADO
ENGENHARIA INFORMÁTICA
ESCOLA
SUPERIOR
DE TECNOLOGIA
E GESTÃO
POLITÉCNICO
DO PORTO

ESCOLA
SUPERIOR
DE TECNOLOGIA
E GESTÃO
POLITÉCNICO
DO PORTO

M

MESTRADO
ENGENHARIA INFORMÁTICA

Secure Remote Storage of Logs with Search Capabilities

Rui Manuel Vieira Araújo

Advisor: António Alberto dos Santos Pinto

Acknowledgments

The conclusion and delivery of this project marks the end of a learning journey, in which several people and institutions are worthy of mention.

First, I would like to acknowledge ESTG for the promotion of this masters degree and consequent opportunity to broaden my knowledge and qualifications.

The recognition of my advisor, Prof. Dr. António Pinto, deserves equal prominence due to his guidance and availability to clarify my doubts and review my work, always in the most immediate and careful manner possible.

To my family, I'm grateful for never conditioning my choices and always supporting my decisions.

Lastly, I must recognise the patience and encouraging attitude in the hardest times of my girlfriend.

Abstract

Along side with the use of cloud-based services, infrastructure and storage, the use of application logs in business critical applications is a standard practice nowadays. Such application logs must be stored in an accessible manner in order to used whenever needed. The debugging of these applications is a common situation where such access is required. Frequently, part of the information contained in logs records is sensitive.

This work proposes a new approach of storing critical logs in a cloud-based storage recurring to searchable encryption, inverted indexing and hash chaining techniques to achieve, in a unified way, the needed privacy, integrity and authenticity while maintaining server side searching capabilities by the logs owner.

The designed search algorithm enables conjunctive keywords queries plus a fine-grained search supported by field searching and nested queries, which are essential in the referred use case. To the best of our knowledge, the proposed solution is also the first to introduce a query language that enables complex conjunctive keywords and a fine-grained search backed by field searching and sub queries.

Keywords: Logging, Cryptography, Searchable Encryption, Privacy, Confidentiality, Integrity, Authenticity

Resumo

A geração de *logs* em aplicações e a sua posterior consulta são fulcrais para o funcionamento de qualquer negócio ou empresa. Estes *logs* podem ser usados para eventuais ações de auditoria, uma vez que estabelecem uma *baseline* das operações realizadas. Servem igualmente o propósito de identificar erros, facilitar ações de *debugging* e diagnosticar *bottlennecks* de performance. Tipicamente, a maioria da informação contida nesses *logs* é considerada sensível.

Quando estes *logs* são armazenados *in-house*, as considerações relacionadas com anonimização, confidencialidade e integridade são geralmente descartadas. Contudo, com o advento das plataformas *cloud* e a transição quer das aplicações quer dos seus *logs* para estes ecossistemas, processos de *logging* remotos, seguros e confidenciais surgem como um novo desafio. Adicionalmente, regulação como a RGPD, impõe que as instituições e empresas garantam o armazenamento seguro dos dados.

A forma mais comum de garantir a confidencialidade consiste na utilização de técnicas criptográficas para cifrar a totalidade dos dados anteriormente à sua transferência para o servidor remoto. Caso sejam necessárias capacidades de pesquisa, a abordagem mais simples é a transferência de todos os dados cifrados para o lado do cliente, que procederá à sua decifra e pesquisa sobre os dados decifrados. Embora esta abordagem garanta a confidencialidade e privacidade dos dados, rapidamente se torna impraticável com o crescimento normal dos registos de *log*. Adicionalmente, esta abordagem não faz uso do potencial total que a *cloud* tem para oferecer.

Com base nesta temática, esta tese propõe o desenvolvimento de uma solução de armazenamento de *logs* operacionais de forma confidencial, integra e autêntica, fazendo uso das capacidades de armazenamento e computação das plataformas *cloud*. Adicionalmente, a possibilidade de pesquisa sobre os dados é mantida. Essa pesquisa é realizada *server-side* diretamente sobre os dados cifrados e sem acesso em momento algum a dados não cifrados por parte do servidor.

Para tal, o uso de *searchable encryption* é considerado. Esta técnica, que tem ganho nos últimos anos uma atenção especial por parte de diversos investigadores devido ao ênfase atual dado à privacidade de dados, garante a capacidade de pesquisa remota efetuada pelo servidor sobre os dados cifrados sem a necessidade de os decifrar em nenhum momento. Em suma, o uso de *searchable encryption* impulsiona a realização de pesquisas cifradas sobre dados cifrados, que retornem resultados igualmente cifrados.

De forma a garantir uma melhor performance e escalabilidade na pesquisa, um esquema de *searchable encryption* baseado em *inverted indexing* é empregue. *Inverted indexing* consiste numa estrutura de dados que mapeia, para este caso concreto, os termos de pesquisa com as linhas de *log* onde estes existem, permitindo a realização de pesquisas mais rápidas.

A solução proposta visa assim a operacionalização do armazenamento seguro de logs, assumindo um modelo baseado em três entidades distintas. O *data owner*, cujos *logs* são armazenados de forma segura no servidor de armazenamento remoto, o *data user*, autorizado a pesquisar sobre os dados armazenados e o servidor de armazenamento remoto, encarregue de armazenar os registos de *log* e devolvê-los em subsequentes operações de pesquisa.

Relativamente à arquitetura da solução, esta compreende seis componentes distintos. O *Conf Manager* (C1), responsável pela disponibilização de todas as configurações da solução, incluindo as chaves criptográficas necessárias. O *Encryption* (C2), necessário para operações de cifra e decifra. O *Indexing* (C3), responsável pela transformação de registos de *log* em registos cifrados e pesquisáveis. O *Search* (C4), para realização de pesquisas sobre esses *logs*. O *Internal Connection* (C5), empregue na comunicação com as aplicações cujos registos de *log* serão armazenados. O *External Connection* (C6), para a criação de uma ponte entre o lado do cliente e o lado do servidor e para a disponibilização de uma camada de abstração sobre a complexidade de integração com os diversos *cloud providers*.

A confidencialidade, integridade e autenticidade dos dados são asseguradas não apenas no seu armazenamento mas também durante a sua transmissão. Assim sendo, dois protocolos de comunicação são especificados. O primeiro é aplicado na conexão entre as aplicações que produzem os registos de *log* e a solução proposta, garantindo, através de criptografia simétrica e assimétrica, o estabelecimento de um canal cifrado entre as partes, devidamente autenticado e autorizado. A integridade da informação enviada nesse canal é também garantida. O segundo protocolo surge na comunicação entre a solução proposta e o *cloud provider*, garantido as mesmas propriedades de segurança.

A confidencialidade dos registos de *log at rest* é certificada no momento da sua indexação através do uso de criptografia simétrica. Cada registo de *log* é cifrado com uma chave secreta, apenas válida para cada registo individual. A integridade dos dados é garantida pela aplicação da técnica de *hash chaining*, na qual o *hash* criptográfico da linha anterior é usado para calcular o *hash* criptográfico da linha atual. Este método não só garante a integridade dos dados como também assegura a correta ordem das linhas de *log*.

A capacidade de pesquisa sobre os registos de *log* é realizado pelo cliente igualmente no momento de indexação. Após uma extração dos termos relevantes, estes são transformados em *trapdoors* de pesquisa e enviados para o servidor remoto de armazenamento de forma cifrada, garantindo a confidencialidade e privacidade das pesquisas. A solução proposta oferece também suporte a tipos de pesquisa mais avançados, como *field searching* e *nested queries*. De forma a garantir uma melhor

experiência de utilização, é definida uma linguagem de *query*, o mais aproximada à linguagem natural humana possível, que garante uma escrita mais simples e direta de todo o tipo de *queries* suportadas pelo motor de pesquisa desenvolvido.

Palavras Chave: *Logging*, Criptografia, *Searchable Encryption*, Privacidade, Confidencialidade, Integridade, Autenticidade

Contents

List of Figures

List of Tables

List of Algorithms

Listings

Acronyms

ABE Attribute Based Encryption

AE Authenticated Encryption

AES Advanced Encryption Standard

API Application Programming Interface

BF Bloom Filter

CKA Chosen Keyword Attack

CTR Counter Mode

DES Data Encryption Standard

EEA European Economic Area

EU European Union

FIPS Federal Information Processing Standard

GCM Galois/Counter Mode

GDPR General Data Protection Regulation

HMAC Hash-based Message Authentication Code

HVE Hidden Vector Encryption

IBE Identity Based Encryption

IDF Inverse Document Frequency

IETF Internet Engineering Task Force

IV Initialisation Vector

KGA Keyword Guessing Attack

LSH Locality Sensitive Hashing

MAC Message Authentication Code

MD5 Message Digest 5

MIT Massachusetts Institute of Technology

NIST National Institute of Standards and Technology

NSA National Security Agency

OAEP Optimal Asymmetric Encryption Padding

PEKS Public Key Encryption With Keyword Search

PIR Private Information Retrieval

PKE Public Key Encryption

PKG Private Key Generator

PRNG Pseudorandom Number Generator

PSS Probabilistic Signature Scheme

RSA Rivest Shamir Adleman

SE Searchable Encryption

SHA Secure Hash Algorithm

SLaS Secure Logging as a Service

SSL Secure Socket Layer

SLS Secure Logging Service

SSE Symmetric Searchable Encryption

TF-IDF Term Frequency - Inverse Document Frequency

TF Term Frequency

TLS Transport Layer Security

US United States

Chapter 1

Introduction

Business critical applications require monitoring. A frequent pillar of application monitoring is the use of logs. These produce a time-stamped recording of events relevant to a particular system and establish a baseline of standard operations for future audit reference, to identify erroneous operations, to diagnose performance bottlenecks, to facilitate application debugging, among other tasks. Frequently, part of the information contained in logs records is sensitive. On one hand, when considering on-premises deployment of logging solutions, considerations related to anonymity, confidentiality and integrity of log records may not be addressed. On the other hand, with the advent of cloud platforms that both house the applications and their logs, secure and confidential remote logging appears as a crucial issue to address.

Depending on the commercial and trust relations established between a client and a remote log service provider, distinct forms of log storage can be envisioned. If it's the case of storing anonymous logs, these may be stored without additional processing. However, if it's the case of storing application or server related logs that might contain sensitive information on them, these may require encryption to guaranty confidentiality. The log encryption may be performed at the user's premises or at the premises of the service provider. Additional guarantees, such as integrity or search capability may also be required. Moreover, if some user related information is comprised in such logs, additional measures are imposed by regulations such as the General Data Protection Regulation (GDPR)[1]. Finally, a business may wish to deploy its applications with one cloud provider and store the operational logs of those applications in a distinct cloud provider.

When remote log confidentiality is required, the most common solution is to use cryptography techniques to encrypt all data before transferring it to a remote cloud storage service. In some particular cases, the data sent can also be digitally signed to ensure its source trustworthiness and a crypto-

[1] European Union Regulation 2016/679, created in 2018 and applicable for all individual citizens of the European Union (EU) and the European Economic Area (EEA)

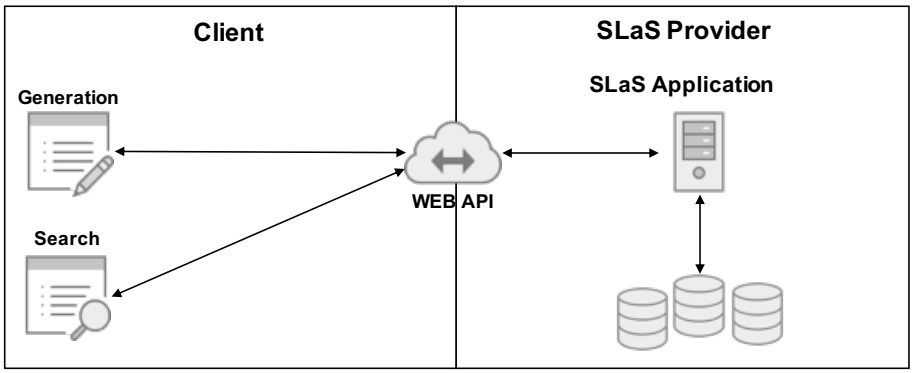

Figure 1.1: Reference scenario

graphic hash can also be computed to assure data integrity and to prevent its manipulation while in transit. If searching within this remotely stored logs is required, the simplest and trivial approach consists in transferring all data back to the client, so that it can be decrypted, allowing search operations to be performed over the clear text and at the client side. Despite the data privacy and confidentiality guarantees offered by this approach and, possible, data integrity when combined with digital signature and hash validation techniques, it can rapidly become impractical with the normal growth of log data. It will also have a negative impact in the latency and performance of the client-side operations since every time a search operation is performed all the log data is transferred to the client. Moreover, this approach does not make use of the full potential of cloud computing.

1.1 Reference Scenario

The reference scenario adopted in this thesis is presented in Figure 1.1. It describes a cloud-based secure log storage service. The Client makes use of a web-based Application Programming Interface (API) to transfer its encrypted operational logs to the cloud, named Secure Logging as a Service (SLaS) Provider. In parallel, the Client can make use of the same WEB API to query, in a encrypted form, its logs and retrieve matching records. A SLaS application is foreseen in order to make use of the cloud's potential and to enable the transfer of some CPU-intensive tasks from the Client to the SLaS Provider. The SLaS Provider stores the client-side encrypted logs in its database and performs search operations on them, without having access to clear text logs.

1.2 Objective

The main objective of this work it to develop a secure, confidential and off-premises storage solution for operational logs that is also capable of assuring the integrity and authenticity of the stored data while

supporting server-side log searching and retrieval.

1.3 Expected Outcomes

At the end of this work, the following results are expected:

1. **Secure Remote Storage Solution**. A cloud-based server solution that provides a confidential storage and retrieval of encrypted log records. The integrity and authenticity of the data shall also be assured by the use of secure hash functions.
2. **Secure Search Algorithm**. A search algorithm shall be made available to the client so that he can submit queries and retrieve matching log records. The queries should be submitted in an encrypted form, performed over encrypted data and return encrypted matches. The queries shall also support complex conjunctive keywords and a fine-grained search backed by field searching and sub queries.
3. **Query Language**. A query language shall be defined in order to enable the construction of any query supported by the search algorithm. The language should be as close to natural language as possible in order to support seamless query definition.

1.4 Organisation of the Document

The document is organised in chapters. Chapter 2 details the concept of cryptography, from the definition to its main applications. It also presents the definition of Searchable Encryption and an overview of its evolution. Chapter 3 presents all relevant work from other researchers that is related to the one discussed herein. Chapter 4 details the proposed solution, its architecture and main features. Chapter 5 is dedicated to the validation of the proposed solution and comprises the performed tests and the validation of the prototype. It also includes a comparison of the proposed solution with the related work and a security analysis. Chapter 6 reviews the performed work, makes global considerations, characterises the results and discusses future work.

Chapter 2

Cryptography and Searchable Encryption

This chapter aims to provide an understanding of searchable encryption. It starts by a definition of cryptography and details its evolution trough time. A description of symmetric and asymmetric encryption, hash functions and authenticated encryption are also included. After, searchable encryption is explained, including an overview of the multiple forms of searchable encryption and a listing of the relevant and groundbreaking achievements in this field of research.

2.1 Cryptography

Secure communication between two parties or private storage are two of many scenarios where one might have a need to make their data unintelligible to third parties. The technique that makes it possible is known as cryptography. The term derives for the ancient Greek words, *kryptos*, that translates to "secret or hidden" and *graphein*, which means "to write". Cryptography is closely associated to encryption, which is a reversible transformation process of intelligible data into unintelligible data, named ciphertext. Decryption allows the transformation of ciphertext back to the original intelligible data, named clear text. However, in information security, alongside data confidentiality, cryptography allows for data integrity, assuring that information was not altered, through the use of hash algorithms. Non-repudiation or the authentication of the involved parts can also be assured by the use of digital signatures [1].

Cryptography can be traced back to the ancients, with the first documented use dating to 1900 BC in ancient Egypt with substituted hieroglyphics carved in stone [2]. However, the most famous classic cipher came from the Roman emperor Julius Caesar in 100 BC. His scheme encrypts a message by applying a substitution cipher where each of the letters is shifted three positions down in the alphabet.

Although this approach is considered very weak for the modern standards, since it can be rapidly broken by applying a reverse shift of three places for each letter, it was highly used by the Roman troops during the dominance of the Roman empire [3].

A meaningful improvement over the Caesar cipher was presented 1500 years later, in the 16th century. The Vigenère cipher is similar to the Caesar cipher, except that letters are not shifted by a fixed number of places rather by values defined by a key. This key, a collection of letters that represent the number of shifts to do based on their position in the alphabet, is considered one of the first cryptographic keys. Although this cipher is clearly more secure than the Caesar cipher, it is also trivial to be broken by the use of frequency analysis, which exploits the uneven distribution of the letters in language and the deterministic property of this cipher. This is, for the same key, the same clear text always encrypt to the same ciphertext. Nevertheless, the Vigenère cipher became popular and was even used during World War I [4].

In the early 20th century, cryptography has benefited from the technological advances, with the advent of the rotating disc mechanisms. This mechanisms formed the basis of the famous German Enigma machine, invented by Arthur Scherbius [1]. It was debuted at the end of World War I and was heavily relied upon by the German military, during World War II, to assure confidential and secret communications between all of their military branches. In the midst of war, the British faced the challenge of deciphering the Enigma code and understanding the inner workings of the Enigma machine. To find a method to decrypt the Enigma communications, the British devoted, for many years, a considerable amount of resources, which culminated on the breakdown of Enigma, which helped the Allied troops to achieve victory and, consequently, end World War II.

Modern cryptography shifted from the linguistic and lexicographic patterns and is now heavily based on mathematical theory and computer science practice. The recent cryptographic algorithms are designed around computational hardness assumptions, such as the integer factorisation or the discrete logarithm problem. These are problems that are easy to state but hard to solve. Algorithms are considered computationally secure even if they are theoretically possible to break by any adversary but unfeasible to do so by any known practical means in useful time. Regardless, the faster computing capacity require these solutions to be continually adapted. The most common adjustment made is the growth in the length of the used keys, ensuring that even a brute force attack can't break the cipher in useful time [5].

The modern field of cryptography can be divided essentially into two different types of cryptography. Symmetric cryptography, the only one available until the 70's, that is based on key sharing between both involved parties and on an encryption algorithm that uses the shared key to transform intelligible data into unintelligible data and vice-versa. These require the use of a secure channel to distribute the key and assume that the encryption/decryption algorithm is known by both parties. Two major drawbacks

unfold from the use of symmetric cryptography. Key disclosure to third parties compromises all communications, both parties are equal and the receivers are able to forge messages claiming it came from the sender. Asymmetric cryptography arises as a different approach and uses cryptographic algorithms that require a pair of two related keys. Asymmetric cryptography might suggest better security or appear as a replacement of symmetric encryption, however this is not true. The security of an asymmetric encryption scheme depends essentially in the size of the key used. On the other hand, symmetric encryption is much faster than asymmetric encryption. For these reasons, asymmetric cryptography is used as a complement to symmetric cryptography.

Cryptography makes intensive use of random numbers, which are used for instance in key generation. Secure random numbers must be uniformly distributed and unpredictable [4]. That is, the number generation must be regular, and the future generated number sequences must not be inferred from previously generated numbers. A good solution is to base random number generation in the natural randomness of the real world by monitoring continuous random events such as radio or audio noise. Other solutions are based on software named Pseudorandom Number Generator (PRNG) that generate almost random numbers. In fact, these numbers are not truly random, but can pass randomness testing.

2.1.1 Symmetric Cryptography

Symmetric cryptography refers to an encryption method in which both parts involved share a secret key that is used for both encryption and decryption operations. A symmetric key scheme can be implemented either as a block cipher or a stream cipher [6, 7].

A block cipher operates over a group of bits of fixed length, called blocks, and consists on two paired algorithms. The encryption algorithm takes as input a key and a clear text block and produces a ciphertext block. The decryption algorithm is the inverse and decrypts a ciphertext back to the original clear text. A block cipher consists of a repetition of rounds of basic transformations that are simple to specify and to implement. When these transformations are iterated several times it becomes hard to reverse without being in the possession of the right key.

A block cipher is characterised by the block size and the key size, on which the security of the algorithm depends on. Blocks must not be too large in order to minimise both the length of the ciphertext and the computation overhead. However, they should not be too small, otherwise they are susceptible to codebook attacks. Codebook attacks are performed by building a lookup table that maps each ciphertext block to its corresponding clear text block, allowing the decryption of an unknown ciphertext block by simply looking in the table for the corresponding clear text block.

A stream cipher works by generating pseudorandom bits from the secret key, creating a keystream. This keystream is used to encrypt the clear text by the computing a XOR[1] operation between the

[1] Exclusive or, a logical operation that outputs true only when the inputs differ, one is true, the other is false

keystream bits and the clear text bits.

From a high level perspective, it is possible to point out two types of stream ciphers. Stateful stream ciphers, that include a secret initial state that initialises from the key and a nonce and evolves throughout the keystream generation. Counter-based stream ciphers produces chunks of the keystream from a key, a nonce and a counter value, without the need to memorise a secret state during the generation of the keystream.

AES

The Advanced Encryption Standard (AES) [8] represents the state-of-the-art regarding symmetric cryptography, being the most used algorithm of this class. Its origin traces back to the end of the 20^{th} century, when National Institute of Standards and Technology (NIST) opened a competition to find an unclassified, publicly disclosed encryption algorithm capable of protecting sensitive government information well into the next century. A new standard was primarily needed because Data Encryption Standard (DES), the standard cipher in use on that date, used a relatively small 56 bit size key and was possibly vulnerable to brute force attacks. In addition, the DES algorithm was designed to be fast on integrated circuits, not on mainstream CPUs. Even the 3DES, a variant of the algorithm that improves on the used key size, remained slow and its unsuitable for platforms with limited resources. In October 2000, NIST announced that the Rijndael algorithm [9], proposed by Vincent Rijmen and Joan Daemen, was selected as the new standard, becoming the world's leading encryption scheme. Nowadays, is supported on most of the cryptographic commercial solutions and approved by the National Security Agency (NSA) for protecting secret and sensitive information.

AES processes block of 128 bits using secret keys with 128, 192 or 256 bits of size, based on a design principle know as a substitution-permutation network. It comprises a series of linked operations, some of which involve replacing inputs by specific outputs and others involve shuffling bits around. AES treats the 128 bits of a clear text block as 16 bytes, which are arranged in four columns and four rows for processing as a two dimensional array. The number of rounds that the algorithm applies is variable and depends on the length of the key. It uses 10 rounds for 128 bit keys, 12 rounds for 192 bit keys and 14 rounds for 256 bit keys. Each of the rounds uses a different 128 bit key, which is calculated from the original AES key.

Each round of the AES encryption process comprises four operations. The AddRoundKey operation applies a XOR operation between the 128 bits of the round key and the 128 bits of the cipher state. The SubBytes operation performs a substitution where each byte is replaced with another according to a lookup table (Rijndael S-box). The ShiftRow is a transposition step where the last three rows of the state are shifted in a cyclic manner. The MixColumns executes a linear mixing operation which operates on the columns of the state, combining the four bytes in each column and outputs four completely new

bytes which replace the original column. In order to decrypt a ciphertext, AES unwinds each operation by applying its inverse operation. The inverse lookup table of SubBytes reverses the SubBytes transformation, ShiftRow shifts in the opposite direction, MixColumns's inverse is applied and AddRoundKey XOR is unchanged because the inverse of an XOR is the same operation.

2.1.2 Asymmetric Cryptography

Asymmetric cryptography, also referred to as public-key cryptography, is a scheme in which two related keys are used. A public key, used for encryption that can be made publicly available and a private key used for decryption that should always remain secret. The public key can be computed from the private key, but the inverse operation cannot be performed, since this computation is easy to perform in one direction but unfeasible to invert. This property forms the basis of asymmetric cryptography [10]. Combined with the confidentiality offered by encryption, asymmetric cryptography can also assure authentication, non-repudiation and integrity. This three properties can be achieved with the use of digital signatures.

To produce a digital signature, usually a cryptographic hash or checksum of the original data is computed. Then the result of that computation is digitally signed with the private key, making it verifiable with the corresponding public key. Assuming the private key remains secret and the entity it is issued is the only holder with access to it, a digital signature can provide authentication since the recipients can trust that the entity was the one who actually applied the signature. Non-repudiation can be assured since the entity is the only with access to the private key used in the signature, that entity cannot later discard the signature authorship. Integrity can be guaranteed when the signature is verified. The context of the message is checked to match what was in the original input of the signature, making illicit changes on the message noticeable.

Asymmetric algorithms are often based on the computational complexity theory, a field of science that appeared on the 70's and is dedicated to the study of hard problems. The concept of a hard problem can be described as an operation with a high level of difficulty that require significant resources to solve and usually cannot be achieved in useful time. Examples of such hard problems are the integer factorisation and the discrete logarithm problems, both used in well known public key algorithms [11].

The difficulty of the underlying problems increase with the length of the numbers used as input to the referred hard problems. Typically asymmetric cryptography algorithms make use of larger key sizes then the ones used in symmetric cryptography. As a consequence, asymmetric key algorithms usually are slower and produce larger ciphertexts. For practicality, an hybrid approach is commonly used. On a hybrid scheme, a symmetric key algorithm, usually faster and with greater performance, is used for the encryption of the message itself and the asymmetric key algorithm is used to encrypt the used symmetric key, that is sent alongside the message ciphertext. With this approach, it is possible

to combine the advantages and minimise the drawbacks of both types of cryptography. It is feasible to have a fast scheme that produces small ciphertexts thanks to the use of symmetric cryptography with a simple and secure key exchange system by the application of an asymmetric key algorithm.

RSA

The Rivest Shamir Adleman (RSA) algorithm [12] was publicly described in 1977 by Ron Rivest, Adi Shamir, and Leonard Adleman at the Massachusetts Institute of Technology (MIT), representing the breakthrough of asymmetric cryptography as the first public key scheme. Its security derives from the difficulty of factorising large integers that are the product of two large prime numbers. This difficulty is the basis of the one-way RSA core function, which is easy to compute in one direction, but almost impossible to compute in reverse. That impossibility hails from the ease of multiplying two numbers and the difficulty of determining the original prime numbers. Therefore, RSA is secure while the discovery of such numbers is unfeasible and not achievable in useful time, regardless of the computing power available. Additionally, encryption strength is directly tied to key size. If the size is doubled, the strength of the algorithm increases exponentially. RSA keys are typically of 2048 or 4096 bits. In addition to encryption, RSA is also used to build digital signatures, where the owner of the private key is the only one able to sign a message and the public key enables anyone to verify the signature's validity.

The RSA key generation function starts by the choice of two distinct random large prime numbers, named p and q. These numbers should be similar in magnitude but differ in length to make factoring harder and consequently delay brute-force attacks. Afterwards, the modulus n is computed as $n = pq$. Once n is obtained, the Carmichael's totient function is applied like $\phi(n) = lcm(p-1, q-1)$, where lcm means the lowest common multiple, the lowest number that both p and q can divide into. Finally, a number e is chosen such that $1 < e < \phi(n)$ and $gcd(e, \phi(n) = 1$, that is e and $\phi(n)$ are co-prime, since the only positive integer that divides both of the numbers is 1, gcd stands for greatest common divider, the largest positive integer that divides both the numbers. Since the public key is shared openly, e is generally 65 537. Finally, the modulo multiplicative inverse, d of e modulo $\phi(n)$ is determined. The public key consists of the modulo n and the public exponent e. The private key consists of the private exponent d. The elements p, q and $\phi(n)$ can be discarded after the key generation operation is done. The ciphertext c is computed by $c = m^e (mod\ n)$, this is, ciphertext c is equal to m raised to the power of e and then reduced modulus n. The decryption of ciphertext c is computed by $m = c^d (mod\ n)$, the original clear text m can be recovered by the raising c to the power of d and then reduced modulus n.

The original RSA encryption is deterministic, this is for the same key pair the same clear text always produce the same ciphertext. This property makes the algorithm vulnerable to a chosen clear text attack, where one can obtain arbitrary ciphertexts from known clear texts and test if they are equal to previously obtained ones. By this comparison an attacker can deduce the original data without deciphering it. To

avoid this problem and make the algorithm semantically secure, random data, named padding, is added to the ciphertext. This padding assures that two ciphertexts produced from the same clear text are indistinguishable. The standard in use is the Optimal Asymmetric Encryption Padding (OAEP), which involves the creation of a bit string as large as the modulo n and by padding the message with extra random data before applying the RSA function. OAEP uses a pseudorandom number generator (PRNG) to ensure the indistinguishability of ciphertexts by making the encryption probabilistic, being secure as long as the PRNG and RSA functions are secure. The padding algorithms are also incorporated on signature schemes, being the Probabilistic Signature Scheme (PSS) the standard in use to provide additional security for RSA signatures.

2.1.3 Hash Functions

Cryptographic hash functions are operations that map a dynamic sized input to a fixed size output through mathematical properties, called the hash value or digest. These functions are one-way functions and unfeasible to reverse. The algorithms behind them must have two crucial properties. First, they need to be fast on the hash value computation and second they must be deterministic, this is, for any given input, the hash function must always produce the same output.

The notion of security for hash functions is different from encryption. Whereas ciphers protect data confidentiality in an effort to guarantee that data sent in the clear can not be read, hash functions protect data integrity in an effort to guarantee that data has not been modified. For an hash function to be secure, it should comply with pre-image resistance, as it should be computationally hard to reverse an hash function. Also, it is necessary to assure second pre-image resistance, which means that given an input and its hash, it should be hard to find a different input with the same hash. This is also known as collision resistance. Additionally, the strength of hash functions stems from the unpredictability of their outputs, which should also not reveal any information about the input [13].

SHA

The Secure Hash Algorithm (SHA) [14] is a family of cryptographic hash functions published by NIST as a Federal Information Processing Standard (FIPS) for use by non-military federal government agencies in the United States (US). It is considered worldwide as the standard replacement of Message Digest 5 (MD5) algorithm, proven not to be secure against collision attacks. The first proposed scheme, SHA-0, published in 1993 under the name ”SHA”, was withdrawn shortly after publication due to an unidentified security issue and replaced by the slightly revised version SHA-1. SHA-1 was published in 1995 in FIPS PUB 180-1 [15], producing 160-bit hash values. Since 2005 SHA-1 has not been considered secure against collision attacks and since 2010 many organisations have recommended its replacement by SHA-2 or SHA-3 [16].

SHA-2 [17], the successor to SHA-1, was designed by the NSA and standardised by NIST. SHA-2 consists of a family of four hash functions: SHA-224, SHA-256, SHA-384, and SHA-512, of which SHA-256 and SHA-512 are the two main algorithms. The three-digit numbers represent the bit lengths of the output hash value. The initial motivation behind the development of SHA-2 was to generate longer hashes and thus deliver higher security when compared to SHA-1. For example, whereas SHA-1 has 160-bit output values, SHA-256 has 256-bit output values. Both SHA-1 and SHA-256 have 512-bit message blocks. However, whereas SHA-1 makes 80 rounds, SHA-256 makes 64 rounds, expanding the 16-word message block to a 64-word message block. The SHA-2 family also includes SHA-224, which is identical to SHA-256 except that its hash value length is 224 bits, instead of 256 bits, and is taken as the first 224 bits of the final hash value. SHA-512 is similar to SHA-256 except that it works with 64-bit words instead of 32-bit words. As a result, it uses 512-bit output values and makes 80 rounds instead of 64. SHA-384 is the same algorithm as the SHA-512 but uses a different initial value and a the final hash is truncated to 384 bits.

Despite no significant attack on SHA-2 has been yet demonstrated, after practical attacks on MD5 and on SHA-1, NIST grew concerned about SHA-2's long-term security due to its similarity to SHA-1. Many even believed that attacks on SHA-2 were just a matter of time. As follow up, NIST announced, in 2007, the Hash Function Competition in order to create a new hash standard, that should be at least as secure and as fast as SHA-2. SHA-3 candidates inner workings should also not be similar to SHA-1 and SHA-2 in order to be immune to attacks that would break SHA-1 and potentially SHA-2. After an extensive analysis of the submissions, the Keccak algorithm [18] was selected as the winner of the competition, becoming the new hashing standard.

The Keccak algorithm is based on a sponge construction, a mode of operation, based on a fixed-length permutation and on a padding rule, which builds a function that maps a variable-length input to a variable-length output. Keccak's core algorithm is a permutation of a 1600-bit state that ingests blocks of 1152, 1088, 832, or 576 bits, producing hash values of 224, 256, 384, or 512 bits, respectively, which are the same four lengths produced by SHA-2 hash functions. However, SHA-3 does not replace SHA-2, the two standards will complement each other and offer more options to designers of both hardware and software.

Keyed Hashing

A cryptographic hash function takes a message and returns its hash value. However, in some scenarios there might be the need to confirm that the message came from the stated sender and has not been changed. Keyed hashing assure this verification, through the use of a Message Authentication Code (MAC). A MAC protects a message's integrity and authenticity by creating a value $T = H(K, M)$, called the authentication tag of the message M, using the secret key K [4].

An Hash-based Message Authentication Code (HMAC) [19, 20] is a specific type of MACs used to simultaneously verify the integrity and authenticity of a message through the use of a cryptographic hash function and a secret key. The calculation of an HMAC may be obtained by the usage of any cryptographic hash function, being the most popular the SHA algorithms. The resulting HMAC algorithm is termed HMAC-X, where X is the hash function used, e.g. HMAC-SHA3. The cryptographic strength of the HMAC depends upon the cryptographic strength of the underlying hash algorithm, the size of its hash output, and the size and randomness of the key.

The construction of an HMAC is obtained by $T = H((K \oplus opad)||H((K \oplus ipad)||M))$, where H is the cryptographic hash function, M is the message to be authenticated and K is the secret key. The $||$ denotes concatenation and the $\oplus$ denotes the XOR operation. The term $opad$, outer padding, consists of repeated bytes valued $0x5c$ and the term $ipad$, inner padding, consists of repeated bytes valued $0x36$. The length of $opad$ and $ipad$ is determined by the block size used by the underlying cryptographic hash function.

The design of the HMAC specification was mostly motivated by the existence of attacks on more trivial and straightforward mechanisms for combining a key with a hash function. One of that techniques is called the secret-prefix construction, which turns a normal hash function into a keyed hash one by prepending the key to the message and returning $T = H(K||M)$. This approach can be insecure when the hash function is vulnerable to length-extension attacks. The alternative, the secret-suffix construction where the key is appended like $T = H(M||K)$ is also insecure, since if an attacker is able to find a collision in the underlying hash function, the same collision will occur after the concatenation with the secret key.

2.1.4 Authenticated Encryption

Authenticated Encryption (AE) [21] is a type of cryptographic technique that enables not only the protection of a message's confidentiality but also its authenticity. The most common approach used to build an authenticated encryption scheme is the combination of a symmetric key block cipher to encrypt the message with a MAC algorithm, to produce the authentication tag.

Three different approaches can be used to obtain an authenticated encryption algorithm. The three combinations differ in the order in which encryption is applied and the authentication tag is generated. The encrypt-and-MAC approach computes a ciphertext and a MAC tag separately and possibly in parallel. Given a clear text P, a ciphertext C is computed like $C = E(K1, P)$, where E is an encryption algorithm and $K1$ is the encryption key. The authentication tag T is calculated from the clear text as $T = MAC(K2, P)$. Both C and T are transmitted to the other part involved. The decryption of C to obtain P is performed through $P = D(K1, C)$. With P, the decrypted clear text, a new authentication tag can be computed and compared with the previously one obtained. If a match do not occur the

verification will fail and the message will be deemed invalid.

The MAC-then-encrypt composition protects a message, P, by first computing the authentication tag T, like $T = MAC(K2, P)$. Afterwards, the ciphertext is created by encrypting the clear text and tag together, according to $C = E(K1, P||T)$. Only C, which contains both the encrypted clear text and tag, is transmitted. Upon receipt, C is decrypted by computing $P||T = D(K1, C)$ to obtain both the clear text and the tag T. In order to confirm that the computed tag is equal to the tag T, a computation of a new tag directly from the clear text is performed according to $T = MAC(K2, P)$. Despite this approach might expose potentially corrupted clear texts to the involved parts, is more secure than encrypt-and-MAC because it hides the authentication tag, thus preventing it from leaking information on the clear text.

The encrypt-then-MAC construction sends two values. The ciphertext produced by $C = E(K1, P)$ and an authentication tag based on the ciphertext, $T = MAC(K2, C)$. Upon receipt, a new tag is computed by $T = MAC(K2, C)$ and verified against the one received. If the values are equal, the clear text is obtained by $P = D(K1, C)$, if they are not, the ciphertext C is discarded. An obvious advantage that comes from this method is that the receiving party only needs to compute a MAC in order to detect corrupt messages, meaning that there is no need to decrypt a corrupt ciphertext. This feature makes the encrypt-then-MAC approach stronger than the encrypt-and-MAC and MAC-then-encrypt schemes.

AES-GCM

The AES-GCM [22] is the most widely used authenticated encryption scheme, being, at the time of this writing, the only algorithm taken as standard by NIST, SP 800-38D. AES-GCM is also part of NSA's Suite B and of the Internet Engineering Task Force (IETF) for secure network protocols. It is based on the AES algorithm and the Galois/Counter Mode (GCM). The GCM is a block cipher mode of operation capable of providing authenticated encryption, achieved with high speeds with low cost and low latency both in hardware and software.

GCM is also capable of acting as a standalone MAC, authenticating messages with no data encryption. More importantly, it can be used as an incremental MAC. If an authentication tag is computed for a message and then part of that message is changed, an authentication tag can be computed for the new message with a computational cost proportional to the number of bits that were changed. Additionally, it accepts an Initialisation Vector (IV) of arbitrary length, which makes it easier for applications to meet the requirement that all IVs be distinct and that both encryption and decryption operations must be able to be run in parallel.

The GCM mode of operation is similar to the normal Counter Mode (CTR) with a tweak that incorporates a small and efficient component to compute an authentication tag. Just as in normal CTR, each block composed of a nonce N concatenated with a counter is XORed with a clear text block to obtain a ciphertext block. Afterwards, the ciphertext blocks are mixed using a combination of XORs

and multiplications. AES-GCM performs an encryption in CTR mode and a MAC over the ciphertext blocks. Therefore, AES-GCM is essentially an encrypt-then-MAC construction, where AES-CTR encrypts using a 128-bit key K and a 96-bit nonce N. To authenticate the ciphertext, GCM applies a XOR operation between the ciphertext and the output of a universal hash function called GHASH, as $T = GHASH(H, C) \oplus AES(K, N||0)$, where C is the ciphertext and H is the hash key, or authentication key, $T = GHASH(H, C) \oplus AES(K, N||0)$, where C is the ciphertext and H is the hash key, or authentication key. GHASH does not use K directly in order to ensure that if GHASH's key is compromised, the master key K remains secret. Given K, it is possible to get H by computing $AES(K, 0)$, but unfeasible to recover K from that value since K acts here as AES's key.

2.2 Searchable Encryption

In a common clear text search operation, an entity sends one or many keywords to the server in order to retrieve matching data. However, after this search operation, the knowledge of both the keywords and the matching data are known to the server. To achieve data confidentiality the most common solution is to use cryptography techniques to encrypt all data before transferring it to the server. To search, the simplest and trivial approach consists in transferring all data back to the client, so that it can be decrypted, allowing search operations to be executed over the clear text at the client side. Although straightforward, this solution is impractical. Consequently, Searchable Encryption (SE) arises as a technique that preserves data confidentially while enabling server-side searching [23].

SE consists in a cryptographic method that encrypts data by such a scheme that enables keyword search to be conducted over the encrypted data. Over the years, various types of constructions have been proposed. In some of them researchers have focused on the efficiency of SE technique, while others focused on the security and privacy [24]. In fact, alongside efficiency and security, the expressiveness of queries are the three main challenges focused within this field of research. On a SE scheme, efficiency can be typically measured by its computational and communication costs. Although there is no common security model, typically, a SE scheme is considered secure if it can assure that the server learns nothing about the queries as well as the matching results. The query expressiveness defines the types of supported searches.

Regarding query expressiveness, a lot of research has been conducted to provide more than single keyword search. Multi keyword search was proven feasible, with the advent of solutions with support for range, subset, fuzzy and wildcard queries. The refinement of the search results has also been studied considering ranked keyword searches, which sort the results in order to present the more relevant ones first. Verifiable keyword search was also studied and enables the verification of the correctness and completeness of the results. Nevertheless, query expressiveness implies some tradeoffs, since typically

document	keyword
1	k1, k3, k5
2	k2,k4
...	...
n	k

Figure 2.1: Forward index

keyword	document
k1	1,3,5
k2	2,4
...	...
k	n

Figure 2.2: Inverted index

it is achieved at the expense of some downgrade on efficiency or security.

SE schemes usually operate based on one of two techniques. Some schemes use an encryption algorithm over the clear text data that allows search operations to be performed directly and sequentially on the ciphertext. Thus, the search time is linear to the size of the data stored on the server. With n documents with k keywords, a linear search scheme yields a complexity of $O(nw)$, since the scheme scans the entire set item by item to find out the items of interest. To obtain more efficient results, some schemes generate a searchable encrypted index, based on the existing keywords. This indexes can significantly increase the search performance, since they allow queries to be performed by the use of a trapdoor function for a keyword k in the complexity of $O(1)$. A trapdoor function consists in a function that is straightforward to compute in one way, but very inefficient to inverse without the knowledge of a secret value. In a SE scheme they are used to generate the search tokens, commonly known as trapdoors, that allow the search to be securely performed.

However, index based solutions typically requires higher computations to be performed at the storage phase, in order to extract keywords from the input documents, encrypt them and add them to the index construction. To build a searchable index, two main approaches exist: forward index and inverted index. A forward index, depicted in Figure 2.1, builds an index of keywords per document, providing a search time of $O(n)$, where n represents the number of documents. An inverted index constructs an index per keyword in the database, achieving a search complexity of $O(|D(k)|)$, where $|D(k)|$ represents the number of documents containing the keyword k, like shown in Figure 2.2 [25].

2.2.1 System Model

A typical SE model is composed by three different parties. As shown in Figure 2.3, those parties are the data owner, the data user and the remote storage server [26].

The data owner is a trusted entity whose data, for instance a collection of documents $D1, D2, ..., Dn$, is to be outsourced to the remote storage server. The data sent by the data owner typically comprises sensitive information, arising the need for the confidentiality and privacy of that information. To obtain the needed confidentiality and privacy, the data owner encrypts the data before transferring it to the remote

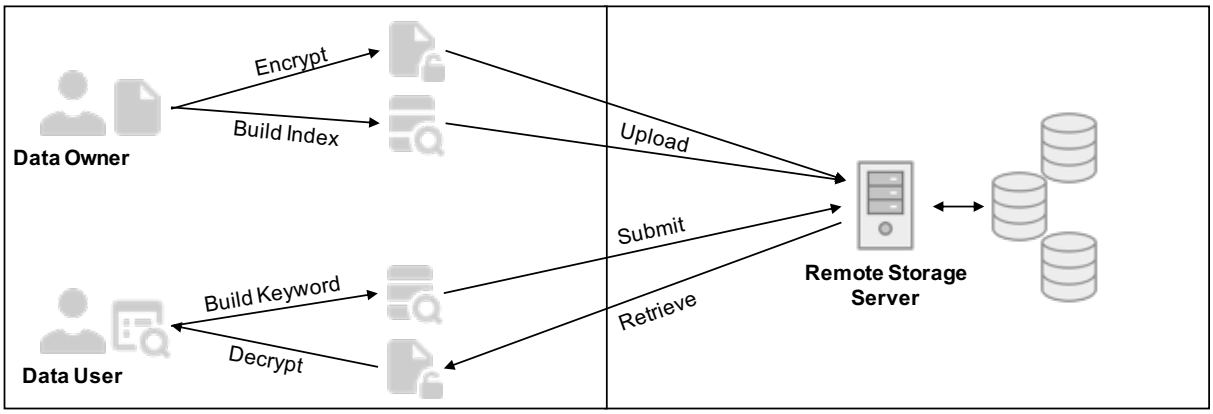

Figure 2.3: System model

storage server. In order to enable search and retrieval operations of that collection of documents, a set of keywords for each encrypted document is sent alongside the outsourced data. The keywords compose a searchable index. The index shares the same confidentiality and privacy requirements as the data from where it was extracted. Thus, the keywords are encrypted in such a manner that allows the search and retrieval to be conducted without the remote storage server learn anything about the submitted query and matching results. The data user is the entity authorised to search the remote stored data for the keywords of interest. To perform the search, the data user applies the same scheme to the keywords that was applied by the data owner, prior to its storage. The encrypted query is submitted to the server that will retrieve the matching results. This results can be decrypted and analysed by the data user. In some schemes, the data user and the data owner are the same entity. The remote storage server is in charge of the search and retrieval operations. It receives an encrypted query, tests it against the stored data and returns the matching results. Commonly, the remote storage server is assumed to be honest-but-curious. That is, the server faithfully follows the search protocol, but may try to analyse the received queries and the matching results to gather information about the data.

SE can be divided into two different classes of schemes: Symmetric Searchable Encryption (SSE), that uses symmetric key cryptography and enables only the secret key owner to produce ciphertexts and search queries; and Public Key Encryption With Keyword Search (PEKS) that enables the creation of ciphertexts with a public key and only the private key owner can perform the encrypted search. PEKS has the particularity of supporting multiple users scenarios due to the fact that anyone can encrypt data with the right public key.

2.2.2 Symmetric Searchable Encryption

The first notion of SSE was brought by Song et al. [27]. In their solution, search is conducted by performing a sequential scan on the entire ciphertext. Each word is encrypted separately and concatenated

with a special format hash value, creating a two-layer encryption method. To carry out a search operation, the server can extract that hash value from the stored ciphertext and test if the value matches the desired one. Although, the scheme can provided clear text and keyword privacy, it has a low search efficiency. Search time is linear to the size of the encrypted stored collection, since the server, for each search operation, needs to scan the entire ciphertext to test the presence of a keyword.

To overcome some of [27] limitations and improve search efficiency, Goh et al. [28] proposed an index per document construction, which is independent from the encryption algorithm used in the clear text. In order to achieve what was mentioned, the proposed scheme makes use of a Bloom Filter (BF) [29]. A BF is a probabilistic data structure that can be used to test whether an element is a member of a set. It can be represented as an array of n bits, which are initialised to 0. The elements (e.g. keywords) are added to the set by the use of k independent hash functions, each of which maps a set element to one of the array positions. The k hash functions yield k array positions. The bits at the positions are set to 1. To test the presence of an element, the same k hash functions produce the k array positions. If any of this bits at those positions is 0, the element is definitely not part of the set. If all of them are setted to 1, either the element is present on the set, or the bits might have been modified to 1 by the insertion of other elements. In other words, a BF query returns that an element is definitely not in set or possibly in set. That is, false negatives are impossible, but false positives are feasible.

On this scheme, each distinct word in a document is handled twice by a pseudorandom function and then inserted into the BF. To assure that all documents BFs are different even with the same keywords, the second run of the pseudorandom function takes as input not only the input of the first but also a unique document identifier. By creating one BF per stored document, a search operation only needs to be performed over the indexes instead of scanning the entire ciphertext. Therefore, the search time becomes linear to the number of documents, contrasting with the number of words on Song scheme. The size of each document index is proportional to the number of different words present in the document.

Chang et al. [30] developed a two-index based scheme. The index on his schemes is an $m - bit$ array, initialised to 0, where each bit position corresponds to a keyword in the dictionary. The dictionary is a prebuilt structure of searchable keywords that enables the construction of an index per document. The main difference between the two schemes is the location of the dictionary. On the first one, it is stored at client side, on the second one it is stored in an encrypted form on the server. Although the second proposal is more space efficient, the computational overhead of the second scheme is bigger. Both constructions can handle secure updates of the documents.

Another secure index based solution was proposed by Curtmola, et al. [31]. On this scheme, an inverted index is applied, mapping any distinct keyword to the document identifiers where it is present. The index is composed by an array A in which each element consists on a linked list L per keyword and a look-up table T that identifies the first node in A. Each encrypted node $N_{i,j}$ of L_i is made of three

fields $(a||b||c)$, where a is the identifier of the document where the keyword is present, b is the secret key $K_{i,j}$, used to encrypt the next node and c is a pointer to the next node or $\emptyset$ if the node is the last on the list. The node $N_{i,j}$ is encrypted with the key $N_{i,j-1}$, which is stored in the previous node $N_{i,j-1}$. The look-up table T stores, per keyword wi, a node $N_{i,0}$ that contains the pointer to the first node $N_{i,0}$ in L_i and the corresponding key $K_{i,0}$. By using a linked list and due to its nature, if the position and correct decryption key for the first node is known, the server is able to find and decrypt all the nodes and obtain the identifiers of the documents. Compared with document-based indexes, keyword-based indexes are more efficient at the search phase. Nevertheless, the update of a keyword-based index to add, delete or modify documents is costly and linear to the number of distinct words per document.

Other novel solutions, focusing on different aspects, can be found in the literature. Amanatidis et al. [32] proposed two schemes based on deterministic message authentication codes, in which each keyword is built with the use of MAC functions. Van Liesdonk et al. [33] developed a scheme that shares the same principal of one index per keyword as [31] but supports dynamic document updates. Kurosawa et al. [34] came up with a verifiable scheme that is constructed by the inclusion of a MAC inside the index in order to bind a search query to a search response. Kamara et al. [35] proposed a new scheme based on PRFs and XORs to support efficient additions, removals and modifications of documents. The same authors in [36], made use of the advances in multi core architectures to create a new dynamic scheme, highly parallelizable. Through a tree-based multi map data structure per keyword, called keyword red-black trees, a sub linear search time was achieved.

2.2.3 Public Key Encryption With Keyword Search

Boneh et al. [37] proposed the first SE scheme using a public key system based on Identity Based Encryption (IBE). IBE is a cryptographic primitive, developed by Boneh and Franklin [38], in which the public key on an entity consists of some unique information about its identity. A trusted third party, the Private Key Generator (PKG), is used to generate the corresponding private keys. To fulfil that, the PKG first computes and makes publicly available a master public key, keeping the master private key secret. Given the master public key, any party can obtain the public key of an identity by the combination of the master public key with the identity information. To obtain a corresponding private key, the PKG uses the master key to generate the required private key. This property allows a sender who has access to the public parameters of the recipient to encrypt any arbitrary message. The recipient obtains the private key from the PKG and is the only one able to decrypt the message. All of this operations can be performed without no prior distribution of keys between the involved parties. However, a secure, confidential and authenticated channel between an entity and the PKG is required for the private keys transmission.

On this searchable scheme, IBE is used in such a manner that the keywords act as the identity. Each sender is allowed to produce ciphertexts with the receiver's public key, but only the private key owner is

capable of generating search trapdoors. To produce a searchable ciphertext, the clear text is encrypted by a standard public key system and each keyword is encrypted by an IBE scheme. The final result that is sent to the server is the concatenation of both ciphertexts. In order to perform search operations, the master private key is used to derive the private key for the keyword to be searched. This key, used as the trapdoor, is sent to the server. The server will try to decrypt all the existing ciphertexts, produced by the IBE scheme. If the decryption is successful, the server assumes that the specified keyword is present.

In the Boneh model, the trapdoors are produced by a deterministic encryption system, which means that for the same keyword the trapdoor will always be the same. This property gives the server the ability to collect previous used trapdoors and test them for future documents. To tackle this limitation, Abdalla et al. [39] came up with a scheme in which a trapdoor is generated only for a specific time interval, making unfeasible for the server to search for a keyword of a different interval.

Boneh scheme also holds the drawback of requiring a secure channel to transmit trapdoors. To overcome this limitation, Baek et al. [40] proposed a Secure Channel Free PEKS model, which does not require a secure channel. The main idea behind their solution is the addition of a server key pair and the use of the aggregation technique from Boneh et al. scheme. By adding the referred key pair, only the server chosen by the data owner is capable of performing searches.

On a typical PEKS scheme, when the amount of data stored increases, the query efficiency is drastically reduced. In order to solve this issue, Bellare et al. [41] proposed a deterministic public key encryption model. This encryption approach supports efficient queries, at the cost of a weaker security model. The authors claim that any public key encryption scheme can be used in combination with any hash function, with deterministic properties. For each keyword, a encryption and hash operations are required. A search operation becomes a simple look-up for the hash value on the database.

Solutions for multiple scenarios can be found in the literature. Crescenzo et al. [42] was first to present a PEKS scheme based on Jacobi symbols instead of bilinear maps. Khader et al. [43] proposes a construction that supports multiple keywords and a secure-channel-free PEKS scheme. Rhee et al. [44] enhanced the security model of the PEKS construction of [40]. The same author proposed a scheme secure against keyword-guessing attacks by making trapdoors indistinguishable [45]. Camenisch et al. [46] presented a public-key encryption with oblivious keyword search, in which an entity can obtain the search token from the secret key owner without revealing the keyword. Liu et al. [47] proposed a scheme to outsource part of the decryption process to the service provider and thus reduce the computational overhead of the client. Tang et al. [48] developed the concept of public key encryption with registered keyword search. This model allows the data user to query for keywords that were previously registered by the data owner. Zhang et al. [49] propose an hybrid model Public Key Encryption (PKE) and PEKS into a single scheme that shares the same key pair for both primitives. Ibraimi et al. [50] proposed a

construction for a public key encryption with delegated search, which allows the server to search for two different kinds of trapdoors. One allows searching for a keyword inside a trapdoor and the other allows the server to search directly for a keyword.

2.2.4 Query Expressiveness

In order to support multi-keyword and other types of search operations, visible research has been conducted to prove its feasibility. From the literature, the identified keyword search strategies are threefold: conjunctive, fuzzy and ranked keyword searches.

Conjunctive Keyword Search

One of the first achieved improvements is the conjunctive keyword search that tries to find documents that contain multiple keywords in a single query. The naive approach would be the submission of the query as a set of separated queries, one with each keyword. After the return of the results, an intersection would be performed at the client side. Although feasible and straightforward, this approach is very inefficient. A better solution is to make the server capable of conjugating queries. Golle et al. [51] was a pioneer on this field and provided two SE schemes with support for conjunctive keyword search. His solution assumes that there are special keyword fields within each document. For instance, in a email there might exist keywords fields such as ”to” and ”subject”. Additionally, the entity who demands the search operation must know, in advance, which keyword field are to be considered while searching.

On a following work, Ballard et. al [52] presented a two conjunctive keyword search construction, based on Shamir's secret sharing technique [53] and on bilinear pairings. This construction requires a trapdoor size linear to the number of documents being searched. Wang et al. [54] presented the first keyword-field-free conjunctive keyword search scheme by using a bilinear map per keyword and per document index. The size of a query trapdoor is linear to the number of keywords contained in the index. Cash et al. [55] extended the search to support boolean queries on arbitrarily structured data. His scheme is based on the inverted index approach of [31]. Example being the searching for the presence of both “A” and “B”, or which contains at least “A” or “B”. Also, the search protocol of this scheme is interactive, since first the server replies to a query with encrypted document identifiers. Then, after those identifiers decryption, the client can communicate again with the server to retrieve the corresponding documents. Faber, et al. [56] extended Cash et al. scheme to support range, wildcards, and phrase queries.

In the public key setting, Park et al. [57] built two schemes with support for conjunctive keyword search and with an efficient computation overhead while maintaining a constant trapdoor size. In [58], the same authors proposed a new scheme with enhanced security. By using an hybrid encryption technique, it is possible to create both a decrypt trapdoor and a search trapdoor for specific keywords. Hwang et

al. [59] also proposed a public key encryption with conjunctive keyword search by introducing the concept of multi user PEKS, in order to minimise the communication and storage overhead for both the server and client. Boneh and Waters [60] presented a scheme based on Hidden Vector Encryption (HVE) that supports comparison queries, subset queries, and arbitrary conjunctive queries. Shi et al. [61] designed a scheme that is able to support multidimensional range queries over multiple attributes. Each encrypted value represents a point in a multidimensional space, which makes a query equivalent to testing whether a point is inside an hyperrectangle or not. To represent ranges, the authors use binary interval trees over integers. Later, Sedghi et al. [62] constructed a SE scheme with wildcards based on bilinear pairing and HVE. Yang et al. [63] proposed a flexible wildcard scheme that supports multiple keyword search in which any keyword may contain zero, one or two wildcards. Their construct is based on homomorphic encryption [64] with the application of the Paillier cryptosystem [65].

Fuzzy Keyword Search

Fuzzy keyword search schemes were proposed to tolerate minor typos and formatting inconsistencies in search operations. Park et al. [66] proposed a method to search for keywords with error tolerance over encrypted data, based on approximate string matching. To search for identical words, each word is encrypted character by character. Then, the test for similar keywords is done by computing Hamming distances [67]. Shen et al. [68] presented an encryption scheme based on inner products. The trapdoor and the encrypted content are viewed as vectors and, during search operations, the product of both vectors is calculated. This scheme does not leak which of the search terms has matched the query. The authors also give a definition for fully secure predicate encryption, which means that during the search operation nothing should be leaked, except for the access pattern, which represents the set of obtained search results (i.e, the collection of documents) that were obtained for a given keyword. Bosch et al. [69] proposed a scheme also based on inner products. It makes use of an index generation technique in combination with somewhat homomorphic encryption [70]. They also separate the query phase from the results retrieval phase and make use of Lattice-based cryptography [71] to achieve better efficiency.

Li et al. [72] presented a fuzzy keyword search scheme for cloud computing that calculates Edit distances [73] and construct fuzzy keyword sets. In 2013, the same authors extended their scheme to support multiple users by using Attribute Based Encryption (ABE) [74]. Later, Wang et al. [75] proposed a multi-keyword fuzzy SE scheme that combines the use of BF and hash functions. The scheme supports fuzzy keyword search and provides proof for verification. The proof can be used by the client to verify if the server returned all possible search results. Kuzu et al. [76] developed a generic construction based on Locality Sensitive Hashing (LSH) and BF. A LSH function outputs the same hash values for similar items. In order to measure the distance between words for the similarity search the Jaccard distance [77] is applied. This scheme is interactive, needing two rounds of communication to retrieve the matching

documents.

To tackle fuzzy keyword searching when using asymmetric cryptography, Bringer et al. [78] also proposed the use of LSH to enable error-tolerant queries. In this case, the function used for LSH measurement was the Hamming distance. The output of that function is added to the BF in an encrypted form. If two keywords are similar, the LSH yields the same hash values, thus allowing error-tolerant queries. The search in this scheme is interactive. First, through a Private Information Retrieval (PIR) [79] protocol, the encrypted BF positions are retrieved. Then, the client decrypts all positions and computes the intersection. Finally, the files can be obtain by its identifiers. Xu, et al. [80] proposed a PEKS scheme that also supported fuzzy keyword search and the use of more than one keyword per fuzzy keyword trapdoor. Such avoids that the server cannot discover the exact keyword that was used. Afterwards, Fu et al. [81] improved the accuracy of the fuzzy multi-keyword search schemes so that more spelling mistakes could be tolerated.

Ranked Keyword Search

Over the last few years, some work was developed towards ranked keyword search. Ranked keyword search can be characterised as an advanced search functionality that is present in several online search engines. It works by returning the most relevant documents first, based on the submitted query. In the single keyword paradigm, Wang, et al. [82, 83] achieved ranked search through the use of order-preserving encryption [84, 85]. This is a deterministic encryption technique which preserves the numerical order of clear texts. It makes use of the Term Frequency - Inverse Document Frequency (TF-IDF) rule. Term Frequency (TF) is used to measure the importance of a given term within a document and is calculated as the number of times a given keyword appears within a document. Inverse Document Frequency (IDF) is used to measure the overall importance of a given term within the whole collection and is calculated by dividing the total number of documents by the number of documents containing the term.

A multi-keyword ranked search scheme was developed by Cao et al. [86] who proposed the use of similarity measure called coordinate matching. Coordinate matching can be described as obtaining the maximum number of possible matches in order to describe the relevance of documents to the search query. Inner product similarity is used for quantitative evaluation of similarities. Despite being characterised as an improvement by ranking the results based on the number of matching keywords, their solution does not consider that keywords may have different importance with respect to the search results. Other relevance measurement methods were introduced to tackle this. Sun et al. [87,88] proposed a multi-keyword ranked search model using the cosine measure technique. This is a metric which can be use to determine the similarity between documents independently of their size. It works by measuring the cosine of the angle between two vectors in a multi-dimensional space. The smaller the angle, the

higher the cosine similarity. Their scheme achieves better-than-linear search efficiency at the expense of search accuracy. Khan et al. [89] combined multi-keyword ranked search with fuzzy search.

Xia, et al. [90] proposed a dynamic multi-keyword ranked search scheme which uses a secure tree. Chen, et al. [91] developed a scheme based on a hierarchical cluster structure, where documents are divided into subcategories and built as trees. Search is conducted through an improved k-mean method. Orencik et al. [92] proposed a scheme which returns the top most relevant matches using the TF-IDF primitive. More recently, Guo et al. [93] presented a multi-keyword ranked search scheme, based on the vector space model combined with TF-IDF rule and cosine similarity measure. To achieve sub-linear search time, the authors use a BF to build a search index tree.

2.3 Summary

Regarding cryptography, within this chapter it was possible to understand the ubiquity of such technique. Symmetric and asymmetric key algorithms are the two distinct types of cryptography and its common to find hybrid approaches to combine the advantages and minimise the drawbacks of both types of cryptography. Hash functions arise with the purpose of data integrity preservation and keyed hashing assures an additional authenticity of the information. Authenticated encryption, usually built by combining a symmetric encryption algorithm and a HMAC, enables the sustenance of confidentiality and authenticity simultaneously.

Searchable encryption is a powerful cryptographic method that, due to current privacy concerns, has gained the attention of multiple researchers over the last few years. Like in cryptography, the proposed algorithms are either based on the symmetric or asymmetric settings. The main evolution on this field of research is related to query expressiveness, being already possible to find solutions that allow a type of search as advanced as what is supported in the clear text field. However, there is room for much improvement since the efficiency and security of such solutions can still be enhanced. Nevertheless, the trade-off between the query expressiveness, security and efficiency is one the relevant challenges researchers face on the design of their solutions.

Chapter 3

Related Work

The present chapter discusses research related to securing logs, either using secure search capabilities or not. The rationale regarding this selection is based on its relation to the solution proposed in this work. Securing logs isn't a new concept since some work can even be found in a pre-cloud era.

The solutions that we present herein are divided in two sets. The first set comprises solutions that provide secure logging frameworks, with confidentiality, integrity and authenticity guarantees. Of which, the most recent solutions are oriented to the remote storage of logs in cloud-based services. The second set solutions, despite enabling similar security properties, have as main characteristic the enabling of searchable encrypted logs, as a specific application of searchable encryption techniques.

Bellere and Yee [94] were the first to propose a solution to enhance the security of logs. The authors define a forward integrity security property and demonstrate its application to secure and verifiable logs. Key issues being intrusion detection, accountability and communications security. A forward secure stream integrity is presented and constructed by the use of a forward-secure MACs scheme, in which the log entries are indexed based on time periods. This solution enables a flow integrity of log entries. Based on this work, multiple research efforts appeared.

3.1 Schneier and Kelsey

Schneier and Kelsey [95] proposed a protocol, using symmetric cryptographic, focused on the storage of audit logs, which has been used as an inspiration by some secure logging solutions. The scheme employs an one-way hash chain [96] used to create a dependency among the log entries. This dependency enables the detection of unauthorised modification in the log sequence, since any alteration would break the consistency of the hash chain. An hash chain, illustrated in Figure 3.1, consists of a successive computation of cryptographic hash functions, where the input of the current hash also includes the output of the previous one.

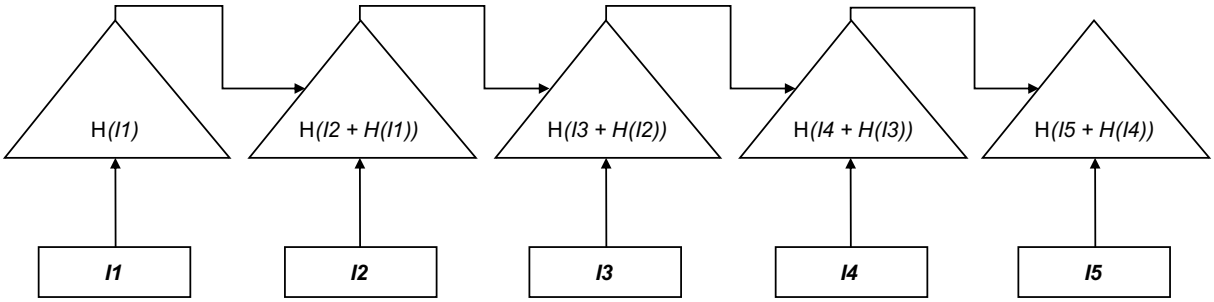

Figure 3.1: Hash chain

The authors also considered the use of evolving cryptography keys [97] to protect the audit trails and support computer forensics. Evolving cryptography can be seen as an encryption technique in which symmetric keys evolve over time, with the goal of limiting the impact of key compromise. Attackers who obtain a key can only decrypt entries where the compromised key was used. Alongside the communication and storage overhead, since an authentication tag is generated and stored for each individual log entry, the scheme was proven to be vulnerable to truncation attacks. A truncation attack consists in the deletion of tail end log entries without being detected, thereby breaking the log entries integrity.

3.2 Forte

In 2005, Forte et al., proposed a generic secure logging solution [98] that makes use of covert channels for log transmission. It aims at addressing the lack of security of the syslog protocol [99] with problems related to the transmission of data, message integrity and message authenticity. A covert channel can be used in any communication channel that may be used to transmit information using methods not originally though of. The authors explored the possibility of transmitting log data using ordinary DNS requests and responses. By controlling a DNS server, it is possible to create a covert channel and send data using specific DNS records. For instance, using the CNAME record, Forte et al. could send or receive 110 bytes of data, per packet.

The communication protocol is achieved by proxying the information through a DNS server on which the syslog clients write information, the log data, by modifying the DNS tables of the server. The real syslog server, in order to obtain the log data, will make successive DNS query requests to obtain the record from the DNS server. On the design of the solution, the guarantees of syslog messages authenticity and integrity was contemplated. They used DNS Security Extensions, asymmetric cryptography and hash functions, such as the MD5 [100] and the SHA-1 [101] algorithms, considered safe at the time of their publication. Encryption also provides message secrecy and prevents unauthorised parties to publish logs. The DNS server publishes its public key through specific DNS records. The client can verify data authenticity.

3.3 Holt

Holt et al. proposed Logcrypt [102], a secure logging protocol. Its proposes that logs are initialised in a known state, stored on an external server, and that the integrity of an earlier state can be used to verify the integrity of a later state. To provide tamper-evidence, once a secret is used and no longer necessary, it is erased from the system. The solution provides two different approaches. The simpler one is based on MACs, in which an initial secret random value is used to initialise an hash chain. Each link of the hash chain is then used to derive secret keys for log data encryption. As soon as those keys are no longer needed, they are erased. Additionally, the hash chain link used to generate the keys for each log entry is deleted right after its usage. This way, only the last link of the hash chain and the keys computed from it require storage. The use of symmetric encryption requires that any entity, who desires to verify the integrity of a log entry, must have the secret key used with the MAC function. Any entity that knows the key can forge log entries and, consequently, break the system's security. To address this problem, Holt proposed the second approach that uses asymmetric encryption combined with identity-based signatures [103]. Replacing MACs with digital signatures enables the verification of log entries by any entity without disclosing private keys.

Logcrypt also presents an alternative to secure multiple log files. The solution is based on a tree like structure of logs in which the parents nodes store their children initial secret values required to compute the hash chain. To add new children, a new secret value needs to be added to the parent node. Holt assumes that he cannot prevent an attacker who controls the system from performing deletion or truncation attacks. However, the author proposes an option to alert whenever logs tampering is detected. The detection is based on metronome entries, an approach inspired by the UNIX utility syslog. A metronome entry consists in a specific log entry appended to the log file at regular intervals of time to indicate that the log is still working as expected. The verification consists of verifying all metronome entries, if one missing, the last valid metronome entry indicates, with a relative time margin, when the truncation attack might have happened. Despite the fact that the use of asymmetric cryptography simplifies the log entry verification, it creates not only a communication overhead, originated from the constant key pair exchanges between the parties involved, but also storage overhead, since asymmetric key signatures are usually larger than MACs.

3.4 Ma

Ma et al. [104] proposed a new approach to secure logging. They state that for an audit logging system to be considered secure, it must assure not only data integrity but also stream integrity, as no reordering of the log entries should be possible. The author also enunciates the log truncation attack, a type of attack that prior schemes [94, 95, 102] failed to mitigate. Plus, he considered that prior schemes are

inefficient both in storage and in communications, making them unsuited for resource-poor devices. The proposed mitigation is based on forward-secure stream integrity for audit logs. This property is achieved by the use of Forward-secure sequential Aggregate (FssAgg), conceived by the same author in [105]. In a FssAgg scheme, signatures, if it is the case of public verification, or MACs, if it is the case of private verification, are combined into a unique aggregated signature, in a sequential manner.

The FssAgg scheme satisfies all the requirements the authors identified for a secure logging system. Forward security is achieved by a one-way function used to update the signature key. Therefore, an attacker is unable to recover previous keys even knowing the current one. Prior signatures cannot be forged. Stream security is assured by the sequential aggregation nature of the scheme that preserves signatures order and makes reordering of the signatures unfeasible. Integrity is accomplished because any message addition, modification or deletion makes the final aggregated signature invalid. Additionally, a FssAgg scheme only foresees append operations, so any selectively modification of an already computed signature is impossible. Regarding its operation, an FssAgg scheme comprises a signature algorithm and a verification algorithm. The former takes as input a private/secret key, a message to be signed and the aggregated signature. It computes the signature of the input data and combines it with the already existing signature, outputting a new aggregated signature. A new key is produced based on the key used to compute the current signature. The latter receives a public/secret key, a set of signed messages and the aggregated signature and outputs a binary value telling whether the signature is valid or not.

Based on this scheme, Ma et al. devised two secure logging schemes, one privately verifiable and another publicly verifiable. The privately verifiable scheme is based on MACs, in which two FssAgg MACs are computed over each log file with different keys in order to avoid the dependency of an always online server. This scheme can be considered efficient since it only uses hashing and symmetric cryptographic operations, both considered fast and not computationally unfeasible. The publicly verifiable scheme bases its operation in asymmetric cryptography and is envisioned mainly for systems that require public auditing. For storage efficiency, only the aggregated signature is kept and the individual ones are erased once they are used in the aggregation, since the verification of one individual log entry implies the verification of the log file. Nevertheless, if a specific log entry must be verified, a more fine-grained verification is required. For that purpose, the authors proposed an extension to their schemes in which a signature is kept for each log entry, providing individual log verification. However, this scheme extension makes it vulnerable to truncation attacks. To mitigate this, Ma et al. developed an immutable FssAgg authentication scheme, making it computationally unfeasible to generate a valid aggregated signature based on already existing ones.

3.5 Ray

The Secure Logging as a Service (SLaS) term was introduced by Ray et al. [106]. They proposed a novel solution for the storage of log records on a remote server operating in a cloud-based environment. They enumerate the properties that a secure logging as service platform should offer, not only during the generation of the log records, but also during its harvesting, transmission, storage and retrieval. The correctness of the data, meaning that it should reflect the reality of the system from where it was created. Tamper resistance, assuring that only records submitted by legitimate customers should be accepted and, after storage, these records should remain unchanged or at least they should not be altered without detection. Verifiability, meaning that it is also necessary to be able to validate that none of the log entries have been manipulated or removed. Forward integrity should be achievable by making all the log entries linked together. Data confidentiality is also crucial, since the information contained in them must remain secret not only during transit and while stored, and readable only by legitimate entities. Log records privacy, meaning that each log entry should not be easily traceable to its origin.

With this properties as the base pillars, the authors presented a solution comprised by three entities: 1) Log Generator, in charge of generating log records; 2) Logging Client, responsible for receiving the logs from the log generators and preparing them for remote storage; 3) Logging Cloud, the remote server accountable for storing and retrieval of the log information. The proposed protocol starts by generating three master keys: A_0 and X_0 used for data integrity in hash calculations, and K_0 used for confidentiality in encryption and decryption operations. These keys are stored across the three entities based on a proactive secret-sharing scheme [107], in which the chunks of the secret are exchanged between the entities, at fixed intervals, thus making it more difficult for an attacker to compromise the confidentiality of the system.

The log records are handled in batches of n, a random value that indicates how many times the master keys can be used, limiting how many log records will exist on each batch. Each batch starts with a special first entry that contains a timestamp and the value n. This entry is encrypted with K_0 and a MAC, using A_0, is calculated over the resulting ciphertext. Then for each log record received, a new set of keys is generated based on the previous ones, later erased after use. The log entry is then encrypted and a MAC is computed with the encrypted log entry as input. An aggregated MAC is also calculated using X_0 and having each encrypted log entry MAC as input. The closing of the batch is marked by a log close entry, which includes a timestamp and the aggregated MAC. The log batch is then uploaded to the remote server for storage.

The upload of log batches can only be accomplished by authorised parties, impacting on its use for anonymous log storage. For that purpose, the authors proposed the use of a k-times anonymous authorisation protocol [108] that imposes a no correlation between a logging client and its transactions. Each log batch is indexed by an upload tag in order to allow for future retrieval of the data. The upload tag

consists of an instance of a hashed Diffie-Hellman [10] key and is calculated based on public information, so it is possible for anyone to retrieve log data using an upload tag. Nevertheless, the data retrieved is in an encrypted form and only the one with the right decryption key will be able to read it. A delete tag, computed using a cryptographic random number, is also included on the uploaded data. That tag gives the ability to erase a log batch. Since it is a critical operation, the logging client in order to be able to delete a log batch needs to fulfil a challenge/response authentication method prior to deletion of the logs. Moreover, the delete tag should be stored in a secret sharing scheme identical to the one used for the master keys [107].

3.6 Zawoad

Zawoad et al. applies SLaS to the digital forensics domain in [109], with a consequent extended version in [110]. Digital forensics is, according to NIST, an applied science used to identify an incident and further collection, examination and analysis of evidence data [111]. Cloud forensics can be defined as the application of a scientific method to identify previous events that on cloud environments by identifying, collecting, preserving and analysing of digital evidence [112]. The inherent properties of cloud platforms establish some challenges that do not exist on traditional forensics.

The authors proposed SecLaaS, a solution to store logs in a secure way, preserving its confidentiality and integrity, while providing an API for forensic investigators to be able to gather their evidence. The log treatment of their solution starts by the acquisition of the log records from multiple sources. Then, those logs entries, to preserve their privacy, are encrypted using an asymmetric encryption algorithm. Some of the fields of the entries are kept in clear in order to allow some search operations to be performed over that data. After, the log of that encrypted entry is fed to log hash chain, which is used to maintain the right order of log records. Then, the tuple composed by the encrypted log entry and its matching hash chain link are stored. In order to generate the Proof of Past Log (PPL)[1], one of three accumulator schemes can be used: Bloom filters, one-way accumulators, or Bloom trees.

In the Bloom filter scheme, a Bloom filter is maintained per IP address, per day. The proof creation starts by retrieving the latest Bloom filter and generating k bit positions for the log entry, based on k different hash functions. These k bits are then setted on the Bloom filter and stored. At the end of each day, the PPL is generated based on an accumulator value, a proof generation time and a digital signature. Verification starts by the calculation of the k bit positions of the Bloom filter, which are then tested against the published accumulator. If all the bits are similar, then the log entry can be considered valid.

The one-way accumulator is a cryptographic accumulator, based on RSA assumption, which enables

[1] PPL is a publicly available information that assures that, after publication, the logs accommodated inside its time span cannot suffer modifications of any kind.

the test of membership of an element in a set, with no false negatives and the possibility of false positives [113]. It is based on two large private prime numbers P and Q and on two public values: N, equal to $P \times Q$, and X, an initial seed, also prime. The proof creation is calculated by the exponentiation of X, or the current accumulator, to a numerical hash value of the log entry reduced modulo N. The PPL is generated based on the same notion as the Bloom filter. The validation is conducted by the verification of the presence of the accumulator value of the specific log entry.

Since Bloom filters conceive the possibility of false positives, the authors designed a Bloom tree that, while requiring less space, provides a smaller percentage of false positives. To build the Bloom tree, for every m number of logs, a new Bloom filter is generated. Creating $\frac{n}{m}$ Bloom filters for n logs records, per IP and per day. The proof creation follows the same pattern as the simple Bloom filter. The PPL is computed by a cumulative addition of the Bloom filters into a higher order Bloom filter. The integrity verification is provided by a set of m logs. However, in this scheme, it is necessary to retrieve additional logs to conduct the validation. The authors claim that in the worst case scenario $2(m-1)$ extra logs records are required.

3.7 Waters

For auditing purposes, Waters et al. proposed a solution [114] that maintains the privacy of the audit logs without losing searching capabilities. The authors defined a secure audit log as a confidential log with tamper resistance, publicly or privately verifiable and, for that reason, they propose both an asymmetric and symmetric key schemes. The proposed solution is constructed as a series of singular audit log records. Each one is, in turn, obtained by the encryption of the log data with a K_i, randomly selected per log entry. To enable integrity and tamper resistance, each encrypted record is concatenated with the hash of the previous record, forming an hash chain. For searchability purposes, it also includes keywords, which are extracted from the audit log as depicted in Figure 3.2. A verification value, also included in each log entry, is a hash value of the complete audit log content is calculated. The verification value is periodically published on multiple servers in order to be publicly verifiable.

The symmetric key scheme, inspired by [27] and [28], indexes each log entry with the generation of a random symmetric key, that shall be used only for the encryption of this entry. After the encryption of a log entry, a random bit string r of some fixed length is created. Then, the set of keywords is extracted from log record. For each keyword k_i, a pseudorandom function is applied having as input k_i and a secret S. The result is then used as input for another pseudorandom function, together with r. The output of this second function is XORed with the key K and a flag, a constant bit string of length l, which yields the final keyword value c_i. The log entry is then formed by the encryption of the log record, the r value and the set of encrypted keywords $c_1, ..., c_n$. The search and decryption operations work as

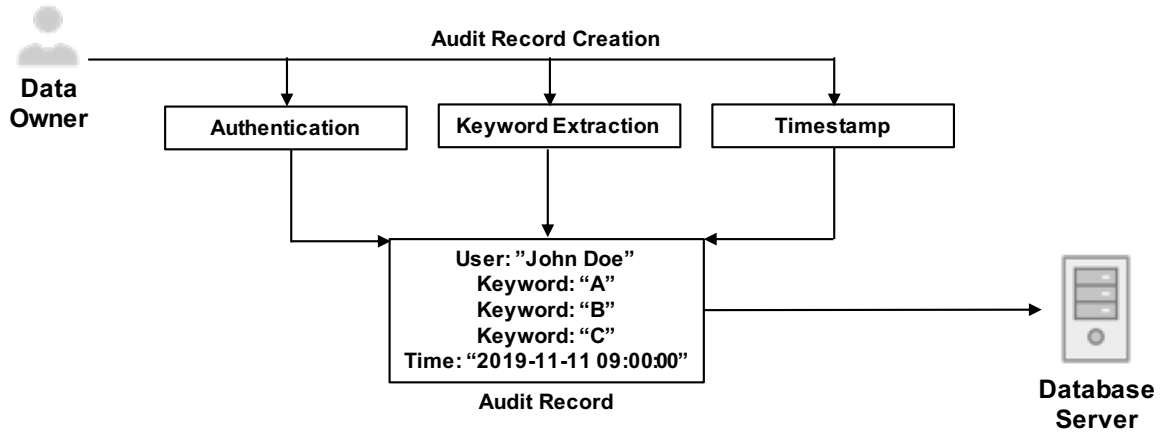

Figure 3.2: Waters keyword extraction scheme

follows. The search for the a keyword k starts by calculating the pseudorandom function with k_i and S as inputs. Then, followed by the encrypted keyword c_i computation, and if the first resulting l bits match with the flag, the key K is extracted from the rest of the result and used to decrypt the log record.

The asymmetric cryptography scheme is based on Identity-Based Encryption (IBE) [38]. The encryption is achieved by generating a random secret key K, used only for the encryption of one entry. Then, for each keyword k_i, it calculates the Identity-Based Encryption c_i of the concatenation of the flag and the key K, using k_i as the public key. To search for a keyword k, first it is necessary to obtain the related private key k and, then, for each encrypted keyword, a decryption operation is performed. If the first output bits of that decryption are identical to the flag, the rest of those bits corresponds to the secret key K, which can be used to decrypt the log entry. This asymmetric scheme was implemented by Gopularam et al. in [115].

3.8 Ohtaki

Ohtaki et al. [116] addressed some of the performance issues present in existing solutions. Those issues are mainly related to the use of asymmetric encryption in search operations. These operations usually require higher computation, making the search time proportional to the number of log records. Based on this premise, the author proposed the use of a partial disclosure scheme that was based on his previous work [117]. Two key pairs $(P_0, S_0), (P_1, S_1)$ are generated and are used to compute a log record for both searching and disclosure. After making the keys P_0 and P_1 publicly available, the scheme signs each log record keyword with key S_0. Then, it concatenates that signature with the log record unique identifier I_i, encrypting those values together by the public key P_1.

The log record is also encrypted with a symmetric key algorithm, being the symmetric key generated

by encrypting Ii with S_0. To search for a specific keyword k, first its digital signature is calculated with S_0. Then, the signature is encrypted together with each log record identifier and compared with the stored values. If they match, it is possible to claim the existence of that keyword in that log record. In order to obtain the relevant log records decryption keys, its identifiers are encrypted again with S_0, which generates the symmetric key used for the log record encryption.

After some experiments, Ohtaki noticed that this scheme is inefficient. The conducted tests indicate that 90 seconds are elapsed for a search operation of a single keyword over a 5,000 log records universe. To reduce this search time, the author studies the feasibility of the use of an encrypted inverted index[2]. The inverted index consists of a linear list, on which each list item is composed by the log record identifier and a pointer to the next list item. To assure privacy, these list items are encrypted. Each index is labelled with the result of the encryption of the keyword k with S_0, followed by a digital signature calculated with P_1. The pointer to the latest list item is encrypted with a symmetric encryption algorithm, whose key is the encryption of keyword k with S_0.

Updating the encrypted inverted index is as follows. Each keyword is first encrypted with S_0 and then signed with P_1. Then it verifies if the resulting signature is already present in the index. If not, a new linear list is created. Otherwise, a new list item is added the existing list, with a consequent update of the list head. Search operations become simpler, since after applying the required encryption and signature to a keyword k, a simple lookup operation is enough to obtain the relevant log records. An embedded scheme is also proposed, on which the log record decryption key is saved on the inverted index alongside its identifier.

In following work, the same author proposed a solution [118] that extends its simple search to one that supports boolean queries, enhancing the search capability. The basic and most straightforward solution was the creation and storage of all possible combinations of keywords conjugations. However, this approach would rapidly become impractical since the required storage space would increase drastically. Thus, the use of Bloom filters was adopted. The designed protocol is initialised by the generation of a key pair (P_0, S_0) and the publication of P_0. The properties of the Bloom filter are also configured. More precisely k, the number of hash functions, and m, the size of the bit array. Afterwards, each log record is symmetrically encrypted, for disclosure, with a key K_i generated by the encryption of the log identifier with S_0.

Then, the Bloom filter construction starts, as illustrated on Figure 3.3. First, all possible combinations of keywords that exist on each log entry are generated. For instance, for the keywords “A”, “B” and “C”, the possible combinations are “A”, “B”, “C”, “A and B”, “A and C”, “B and C” and “A and B and C”. Next, a normalisation is applied to the patterns, making the order of the keywords on the query not important (e.g. “A and B” is the same as “B and A”). The normalised patterns, treated as individual keywords, are

[2] An inverted index is a common data structure used for plain text search speed optimisation by switching from sequential data comparison to a table lookup

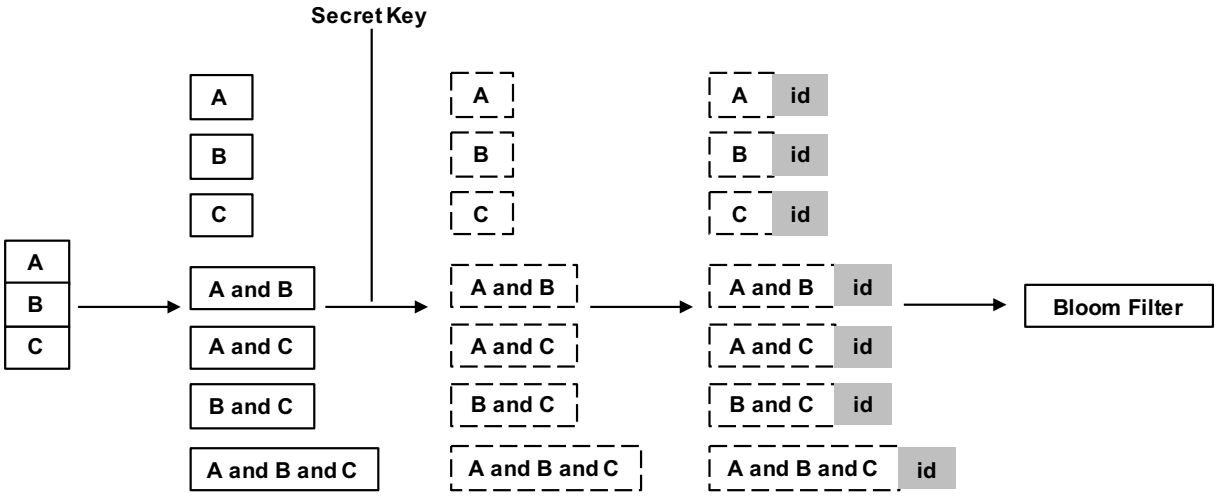

Figure 3.3: Ohtaki Bloom Filter construction

encrypted with S_0 and concatenated with the log identifier. The final value is added to the Bloom filter.

3.9 Sabbaghi

Sabbaghi et al. [119] proposed a scheme to build an audit log that should guarantee tamper resistance, verification capability, logging speed, search speed and correctness of the search results. They propose the use of a record authenticator generated with hash functions. The schema is focused on SQL commands and starts with the generation of an asymmetric key pair (P_0, S_0) and the sequential publication of the public key P_0. Then, the extraction of keywords from SQL queries is performed like detailed in Figure 3.4. Five groups G_i of distinct types of keywords are created. The first group is reserved for keywords of the "SELECT" part, the second for keywords of the "FROM" clause, the third for keywords of the "WHERE" condition, the fourth group is dedicated to the values of that condition and the fifth, and last group, is used for metadata, such as the time when the query was executed or by whom. For instance, for the statement "SELECT NAME FROM USERS WHERE ID=1", executed by user U_i at 01/11/2019, after extraction, the first group would be "NAME", the second would be "USERS", the third would be "ID", the fourth would be "1" and the fifth would be "U_i;01/11/2019".

Following the creation of such groups, the record authenticator is generated by the use of three hash functions and a dedicated hash space H, which consists on a string of bits, setted initially to zero. First, a unique random key K, valid only for that log entry, is generated by the encryption of the log record identifier with the key S_0. Then, the generation of the record authenticator is achieved by hashing the extracted keywords. In order to enhance the security of the scheme, each keyword, prior to its hash calculation, is concatenated with key K. This assures that the same keyword, even on the same group,

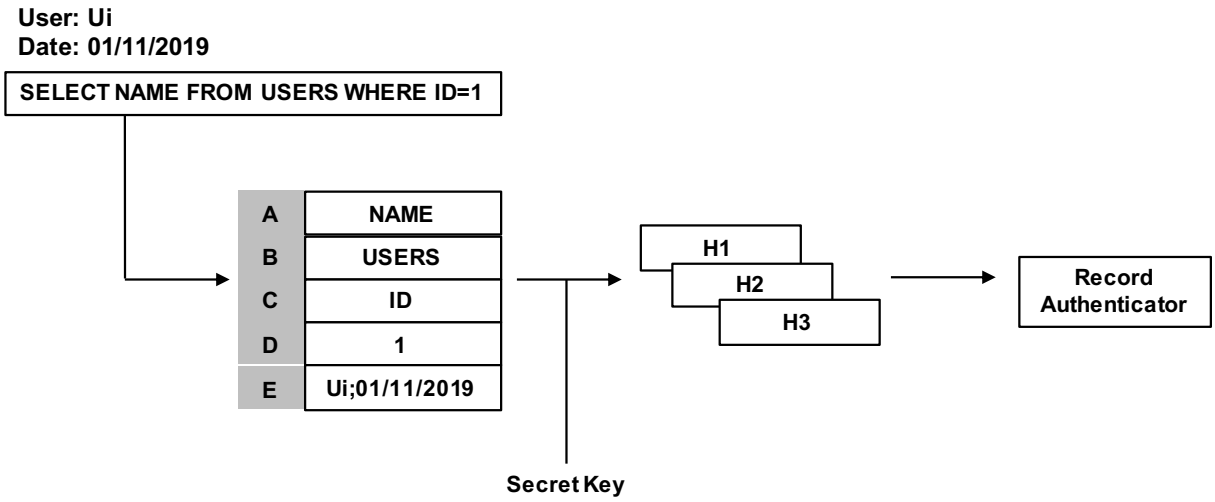

Figure 3.4: Sabbaghi Record Authenticator construction

for two different log records hashes to different places of H. The used hash functions parameters are variable. If the hash space H length is fixed, the hash functions only receive the keyword and its group as input, if the hash space H length is variable, the hash functions must also receive the length of H as input. Subsequently, the entire SQL command is encrypted using K and a symmetric encryption algorithm. Both the result of the encryption and the record authenticator are stored.

At search phase, a set of keywords and related groups is presented and an hash space Hs is created and setted to zero. Then, each keyword and group are passed to the hash functions, whose output converts some of bits of the hash space Hs to one. To verify if those keywords are present on any log record, a logical AND operation between the hash space Hs and each log record hash space H is computed, generating a new hash space Hr. If Hs and Hr are similar, the presence of that keyword on that log record is confirmed. The decryption key of each log record is generated by encrypting the identifier of the log record with S_0.

3.10 Accorsi

Accorsi addresses log privacy [120, 121] with focus on devices with low resources but also addressing inner and outer privacy. Outer privacy is described as outside threats that arise from the lack of control over the information. Inner privacy is defined as the attempt to gather private information by log data tampering. Accorsi also states that, a secure remote storage of logs is required, since the typical amount of data logged by a device is immense and the considered devices are resource-poor devices. As logs are remotely stored, data should be encrypted in order to prevent unauthorised third-party access.

Accorsi's proposed solution, inspired by [95], starts on the device, that is in charge of the initialisation

and construction of the log file. The device is expected to apply the necessary cryptographic techniques to safeguard the privacy, integrity and uniqueness of its log file. The privacy is achieved by encrypting each log entry with a symmetric encryption algorithm. The encryption algorithm also enables forward integrity, meaning that if an attacker can compromise the log data at instant t, all log data stored prior to t will not be compromised.

The integrity is guaranteed by the construction of an hash chain composed by message authentication codes that are calculated per entry and based on a secret random value computed inside the device. Uniqueness is ensured by the use of timestamps, which also prevent replay attacks. The designed protocol also contemplates mutual authentication of the involved parties and remote storage confirmation by the server.

Based on his previous work [120, 121], Accorsi designed BBox [122], a digital black box, inspired by the flight recorders used in air-planes, based on asymmetric cryptography, that aims to guarantee the authenticity of log records used in distributed systems. It assures a reliable data origin by only considering log records from legitimate sources. The log records, in order to guarantee forward secrecy, are stored in a encrypted state, each one encrypted with a randomly generated key. Tamper-evidence is also accomplished through the maintenance of an hash chain. BBox allows single keyword searches, with the use of log views, a mechanism similar to views of relational databases.

The architecture of this solution is illustrated on Figure 3.5 and it comprises: the Log message handler (LMH), the Entry append handler (EAH), the Log view handler (LVH), the Entry retrieval handler (ERH), the Operational log file (OLF), the Crypto module (CM), and the Secure log file (SLL). The LMH receives log records from the clients and assures that those log records are valid and from legitimate sources. Then, the log records are encrypted by the EAH and stored in the SLL container. The LVH is responsible for the disclosure of selected log records to legitimate and authenticated entities. The queries capable of retrieving such information are in the scope of ERH. The CM is responsible for storing the cryptographic keys and the OLL records the events occurred during the operation of BBox. After its initialisation, where the required cryptographic keys are computed, the BBox is ready to receive logs, which arrive encrypted with its public key and signed with the client's private key. After decrypting it and assuring its integrity, each log entry is firstly encrypted with a unique symmetric key K_i, generated from an hash of the combination of an evolving key G_i and the log entry identifier I. Each key K_i is computed independently for each received entry. Afterwards, the encryption output is stored alongside its keyword hash and its signed hash chain link.

The search and subsequent retrieval of log records is achieved through the use of log views. The search operation requires a mutual authentication between BBox and the entity who desires to perform the search. To search for a keyword of interest, BBox starts by computing the hash of that keyword and performing a linear search over the stored log records. When a match occurs, the symmetric encryption

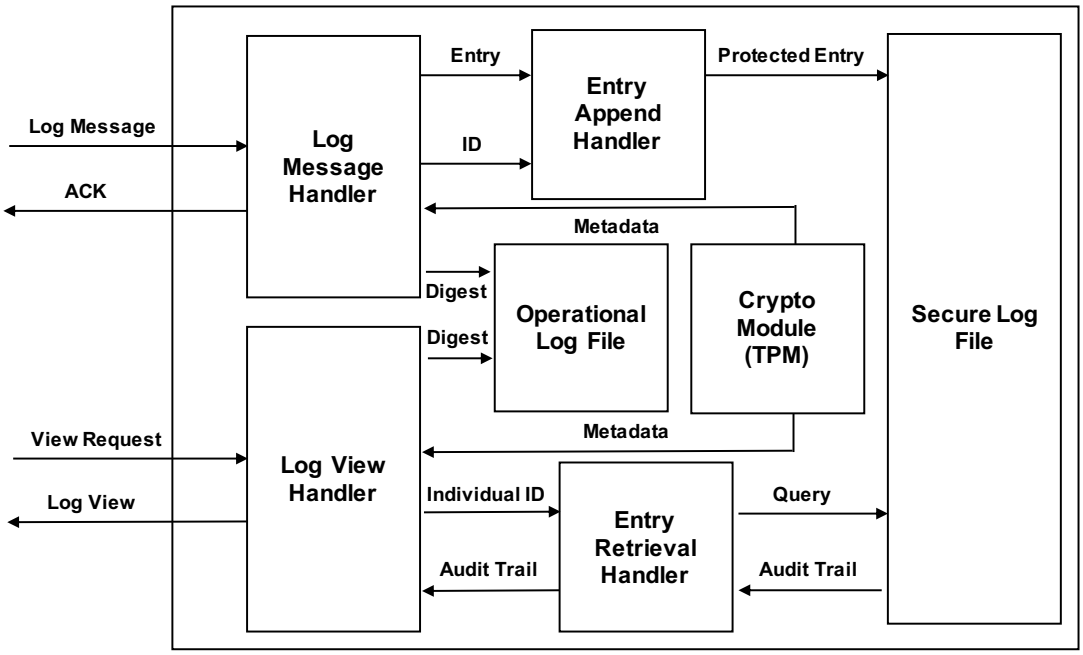

Figure 3.5: BBox architecture

key K_i of the log entry is computed and the subsequent decryption is applied.

3.11 Savade

Based on a system of linear equations, Savade et al. [123] proposed a technique to protect log records from tampering while still permitting search operations to be conducted over those records. The developed schemes comprises three entities: a Trusted Key Generator, which generates a key pair (PU, PR); a User, which has access to the remote stored data; and a Server, which stores the encrypted log records. Four distinct functions are envisioned on the proposed solution: $KeyGen(s)$ that returns a key pair (PU, PR) using a security parameter s; $PKE(PU, m)$ that encrypts a message m, using PU, and outputs s; $Trapdoor(PR, m)$ that outputs the trapdoor t using m and PR as input; and $Test(PU, s, t)$ that tests if an encrypted keyword t is present in the log records. This verification is conducted by a generalisation of the inverse matrix concept [124].

The storage of log records, depicted in Figure 3.6, starts by $KeyGen(s)$ being executed by the Trusted Key Generator entity. Then, the User, using the public key PU, encrypts the log record alongside with the keywords of interest. All encrypted data is then stored by the Server. The search protocol, depicted in Figure 3.7, is initiated by the User who encrypts a keyword using PR, generates its search trapdoor and sends it to the Server. In its turn, the Server verifies if the keyword trapdoor matches any encrypted stored keyword and, if so, the related encrypted log record is retrieved.

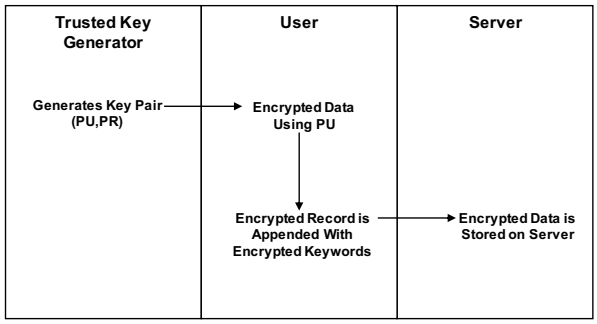

Figure 3.6: Savade's scheme storage flow

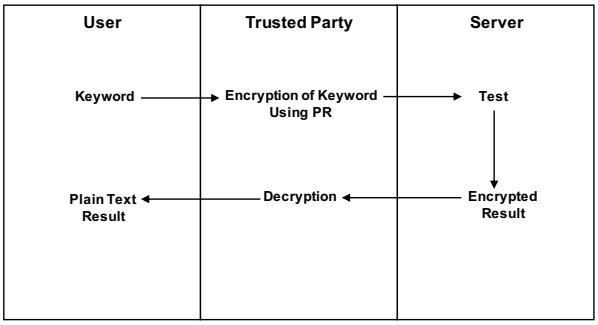

Figure 3.7: Savade's scheme search flow

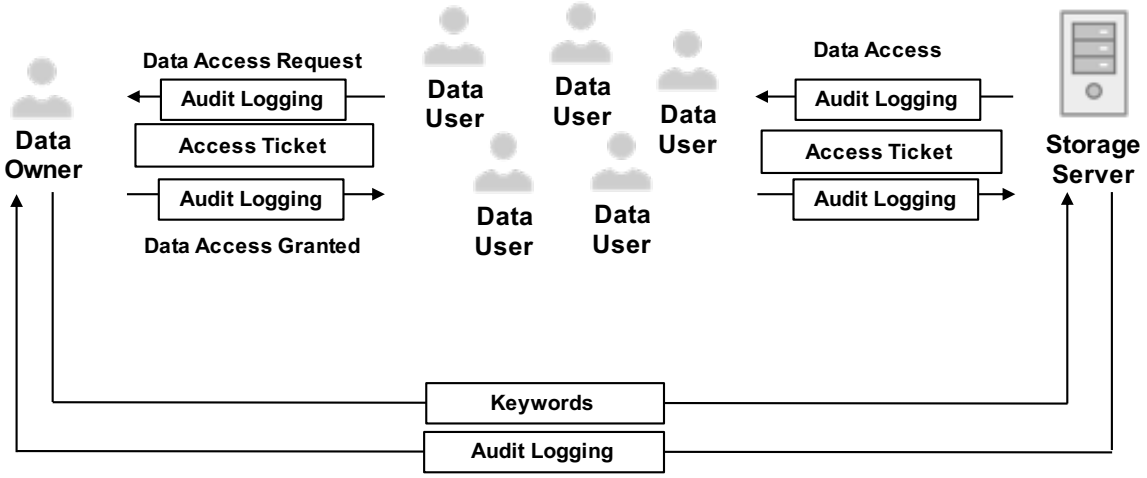

Figure 3.8: Zhao's scheme

3.12 Zhao

More recently, the scheme presented by Zhao et. al. addresses encrypted log searching while preserving privacy [125]. The proposed scheme, shown in Figure 3.8, involves three distinct entities, which are the trusted data owner, the remote storage server, assumed to be semi-honest, and the data users. The scheme assures that the log records can only be produced by the data owner and that unauthorised parties cannot forge or tamper with them without being detected.

All data users must be previously approved by the data owner and each operation performed by them is logged. After the data owner has outsourced his data to the remote storage server, the sharing and access to that data by data users is possible. Each data user must request access to the data owner. If granted, the data owner produces a searchable encrypted audit log that comprises an access ticket, an encrypted authorisation for a specific data user. The access ticket includes information about the data user's identity, the type of operations he is allowed to perform and the expiration date of the authorisation. The encrypted audit log is then submitted to the server, which grants access to the data user if the log verification is positive. The audit log is stored by the server. In parallel, the data owner

is able to query about what operations are being performed over his data. To do that, a trapdoor of keywords such as ”who/when/where/what” is generated and submitted to the server that returns any matching encrypted audit log records.

The proposed solution is initialised with a setup phase, on which two random master keys are computed. The generation of a searchable encrypted audit log l for a specific access request R starts by the selection of two random numbers s, r, used on hash and encryption operations. Then, the access ticket is computed using an hash function that contains: the identity and the IP address of the requesting data user, the request expiration date, and information regarding the type of operation desired to be executed over the data. This information is then used on the construction of the keyword set that is appended to the encrypted audit log and enables the server to answer each one of the ”who/when/where/what” data owner queries. The final searchable encrypted audit log is passed to the data user who presents it to the remote storage server for data access grant. The server confirms the validity of that audit log properties and grants the access if the validation is positive.

When the data owner desires to query his encrypted audit logs, it creates a trapdoor of the keywords of interest by applying the same algorithm used on the audit log generation. The master keys are used on this generation, assuring that no one besides the data owner can create search trapdoors. Upon receiving such trapdoors, the server tests them against each stored log record. All matching results are returned to the data owner, whom, after assuring the validity and integrity of such results, decrypts them and obtains the information regarding the kind of access that has been done over his data.

3.13 Comparison

Name	Confidentiality	Integrity	Authenticity	Simple Search	Conjunctive Search	Nested Search	Field Search
Schneier	Yes	Yes	Yes	No	No	No	No
Forte	No	Yes	Yes	No	No	No	No
Holt	Yes	Yes	Yes	No	No	No	No
Ma	Yes	Yes	Yes	No	No	No	No
Ray	Yes	Yes	Yes	No	No	No	No
Zawoad	Yes	Yes	No	No	No	No	No
Waters	Yes	Yes	No	Yes	No	No	No
Ohtaki	Yes	No	No	Yes	Yes	No	No
Sabbaghi	Yes	No	No	Yes	Yes	No	No
Accorsi	Yes	Yes	Yes	Yes	No	No	No
Savade	Yes	Yes	No	Yes	No	No	No
Zhao	Yes	Yes	Yes	Yes	Yes	No	No

Table 3.1: Comparison of the related work

Table 3.1 compares the identified related work. Except Forte et al. [98], whose scheme is focused

on the transmission phase, all identified solutions enable data confidentiality and privacy by encrypting the log data prior to its storage. Almost all solutions enable data integrity and authenticity by the use of a cryptographic hash chain. Only Ma et al. [104] devises a different approach for data integrity and authentication with the development of the authentication technique FssAgg.

Regarding encrypted log searching, the solution proposed by Waters et al. [114], although source of inspiration for subsequent solutions, only supports single keyword search and has both high computational and high storage costs. The solutions proposed by Accorsi et al. [122] and Savade et al. [123] while being more computational and storage efficient, only allow for single keyword search. Ohtaki et al. [118] enhances the search capability with support for boolean queries, more precisely for the “AND” and “OR” operators. Nevertheless, this boolean queries are achieved not by performing boolean operations but by adding extra searchable indexes. For instance, in order to able to perform the query “A and B”, an index of such pattern is required to have been previous computed. Ohtaki uses Bloom filters that admit the occurrence of false positives that might disclose more log records than the ones that are relevant for the executed query. Sabbaghi et al. [119] also provides the execution of conjunctive queries, however their work is fully focused on SQL commands. Hence the keywords may only be searched within the SQL clauses. The solution designed by Zhao et al. [125] only allows searches to be performed over the metadata appended to each log record and only to answer the ”who/when/where/what” questions.

None of the presented solutions supports all the boolean operators nor fine-grained operations like field search and nested queries. Field search can be used to search for a value on a specific part of the log record. For instance, if one wants all log entries from November, fielded search enables the non-occurrence of false positives by not matching log entries that contains the keyword ”November” in any other part of the log. Nested queries can be used to retrieve log records, for example, from the month of November but originating from a specific set of IP addresses. Moreover, none of the solutions proposed in the related work offer all the desired security properties alongside an advanced searching capability.

Chapter 4

Secure Logging Service

With the advent of cloud platforms that both house the applications and their logs, secure and confidential remote logging appears as a crucial issue to address. When remote log confidentiality is required, the most common solution is to use cryptography techniques to encrypt all data before transferring it to a remote cloud storage service. If searching within this remotely stored logs is required, the simplest and trivial approach consists in transferring all data back to the client, so that it can be decrypted, allowing search operations to be performed over the clear text and at the client side. Despite the data privacy and confidentiality guarantees offered by this approach, it can rapidly become impractical with the normal growth of log data. Moreover, this approach does not make use of the full potential of cloud computing.

The reference scenario adopted is presented in Figure 1.1 on Chapter 1. It describes a cloud-based secure log storage service. The Client makes use of an API to transfer its encrypted logs to the cloud, named SLaS Provider. In parallel, the Client can make use of the same API to query, in a encrypted form, its logs and retrieve matching records. A SLaS application is foreseen in order to make use of the cloud's potential and to enable the transfer of some CPU-intensive tasks from the Client to the SLaS Provider. The SLaS Provider stores the client-side encrypted logs in its database and performs search operations on them, without having access to clear text logs.

The intention of this chapter is the proposition of the solution presented in Chapter 1. First, the system requirements are enumerated, followed by the presentation of the system model. Next, we describe the architecture of all the elements and their functionality. Concluding this chapter, the application protocols are described.

4.1 System Requirements

In order to enumerate the system requirements and to aid on the development of a solution, the following scenarios were considered: 1) an authorised entity desires to transfer, in an encrypted form, its operational logs to the cloud; 2) an authorised entity desires to perform search operations on such logs; 3) an unauthorised entity tries to tamper the remote stored logs; 4) an unauthorised entity tries to obtain information regarding the stored logs. Based on that, the following requirements were identified:

1. Privacy Preserving - the privacy and confidentiality of log records content must be protected, since such logs contain sensitive information;
2. Authentication - it must be only possible for authenticated entities to submit log records for storage. Additionally, it must be unfeasible to any unauthorised party to generate and index valid encrypted log records;
3. Verifiability - it must be possible to verify each individual log record integrity and authenticity;
4. Forward Integrity - each log record must be linked together in such way that makes it possible to verify if any of those records was modified or deleted;
5. Search - it must be possible to search within the encrypted log records and retrieve matching results;
6. Query Confidentiality - it must be possible to submit queries in an encrypted form, perform the search over encrypted data and retrieve encrypted matches.
7. Query Expressiveness - it must be possible to submit complex and fine-grained queries to obtain the most accurate results.
8. Reduced Log Size - it must be possible to store searchable encrypted log records in the most secure way using the low storage space required;
9. Logging Speed - it must be possible to store log records in a secure way with acceptable speed and without compromising the normal operation of the source applications;
10. Search Speed - it must be possible to perform elaborated search queries and retrieve matching records in useful time;

The privacy of logs records (Requirement 1) can be achieved with the use of encryption. The log records must be encrypted not only at rest but also during its transfer to the remote storage server. The Requirements 8, 9 and 10 may suggest the adoption of symmetric cryptography techniques since symmetric key algorithms typically require less computational power and smaller storage space.

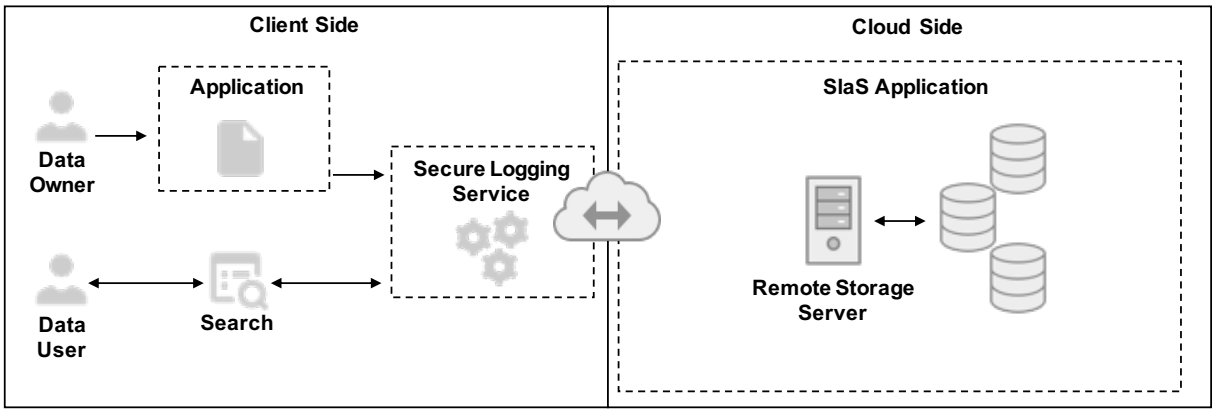

Figure 4.1: Proposed system model

Requirement 2 can be fulfilled through an authentication system in which both the data owner and the remote storage server authenticate each other, prior to any log record or search query. Additionally, the data owner must sign all requests, because the server will only process requests with a valid signature. The responses of the server must also be signed and verified by the data owner prior its consideration.

The satisfaction of Requirements 3 and 4 can be realised by the computation of hash functions. More specifically, the use of keyed hashing techniques assure that not only the integrity of each log record is preserved but also its authenticity (Requirement 3). The forward integrity (Requirement 4) is attained by the construction of an hash chain, where the successive computation of HMACs makes possible to detect any unauthorised modification in the log sequence.

The searchability (Requirement 5) can be addressed by parsing each log record for the extraction of the keywords and its conversion to searchable encrypted trapdoors. To assure Requirement 6, the trapdoors are generated on the client side and submitted to the server only in encrypted form. The server tests its presence over encrypted log records, returning the ones that match the query. A query language can be used to address the query expressiveness (Requirement 7) and to support conjunctive keyword search, field search and nested queries.

Alongside all the requirements listed above, there is a non functional requirement that the proposed solution shall meet. That requirement is security. This is, the proposed solution shall be secure against unauthorised, accidental or unintended threats and provide access only to authenticated entities.

4.2 System Model

The system model of the proposed solution is depicted in Figure 4.1. The system is physically divided between the client side, where the log records generation occur and the consequent search operations are originated, and the cloud side, where the searchable encrypted log records are stored and retrieved. The client side contains the business applications from which the logs are forwarded to the Secure

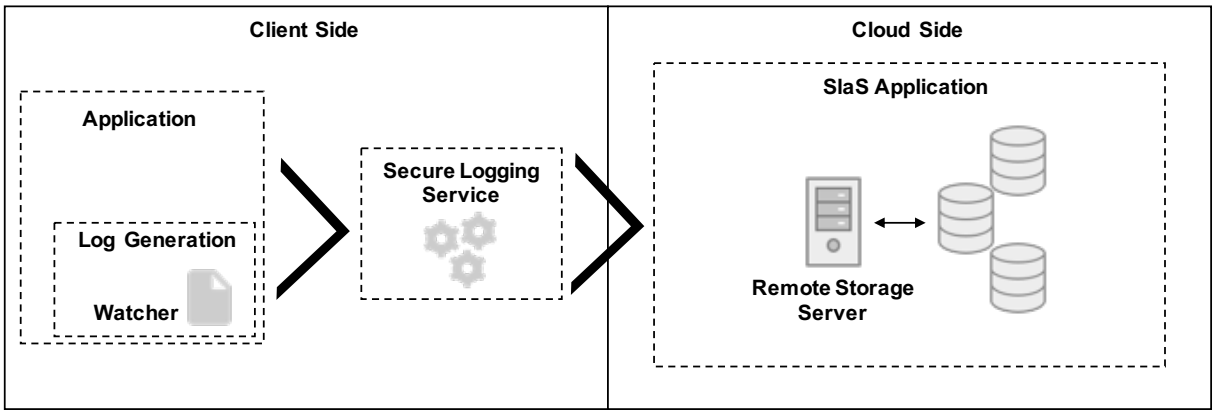

Figure 4.2: Flow of data within the pipeline

Logging Service (SLS). These logs, after conversion to searchable encrypted ones, are transferred over to the cloud side. The SLaS application will securely store them on the cloud side. The search operations are also originated on the client side, using the SLS, sent over to the SLaS application, which returns the encrypted matching results to the client side.

Figure 4.1 depicts the three involved entities of the proposed solution, these being the data owner, the data user, and the remote storage server. The data owner possesses a collection of log records $L1, L2, ..., Li$, produced by his applications, that shall be securely stored on the remote storage server. To assure the desired confidentiality, integrity and authenticity, all the necessary cryptographic operations are performed by the data owner prior to transferring the logs to the cloud. To enable search and retrieval operations of such log records, the data owner also devises a set of searchable encrypted keywords, sent alongside the log records. All this information is sent over to the remote storage server, that additionally to the provisioning of a secure storage, receives encrypted search queries, tests them against the stored data and returns the matching results. The data user represents the entity that is authorised to search the remote stored data for the keywords of interest. On the proposed solution, we assume that the data owner and the data user are the same entity.

4.2.1 Data Pipeline

On the storage phase, the proposed solution is envisioned as a part of a data pipeline, as shown in Figure 4.2. This pipeline is initialised by the harvesting of log records produced by applications of the data owner. Typically, these applications write their operational logs to file, thus, the harvesting of such logs is performed through file watchers. These file watchers serve as plugins for such applications that watch for changes of the log files, raising events every time a new log line is added. The events are then handled by the SLS, that receives clear text log records, transforms them into encrypted searchable data and sends them to the remote storage server.

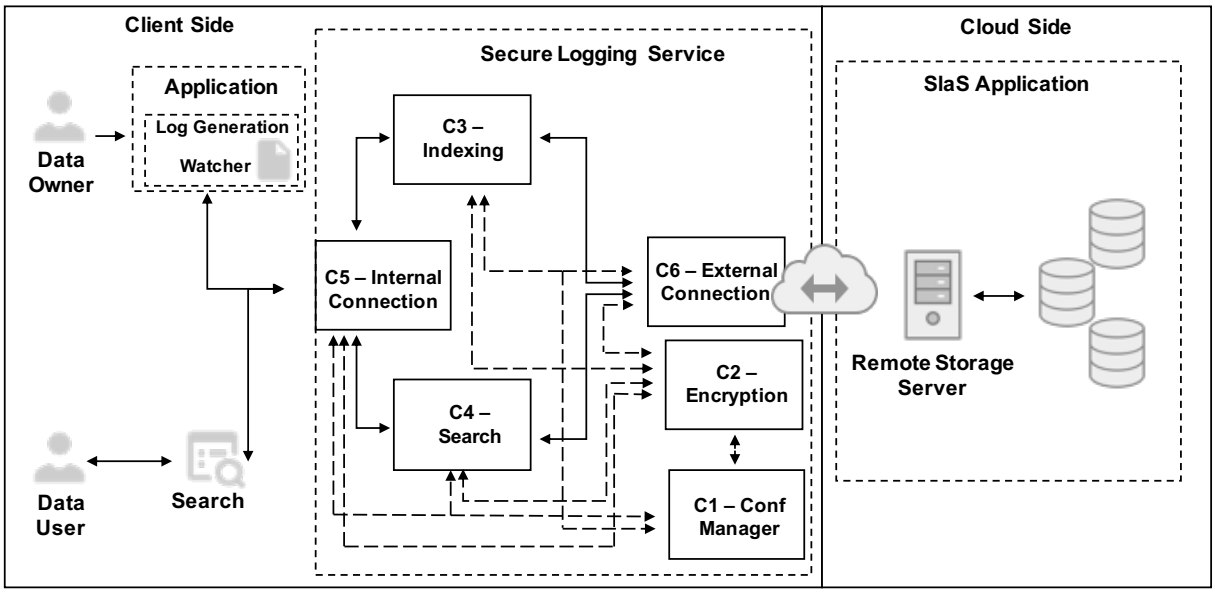

Figure 4.3: Architecture of the proposed solution

Although the transferred events include log records in clear text, the communication between the file watchers and the logging service is performed through encrypted channels, for which only the two involved parties possess the correct decryption keys and are able to see the information in transit. Moreover, only file watchers that are previously registered with the SLS are able to submit log records. That submission is performed in an authenticated manner, that makes unfeasible for unauthorised parties to submit bogus log records. Additionally, the authorisation of any file watcher can be revoked at any time by the SLS.

4.3 Architecture

The SLS architecture is composed by several components. Each one of this building blocks has a very specific role on the operation of the proposed solution. Figure 4.3 depicts the six components (C1 to C6): the Conf Manager, the Encryption, the Indexing, the Search, the Internal Connection and the External Connection component. Figure 4.3 also illustrates, through the use of arrows, the interaction between that components. The filled arrows represent the communication that exists during the two main operations of the solution, namely the indexing and search. The dashed arrows represent the interactions between components required during the execution of that two main operations.

Conf Manager

The Conf Manager (C1) component is the one responsible for managing all configuration properties of the SLS. Moreover, it is in charge of the secure computation and availability of all cryptographic keys required by the SLS. It generates secure random keys with a previously agreed size and makes them available to the remaining components. These components, in turn, will use them in encryption, decryption, digest and communication operations. We assume that the keys and all the remaining configuration properties are saved on an controlled location, only accessible by the adjacent components using a secure channel. Moreover, we consider that the cryptographic keys always remain is the possession of the data owner and are not shared with any entity external to the described scenario.

Encryption

The Encryption (C2) component is responsible for all the encryption and decryption operations performed throughout the normal operation of the SLS. Thus, this component acts as a dependency of all the remaining ones. In detail, it collaborates with the Indexing component in order to encrypt all the log records that are sent for storage on the remote storage server. Furthermore, this component decrypts all the information fetched from the remote storage server during search operations. Additionally, the Encryption component assists the Internal and External Connection components on the execution of the required cryptographic operations during the communication between the SLS, the file watchers and the remote storage server.

Indexing

The Indexing (C3) component is in charge of the transformation of the clear text log records into encrypted and searchable log records. It is envisioned as a node of a data pipeline, acting as a proxy, that receives, as input, clear text log records produced by applications, transforms them into encrypted searchable data and outputs it to the remote storage server that will store it. On the cloud side, the server is not only responsible for the storage of the encrypted log records but also for updating the encrypted inverted index to include the search terms that are continuously being submitted by this component.

Search

The Search (C4) component is the one responsible for submitting queries to the remote storage server and for the retrieval of matching log records. In short, it receives, as input, search terms that are submitted by the client, applies the required transformations to convert them into searchable trapdoors, and then, submits the transformed search terms to the remote storage server. On the server side, the searched terms are looked up on the encrypted inverted index and the matching log records are

returned. After which, the matching log records are returned to the client that will send them to the Encryption component that, in turn, will decrypt them.

Internal Connection

The Internal Connection (C5) component is the only one open to communication with other systems on the client side. In detail, it is accountable for assuring a secure and authenticated communication between the multiple file watchers and the SLS. For every log record submitted for storage this component will verify the request source trustworthiness and authenticity. If that validation is successful the Internal Connection component will forward the log record to the Indexing component. The search requests are also originated by this component. If such requests are compliant with the proposed solution security requirements, the Internal Connection will forward the search requests to the Search component.

External Connection

The External Connection (C6) component represents the bridge between the client side and the cloud side. This component acts as a dependency of the Indexing and Search components, since it is the only possible gateway for log records to be stored and search queries to be performed. The External Connection component deals with the complexity of the cloud side connection and ensures the establishment of a secure channel between the two different sides of the proposed solution scenario. Furthermore, the External Connection component enables an authentication protocol that assures that the SLS and the SLaS Application are authentic and trustworthy. Additionally, this component makes the adoption and integration with different cloud providers more seamless.

4.4 System Design

Prior to the description of the inner workings of the proposed solution, it is important to describe the notation adopted throughout this work, which is depicted in Table 4.1. SLS represents the proposed Secure Logging Service. $SLaS$ represents the SLaS application, running on the cloud side and storing and retrieving log records. LS represents a log source. X_i represents the unique identifier generated for entity X. X_{yK} represents the secret key used for operations of type y by entity X. X_{pubK} and X_{privK} represents, respectively, the public and private key of entity X. P represents a payload, such as a log record or a search request, and PE represents the encrypted version of P. H_P represents the HMAC produced for P. S_P represents a signature produced for P. TS represents a timestamp. $A + B$ represents field A concatenated with field B. $[A, B]$ represents the serialisation of fields A and B in order to be stored. $\{A : B\}$ represents a key value tuple, where A is the key and B is the value.

SLS	SLS
$SLaS$	SLaS application
LS	Log Source
X_i	Identifier of entity X
X_{yK}	Secret key of entity X for operation Y
X_{pubK}	Public key of entity X
X_{privK}	Private key of entity X
P	Payload
PE	Payload encrypted
H_P	HMAC of P
S_P	Signature of P
TS	Timestamp
$A + B$	Concatenation of fields A and B
$[A, B]$	Serialization of fields A and B, for storage
$\{A : B\}$	Key: Value tuple

Table 4.1: Adopted notation

4.4.1 Initialisation

The initialisation process comprises two distinct actions. The first is the setup of the SLS itself and the second is the setup conducted for every source of log records (LS) integrated with the SLS. On the design of the solution we assume that a source of log records corresponds to a file watcher that only forwards data from one client application. That file watchers must be previously authorised by the SLS to send log records.

The setup of the SLS starts by the computation of a pair of asymmetric keys ($SLSe_{pubK}$ and $SLSe_{privK}$) to assure a confidential and authenticated communication between the SLS and the SLaS Application. The SLaS Application also computes a set of asymmetric keys ($SLaS_{pubK}$ and $SLaS_{privK}$) and performs an handshake with the SLS. This handshake consists in the exchange of the public keys of both entities in order to implement an authentication system between them. Additionally, a symmetric key SLS_{kK} is randomly generated by the SLS to be used in the encrypted trapdoors creation.

The setup of a LS stars by the configuration of its properties. If the log records follow a fixed structure, and thus allow field search, a regular expression (LS_{regex}) can be configured in order to extract specific information from each log record. Alongside, a mapping of the extracted fields (LS_{map}) must be added. That mapping includes, for each field, a name that is used on the client side and a unique identifier that is applied on the cloud side and does not reveal the type of each field. If the log records do not follow a fixed structure, the SLS will split each log record by a delimiter (e.g. spaces). Hence, a delimiter (LS_{dlm}) must be configured. Additionally, a set of blacklist characters (LS_{blk}) is added to allow the SLS to remove unwanted characters and sanitize the log records prior to its storage. Finally a name (LS_{name}) and a description (LS_{desc}) is also setted for the log source, in order to facilitate its identification by the data owner.

The following step of the LS setup is the generation of a unique identifier (LS_i) for that specific LS. Also, the computation of the necessary cryptographic keys is performed. A pair of asymmetric keys ($SLSi_{pubK}$ and $SLSi_{privK}$) is generated and will be used to assure confidentiality between the file watcher and the SLS. Then, three symmetric keys are computed. The first key, LS_{aK}, is used for the calculation of HMACs that authenticate the file watcher against the SLS. The second key, LS_{hK}, is used to compute each log record HMAC, creating an hash chain for integrity and authenticity validations. The third key, LS_{eK}, is used to compute an encryption key K_{Ln} for each log record. A random seed value LS_{seed} is also computed in order to initialise the LS hash chain. LS_i, $SLSi_{pubK}$ and LS_{aK} are shared with the file watcher.

4.4.2 Communication

The proposed SLS devises two communication protocols to assure confidentiality, integrity and authenticity of data in transit and to prevent unauthorised accesses to the system. The Internal Communication protocol is applied on the connection established between the file watchers and the SLS. The External Communication protocol is used for data transmission between the SLS and the SLaS Application. The former protocol assumes that the communication is always performed on the premises of the client. The latter predicts a connection between the client side and the cloud side. Moreover, all communications between the involved parties are performed over Secure Socket Layer (SSL)/Transport Layer Security (TLS) connections.

Internal Communication

Algorithm 4.1: File watcher internal communication algorithm

Input : L_n
Output: LE_n, LS_i

1 $TS \leftarrow$ generateTimestamp()
2 $AT \leftarrow$ HMAC.create(LS_{aK}, $L_n + TS + LS_i$)
3 $LE_n \leftarrow SLSi_{pubK}$.encrypt([$L_n, TS, AT$])
4 SLS.send(LE_n, LS_i)

Algorithm 4.1 describes the process executed by the file watchers for each log record they desire to submit to the SLS. The first step (step 1) is the generation of a timestamp TS, which indicates when the log record is submitted for storage and prevents replay attacks. This timestamp is used also as input to the computation of the authentication tag (AT), assuring the indistinguishability between two tags of two distinct log records whose content is the same. Additionally, the inclusion of the log source identifier (LS_i), unique for every file watcher, makes that AT indistinguishability between two equal log

Algorithm 4.2: SLS internal communication algorithm

Input : LE_n, LS_i
Output: L_n, TS, LS_i

1 **if** *LogSources.includes(LS_i)* **then**
2 $LS \leftarrow$ LogSources.get(LS_i)
3 $L_n, TS, AT \leftarrow LS.SLSi_{privK}$.decrypt($LE_n$)
4 logVerified $\leftarrow$ HMAC.verify($LS.LS_{aK}$, $L_n + TS + LS.LS_i$, AT)
5 **if** *logVerified* **then**
6 Index(L_n, TS, $LS.LS_i$)
7 **end**
8 **end**

records of two different file watchers. AT, which assures that only authorised parties are capable of submitting log records for storage, is obtained by the computation of an HMAC using the key LS_{aK} (step 2). Next, the log record L_n, alongside TS and AT are serialised together and asymmetrically encrypted with $SLSi_{pubK}$ key (step 3), resulting in LE_n. LE_n and LS_i are sent over to the SLS (step 4), where the Algorithm 4.2 is executed.

Algorithm 4.2 initialises with the verification of the existence of the requesting log source (i.e. file watcher) LS in the SLS known log sources (step 1). If such verification is successful the log source properties are loaded (step 2). Next, with $SLSi_{privK}$ key, the received data is decrypted and deserialized (step 3). The SLS, then, validates the received AT (step 4), which allows not only to authenticate the log source but also to assure the integrity of the received information. If this validation is successful (step 5), the received data is forwarded for indexing and consequent storage (step 6). If any of the validations performed within this algorithm is unsuccessful the received information is discarded.

Algorithms 4.1 and 4.2 describe the communication process conducted by file watchers during log transmission for indexing and storage. Search operations, carried out by the data user, are assumed to be performed directly on the SLS, thus a communication protocol for such operations is not envisioned.

External Communication

Communication between the SLS and the SLaS Application happens during indexing and searching operations. The Algorithms 4.3 and 4.4 are applied to both operations, since they are agnostic to the type of information that is being sent.

Algorithm 4.3 describes the actions performed by the Sender entity. In other words, this algorithm is executed by the SLS in indexing and search requests and performed by the SLaS Application in the corresponding responses. The first step of this algorithm is the generation of a secure random symmetric

Algorithm 4.3: Sender external communication algorithm

Input : P_n
Output: PE_n, KE_{Pn}, S_{Pn}

1 $K_{Pn} \leftarrow$ generateSecretKey()
2 $PE_n \leftarrow K_{Pn}$.encrypt(P_n)
3 $KE_{Pn} \leftarrow Rec_{pubK}$.encrypt($K_{Pn}$)
4 $H_{Pn} \leftarrow$ hash([PE_n, KE_{Pn}])
5 $S_{Pn} \leftarrow Send_{privK}$.sign($H_{Pn}$)
6 $SLaS$.send(PE_n, KE_{Pn}, S_{Pn})

Algorithm 4.4: Receiver external communication algorithm

Input : PE_n, KE_{Pn}, S_{Pn}
Output: -

1 $H_{Pn} \leftarrow$ hash([PE_n, KE_{Pn}])
2 logSignValid $\leftarrow Send_{pubK}$.verify(H_{Pn}, S_{Pn})
3 **if** *logSignValid* **then**
4 $K_{Pn} \leftarrow Rec_{privK}$.decrypt($KE_{Pn}$)
5 $P_n \leftarrow K_{Pn}$.decrypt(PE_n)
6 Process(P_n)
7 **end**

key K_{Pn}, valid only for one communication trip (step 1). K_{Pn} is used to encrypt the payload P_n (step 2). K_{Pn} is also encrypted (step 3) with the receiver public key Rec_{pubK} (on a request Rec_{pubK} is $SLaS_{pubK}$, on a response Rec_{pubK} is $SLSe_{pubK}$). Afterwards, an hash H_{Pn} of the serialisation of PE_n and KE_{Pn} is computed (step 4) and signed (step 5) with the sender's private key $Send_{privK}$ (on a request $Send_{privK}$ is $SLSe_{privK}$, on a response $Send_{privK}$ is $SLaS_{privK}$) resulting in S_{Pn}. Finally, the produced signature S_{Pn} alongside PE_n and KE_{Pn} are sent over to the receiver.

Upon information arrival, the receiver entity executes Algorithm 4.4. First, it computes the hash (H_{Pn}) of the received PE_n and KE_{Pn} values (step 1). H_{Pn} is then used as input for the verification of the received signature (step 2). That signature verification is performed with the sender's public key $Send_{pubK}$ (on a request $Send_{pubK}$ is $SLSe_{pubK}$, on a response $Send_{pubK}$ is $SLaS_{pubK}$). If such signature is valid (step 3), K_{Pn} is decrypted (step 4) with Rec_{privK} (on a request Rec_{privK} is $SLaS_{privK}$, on a response Rec_{privK} is $SLSe_{privK}$). Next to obtaining K_{Pn}, the received PE_n is also decrypted (step 5) and processed (step 6). If the signature validation fails, the algorithm stops and the received information is discarded.

For external communication both a hybrid encryption and an authentication schemes are applied. The hybrid encryption scheme uses a symmetric key to encrypt log data, since symmetric key algorithms are usually faster and produce smaller ciphertexts. Nevertheless, in order to decrypt the message, this key must be sent over to the other involved party, which is accomplished by the encryption of such key with an asymmetric key algorithm. This way, the confidentiality of the data in transit between the client and the cloud side is assured.

The authentication technique is achieved by the use of asymmetric computed signatures. Both entities possess a pair of keys and share between them the public key. This way, one entity authenticates the other by validating the signatures received. We assume that the private keys always remain in control of their owners, so the signatures produced by them validates their identity. Moreover, since the signatures are computed over the content of each communication process, the verification of such signatures also assures the integrity of the information in transit.

4.4.3 Indexing

The Indexing operation is carried out by the SLS and for each log record sent from the file watchers. The log data undergoes a series of cryptographic operations which transforms it into searchable encrypted information, as described in Algorithm 4.5. This algorithm can be divided into three different sections, each one in charge of different log security properties. From line 2 to line 4, the privacy and confidentiality of log records content is protected. Line 5 assures each individual log record integrity and authenticity. Lines 6 and 7 achieve Forward Integrity, enabling the verification of changes or removal of any log records. The rest of the algorithm adds the ability to search within the encrypted log records.

Algorithm 4.5: SLS indexing algorithm

Input : L_n, TS, LS_i
Output: L_{ni}, LE_n, H_{Ln}, HC_{Ln}, T_{Ln}, LS_i

$LS \leftarrow$ LogSources.get(LS_i)
$L_{ni} \leftarrow$ generateId()
$K_{Ln} \leftarrow$ HMAC.create($LS.LS_{eK}$, L_{ni})
$LE_n \leftarrow K_{Ln}$.encrypt([L_n, TS])
$H_{Ln} \leftarrow$ HMAC.create($LS.LS_{hK}$, $LE_n + LS_i + L_{ni}$)
$HC_{Ln} \leftarrow$ HMAC.create($LS.LS_{hK}$, $H_{Ln} + LS.HC_{Ln-1}$)
$LS.HC_{Ln-1} \leftarrow HC_{Ln}$
$T_{Ln} \leftarrow$ []
if *exists*($LS.LS_{regex}$) **then**
    $L_{nparsed} \leftarrow LS.LS_{regex}$.parse($L_n$)
    for *i* $\leftarrow$ *0* **to** *length*($L_{nparsed}$) **do**
        $L_{nfield} \leftarrow L_{nparsed}[i]$
        $L_{nfieldId} \leftarrow LS_{map}$.convert($L_{nfield}$)
        $L_{ntrapdoor} \leftarrow$ HMAC.create(SLS_{kK}, L_{nfield})
        T_{Ln}.add({$L_{nfieldId}$, $L_{ntrapdoor}$})
    end
else
    $L_{nsanitized} \leftarrow LS_{blk}$.sanitize($L_n$)
    $L_{nparsed} \leftarrow LS_{dlm}$.split($L_{nsanitized}$)
    for *i* $\leftarrow$ *0* **to** *length*($L_{nparsed}$) **do**
        $L_{nfield} \leftarrow L_{nparsed}[i]$
        $L_{ntrapdoor} \leftarrow$ HMAC.create(SLS_{kK}, L_{nfield})
        T_{Ln}.add({$L_{ntrapdoor}$})
    end
end
$SLaS$.index(L_{ni}, LE_n, H_{Ln}, HC_{Ln}, T_{Ln}, LS_i)

The first instruction of Algorithm 4.5 is the attainment of LS_i properties (step 1). Next, a unique identifier L_{ni} is generated for log record L_n (step 2). This identifier forms the basis of the K_{Ln} computation (step 3), the key specifically used to encrypt the entire log record L_n alongside its timestamp TS (step 4). Afterwards, an HMAC H_{Ln} of the concatenation of LE_n, LS_i and L_{ni} is calculated (step 5) in order to maintain both the integrity and authenticity of that specific log record. The obtained H_{Ln} is then used to produce the current log record hash chain link HC_{Ln} (step 6). HC_{Ln} consists in an HMAC of H_{Ln} concatenated with the previous log record hash chain link HC_{Ln-1}. HC_{Ln} is setted as the HC_{Ln-1} of that LS, which will be used on the subsequent indexing operations (step 7).

Following, the algorithm moves on to the preparation of the search capability. First, it creates an empty set of trapdoors T_{Ln} for that log record (step 8). T_{Ln} is populated with one of two ways. If the log record follows a fixed structure and LS has a regular expression LS_{regex} configured (step 9), the log is parsed by that. If not, the log record is splitted by a pre-configured delimiter LS_{dlm} (step 17). More specifically, if LS_{regex} exists, for each extracted field L_{nfield} of the log record, a lookup operation is performed (step 13) in order to obtain the identifier $L_{nfieldId}$ of such field. Then, the trapdoor $L_{ntrapdoor}$ for that specific field is built by computing an HMAC, enabling its future search (step 14). Finally, a tuple containing $L_{nfieldId}$ and $L_{ntrapdoor}$ is added to T_{Ln} (step 15). If no LS_{regex} is configured, the log record is sanitized by the removal of the characters present in the configured blacklist LS_{blk} (step 18) and splitted by LS_{dlm} (step 19). Each part L_{nfield} of the splitted log corresponds to a keyword (step 21), thus, $L_{ntrapdoor}$ for each of that keywords is built by computing an HMAC (step 22), which is then added to T_{Ln} (step 23), similar to what is done in the regular expression parsing mode.

In order to have a more clear understanding of the previously explained search capability, assume the reception of the following log record L_n, representing a common request to an HTTP web server.

L_n: **172.217.17.3 - - [12/Oct/2019:15:25:15 -0800] "GET /home HTTP1.1" 200 2314**

L_n is assumed to be sent by the log source $LS_i = GS0XE0Fe$, whose properties are listed in Table 4.2. Since, L_n follows a fixed structure and the specified LS has a regular expression LS_{regex} configured, the data is parsed by that LS_{regex}. This way, L_n is transformed in the following L_{parsed}:

$L_{nparsed}$: **[172.217.17.3, 12/Oct/2019:15:25:15, GET, /home, 200]**

On L_{parsed}, the fields are organized like the following:

[ip, date, method, path, status]

Afterwards, the mapping LS_{map} of such fields is applied, converting the name of the fields to the correspondent identifiers $L_{nfieldId}$. Also, the trapdoors $L_{ntrapdoor}$ for each field value are computed. The tuples containing $L_{nfieldId}$ and $L_{ntrapdoor}$ are then used to create the log record trapdoor list T_{Ln}, like Table 4.3 exemplifies.

LS_i	GS0XE0Fe
LS_{name}	web
LS_{desc}	Web Server 1
LS_{regex}	(.*)\s-\s-\s[(.*)-\d{4}]\s\"(GET|POST)\s(.*)\sHTTP1.1\"\s(\d{3})
LS_{map}	$name$ ip id hEzCbMn7u $name$ date id CxtZ8vVUy $name$ method id C2nyVaqx1 $name$ path id UQdYgEtoZ $name$ status id zL4r5Gbul
LS_{dlm}	\s
LS_{blk}	][\"-
LS_{seed}	MzA1OTY4YTQtMDg0Zi00Yjgx
LS_{hK}	Y2JjMWFkOWEtMzliNi00ZmE1
LS_{eK}	ODgwNWZmNGItODA4ZS00NWE5
HC_{Li-1}	Njc5N2I2YmQtN2QxNS00Ymtr
$SLSi_{pubK}$	OGQ4MzYxYWEtOWMyOC00LprT
$SLSi_{privK}$	ZDQyMmI1YjYtNTgxNS00RTee

Table 4.2: Log source properties example

$L_{nfieldId}$	hEzCbMn7u	$L_{ntrapdoor}$	YWJiYWI4NGYtOTcyZi
$L_{nfieldId}$	CxtZ8vVUy	$L_{ntrapdoor}$	NTdiZjZjZmYtOGQyNS
$L_{nfieldId}$	C2nyVaqx1	$L_{ntrapdoor}$	ZTgyMjQzYTYtNmZhYS
$L_{nfieldId}$	UQdYgEtoZ	$L_{ntrapdoor}$	ZWEwYjg4MmQtZGZmMC
$L_{nfieldId}$	zL4r5Gbul	$L_{ntrapdoor}$	Y2QyNTg1NTctYjE0NC

Table 4.3: Log record trapdoor list with regular expression example

If we assume that no LS_{regex} is present, L_n is first sanitized by the removal of blacklist characters $LS_{blk} =$] [" -, resulting in the following $L_{nsanitized}$:

$L_{nsanitized}$: **172.217.17.3 12/Oct/2019:15:25:15 0800 GET /home HTTP/1.1 200 2314**

$L_{nsanitized}$ is then splitted by a white space, the configured delimiter LS_{dlm} for the specified LS producing the following $L_{nparsed}$:

$L_{nparsed}$: **[172.217.17.3, 12/Oct/2019:15:25:15, 0800, GET, /home, HTTP/1.1, 200, 2314]**

Since LS_{regex} is applied, a field mapping conversion is not necessary, thus only a HMAC computation for each keyword to produce $L_{ntrapdoor}$ occurs, producing the trapdoors list T_{Ln} exemplified on Table 4.4:

$L_{ntrapdoor}$	YWJiYWI4NGYtOTcyZi
$L_{ntrapdoor}$	YTkzNjBkNjUtZmFlNS
$L_{ntrapdoor}$	ZGExNDYwYmYtMWIwNi
$L_{ntrapdoor}$	ZTgyMjQzYTYtNmZhYS
$L_{ntrapdoor}$	ZWEwYjg4MmQtZGZmMC
$L_{ntrapdoor}$	ZmEwODM5ZjctOTA5NS
$L_{ntrapdoor}$	Y2QyNTg1NTctYjEONC
$L_{ntrapdoor}$	ZGUONGFiZGMtMzUOMC

Table 4.4: Log record trapdoor list without regular expression example

This information is, afterwards, transferred to the cloud, where the SLaS Application will apply Algorithm 4.6. This algorithm starts with the storage of L_{ni}, LE_n, H_{Ln}, HC_{Ln} and LS_i (step 1) on the SLaS Application database DB. This operation produces a timestamp $TS_{storage}$ indicating when the data was officially stored, which is returned on the end of the algorithm to the SLS as an acknowledgement of the operation. Afterwards, the encrypted inverted index $InvIndex$, which forms the basis of the forthcoming search operations, is updated. The content of each received trapdoor tuple $T_{Ln}[i]$ is analysed. If $T_{Ln}[i]$ includes the field identifier $L_{nfieldId}$ (step 3), the $InvIndex$ entry for the combination of $L_{nfieldId}$ and $L_{ntrapdoor}$ is affected. If $T_{Ln}[i]$ is composed by $L_{ntrapdoor}$ alone, only the $InvIndex$ entry for $L_{ntrapdoor}$ is influenced (step 9). In both situations, a verification is executed to find if $T_{Ln}[i]$ already exists on $InvIndex$ (steps 4 and 10). If this verification is successful, the respective L_{ni} is added to the set of the already existing identifiers (steps 5 and 11). If not, a new entry for $T_{Ln}[i]$ is created and initialised with the respective L_{ni} (steps 7 and 13).

To better understand how the information is handled and stored by the SLaS Application, let's assume that the structure present in Table 4.5 arrives at the SLaS Application. Upon receiving such structure, the SLaS Application starts by adding it to its DB, the log identifier L_{ni} with its encrypted content LE_n, HMAC tag H_{Ln}, hash chain link HC_{Ln} and log source identifier LS_i, as depicted in Table 4.6. Then, the log record trapdoor list T_{Ln} is created and/or updated on the encrypted inverted index $InvIndex$, as exemplified by Table 4.7.

Algorithm 4.6: SLaS Application indexing algorithm

Input : L_{ni}, LE_n, H_{Ln}, HC_{Ln}, T_{Ln}, LS_i
Output: L_{ni}, $TS_{storage}$

$TS_{storage} \leftarrow DB$.save($L_{ni}$, LE_n, H_{Ln}, HC_{Ln}, LS_i)
for *i* $\leftarrow$ *0* **to** *length*(T_{Ln}) **do**
  if *exists*($T_{Ln}[i].L_{nfieldId}$) **then**
    if $InvIdx$.*includes*($T_{Ln}[i].L_{nfieldId}, T_{Ln}[i].L_{ntrapdoor}$) **then**
      $InvIdx$.update($T_{Ln}[i].L_{nfieldId}, T_{Ln}[i].L_{ntrapdoor}$, L_{ni})
    else
      $InvIdx$.create($T_{Ln}[i].L_{nfieldId}, T_{Ln}[i].L_{ntrapdoor}$, L_{ni})
    end
  else
    if $InvIdx$.*includes*($T_{Ln}[i].L_{ntrapdoor}$) **then**
      $InvIdx$.update($T_{Ln}[i].L_{ntrapdoor}$, L_{ni})
    else
      $InvIdx$.create($T_{Ln}[i].L_{ntrapdoor}$, L_{ni})
    end
  end
end
SLS.send(L_{ni}, $TS_{storage}$)

L_{ni}	Qi0xb			
LE_n	NjA5YmM5YTktM2UyMC			
H_{Ln}	ZTk2NmQ3MjItOWM3NS			
HC_{Ln}	ZmY1MzczMTgtZjA4My			
LS_i	NjhiNDhkNmUtOTUwNC			
T_{Ln}	$L_{nfieldId}$	CxtZ8vVUy	$L_{ntrapdoor}$	hEzCbMn7u
	$L_{nfieldId}$	UQdYgEtoZ	$L_{ntrapdoor}$	CxtZ8vVUy
	$L_{nfieldId}$	zL4r5Gbul	$L_{ntrapdoor}$	C2nyVaqx1

Table 4.5: SLaS Application indexing input request example

L_{in}	LE_n	H_{Ln}	HC_{Ln}	LS_i
SS1Xa	MDZhYmEyMGItMmJkNy	N2IyNDFhYjEtMmUzNS	MGY2ZjI4MzktYmQ4Zi	MTViOWVkNzktNzYxYS
Rk1tV	MGY2ZjI4MzktYmQ4Zi	OGE5ZjMwMzQtZDFmMC	ODdkZmEzNjItMGJhYy	YTgyYzFmYzQtOWFjMy
b3B4W	NzAwODAwYTctNDdlZS	YTgyYTBmMDctOGRmNC	ODVjYTZkMmItZTMxYi	ODg4ZGI2YmQtY2ExNS
Qi0xb	NjA5YmM5YTktM2UyMC	ZTk2NmQ3MjItOWM3NS	ZmY1MzczMTgtZjA4My	NjhiNDhkNmUtOTUwNC

Table 4.6: SLaS Application database sample

$L_{nfieldId}$	$L_{ntrapdoor}$	L_{ni}
C2nyVaqx1	ZTgyMjQzYTYtNmZhYS	[SS1Xa,b3B4W]
--	YWJiYWI4NGYtOTcyZi	[Rk1tV]
CxtZ8vVUy	NTdiZjZjZmYtOGQyNS	[Rk1tV, Qi0xb]
UQdYgEtoZ	ZWEwYjg4MmQtZGZmMC	[Qi0xb}]
zL4r5Gbul	Y2QyNTg1NTctYjEONC	[Qi0xb]

Table 4.7: SLaS Application inverted index sample

4.4.4 Search

The Search operation is assumed to be executed whenever a query over the encrypted log records is required. From the perspective of the data user, he submits a clear text query and receives matching clear text log records. However, the SLS applies a series of cryptographic operations to assure that no one besides him has knowledge about the query and matching results. These operations are described in Algorithms 4.7 and 4.8. Algorithm 4.7 explains the process executed by the SLS to transform a clear text query into an encrypted query. Algorithm 4.8 demonstrates how the SLS, using the encrypted query, retrieves matching log records from the SLaS Application and presents them to the data user.

Algorithm 4.7 receives as input a clear text query Q comprised of operators and terms. The term represents the values to be searched. More specifically, those values can be either the keyword content itself or a combination of the keyword content alongside the field identifier of where that specific keyword should be looked up. For instance, a query term can be either ”GET” or ”method=GET”. The former indicates a search of the keyword ”GET” in all the log records universe, the latter indicates that the value ”GET” should be only looked up on the field ”method”. Moreover, the operator represents the boolean operators ”AND”, ”OR” or ”NOT” that enable conjunctive keywords queries. If a nested query is required it must be submitted inside parentheses.

Algorithm 4.7 is initialised by the division of Q into separated parts (step 3). Next, each $Qpart$ is verified to check if it contains a nested query. If such verification is successful, Algorithm 4.7 is applied in a recursive manner to $Qpart$. If $Qpart$ is a query operator (step 8) (i.e. ”AND”) $Qpart$ is added to the encrypted query operators QE_{ops} (step 9). If $Qpart$ is not a nested query nor an operator, it is assumed to be a query term (step 10). Thus, a technique similar to the one applied in Algorithm 4.5 for the creation of the keywords trapdoors is applied. More specifically, it is verified if Q_{part} includes a field name Q_{name} (step 11). If so, a lookup operation is performed (step 11) to obtain the identifier $Q_{fieldId}$ of such field in LS_{map}. Then, the trapdoor $Q_{trapdoor}$ for that specific Q_{value} is built by the computation of an HMAC with SLS_{kK} (step 13). Finally, a tuple containing $Q_{fieldId}$ and $Q_{trapdoor}$ is added to the encrypted query terms QE_{terms} (step 14). If Q_{part} includes Q_{value} alone, only $Q_{trapdoor}$ is obtained by the HMAC computation (step 16) and added to QE_{terms} (step 17).

Both QE_{terms} and QE_{ops} are used as input for Algorithm 4.8 that starts by sending them to the SLaS

Algorithm 4.7: SLS query builder algorithm

Input : Q
Output: QE_{terms}, QE_{ops}

$QE_{terms} \leftarrow$ []
$QE_{operatos} \leftarrow$ []
$Q_{parsed} \leftarrow$ splitQuery(Q)
for *i* $\leftarrow$ *0* **to** *length*(Q_{parsed}) **do**
    $Q_{part} \leftarrow Q_{parsed}[i]$
    if *isNestedQuery*(Q_{part} **then**
        BuildQuery(Q_{part})
    else if *isOperator*(Q_{part}) **then**
        $QE_{operatos}$.add(Q_{part})
    else
        if *exists*($Q_{part}.Q_{name}$) **then**
            $Q_{fieldId} \leftarrow LS_{map}$.convert($Q_{part}.Q_{name}$)
            $Q_{trapdoor} \leftarrow$ HMAC.create(SLS_{kK}, $Q_{part}.Q_{value}$)
            QE_{terms}.add($\{Q_{fieldId}, Q_{trapdoor}\}$)
        else
            $Q_{trapdoor} \leftarrow$ HMAC.create(SLS_{kK}, $Q_{part}.Q_{value}$)
            QE_{terms}.add($\{Q_{trapdoor}\}$)
        end
    end
end

Application in order to retrieve a set encrypted log records LE (step 1). Each of the n encrypted log records on LE is composed by the log record identifier L_{ni}, the encrypted content LE_n, the HMAC H_{Ln} and the log source identifier LS_i (step 3). L_{ni} is used to load the specific LS properties (step 4), which enables the consequent verification of H_{Ln} (step 5). If H_{Ln} verification is successful (step 6), the n log record encryption key K_{Ln} is produced by the HMAC computation of L_{ni} with LS_{eK} key (step 7). K_{Ln} enables the encryption of the clear text log record L_n and its timestamp TS (step 8), which are finally presented to the data user (step 9).

Algorithm 4.8: SLS search algorithm

Input : QE_{terms}, QE_{ops}
Output: -

1 $LE \leftarrow SLaS$.search(QE_{terms}, QE_{ops})
2 **for** *n* $\leftarrow$ *0* **to** *length(LE)* **do**
3 $L_{ni}, LE_n, H_{Ln}, LS_i \leftarrow LE[n]$
4 $LS \leftarrow$ LogSources.get(LS_i)
5 logVerified $\leftarrow$ HMAC.verify($LS.LS_{hK}$, $LE_n + LS_i + L_{ni}$, H_{Ln})
6 **if** *logVerified* **then**
7 $K_{Ln} \leftarrow$ HMAC.create($LS.LS_{eK}$, L_{ni})
8 $L_n, TS \leftarrow K_{Ln}$.decrypt($LE_n$)
9 SLS.print(L_n, TS)
10 **end**
11 **end**

Algorithm 4.9: SLaS Application search algorithm

Input : QE_{terms}, QE_{ops}
Output: LE

1 $L_I \leftarrow$ InvIdx.query(QE_{terms}, QE_{ops})
2 $LE \leftarrow$ []
3 **for** *i* $\leftarrow$ *0* **to** *length(L_I)* **do**
4 $L_{ni}, LE_n, H_{Ln}, LS_i \leftarrow$ DB.get($L_I[i]$)
5 LE.add($\{L_{ni}, LE_n, H_{Ln}, LS_i\}$)
6 **end**
7 SLS.send(LE)

The set of operations that occur on the SLaS Application during the search procedure are described in Algorithm 4.9. First the received encrypted query terms QE_{terms} and operators QE_{ops} are checked against the SLaS Application encrypted inverted index $InvIdx$ to obtain a set of identifiers L_I of log records that satisfy the received query (step 1). Then, for each identifier i, the SLaS application retrieves

L_{ni}, LE_n, H_{Ln} and LS_i from its DB (step 4) and adds them to set of encrypted log records LE (step 5). At the end, LE is sent over to the SLS (step 7).

To have a better perception of the type of operations that occur during a search procedure, suppose the reception of the below query Q, that searches for log records whose IP address matches with "172.217.17.3" and contains the keywords "GET" or "PUT" in any part of the log content.

Q: **ip=172.217.17.3 AND (GET OR PUT)**

After Q splitting into terms, operators and nested queries, the following Q_{parsed} is obtained:

Q_{parsed}: **[ip=172.217.17.3, AND, (GET OR PUT)]**

Then, for the first element of Q_{parsed}, which is a term that contains both the field name Q_{name} and its value Q_{value} the following conversions are executed. The field name Q_{name} is converted to its identifier $Q_{fieldId}$ by a LS_{map} lookup operation and Q_{value} is converted to its searchable trapdoor $Q_{trapdoor}$ by an HMAC computation. The second element of Q_{parsed} is an operator, so no additional processing is done. The third and final element is a nested query so the build algorithm is executed in a recursive manner. This is, the nested query is again broken into terms, operators and other possible nested queries. Since it only includes two terms separated by an operator, no additional recursive iteration is executed. The terms of the referred nested query are build by Q_{value} alone, thus just an HMAC computation is required to obtain the respective $Q_{trapdoor}$. The final encrypted query terms QE_{terms} and operators QE_{ops} are structured as depicted in Table 4.8.

$Q_{fieldId}$	hEzCbMn7u	$Q_{trapdoor}$	MTg0LjI1LjQxLjE1
AND			
$Q_{trapdoor}$	R0VU		
OR			
$Q_{trapdoor}$	UFVU		

Table 4.8: Encrypted query sample

QE_{terms} and QE_{ops} are then sent to the SLaS Application that tests them against its searchable encrypted inverted index $InvIdx$ and returns a set LE of matching encrypted log records, containing the log identifiers L_{ni} alongside its encrypted content LE_n, HMAC tag H_{Ln} and log source identifier LS_i, as depicted in Table 4.9.

L_{in}	LE_n	H_{Ln}	LS_i
SS1Xa	MDZhYmEyMGItMmJkNy	N2IyNDFhYjEtMmUzNS	MTViOWVkNzktNzYxYS
Qi0xb	NjA5YmM5YTktM2UyMC	ZTk2NmQ3MjItOWM3NS	NjhiNDhkNmUtOTUwNC

Table 4.9: SLaS Application search response example

Upon receiving such information, the SLS validates each log record H_{Ln} and decrypts LE_n to obtain the clear text log record L_{ni} and its timestamp TS, presenting the information to the data user like Table 4.10 depicts.

TS	L_{ni}
2019-10-12 15:25:15.77	172.217.17.3 - - [12/Oct/2019:15:25:15 -0800] ”GET /home HTTP1.1” 200 2314
2019-10-12 15:26:00.77	172.217.17.3 - - [12/Oct/2019:15:26:00 -0800] ”PUT /user HTTP1.1” 200 2314

Table 4.10: Search response clear text example

Query Language

In order to enable the construction of any query supported by the search algorithm, the definition of a query language, close to natural language as possible in order to support seamless query definition, is envisioned. The SLS queries are broken up into terms and operators and may be combined like ”TERM OPERATOR (TERM OPERATOR TERM)” where the ”TERM” represents the keywords to be searched, the ”OPERATOR” represents the boolean operators and parentheses indicate the existence of a nested query.

Regarding the OPERATOR clause, the algorithm supports the three basic boolean operators. The AND operator enables the retrieval of log records that simultaneously match two or more keywords. To obtain log records which contain either or both keywords the OR operator is applied. In order to fetch log records that exclude a specific keyword or set of keywords the NOT operator is used.

The TERM clause is used to indicate what keywords shall be searched within the encrypted log records. Each term can be composed either for the single keyword value or for the combination of the keyword value and the identifier of the field where the search should focus. If only the keyword value is present, the query will test the presence of such keyword in any part of the log record, which may return results that are not relevant for the desired search. Thus, the combination of keyword value and the field identifier enables field search and consequent more accurate results. To perform a field query, the field identifier and the keyword value must be combined like ”ID=VALUE”. The SLS search algorithm supports the simultaneously inclusion of single and field terms within the queries.

Nested queries, that allow more fine-grained search results, are also possible. To create a nested query the only rule is to write it inside parentheses to allow it to be parsed correctly by the search engine.

4.4.5 Forward Integrity

The proposed SLS links each log record in such way that makes it possible to verify if any of those records was modified or deleted. In other words, the SLS enables Forward Integrity. This property is achieved with the creation of a log record hash chain, built by a successive computation of an HMAC

Algorithm 4.10: SLS forward integrity verification algorithm

Input : LS_i
Output: -

$LS \leftarrow$ LogSources.get(LS_i)
$HC \leftarrow SLaS$.getHashChain(LS_i)
$HC_{Ln-1} \leftarrow LS.LS_{seed}$
for *n* $\leftarrow$ *0* **to** *length*(HC) **do**
    $L_{ni}, H_{Ln}, HC_{Ln} \leftarrow HC[n]$
    hashChainVerified $\leftarrow$ HMAC.verify($LS.LS_{hK}$, $H_{Ln} + HC_{Ln-1}$, HC_{Ln})
    if *hashChainVerified* **then**
        $HC_{Ln-1} \leftarrow HC_{Ln}$
    else
        SLS.print(L_{ni})
        break
    end
end
if $HC_{Ln-1} = LS.HC_{Ln-1}$ **then**
    SLS.print($True$)
else
    SLS.print($False$)
end

function where the input of the current log record HMAC also includes the output of the previously stored log record.

The hash chain is constructed throughout the indexing of log records as described in Algorithm 4.5. If the data owner desires to acknowledge if his log records remain authentic, Algorithm 4.10 is executed by the SLS. This algorithm starts by obtaining the properties of the log source LS whose hash chain is to be verified (step 1). Then, it requests the hash chain HC of LS from the SLaS Application (step 2). The verification phase starts with the initialisation of the hash chain with the random seed value LS_{seed} computed on the configuration of the specified LS (step 3). Next, for each n link in the hash chain, the verification of the current link HC_{Ln} is performed by verifying the HMAC H_{Ln} of the log record concatenated with the previous hash chain link HC_{Ln-1} (step 6). If successful, HC_{Ln} overrides HC_{Ln-1} and the hash chain validation moves on to the next link. If the verification fails, Algorithm 4.10 stops and the identifier L_{ni} of the failing log record is presented to the data owner (step 10).

If Algorithm 4.10 is capable of traversing all the hash chain without detecting tampering, it is assumed that no log modification or deletion has happened. Lastly, in order to detect any possible truncation attack, the final computed hash link HC_{Ln-1} is compared with HC_{Ln-1} obtained from LS properties. If the two values match, no truncation attack occurred. If not, some tail end log records were deleted from the SLaS Application DB.

4.5 Summary

This chapter describes the proposed solution, named SLS, which enables a secure remote storage of encrypted log records with privacy, integrity and authenticity guarantees. Moreover, searchability of such encrypted log records is possible without access to the clear text.

The proposed solution assumes a system model composed by three different entities. The data owner, whose log records are securely stored on the remote storage server, the data user, authorised to search the stored data for the keywords of interest and the remote storage server, who securely stores the log records and retrieves them in search operations.

The architecture of the SLS comprises six different components. The Conf Manager (C1) for configuration properties storage and retrieval. The Encryption (C2) for data encryption and decryption. The Indexing (C3) for clear text log records transformation into searchable encrypted logs. The Search (C4) for querying the logs. The Internal Connection (C5) for communication with the applications whose logs are to be securely stored. The External Connection (C6) for a communication between the SLS and the SLaS Application. The SLS assures the confidentially, integrity and authenticity of data not only at rest but also in transit. Hence, two communication protocols are envisioned. One is applied to the connection between the file watches and the SLS, assuring the authentication of such file watchers based

on HMAC computations and the privacy of the information by the use of asymmetric cryptography. The second protocol is used on the communication between the SLS and the SLaS Application and is based on a hybrid encryption scheme with authentication.

The indexing of the log records assures its confidentiality by the use of symmetric cryptography and secret keys different for each log record. The integrity of each isolated log record is achieved by the computation of an HMAC. Forward Integrity arises from the construction of an hash chain, in which each link consists in an HMAC of the current log record HMAC concatenated with the previous log record HMAC. To assure searchability, the logs records are either parsed from a regex or splitted by a delimiter. The former assures field search, by adding each field identifier alongside the specific search trapdoors. The latter generates the search trapdoors for each part of the splitted log record. This search data is maintained by the SLaS Application on a encrypted inverted index in order to assure fast search operations. In order to facilitate the construction of any query supported by the search algorithms a query language is proposed, supporting field search, boolean operators and nested queries.

Chapter 5

Validation

This chapter is focused on the evaluation of the proposed SLS, achieved by the implementation of a prototype, consequently tested in terms of its performance and functional aspects. The prototype was implemented using version 12.6.0 of the NodeJS runtime environment. All cryptographic operations used the *crypto* native module, which provided methods to implement the solution with secure cryptographic algorithms. The tests performed used the AES algorithm in the GCM scheme [22]. The HMAC computations were based on the SHA-3 algorithm [14]. The asymmetric cryptography applied on the communication phase used the RSA [12] algorithm. In terms of hardware, the prototype was executed on a computer running Linux with an *Intel Core i7 4700MQ 3.4Ghz* processor and 8GB of *DDR3 RAM* memory. For storage, both a single instance of a *MongoDB* and a *PostgreSQL* database servers were adopted. Moreover, during the prototype development, we assume the existence of a single log source.

The chapter starts by introducing the functional tests used to validate that the requirements listed in Chapter 4 are met. The performance tests, focused on the time required by the indexing and search operations and the required storage space for the encrypted data, are detailed next so that a comparison with the related work, identified in Chapter 3, could be performed. Lastly, a security analysis is performed in order to validate if the proposed solution achieves the desired security properties.

5.1 Functional Tests

These functional tests aim to demonstrate that the requirements identified in Chapter 4 are met. Such verification is done by the presentation and consequent description of the implementation of the proposed algorithms.

```
1 const sendSLS = async L => {
```

```
2    const TS = Date.now();
3    const AT = hmacCreate('sha3-512', `${L}${TS}${LS.I}`, LS.LSaK);
4    const LE = rsaEncrypt(JSON.stringify({L, TS, AT}), LS.SLSi.publicKey, '2048');
5    const resSLS = await SLS.indexLog(LE, LS.I);
6    return resSLS;
7  };
```

Listing 5.1: File Watcher internal communication

```
1  const sendSLaSStorage = async P => {
2    const KP = crypto.randomBytes(32);
3    const PE = aesEncrypt('aes-256-gcm', P, KP);
4    const KPE = rsaEncryptKey(KP, SLS.SLaS.publicKey, '2048');
5    const SPN = rsaSign(`${PE}${KPE}`, SLS.SLSe.privateKey, '2048');
6    return SLaS.storeLog(PE, KPE, SPN);
7  };
```

Listing 5.2: SLS external indexing communication

Requirement 1 addresses log privacy. Such privacy must be assured not only at rest but also during the communication between the involved parties. In both situations, privacy is achieved by the use of encryption. Between the File Watchers and the SLS, the log privacy is achieved by the use of the RSA algorithm, as line 4 in Listing 5.1 shows. Each log record L, alongside its timestamp TS and authentication tag AT are encrypted using the SLS public key before its transmission. Regarding the communication between the SLS and the SLaS Application, an hybrid encryption scheme, using AES and RSA algorithms, is implemented, as Listing 5.2 depicts. A random AES key KP is generated (line 2) and used to encrypt the payload P (line 3). The key KP is encrypted using RSA and the public key of the SLaS Application (line 4). Log privacy at rest is obtained during the indexing operation, described in Listing 5.3. Each log record L, prior to its transfer to the SLaS Application for storage, is encrypted with the AES algorithm (line 5) using a key KLN that results from a SHA-3 HMAC computation of the unique identifier LI of the log record.

```
1  const indexLog = async (L, TS) => {
2    if (hmacVerify('sha3-512', `${L}${TS}${LSI}`, LS.LSaK, AT)) {
3      const LI = uuid();
4      const KLN = hmacCreate('sha3-512', LI, LS.LSeK).substring(0, 32);
5      const LE = aesEncrypt('aes-256-gcm', JSON.stringify({L, TS}), KLN);
6      const H = hmacCreate('sha3-512', `${LE}${LSI}${LI}`, LS.LShK);
```

```
7       const HC = hmacCreate('sha3-512', `${H}${HC_1}`, LS.LShK);
8       LS.HC_1 = HC;
9       const TL = [];
10      const REGEX = new RegExp(LS.REGEX, 'gi');
11      let L_PARSED = REGEX.exec(L);
12      if (L_PARSED) {
13        L_PARSED = L_PARSED.splice(1, LS.MAP.length);
14        L_PARSED.forEach((L_FIELD, L_IDX) => {
15          const I = LS.MAP[L_IDX].id;
16          const T = T = hmacCreate('sha3-512', L_FIELD, SLS.SLSkK);
17          TL.push({I, T});
18        });
19      } else {
20        L_SANITIZED = L.replace(new RegExp(LS.BLK, 'gi'), LS.DLM);
21        L_PARSED = L_SANITIZED.split(LS.DLM).filter(T => (new RegExp(/\w/).test(T)));
22        L_PARSED.forEach(L_FIELD => {
23          const T = hmacCreate('sha3-512', L_FIELD, SLS.SLSkK);
24          TL.push({T});
25        });
26      }
27      return sendSLaSStorage(JSON.stringify({LSI, LI, LE, H, HC, TL}));
28    }
29    return null;
30  }
```

Listing 5.3: SLS indexing

Authentication, imposed by requirement 2, is accomplished by the file watchers prior to sending any log record to the SLS. That operation is implemented as described in Listing 5.1, on which an authentication tag AT is generated by calculating the SHA-3 HMAC of the log record, alongside its timestamp and log source identifier (line 3). Since such HMAC computation uses a secret key $LSaK$, that we assume that no one besides the specific file watcher has access, only that file watcher is able to generate a valid AT. Thus, is unfeasible for any unauthorised party to submit log records, since the verification of AT is the first instruction (line 2) of the indexing algorithm, as described in Listing 5.3 and the SLS only processes received log records if such AT is valid.

```
1  const storeLog = (PE, KPE, SPN) => {
2    if (rsaVerify(`${PE}${KPE}`, SLS.SLSe.publicKey, SPN, '2048')) {
```

```
3       const KP = rsaDecryptKey(KPE, SLS.SLaS.privateKey, '2048');
4       const {LSI, LI, LE, H, HC, TL} = JSON.parse(aesDecrypt('aes-256-gcm', PE, KP));
5       await STORAGE.query(
6         'INSERT INTO storage (LSI, LI, LE, H, HC) VALUES ($1, $2, $3, $4, $5)',
7         [LSI, LI, LE, H, HC]
8       );
9       await Promise.all(TL.map(S => (INDEX.update(
10        S.I ? { I: S.I, T: S.T } : { T: S.T },
11        { $push: { L: LI } },
12        { upsert: true }
13      ))));
14      return { LI, TSStorage: Date.now() };
15    }
16    return null;
17  }
```

Listing 5.4: SLaS Application storage

Requirement 2 also imposes authentication between the SLS and the SLaS Application, since it must not be possible for unauthenticated entities to send log records for storage. Listing 5.2 and Listing 5.4 exemplify the application of such constraint on a communication originated by the SLS to the SLaS Application. On Listing 5.2, a RSA signature SPN is computed over the payload using the private key of the SLS (line 5). Then, the SLaS Application (Listing 5.4) only processes the storage requests if the received SPN is valid (line 2). Since we assume that the private key of the SLS always remains secret and in its possession, the verification of the produced SPN authenticates the SLS. In a reverse communication from the SLaS Application to the SLS the same authentication scheme is applied.

```
1  const BuildQuery = (Q) => {
2    const Qparsed = Q.split(' ');
3    const QE = {
4      TERMS: [],
5      OPS1: [],
6      OPS2: []
7    };
8    for (let index = 0; index < Qparsed.length; index++) {
9      const Qpart = Qparsed[index];
10     if (Qpart.startsWith('(')) {
11       QE.TERMS.push([]);
```

```
12        let Qpart2 = Qpart;
13        let Qtmp = [];
14        let i = 0;
15        while (!Qpart2.endsWith(')')) {
16          Qpart2 = Qparsed[index + i];
17          Qtmp.push(Qpart2);
18          i++;
19        }
20        Qtmp = Qtmp.join(' ').replace(/\(/g, '').replace(/\)/g, '').split(' ');
21        for (let index2 = 0; index2 < Qtmp.length; index2++) {
22          const Qpart3 = Qtmp[index2];
23          if (Qpart3 === 'AND' || Qpart3 === 'OR' || Qpart3 === 'NOT') {
24            QE.OPS2.push([Qpart3]);
25          } else {
26            const Qfield3 = Qpart3.split('=');
27            if (Qfield3.length === 2) {
28              QE.TERMS[QE.TERMS.length - 1].push({name: Qfield3[0], value: Qfield3[1]});
29            } else {
30              QE.TERMS[QE.TERMS.length - 1].push({value: Qfield3[0]});
31            }
32          }
33        }
34        index += Qtmp.length - 1;
35      } else if (Qpart === 'AND' || Qpart === 'OR') {
36        QE.OPS1.push(Qpart);
37      } else {
38        const QQQ = Qpart.split('=');
39        if (QQQ.length === 2) {
40          QE.TERMS.push({
41            id: LS.MAP.find(match => (match.name === QQQ[0])).id,
42            value: hmacCreate('sha3-512', QQQ[1], SLS.SLSkK)
43          });
44        } else {
45          QE.TERMS.push({
46            value: hmacCreate('sha3-512', QQQ[0], SLS.SLSkK)
47          });
48        }
49      }
```

```
50      }
51      return QE;
52  }
```

Listing 5.5: SLS query builder

Requirements 3 and 4 are accomplished during the SLS indexing operation, outlined in Listing 5.3. Requirement 3 addresses verifiability, which line 6 of Listing 5.3 assures by the computation of a SHA-3 HMAC H, having as input the encrypted log record LE, its unique identifier LI and the log source identifier LSI. The referred HMAC enables the possibility to verify not only each individual log record integrity but also its authenticity, since the HMAC computation requires a secret key, that we assume to always remains in the SLS possession, thus no one besides it is able to generate valid HMACs. The forward integrity, specified by requirement 4, attained by the construction of an hash chain based on successive computation of HMACs, is achieved on lines 7 and 8 of Listing 5.3. The log record hash H is concatenated with the previous log record hash chain link HC_1 to produce, through a SHA-3 HMAC computation, the current log record hash chain link HC. Then, HC is setted as HC_1, which is used on subsequent indexing operations. Since the key to compute HC and HC is the same, the hash chain can only be constructed and verified by the SLS.

```
1  const SendSLaSSearch = async P => {
2    const KP = crypto.randomBytes(32);
3    const PE = aesEncrypt('aes-256-gcm', P, KP);
4    const KPE = rsaEncryptKey(KP, SLS.SLaS.publicKey, '2048');
5    const SPN = rsaSign(`${PE}${KPE}`, SLS.SLSe.privateKey);
6    return SLaS.queryLog(PE, KPE, SPN);
7  }
```

Listing 5.6: SLS external search communication

Requirement 5 says it must be possible to search within the encrypted log records and retrieve matching results. Such was implemented on Listings 5.3 and 5.3. On the developed prototype, we assumed that a regex exists, so each log line was first tested against it (line 11). If the log line matched with the regex, each extracted keyword name was converted to its identifier (line 15) and the trapdoor created by the computation of a SHA-3 HMAC (line 16). If no match between the log record and regex exists, the log record is first sanitised by the removal of blacklist characters (line 20) and splitted by a delimiter line (21). Then, a SHA-3 HMAC of each log record part is created, generating the trapdoors (line 23). Since the trapdoors are generated on the client-side, with a secret key only in the possession of the SLS and submitted to the server only in encrypted form, requirement 6, concerning query confidentiality, is also partially fulfilled.

```
1  const queryLog = (PE, KPE, SPN) => {
2    if (rsaVerify(`${PE}${KPE}`, SLS.SLSe.publicKey, SPN)) {
3      const KP = rsaDecryptKey(KPE, SLS.SLaS.privateKey);
4      const {TERMS, OPS1, OPS2} = JSON.parse(aesDecrypt(CRYPT, PE, KP));
5      const QE = BuildIndexQuery(TERMS);
6      const QES = await ResolveIndexQuery(QE);
7      const QEF = [QES[0]];
8      CreateStorageQuery({OPS1, OPS2}, QES, QEF);
9      const {rows: L} = await STORAGE.query(
10       `SELECT LSI, LI, LE, H
11       FROM storage
12       WHERE ${QEF.join(' ')}`
13     );
14     return L;
15   }
16   return [];
17 }
```

Listing 5.7: SLaS Application search

Query confidentiality, from requirement 6, is completed at the search phase, as described in Listings 5.5, 5.6, 5.8 and 5.7. Such requirement states that is must be possible to submit queries in an encrypted form. That is achieved by Listing 5.5, on which, in order to build the queries, each keyword value is transformed into a searchable SHA-3 HMAC (lines 42 and 46), prior to its transfer. Moreover, the query is sent to the SLaS Application as described in Listing 5.6, hence an additional layer of encryption is assured during the transfer of that information. Requirement 6 also says that the search must be conducted over encrypted data. As described in Listing 5.7, the received encrypted query terms are directly looked up on the inverted index Mongo database (lines 5 and 6) in order to obtain the log records identifiers. Afterwards, based on that identifiers, a SQL query is built to retrieve the matching results. The final part of requirement 6 addresses the retrieval of encrypted matches, which is also verifiable on the produced SQL query of Listing 5.7. The output of that query consists in a list containing the log record identifiers LI, alongside its encrypted content LE, log source identifiers LSI and hash H, which is directly sent to the SLS. The log records inside that list are only decrypted at the client premises, as described in Listing 5.8.

```
1  const searchLog = Q => {
2    const QE = BuildQuery(Q);
3    const L = await SendSLaSSearch(JSON.stringify(QE));
```

```
4     const LN = [];
5     L.forEach(({lsi: LSI, li: LI, le: LE, h: H}) => {
6       if (hmacVerify('sha3-512', `${LE}${LSI}${LI}`, LS.LShK, H)) {
7         const KLN = hmacCreate('sha3-512', LI, LS.LSeK).substring(0, 32);
8         LN.push(JSON.parse(aesDecrypt('aes-256-gcm', LE, KLN)));
9       }
10    });
11    return LN;
12  }
```

Listing 5.8: SLS search

Requirement 7 says that it must be possible to submit complex and fine-grained queries to obtain the most accurate results. This is achieved by the creation of a query language, which enables the construction of queries closer to the natural human language. Nevertheless, from the possibility to write queries in a natural language arises the need to build a mechanism to convert that queries in structures that are simpler to be parsed by a computer. This mechanism is implemented on Listing 5.5, which describes the process executed by the SLS to transform a query Q into an encrypted query QE. On the development of the prototype we assumed that only one level of query nesting to be necessary, so the algorithm was developed to support that specific case, without the use of recursive functions.

5.2 Performance Tests

The performance tests used real-life log information from a publicly accessible web server with, about, thirty thousand (30,000) records per day, on average. The average number of records per day was calculated from a period of three weeks. The log files were segmented by day and the number of log records per day was counted and then averaged. The tests were performed using a combination of different key sizes for the two algorithms used in indexing and searching operations.

Experiments were performed with AES with 128, 192 and 256 bit size keys and SHA-3 with 256, 384 and 512 bit size keys. For the RSA keys, since it was only applied on the communication between the SLS and SLaS Application, a unique size of 2048 bits was used. In particular, tests were carried out with the following combinations: AES 128 and SHA-3 256; AES 192 and SHA-3 384; AES 256 and SHA-3 512. On each experiment, three test runs per each key size combination were carried out and the average values presented.

The first set of tests focused on time elapsed for the indexing of the 30,000 log records and the storage space it required when using multiple combinations of keys sizes of the AES and SHA-3 algorithms. As illustrated in Figure 5.1, when using the minimum key sizes of 128 bits for AES and 256 for SHA-3,

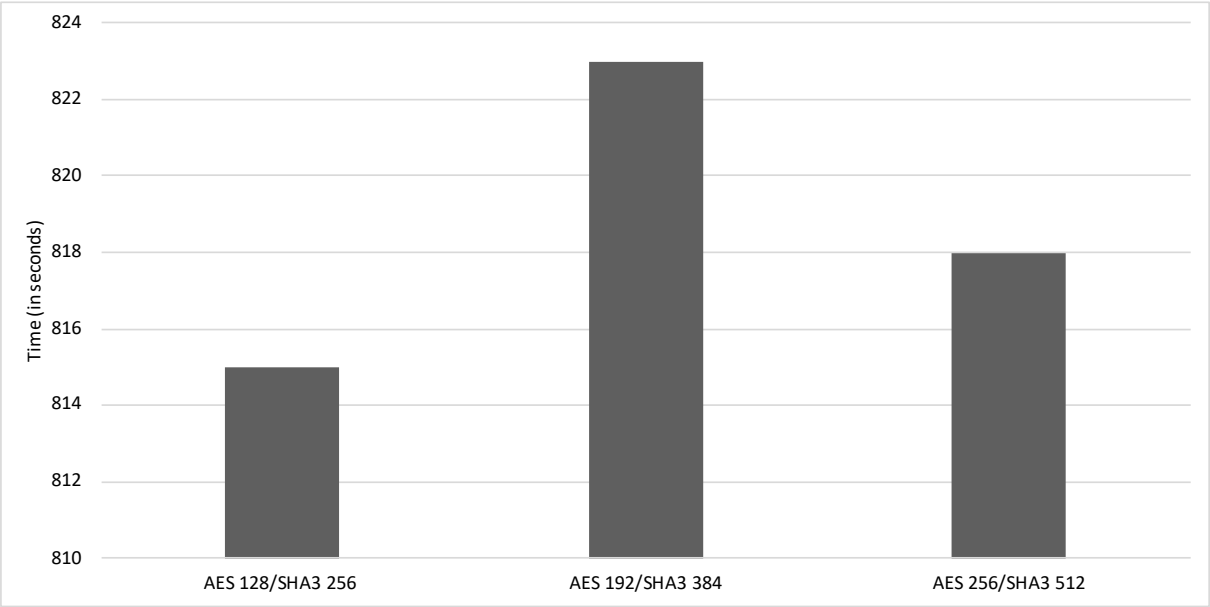

Figure 5.1: Time required to index 30,000 records, per key size

the proposed solution can index the required log entries in 815 seconds. The medium key sizes of 192 bits for AES and 384 for SHA-3 are the ones that take longer, demanding 823 seconds. When using maximum key size of 256 bits for AES and 512 for SHA-3, the proposed solution takes 818 seconds (roughly 13 minutes) to do the same operation. If we consider that a day comprises 1440 minutes, we can conclude that the solution, when using the maximum key sizes for each algorithm, which at the time of writing is considered safe, takes about 0,9% of a day to securely store the log entries generated on that day. Moreover, it is appropriate to notice that the disparity between the elapsed times of the different key combinations is small, as expected by the use of symmetric cryptography. Thus, it is possible to use the biggest key sizes for each algorithm, guaranteeing a higher level of security and without compromising the performance of the solution.

Regarding the storage space requirement, Table 5.1 shows the values achieved for the 30,000 log records, for the used key sizes. Additionally, it is shown the space required to index the log records in clear. Without any encryption or keyed hashing operations, the proposed solution uses 19 MB for the inverted index and 9 MB for the storage of the log records, resulting in an aggregate total of 28 MB.

For the minimum key sizes, the proposed solution uses 20 MB for the inverted index and 18 MB for the storage of the encrypted log records, with a total value of 38 MB. When using 192 bit size keys for AES and 384 bit size keys for SHA-3, the solution requires a storage space of 22 MB for the inverted index and 21 MB for the encrypted log records, with a total of 43 MB. When using the maximum key sizes for each algorithm, both the inverted index and the log records occupy 23 MB each, culminating in a total storage value of 46 MB.

With those values, it is possible to denote a very small discrepancy between the different tested key sizes combinations. Additionally, it is interesting to see that the required stored space for the clear text inverted index is very similar to the encrypted ones. Regarding the database required space, the clear text version occupies less since the log record is stored without no encryption and with no inclusion of its hash value and hash chain link.

Key Size	Inverted Index (MB)	Database (MB)	Total (MB)
Clear text	19	9	28
AES 128/SHA3 256	20	18	38
AES 192/SHA3 384	22	21	43
AES 256/SHA3 512	23	23	46

Table 5.1: Storage requirements per key size on a day

Based on the values presented in Table 5.1, a projection, depicted in Figure 5.2, of the required space for the storage of log records generated up to a period of one year was performed. The numbers show that if the log records are stored in clear text, roughly 10 GB of storage space is required for one year. When using the smallest key sizes for both algorithms, roughly 14 GB of storage space is required for one year. When using 192 bit size keys for AES and 384 bit size keys for SHA-3, almost 16 GB of storage space is required for the same period. When using the biggest key sizes, 256 for AES and 512 for SHA-3, roughly 17 GB of storage space is required for the same period. Based on the values projected in Figure 5.2 it is possible to conclude that if 256 bit size keys are used for AES and 512 bit size keys are used for SHA-3, an additional 30% of space is required when compared with the clear text version. Although, the percentage achieved is larger than expected, we consider that the impact on the required storage of using encryption and keyed hashing with the biggest keys does not outweigh the increase in security.

Performance tests of the search operations were also executed. The tests were executed with the same combination of algorithms, key sizes and techniques used on the indexing of the sample 30,000 log entries.

For each key size, several search terms were tested. These search terms would return different numbers of matching log lines. The times shown in Figure 5.3 are the sum of both the time elapsed for the search conducted by the SLaS Application and the consequent data decryption executed by the SLS. Analysing the results, we can take two different conclusions. First, we can conclude that the different keys sizes do not influence the speed of the search and decryption operations, being possible to achieve similar results in all the tested combinations. This behaviour was somewhat expected since the symmetric cryptographic algorithms performance is not heavily influenced by the used key sizes.

Then, we can denote that the time used in search operations does not grow linearly to the number of matching results. For instance, to obtain the clear text result of a search operation that returned 3045

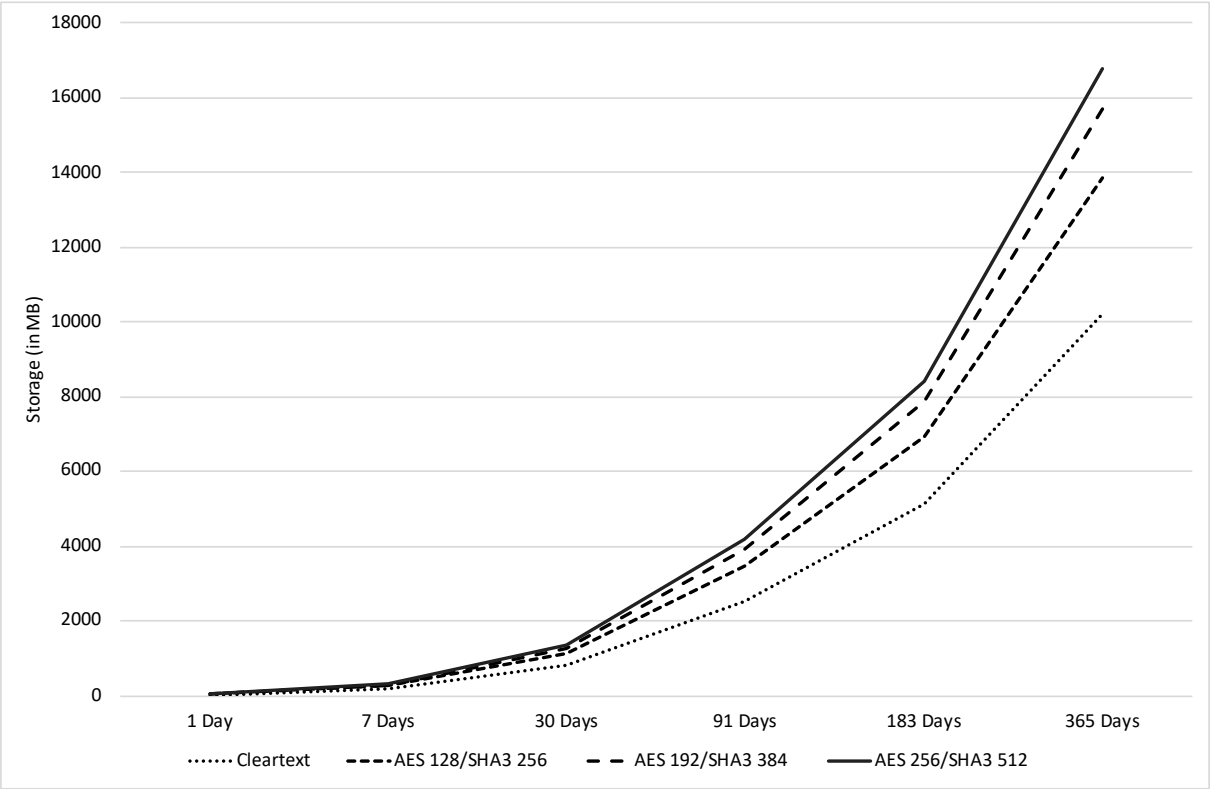

Figure 5.2: Storage requirements per key size

encrypted log lines, representing roughly 10% of all log records, the proposed solution took about 200 milliseconds. Other search operations, such as the ones returning 3000 or less matches, were executed with an approximate average time of the same 200 milliseconds. Lastly, it is important to consider the values achieved on the search operations that returned 26084 matches, representing roughly 87% of all the log records. The proposed solution was able to process this result in a time below 1.2 seconds. Based on that, it is possible to estimate that in order to search and decrypt all the 30,000 log records, the solution would take less than 1.5 seconds.

The final tests executed are related to forward integrity. These aim at validating if any of the 30,000 sample log records were altered or deleted. In order to fulfill this, the implemented prototype retrieves the hash value and hash chain link of each log record and validates them. Figure 5.4 depicts the values achieved on the referred test. For the minimum key sizes of 128 for AES and 256 for SHA-3, the solution took 305 milliseconds to verify the log records. When using 192 bit size keys for AES and 384 bit size keys for SHA-3 an elapsed time of 295 milliseconds was achieved. Finally, for the maximum key sizes of 256 for AES and 512 for SHA-3 the solution took 298 milliseconds. Despite some discrepancy between the different key sizes combinations, this specific test proves that an average value of 300 milliseconds is required to verify the forward integrity of the 30,000 sample log records.

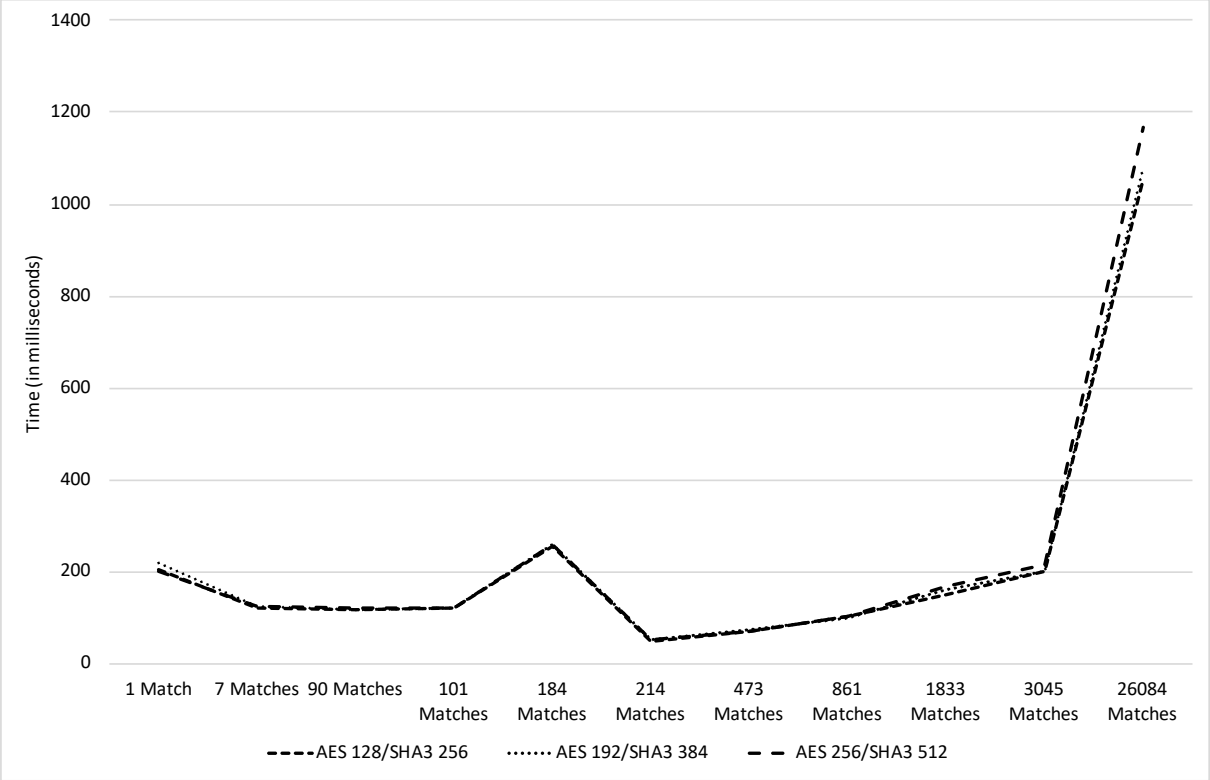

Figure 5.3: Searching times per key size and matches

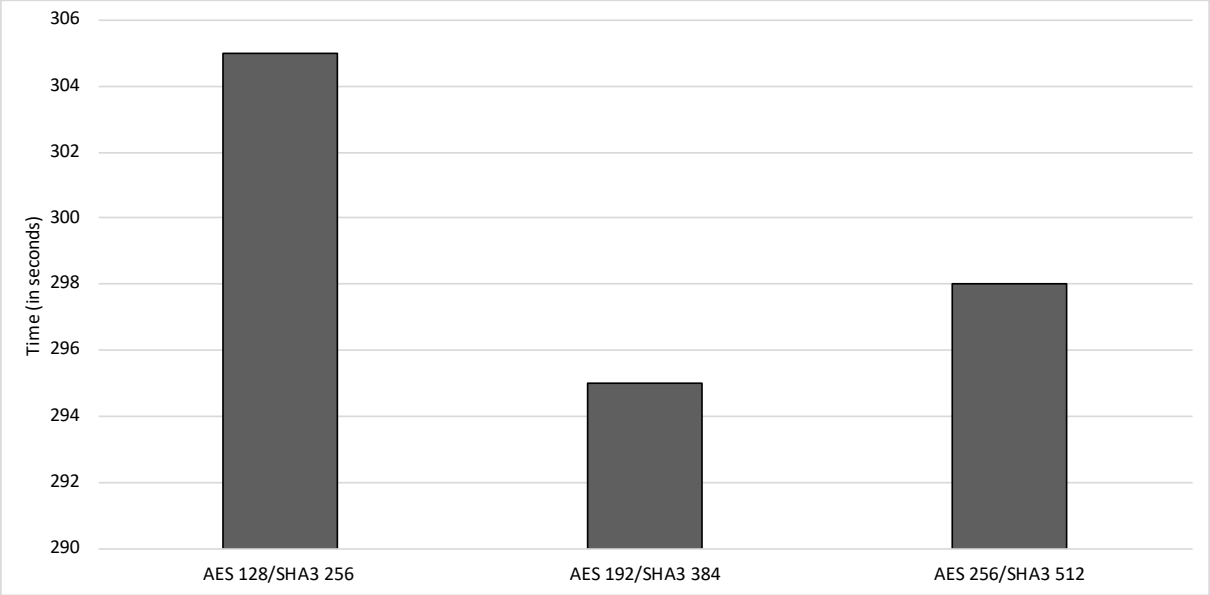

Figure 5.4: Time required to verify 30,000 records, per key size

5.3 Comparison with Related Work

A comparison with the solutions identified in Chapter 3 is interesting in order to validate where our solution fits in terms of performance and storage requirements. Moreover, only the solutions that have the ability to search within the encrypted data were analysed since these were the ones that were deemed equivalent, with respect to their basic functionality, to the one proposed herein.

Name	Indexing	Search
Waters	E_s + $2KE_a$	$2KE_a$ + ND_s
Ohtaki	E_a + $(2K+1)E_s$	E_a + $K(2ND_s)$
Sabbaghi	E_s + E_a + $3KH$	E_a + $3KNH$ + ND_s
Accorsi	E_s + $3H$ + KH	H + KH + ND_s
Savade	E_a + KE_a	KE_a + ND_a
Zhao	E_a + $3KH$	KH + ND_a
Our Solution	E_s + $3H$ + KH	KH + NH + ND_s

Table 5.2: Comparison of the related work indexing and search operations

The first conducted observation is related to the number of cryptographic operations required for the index and search operations, which is useful to measure the distance between the computational cost of our solution and the ones in the related work. Table 5.2 depicts the number of cryptographic operations for both indexing and searching, where E_s represents a symmetric encryption operation, E_a represents an asymmetric encryption operation, D_s represents a symmetric decryption operation, D_a represents an asymmetric decryption operation, H represents an hash operation, K represents the number of keywords and N represents the number of matching results in a search operation. The formulas presented in Table 5.2 represent the computational cost to index one log record and to perform one search operation.

Analysing Table 5.2 it is possible to denote that the solutions of Waters, Ohtaki and Savade only require encryption operations on indexing. The first uses 1 symmetric encryption plus 2 asymmetric encryption operations for each keyword K, the second uses 1 asymmetric encryption alongside $2(K+1)$ symmetric encryption operations and the third uses 1 asymmetric encryption plus an additional asymmetric encryption for each K. Our solution only requires 1 symmetric encryption operation to index a log record. Additionally, our solution needs 3 hash operations plus a new hash operation for each K. The remaining solutions that also use hash operations for indexing are Sabbaghi, Accorsi and Zhao. The first and the third use 3 hash operations per each K and the second uses the same number of hash operations that our solution.

Based on that values, a projection of the time required to index various numbers of log lines on the multiple identified solutions was made. In order to perform it, average values of times elapsed for the cryptographic operations were necessary. Such values were obtained by running 50 tests for each operation and then calculating the average elapsed time for each one. Based on that, it is possible to assume

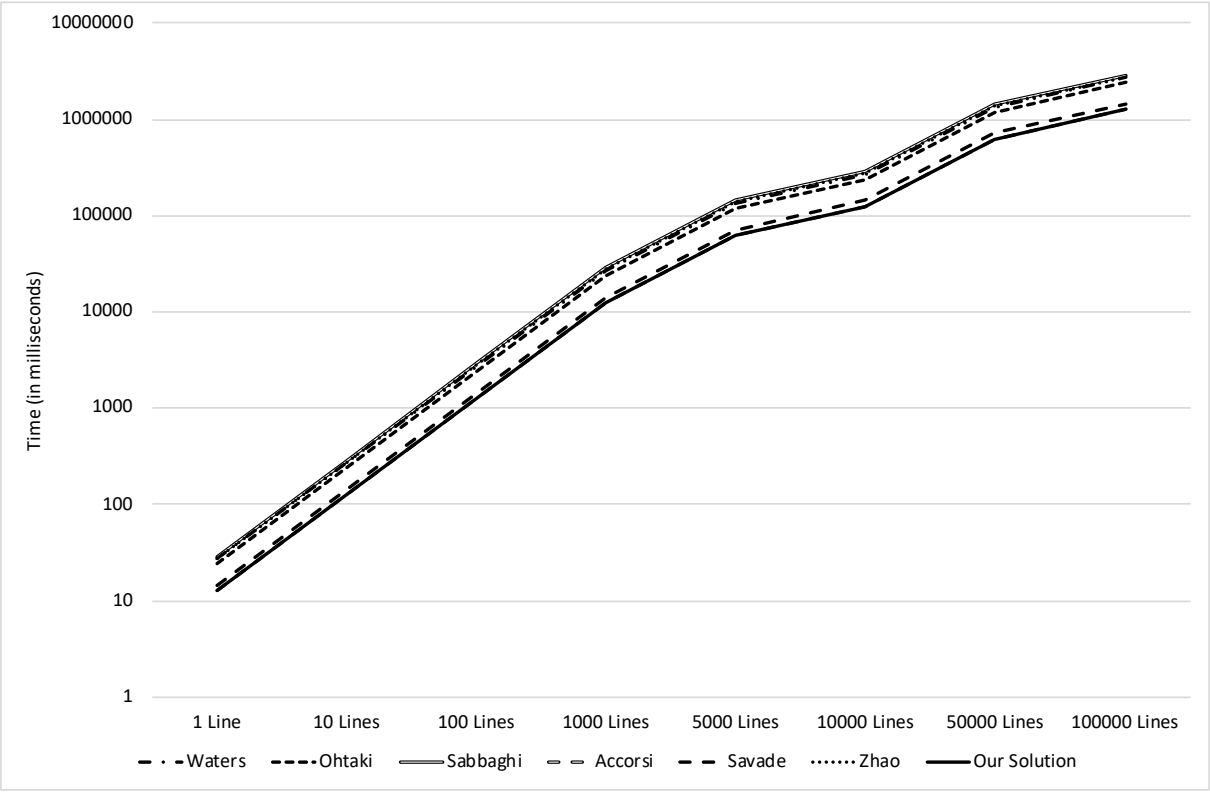

Figure 5.5: Comparison of the related work indexing times

that an hash operation takes 0.88 milliseconds, a symmetric encryption operation takes 1.08 milliseconds, a symmetric decryption operations takes 1.50 milliseconds, an asymmetric encryption operation takes 1.30 milliseconds, and a symmetric decryption operations takes 1.75 milliseconds. Figure 5.5 depicts the indexing projection from 1 to 100 000 lines. A value of $K = 10$ for the number of keywords present in each log line was assumed. This is, each log line comprises 10 keywords that must be indexed. Figure 5.5 uses a logarithmic scale. Based on the values presented, it is possible to conclude that the computation cost of the indexing operation of the proposed solution is equal to the lowest one (Accorsi).

Regarding the number operations required for the search phase, the proposed solution does not demand any encryption operation, unlike the solutions of Waters, Ohtaki, Sabbaghi and Savade. The first uses 2 asymmetric encryption operations per keyword K, the second and the third use a single asymmetric encryption operation and third uses 1 asymmetric encryption operation per keyword K. Hash operations are used by Sabbaghi, Accorsi and Zhao. The first uses 3 hashes per each keyword K and number N of matched log records, the second and the third use K hash operations. Decryption operations exist on all solutions. Except Ohtaki, which requires $2ND_s$ operations per each K, all solutions use a single decryption operation per N number of returned log records. Waters, Sabbaghi and Accorsi use symmetric encryption, Savade and Zhao use asymmetric encryption. The proposed solution requires 1

hash operation per each keyword K, plus an hash and symmetric decryption operation per each number N of matches log records.

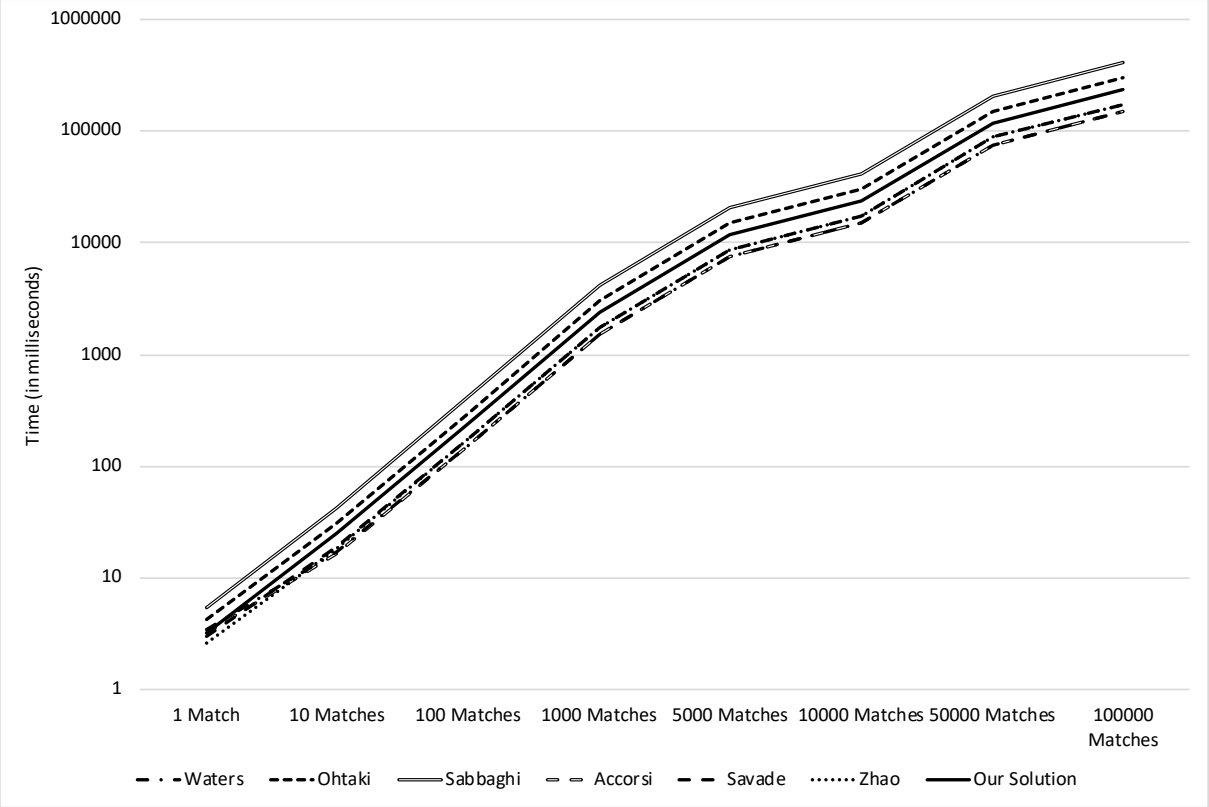

Figure 5.6: Comparison of the related work searching times

A projection of the time required to search and decrypt various numbers (N) of matching results was also done (see Figure 5.6). This projection uses the same time values for the different cryptographic operations that the ones used in the indexing projection. Additionally, a value $K = 1$ is assumed for the number of keywords comprised in the search query. Figure 5.6 uses a logarithmic scale. It shows that solutions with lesser computational cost, than the proposed solution, exist. Nonetheless, the performance of the proposed solution was deemed acceptable, since the level of search enhancement achieved by outweigh the differential computational cost of the search operation, the least frequent ones.

A second comparison motivated by the analysis of the validation sections of the solutions identified in Chapter 3 was done. That comparison is focused on the storage and time requirements for both index and search operations present on the validation sections of the related work. Nevertheless, not every author, whose work supports search, are referred on this comparison due to the lack of information regarding that type of testing of their solutions.

The solution proposed by Waters [114] presents some information regarding the storage space requirements. The author states that a 100 MB storage is capable of storing 800,000 public keys that cor-

respond to 800,000 keywords. The space needed to store the encrypted log records is not addressed. Our solution requires for the maximum key sizes (256 bits for AES, 512 bits for SHA-3) a storage space of 23 MB for the inverted index.

Regarding the indexing elapsed times, Waters states that for each keyword, his solution requires 180 milliseconds to compute its trapdoor, if that trapdoor is not already in a cache. If it is present on a cache, this operation takes 5 milliseconds. If we consider the 30,000 sample log records used on the performance tests and that only one keyword exists per log record, Waters solution would require roughly 25 minutes to index the keywords, considering a 5 milliseconds time per keyword. Our solution was capable of indexing the 30,000 sample log records, which contained more that one keyword per log record, in under 15 minutes.

In terms of searching, the solution proposed by Waters demands a time of 81 milliseconds to execute the required search operations for each entry. Although when searching for one singular entry, Waters solution is faster than our solution, when searching for multiple entries our solution becomes faster, since it does not follow a linear growth of the time elapsed for searching operations.

Ohtaki's first solution [116] only presents details regarding searching times. His solution depends on the number of records in the entire log sample and presents a search time almost linear to the number of matching records. Thus, although Ohtaki's solution presents faster search times for smaller number of matching records, our solution is not affected by the size of the entire log record and presents faster search times for bigger numbers of matching records. For instance, our solution requires an average value of 200 milliseconds to retrieve 3045 matching records and Ohtaki's solution takes 742 milliseconds to obtain 1007 matching records in a universe of 100,000 log records.

On Ohtaki's second work [118], which uses Bloom filters, the author states that for each log record entry a 2.6 MB storage is required for a set of twenty keywords. If such scheme was applied to the sample 30,000 log records, a 78 GB storage space would be needed in contrast to the 23 MB required by our solution to store the inverted index. Times used in indexing and searching operations were not detailed by the authors on their work.

Sabbaghi [119] presents tests for various numbers of stored log records, which indicate that his solution is affected by the existing total number of logs. For instance, in order to perform a search within 400,000 log lines his solution would take roughly 20 seconds (0,05ms per line) with a fixed length hash space and roughly 40 seconds (0,1ms per line) with a variable length hash space. Our solution was able to perform a search and decryption operation of 26,084 sample log lines in about 1,2 seconds (0,046 ms per line). If we assume an expected linear growth of the proposed solution on the time consumed to perform the search, it's possible to estimate that, for the 400,000 log lines, the proposed solution would require 18,4 seconds to search through and decrypt all lines.

Regarding the storage size required, Sabbaghi's solution needs roughly 20 MB to store the search-

able record authenticator when using a fixed length hash space, and about 50 MB when using a variable length hash space. Our solution requires 20 MB to store the encrypted inverted index using 128 bit size keys for AES and 256 for SHA-3 and 23 MB using 256 bit size keys for AES and 512 for SHA-3.

Accorsi, in [122], presents information regarding the times elapsed for the validation of the log records forward integrity. The author presents different values for multiple numbers of log entries. For instance, in order to verify 3,000 log records, his solutions requires roughly 950 milliseconds. Our solution achieved an average value of 300 milliseconds to verify the forward integrity of the 30,000 sample log records.

This comparison shows that the proposed solution is feasible. Regarding the times elapsed in indexing, searching and verification, our solution outperforms all related work that presented performance results in their work. Regarding storage size, our proposed solution also achieves the lowest storage requirements.

Finally, this comparison allow us to demonstrate that the proposed solution complies with requirements 8, 9 and 10, identified in Chapter 4. Requirement 8 says that it must be possible to store searchable encrypted log records in the most secure way using the low storage space required. We can assume that our solution achieves a reduced log size, since all the identified related work presents higher storage space numbers

Requirement 9 indicates that it must be possible to store log records in a secure way with acceptable celerity and without compromising the normal operation of the applications generating the logs. Requirement 10 says it must be possible to perform the elaborated search queries and retrieve matching records in useful time. Since our solution outperforms all related work, that present performance results in their work, we can assume that our solution is also compliant with this two requirements.

5.4 Security Analysis

As a system, our solution has to accomplish security. In general, this requirement comprises: indexing and query confidentiality and privacy. Moreover, it is necessary to prevent leakage due to index information, leakage due to search patterns, and leakage due to access patterns.

Index information leakage refers to the keywords comprised in the encrypted searchable index. Search pattern leakage consists in the information that can be derived from knowledge of whether two search results are from the same keyword, revealing that the same search was already performed in the past or not. The use of deterministic techniques for trapdoor generation directly leaks the search pattern, since the remote storage server may use statistical analysis and infer information about the query keywords. This requirement is known as predicate privacy [68]. Access patterns can be described as the set of search results (i.e, the collection of documents) that were obtained for a given keyword. This type of information might aid an attacker to learn information about the keywords since the remote storage

server will always return the same set of encrypted documents for the same encrypted keywords. In practice, the leakage of search and access patterns can be reduced but not totally eliminated [24].

5.4.1 Threat Model

In scientific literature, there is no standard security model regarding solutions based on searchable encryption [26]. The proposed solution was based assuming an honest-but-curious server that faithfully follows the protocol, but it is eager to learn confidential information by analysing the received encrypted log records, search queries and matching results in order to obtain information about the content of those log records.

Moreover, we assume the existence of external threats. Respectively, we consider that an attacker can attempt to read, alter or delete data, not only at rest but also in transit. Based on that, we focus on the protection against violation of privacy or integrity, data leakage, replay attacks, unauthorised access and spoofing. Additionally, the proposed solution must offer protection against two of the most common referred attacks in the searchable encryption field, these being the Chosen Keyword Attack (CKA) and the Keyword Guessing Attack (KGA). A CKA attack occurs when an attacker gathers knowledge about the stored information by obtaining the decryption of chosen keyword ciphertexts. The attacker choose an encrypted keyword and is handed the corresponding clear text. To assure the confidentiality of the keywords, the searchable encryption scheme must be semantically secure under chosen-keyword attack (SS-CKA) [80]. In a SS-CKA scheme, an attacker cannot learn information about the keywords that are present on the stored ciphertexts, if that keyword is not known to the server. This is, the knowledge of the trapdoor must not leak the knowledge of the keyword. A KGA attack can be performed on a searchable encryption scheme if the ciphertexts of all keywords were produced by the attacker. By knowing one trapdoor, an attacker can search the remaining ciphertexts for corresponding results. If the ciphertext that contains the keyword is found once, an attacker can guess the keyword that is correlated to the trapdoor. This type of attack is more frequent when the keyword space is small since the attacker can rapidly generate ciphertexts for of all keywords [126].

It also important to assert that we inspired this security analysis on the security definitions brought by Curtmola [31]. He pointed out that the security of both the indexes and the trapdoors are inherently linked, introducing two security notions, non-adaptive security against chosen-keyword attack (IND-CKA1) and adaptive security against chosen-keyword attack (IND-CKA2). This two definitions state that nothing should be leaked from the remotely stored documents and searchable index except the outcome of previously searched queries, providing security for trapdoors and assuring that the trapdoors do not leak information about the keywords, except for what can be inferred from the search and access patterns [25]. IND-CKA1 is a non-adaptive notion, in which the attacker does not contemplate previous search results. IND-CKA2 is a adaptive one, since the attacker chooses its queries with the knowledge

of the search results previously obtained.

5.4.2 Analysis

We assume that all cryptographic keys are saved on controlled locations, always remain is the possession of their owners and are not shared with any unauthorised entity. The security analysis of the proposed solution starts by the verification of the security properties that must be attained during the communication phase. The proposed solution devises two different communication protocols, one established between the file watchers and the SLS and the other established between the SLS and the SLaS Application. All communications between the involved parties are performed over SSL/TLS connections. Nevertheless, those protocols must assure not only the privacy of the data in transit but also integrity and authenticity of the involved parties.

Regarding the communication between the file watchers and the SLS, the integrity of the transmitted data is assured by the computation of an HMAC, named authentication tag (AT), over that data by the file watcher. The secret key, used on the AT computation, always remains secret and only in the possession of the file watcher and the SLS, so the authenticity of the file watcher is assured by the verification of such AT, performed by the SLS. Moreover, during this communication protocol, a timestamp TS is generated and used on the AT computation, assuring the indistinguishability between two tags of two distinct log records and preventing replay attacks. The inclusion of the log source identifier (LS_i), unique for every file watcher, makes that AT indistinguishability between two equal log records of two different file watchers. In order to enhance the privacy between the file watchers and the SLS, the data is asymmetrically encrypted with the public key of the SLS, which assures that only the SLS is able to decrypt it, since the corresponding private key always remains in its possession. Additionally, the authorisation of any file watcher can be revoked at any time by the SLS, thus any attempt of unauthorised access or spoofing can also be promptly mitigated, after detection.

For the communication between the SLS and the SLaS Application, a combination of a hybrid encryption scheme and an authentication scheme is used. The hybrid encryption scheme uses a symmetric key to encrypt log data and assure its privacy. The symmetric key used here is randomly generated, valid only for one communication trip and transferred between the involved parties asymmetrically encrypted with the receiver's public key. This way, only the entity with the corresponding private key is able to correctly decrypt the symmetric key and consequently decrypt the transferred information. The authentication technique is achieved by the use of asymmetrically computed signatures. Both entities possess a pair of keys and share their public keys between them, in an initial handshake. This way, one entity authenticates the other by validating the received signatures. Since the signatures are computed over the content of each communication, the verification of such signatures also assures the integrity of the information in transit.

The indexing operation of the proposed solution must assure privacy and integrity of the data at rest, but also index confidentiality and data leakage prevention due to index information must be accomplished. The data privacy at rest is given by the symmetric encryption of each log record prior to its transmission to the SLaS Application. Integrity comes from the computation of an HMAC H_{Ln} based on the encrypted log record LE_n, its identifier L_{ni} and its log source identifier LS_i. Additionally, each log record H_{Ln} is used on the construction of an hash chain. This hash chain links the log records in such way that makes it possible to detect any unauthorised modification or deletion.

The index confidentiality is assured since every keyword trapdoor that is to be added to the searchable index is computed, at the client side, through an HMAC, using a secret key that is only known by the SLS, hence no entity other than itself is able to generate trapdoors. Moreover, since an HMAC is a one-way function, it is unfeasible to an attacker to obtain the original keywords even if he has access to the trapdoors. Additionally, in order to enable field searching, alongside the keyword trapdoor, the SLS might include the type of each field. Each field name is mapped to a unique identifier and the unique identifier is then used, which hides the type of each field to the SLaS Application. Lastly, since every cryptographic operation necessary is performed at the client side, the possibility of data leakage due to index information is eliminated.

At search phase, the proposed solution must assure prevention against leakage due to search patterns and leakage due to access patterns. Moreover, it must be resilient against CKA and KGA attacks. Since the creation of the search capability uses deterministic techniques for trapdoor generation and the SLaS Application has knowledge about the search results obtained for a given keyword, our solution is in part vulnerable to search and access patterns leakage. However, that leakage is only made to the SLaS Application since the confidentiality of the search queries and consequent matching results is assured in transit by the devised communication protocol. Additionally, based on that leakage, the SLaS Application is only able to produce statistical information about the search activity, since the plaintext content of both the search queries and the matching log records is preserved. In detail, search queries are built by one-way functions, thus only a brute-force attack would be able to obtain the plaintext of that queries. The matching log records are symmetrically encrypted and decrypted at client side thus only the SLS is able to see the plaintext of such log records. Moreover, while designing the proposed solution we assumed that the benefits of constructing a more advanced and expressive query engine outweigh the search and access patterns leakage to an honest SLaS Application. In practical terms, we assumed that entities contract cloud services with providers that they have established some degree of trust.

Regarding CKA attacks, the proposed solution is not vulnerable, since in order to be able to perform such action, an attacker must gather knowledge about the stored information by obtaining the decryption of chosen keywords. Our solution makes use of HMACs to build the query trapdoors. Since HMACs are one-way functions produced with a secret key, always in possession of the SLS, it is unfeasible for an

attacker to obtain the original content of chosen trapdoors. KGA attacks are only possible if all the ciphertexts of all keywords were produced by the attacker. Since the SLS is the only entity able to generate keyword trapdoors, the proposed solution is also secure against KGA attacks. Regarding the IND-CKA1 and IND-CKA2 security definitions, the proposed solution is compliant with both since the security of trapdoors is assured and the only conceivable leakage is of the search and access patterns to the SLaS Application.

5.5 Summary

This chapter describes the validation of the proposed solution. The first set of tests focused on the functional aspects of the solution and prove the fulfilment of the requirements identified in Chapter 4. Those requirements were attained by the development and consequent appreciation of a prototype. All cryptographic operations of the prototype used standard algorithms. In particular, AES was used for data encryption, SHA-3 was used for data integrity and query trapdoor generation and RSA was used for communications.

The solution was also validated with respect to performance. The performance tests focused on the time required by the indexing and search operations and the required storage space for the encrypted data. Moreover, real-life log information from a publicly accessible web server with, about, thirty thousand (30,000) records per day, on average, was used. Experiments were performed with the following combinations: AES 128 and SHA-3 256; AES 192 and SHA-3 384; AES 256 and SHA-3 512. For communication purposes only RSA with 2048 bit size keys was used. Regarding the indexing, the use of symmetric cryptography with the respective biggest key sizes, guaranteeing a higher level of security, without compromising the performance of the solution is feasible. The symmetric cryptographic algorithms performance is not heavily influenced by the used key sizes. The time elapsed in all the tested combinations is similar. When using 256 bits keys for AES and 512 bit keys for SHA-3, the proposed solution took roughly 13 minutes to index the 30,000 sample log records.

In terms of storage space, when using the maximum key sizes for each algorithm, both the inverted index and the log records occupy 23 MB each, culminating in a total storage value of 46 MB. Based on that value, a projection of the required space for the storage of log records generated up to a period of one year was performed. The numbers show that if the log records are stored in clear text, roughly 10 GB of storage space is required. For the biggest key sizes, 256 for AES and 512 for SHA-3, roughly 17 GB of storage space is required annually. This comparison allowed to conclude that an additional 30% of space is required when compared with the clear text version. Although, the percentage achieved is larger than expected, we consider that the impact on the required storage of using encryption and keyed hashing with the biggest keys does not outweigh the increase in security.

Performance tests of the search operations used the same key sizes combinations and multiple search terms that would return different numbers of matching log lines. Following the behaviour already seen on the indexing tests, the search tests showed that the different keys sizes do not influence the speed of the search and decryption operations. A search operation that returned 26084 matches, representing roughly 87% of all the log records was done in an average of 1.2 seconds. Other search operations, such as the ones returning 3000 or less matches, were executed with an approximate average time of the same 200 milliseconds. An additional set of tests was executed in order to evaluate the performance of the forward integrity verification, an operation that enables detection if any of the stored log records were altered or deleted. Essentially, this test consists in the recalculation and verification of the hash chain. The tests prove that an average value of 300 milliseconds is required to verify the forward integrity of the 30,000 sample log records.

A comparison with the related work was also presented and was useful to show that the proposed solution is feasible. In a first observation, the number of cryptographic operations required for the index and search operations was analysed and was helpful to measure the distance between the computational cost of our solution and the ones in the related work. Based on the values presented, it was possible to conclude that the computational cost of the proposed solution in indexing is acceptable since it matches with the lowest one identified. In terms of searching, although solutions with less computational cost exist, we still acknowledge our solution acceptable, since the level of search enhancement achieved by outweigh the differential computational cost of the search operation. Regarding indexing, searching and verification elapsed times, our solution outperforms all related work that presented performance results in their work. Regarding storage size, our proposed solution also achieves the lowest storage requirements.

A security analysis was also performed in order to validate if the proposed solution achieves the desired security properties. This analysis showed that the proposed solution achieves the required confidentially, integrity and authenticity of data, not only at rest but also in transit, without loosing the search capability. The search capability also assures the privacy of the query and matching results. Moreover, it is not vulnerable to the most common attacks and compliant with the security definitions identified for searchable encryption solutions.

Chapter 6

Conclusion

The main goal of this work was the proposition of a new approach for storing critical operational logs in a cloud-based remote storage in order to achieve, in a unified way, the needed privacy, integrity and authenticity while maintaining server side searching capabilities by the logs owner.

The designed search algorithm enables conjunctive keywords queries plus a fine-grained search supported by field searching and nested queries, which are essential in the referred use case. To the best of our knowledge, the proposed solution is the first to introduce a query language that enables the construction of any query supported by the search algorithm.

The adopted reference scenario describes a cloud-based secure log storage service. The Client makes use of a web-based API to transfer its encrypted operational logs to the cloud, named SLaS Provider. In parallel, the Client can make use of the same WEB API to query, in a encrypted form, its logs and retrieve matching records. A SLaS application is foreseen in order to make use of the cloud's potential and to enable the transfer of some CPU-intensive tasks from the Client to the SLaS Provider. The SLaS Provider stores the client-side encrypted logs in its database and performs search operations over them, without having access to clear text logs.

6.1 Work Revision

The basis of the proposed solution is searchable encryption, a technique that encrypts data by such a scheme that enables keyword search to be conducted over the encrypted data. Chapter 2 describes the main aspects of cryptography, useful for a more clearer understanding of the concepts that are presented next. In detail, a definition of cryptography and its evolution trough time is detailed. A description of symmetric and asymmetric encryption, hash functions, keyed hashing and authenticated encryption are also included.

Searchable encryption is also detailed within Chapter 2. The analysis of this technique allows the

reader to understand its cryptographic potential, that has gained the attention of multiple researchers over the last few years, mainly due to the current data privacy concerns. The proposed searchable encryption algorithms are either based on the symmetric or asymmetric settings. The main evolution on this field of research is related to query expressiveness, being already possible to find solutions that allow a type of search as advanced as what is supported in the clear text field. However, there is room for much improvement since the efficiency and security of such solutions can still be enhanced. Our analysis allow us also to perceive that the trade-off between the query expressiveness, security and efficiency is one the most relevant challenges researchers face on the design of their solutions.

Chapter 3 addresses the current solutions related to the concept of securing logs. The solutions presented can be divided in two sets. The first set comprises solutions that provide secure logging frameworks, with confidentiality, integrity and authenticity guarantees. Of which, the most recent solutions are oriented to the remote storage of logs in cloud-based services. The second set solutions, despite enabling similar security properties, have as main characteristic the enabling of searchable encrypted logs, as a specific application of searchable encryption techniques.

The evaluation of the identified related work, enabled the recognition that the most common approach to enable data confidentiality and privacy is the encryption of the log records prior to its storage. Additionally, almost all solutions enable data integrity and authenticity by the use of a cryptographic hash chain. Regarding encrypted log searching, none of the presented solutions supports all the boolean operators nor fine-grained operations like field search and nested queries. Moreover, none of the solutions proposed in the related work offer all the desired security properties alongside an advanced searching capability, nor do they offer a query language that facilitates queries construction.

In Chapter 4 the proposed solution was presented, named SLS. The proposed solution assures the confidentiality, integrity and authenticity of data not only at rest but also in transit. Hence, two communication protocols are envisioned. One is applied to the connection between the file watches and the SLS, assuring the authentication of such file watchers based on HMAC computations and the privacy of the information by the use of asymmetric cryptography. The second protocol is used on the communication between the SLS and the SLaS Application and is based on a hybrid encryption scheme with authentication.

The indexing of the log records assures its confidentiality by the use of symmetric cryptography and secret keys, different for each log record. The integrity of each isolated log record is achieved by the computation of an HMAC. Forward Integrity arises from the construction of an hash chain, in which each link consists in an HMAC of the current log record, concatenated with the HMAC of the previous log record. To assure searchability, the logs records are either parsed with a regular expression or splitted using a delimiter. The former assures field search, by adding each field identifier alongside the specific search trapdoors. The latter generates the search trapdoors for each part of the splitted log record. This

search data is maintained by the SLaS Application on an encrypted inverted index in order to assure fast search operations. In order to facilitate the construction of any query supported by the search algorithms, a query language is proposed, supporting field search, boolean operators and nested queries.

Chapter 5 presents the validation of the proposed solution. A prototype was developed and tested, at first, in terms of the functional aspects, which proved that the requirements identified are met. The performance tests were next and focused on the time required by the indexing and search operations and the required storage space for the encrypted data. Experiments were performed with 30,000 log records and with the following cryptographic key combinations: AES 128 and SHA-3 256; AES 192 and SHA-3 384; AES 256 and SHA-3 512. For communication purposes only RSA with 2048 bit size keys was used.

Regarding the indexing, the use of symmetric cryptography enables the use of the larger key sizes of each algorithm, guaranteeing a higher level of security, without compromising the performance of the solution. The storage tests also showed a very small discrepancy between the values achieved when using all the tested key sizes. An additional set of tests was executed in order to evaluate the performance of the forward integrity verification. These tests prove that an average value of 300 milliseconds is required to verify the forward integrity of the 30,000 sample log records.

A comparison with the related work was also presented and was useful to show that the proposed solution is feasible. In a first observation, the number of cryptographic operations required for the index and search operations was analysed and was helpful to measure the distance between the computational cost of the proposed solution and the ones in the related work. In terms of indexing operations, the proposed solution has a computational cost equal to the lowest one identified. In terms of searching, although a solution with less operations exist, since the level of search enhancement achieved by outweigh the differential computational cost of the search operation. Moreover, the number of search operations, when compared to the number of indexing operations on a daily basis, are to few to have a real impact. Regarding indexing, searching and verification elapsed times and storage requirements, our solution outperforms all related work that presented performance results in their work.

As a system, the proposed solution has to accomplish security, hence a security analysis was performed in order to validate if the proposed solution achieves the desired security properties. The analysis showed that the proposed solution achieves the required confidentially, integrity and authenticity of data, not only at rest but also in transit, without hampering the search capability. The search capability also assures the privacy of both the query and matching results and it is not vulnerable to the most common referred attacks and compliant with the identified security definitions.

6.2 Results Characterisation

The work carried out in this master thesis lead to three main results: 1) secure remote storage solution; 2) secure search algorithm; 3) query language. To the best of our knowledge the third result – the query language – is new and it constitutes a contribution.

The secure remote storage solution supports the storage and retrieval of encrypted log records, providing, simultaneously, the integrity and authenticity of the data. The proposed solution aims at assuring such security properties not only for data at rest but also while data is in transit between the parties involved.

The secure search algorithm allows the logs owner to submit queries and retrieve matching log records. All the queries are submitted to the cloud in an encrypted form and performed over the encrypted data without any decryption or access to the clear text. The matching results are also returned in an encrypted form and only decrypted and visualised by the logs owner. Additionally, all the communication required by these operations is conducted over encrypted channels with authentication of the involved parties.

The query language enables the construction of any query supported by the search algorithm. In a language close to natural language as possible, the logs owner can describe queries as seamless as possible, which support complex conjunctive keywords, backed by boolean operators, field searching and sub queries. It allows a more fine-grained search with enhanced results.

6.3 System Limitations and Future Work

In the course of the work presented herein, future work opportunities arose. They can be described from two perspectives: improvement of the current scenario, and adaptation of the current solution to different scenarios.

Several improvements can be made: 1) eliminate encrypted search and access patterns leakage; 2) support ranked or proximity queries; 3) produce statistical information about the log records. The first could be obtained with the use of non deterministic encryption techniques to search the encrypted log records. Ranked searches can be achieved with a TF-IDF technique, which returns the most relevant documents first. The use of homomorphic encryption to measure the distance between the keywords could be explored to implement proximity queries. The third can be attained by the construction of dashboards.

The adaptation of the proposed solution to other types of data also arises as interesting, particularly when considering its search capability. An example could be the adaptation of the solution to medical data, which typically comprises sensitive information that requires privacy and confidentiality but still needs to be searchable.

Bibliography

[1] W. Stallings, *Network and internetwork security: principles and practice*. Prentice Hall Englewood Cliffs, NJ, 1995, vol. 1.

[2] D. Das, U. Lanjewar, and S. Sharma, "The art of cryptology: From ancient number system to strange number system," *International Journal of Application or Innovation in Engineering & Management (IJAIEM)*, vol. 2, no. 4, 2013.

[3] D. Luciano and G. Prichett, "Cryptology: From caesar ciphers to public-key cryptosystems," *The College Mathematics Journal*, vol. 18, no. 1, pp. 2–17, 1987.

[4] J.-P. Aumasson, *Serious Cryptography: A Practical Introduction to Modern Encryption*. No Starch Press, 2017.

[5] J. Katz and Y. Lindell, *Introduction to modern cryptography*. Chapman and Hall/CRC, 2014.

[6] M. Agrawal and P. Mishra, "A comparative survey on symmetric key encryption techniques," *International Journal on Computer Science and Engineering*, vol. 4, no. 5, p. 877, 2012.

[7] S. Chandra, S. Bhattacharyya, S. Paira, and S. S. Alam, "A study and analysis on symmetric cryptography," in *2014 International Conference on Science Engineering and Management Research (ICSEMR)*. IEEE, 2014, pp. 1–8.

[8] A. E. Standard, "Federal information processing standards publication 197," *FIPS PUB*, pp. 46–3, 2001.

[9] J. Daemen and V. Rijmen, *The design of Rijndael: AES-the advanced encryption standard*. Springer Science & Business Media, 2013.

[10] W. Diffie and M. E. Hellman, "Multiuser cryptographic techniques," in *Proceedings of the June 7-10, 1976, national computer conference and exposition*. ACM, 1976, pp. 109–112.

[11] M. Bellare, A. Boldyreva, and S. Micali, "Public-key encryption in a multi-user setting: Security proofs and improvements," in *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2000, pp. 259–274.

[12] R. L. Rivest, A. Shamir, and L. Adleman, “A method for obtaining digital signatures and public-key cryptosystems,” *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[13] S. Bakhtiari, R. Safavi-Naini, J. Pieprzyk *et al.*, “Cryptographic hash functions: A survey,” Citeseer, Tech. Rep., 1995.

[14] Q. H. Dang, “Secure hash standard,” National Institute of Standards & Technology, Tech. Rep., 2015.

[15] D. Eastlake and P. Jones, “Us secure hash algorithm 1 (sha1),” 2001.

[16] B. Schneier, “Schneier on security: cryptanalysis of sha-1,” *Schneier. com*, 2005.

[17] Q. Dang, “Changes in federal information processing standard (fips) 180-4, secure hash standard,” *Cryptologia*, vol. 37, no. 1, pp. 69–73, 2013.

[18] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, “Keccak,” in *Annual international conference on the theory and applications of cryptographic techniques*. Springer, 2013, pp. 313–314.

[19] H. Krawczyk, R. Canetti, and M. Bellare, “Hmac: Keyed-hashing for message authentication,” *Network Working Group RFC*, 1997.

[20] J. M. Turner, “The keyed-hash message authentication code (hmac),” *Federal Information Processing Standards Publication*, pp. 198–1, 2008.

[21] M. Bellare and C. Namprempre, “Authenticated encryption: Relations among notions and analysis of the generic composition paradigm,” in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2000, pp. 531–545.

[22] M. J. Dworkin, “Sp 800-38d. recommendation for block cipher modes of operation: Galois/counter mode (gcm) and gmac,” *IACR Cryptology ePrint Archive*, 2007.

[23] Y. Wang, J. Wang, and X. Chen, “Secure searchable encryption: a survey,” *Journal of Communications and Information Networks*, vol. 1, no. 4, pp. 52–65, 2016.

[24] R. Zhang, R. Xue, and L. Liu, “Searchable encryption for healthcare clouds: a survey,” *IEEE Transactions on Services Computing*, vol. 11, no. 6, pp. 978–996, 2017.

[25] C. Bösch, P. Hartel, W. Jonker, and A. Peter, “A survey of provably secure searchable encryption,” *ACM Computing Surveys (CSUR)*, vol. 47, no. 2, p. 18, 2015.

[26] J. Zhang, “Semantic-based searchable encryption in cloud: issues and challenges,” in *2015 First International Conference on Computational Intelligence Theory, Systems and Applications (CCITSA)*. IEEE, 2015, pp. 163–165.

[27] 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.

[28] E.-J. Goh *et al.*, “Secure indexes.” *IACR Cryptology ePrint Archive*, vol. 2003, p. 216, 2003.

[29] B. H. Bloom, “Space/time trade-offs in hash coding with allowable errors,” *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[30] Y.-C. Chang and M. Mitzenmacher, “Privacy preserving keyword searches on remote encrypted data,” in *International Conference on Applied Cryptography and Network Security*. Springer, 2005, pp. 442–455.

[31] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky, “Searchable symmetric encryption: improved definitions and efficient constructions,” *Journal of Computer Security*, vol. 19, no. 5, pp. 895–934, 2011.

[32] G. Amanatidis, A. Boldyreva, and A. O’Neill, “Provably-secure schemes for basic query support in outsourced databases,” in *IFIP Annual Conference on Data and Applications Security and Privacy*. Springer, 2007, pp. 14–30.

[33] P. Van Liesdonk, S. Sedghi, J. Doumen, P. Hartel, and W. Jonker, “Computationally efficient searchable symmetric encryption,” in *Workshop on Secure Data Management*. Springer, 2010, pp. 87–100.

[34] K. Kurosawa and Y. Ohtaki, “Uc-secure searchable symmetric encryption,” in *International Conference on Financial Cryptography and Data Security*. Springer, 2012, pp. 285–298.

[35] S. Kamara, C. Papamanthou, and T. Roeder, “Dynamic searchable symmetric encryption,” in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 965–976.

[36] S. Kamara and C. Papamanthou, “Parallel and dynamic searchable symmetric encryption,” in *International Conference on Financial Cryptography and Data Security*. Springer, 2013, pp. 258–274.

[37] D. Boneh, G. Di Crescenzo, R. Ostrovsky, and G. Persiano, “Public key encryption with keyword search,” in *International conference on the theory and applications of cryptographic techniques*. Springer, 2004, pp. 506–522.

[38] D. Boneh and M. Franklin, “Identity-based encryption from the weil pairing,” in *Annual international cryptology conference*. Springer, 2001, pp. 213–229.

[39] M. Abdalla, M. Bellare, D. Catalano, E. Kiltz, T. Kohno, T. Lange, J. Malone-Lee, G. Neven, P. Paillier, and H. Shi, "Searchable encryption revisited: Consistency properties, relation to anonymous ibe, and extensions," in *Annual International Cryptology Conference*. Springer, 2005, pp. 205–222.

[40] J. Baek, R. Safavi-Naini, and W. Susilo, "Public key encryption with keyword search revisited," in *International conference on Computational Science and Its Applications*. Springer, 2008, pp. 1249–1259.

[41] M. Bellare, A. Boldyreva, and A. O'Neill, "Deterministic and efficiently searchable encryption," in *Annual International Cryptology Conference*. Springer, 2007, pp. 535–552.

[42] G. Di Crescenzo and V. Saraswat, "Public key encryption with searchable keywords based on jacobi symbols," in *International Conference on Cryptology in India*. Springer, 2007, pp. 282–296.

[43] D. Khader, "Public key encryption with keyword search based on k-resilient ibe," in *International Conference on Computational Science and Its Applications*. Springer, 2007, pp. 1086–1095.

[44] H. S. Rhee, J. H. Park, W. Susilo, and D. H. Lee, "Improved searchable public key encryption with designated tester." in *AsiaCCS*, 2009, pp. 376–379.

[45] ——, "Trapdoor security in a searchable public-key encryption scheme with a designated tester," *Journal of Systems and Software*, vol. 83, no. 5, pp. 763–771, 2010.

[46] J. Camenisch, M. Kohlweiss, A. Rial, and C. Sheedy, "Blind and anonymous identity-based encryption and authorised private searches on public key encrypted data," in *International Workshop on Public Key Cryptography*. Springer, 2009, pp. 196–214.

[47] Q. Liu, G. Wang, and J. Wu, "An efficient privacy preserving keyword search scheme in cloud computing," in *2009 International Conference on Computational Science and Engineering*, vol. 2. IEEE, 2009, pp. 715–720.

[48] Q. Tang and L. Chen, "Public-key encryption with registered keyword search," in *European Public Key Infrastructure Workshop*. Springer, 2009, pp. 163–178.

[49] R. Zhang and H. Imai, "Combining public key encryption with keyword search and public key encryption," *IEICE transactions on information and systems*, vol. 92, no. 5, pp. 888–896, 2009.

[50] L. Ibraimi, S. Nikova, P. Hartel, and W. Jonker, "Public-key encryption with delegated search," in *International Conference on Applied Cryptography and Network Security*. Springer, 2011, pp. 532–549.

[51] P. Golle, J. Staddon, and B. Waters, “Secure conjunctive keyword search over encrypted data,” in *International Conference on Applied Cryptography and Network Security*. Springer, 2004, pp. 31–45.

[52] L. Ballard, S. Kamara, and F. Monrose, “Achieving efficient conjunctive keyword searches over encrypted data,” in *International Conference on Information and Communications Security*. Springer, 2005, pp. 414–426.

[53] A. Shamir, “How to share a secret,” *Communications of the ACM*, vol. 22, no. 11, pp. 612–613, 1979.

[54] P. Wang, H. Wang, and J. Pieprzyk, “Keyword field-free conjunctive keyword searches on encrypted data and extension for dynamic groups,” in *International conference on cryptology and network security*. Springer, 2008, pp. 178–195.

[55] D. Cash, S. Jarecki, C. Jutla, H. Krawczyk, M.-C. Roşu, and M. Steiner, “Highly-scalable searchable symmetric encryption with support for boolean queries,” in *Annual Cryptology Conference*. Springer, 2013, pp. 353–373.

[56] S. Faber, S. Jarecki, H. Krawczyk, Q. Nguyen, M. Rosu, and M. Steiner, “Rich queries on encrypted data: Beyond exact matches,” in *European Symposium on Research in Computer Security*. Springer, 2015, pp. 123–145.

[57] D. J. Park, K. Kim, and P. J. Lee, “Public key encryption with conjunctive field keyword search,” in *International Workshop on Information Security Applications*. Springer, 2004, pp. 73–86.

[58] D. J. Park, J. Cha, and P. J. Lee, “Searchable keyword-based encryption.” *IACR Cryptology ePrint Archive*, vol. 2005, p. 367, 2005.

[59] Y. H. Hwang and P. J. Lee, “Public key encryption with conjunctive keyword search and its extension to a multi-user system,” in *International conference on pairing-based cryptography*. Springer, 2007, pp. 2–22.

[60] D. Boneh and B. Waters, “Conjunctive, subset, and range queries on encrypted data,” in *Theory of Cryptography Conference*. Springer, 2007, pp. 535–554.

[61] E. Shi, J. Bethencourt, T. H. Chan, D. Song, and A. Perrig, “Multi-dimensional range query over encrypted data,” in *2007 IEEE Symposium on Security and Privacy (SP’07)*. IEEE, 2007, pp. 350–364.

[62] S. Sedghi, P. Van Liesdonk, S. Nikova, P. Hartel, and W. Jonker, “Searching keywords with wildcards on encrypted data,” in *International Conference on Security and Cryptography for Networks*. Springer, 2010, pp. 138–153.

[63] Y. Yang, X. Liu, R. H. Deng, and J. Weng, “Flexible wildcard searchable encryption system,” *IEEE Transactions on Services Computing*, 2017.

[64] P. V. Parmar, S. B. Padhar, S. N. Patel, N. I. Bhatt, and R. H. Jhaveri, “Survey of various homomorphic encryption algorithms and schemes,” *International Journal of Computer Applications*, vol. 91, no. 8, 2014.

[65] P. Paillier, “Public-key cryptosystems based on composite degree residuosity classes,” in *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 1999, pp. 223–238.

[66] H.-A. Park, B. H. Kim, D. H. Lee, Y. D. Chung, and J. Zhan, “Secure similarity search,” in *2007 IEEE International Conference on Granular Computing (GRC 2007)*. IEEE, 2007, pp. 598–598.

[67] A. Bookstein, V. A. Kulyukin, and T. Raita, “Generalized hamming distance,” *Information Retrieval*, vol. 5, no. 4, pp. 353–375, 2002.

[68] E. Shen, E. Shi, and B. Waters, “Predicate privacy in encryption systems,” in *Theory of Cryptography Conference*. Springer, 2009, pp. 457–473.

[69] C. Bösch, Q. Tang, P. Hartel, and W. Jonker, “Selective document retrieval from encrypted database,” in *International Conference on Information Security*. Springer, 2012, pp. 224–241.

[70] D. Boneh, C. Gentry, S. Halevi, F. Wang, and D. J. Wu, “Private database queries using somewhat homomorphic encryption,” in *International Conference on Applied Cryptography and Network Security*. Springer, 2013, pp. 102–118.

[71] Z. Brakerski and V. Vaikuntanathan, “Fully homomorphic encryption from ring-lwe and security for key dependent messages,” in *Annual cryptology conference*. Springer, 2011, pp. 505–524.

[72] J. Li, Q. Wang, C. Wang, N. Cao, K. Ren, and W. Lou, “Fuzzy keyword search over encrypted data in cloud computing,” in *2010 Proceedings IEEE INFOCOM*. IEEE, 2010, pp. 1–5.

[73] W. J. Masek and M. S. Paterson, “A faster algorithm computing string edit distances,” *Journal of Computer and System sciences*, vol. 20, no. 1, pp. 18–31, 1980.

[74] J. Li and X. Chen, “Efficient multi-user keyword search over encrypted data in cloud computing,” *Computing and Informatics*, vol. 32, no. 4, pp. 723–738, 2013.

[75] B. Wang, S. Yu, W. Lou, and Y. T. Hou, “Privacy-preserving multi-keyword fuzzy search over encrypted data in the cloud,” in *IEEE INFOCOM 2014-IEEE Conference on Computer Communications*. IEEE, 2014, pp. 2112–2120.

[76] M. Kuzu, M. S. Islam, and M. Kantarcioglu, “Efficient similarity search over encrypted data,” in *2012 IEEE 28th International Conference on Data Engineering*. IEEE, 2012, pp. 1156–1167.

[77] S. Niwattanakul, J. Singthongchai, E. Naenudorn, and S. Wanapu, “Using of jaccard coefficient for keywords similarity,” in *Proceedings of the international multiconference of engineers and computer scientists*, vol. 1, no. 6, 2013, pp. 380–384.

[78] J. Bringer, H. Chabanne, and B. Kindarji, “Error-tolerant searchable encryption,” in *2009 IEEE International Conference on Communications*. IEEE, 2009, pp. 1–6.

[79] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan, “Private information retrieval,” in *Proceedings of IEEE 36th Annual Foundations of Computer Science*. IEEE, 1995, pp. 41–50.

[80] P. Xu, H. Jin, Q. Wu, and W. Wang, “Public-key encryption with fuzzy keyword search: A provably secure scheme under keyword guessing attack,” *IEEE Transactions on computers*, vol. 62, no. 11, pp. 2266–2277, 2012.

[81] Z. Fu, X. Wu, C. Guan, X. Sun, and K. Ren, “Toward efficient multi-keyword fuzzy search over encrypted outsourced data with accuracy improvement,” *IEEE Transactions on Information Forensics and Security*, vol. 11, no. 12, pp. 2706–2716, 2016.

[82] C. Wang, N. Cao, J. Li, K. Ren, and W. Lou, “Secure ranked keyword search over encrypted cloud data,” in *2010 IEEE 30th international conference on distributed computing systems*. IEEE, 2010, pp. 253–262.

[83] C. Wang, N. Cao, K. Ren, and W. Lou, “Enabling secure and efficient ranked keyword search over outsourced cloud data,” *IEEE Transactions on parallel and distributed systems*, vol. 23, no. 8, pp. 1467–1479, 2011.

[84] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu, “Order preserving encryption for numeric data,” in *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. ACM, 2004, pp. 563–574.

[85] A. Boldyreva, N. Chenette, Y. Lee, and A. O’neill, “Order-preserving symmetric encryption,” in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2009, pp. 224–241.

[86] N. Cao, C. Wang, M. Li, K. Ren, and W. Lou, “Privacy-preserving multi-keyword ranked search over encrypted cloud data,” *IEEE Transactions on parallel and distributed systems*, vol. 25, no. 1, pp. 222–233, 2013.

[87] W. Sun, B. Wang, N. Cao, M. Li, W. Lou, Y. T. Hou, and H. Li, "Privacy-preserving multi-keyword text search in the cloud supporting similarity-based ranking," in *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*. ACM, 2013, pp. 71–82.

[88] ——, "Verifiable privacy-preserving multi-keyword text search in the cloud supporting similarity-based ranking," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 11, pp. 3025–3035, 2013.

[89] N. S. Khan, C. R. Krishna, and A. Khurana, "Secure ranked fuzzy multi-keyword search over outsourced encrypted cloud data," in *2014 International Conference on Computer and Communication Technology (ICCCT)*. IEEE, 2014, pp. 241–249.

[90] Z. Xia, X. Wang, X. Sun, and Q. Wang, "A secure and dynamic multi-keyword ranked search scheme over encrypted cloud data," *IEEE transactions on parallel and distributed systems*, vol. 27, no. 2, pp. 340–352, 2015.

[91] C. Chen, X. Zhu, P. Shen, J. Hu, S. Guo, Z. Tari, and A. Y. Zomaya, "An efficient privacy-preserving ranked keyword search method," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 4, pp. 951–963, 2015.

[92] C. Orencik, A. Selcuk, E. Savas, and M. Kantarcioglu, "Multi-keyword search over encrypted data with scoring and search pattern obfuscation," *International Journal of Information Security*, vol. 15, no. 3, pp. 251–269, 2016.

[93] C. Guo, R. Zhuang, C.-C. Chang, and Q. Yuan, "Dynamic multi-keyword ranked search based on bloom filter over encrypted cloud data," *IEEE Access*, vol. 7, pp. 35 826–35 837, 2019.

[94] M. Bellare and B. Yee, "Forward integrity for secure audit logs," Technical report, Computer Science and Engineering Department, University of . . . , Tech. Rep., 1997.

[95] B. Schneier and J. Kelsey, "Secure audit logs to support computer forensics," *ACM Transactions on Information and System Security (TISSEC)*, vol. 2, no. 2, pp. 159–176, 1999.

[96] Y.-C. Hu, M. Jakobsson, and A. Perrig, "Efficient constructions for one-way hash chains," in *International Conference on Applied Cryptography and Network Security*. Springer, 2005, pp. 423–441.

[97] M. Franklin, "A survey of key evolving cryptosystems," *International Journal of Security and Networks*, vol. 1, no. 1-2, pp. 46–53, 2006.

[98] D. V. Forte, C. Maruti, M. R. Vetturi, and M. Zambelli, “Secsyslog: An approach to secure logging based on covert channels,” in *Systematic Approaches to Digital Forensic Engineering, 2005. First International Workshop on*. IEEE, 2005, pp. 248–263.

[99] R. Gerhards *et al.*, “Rfc 5424: The syslog protocol,” *Request for Comments, IETF*, 2009.

[100] R. L. Rivest, “The MD5 Message-Digest Algorithm,” RFC 1321, Apr. 1992. [Online]. Available: https://rfc-editor.org/rfc/rfc1321.txt

[101] D. E. E. 3rd and P. Jones, “US Secure Hash Algorithm 1 (SHA1),” RFC 3174, Sep. 2001. [Online]. Available: https://rfc-editor.org/rfc/rfc3174.txt

[102] J. E. Holt, “Logcrypt: forward security and public verification for secure audit logs,” in *ACM International Conference Proceeding Series*, vol. 167, 2006, pp. 203–211.

[103] A. Shamir, “Identity-based cryptosystems and signature schemes,” in *Workshop on the theory and application of cryptographic techniques*. Springer, 1984, pp. 47–53.

[104] D. Ma and G. Tsudik, “A new approach to secure logging,” *ACM Transactions on Storage (TOS)*, vol. 5, no. 1, p. 2, 2009.

[105] D. Ma, “Practical forward secure sequential aggregate signatures,” in *Proceedings of the 2008 ACM symposium on Information, computer and communications security*. ACM, 2008, pp. 341–352.

[106] I. Ray, K. Belyaev, M. Strizhov, D. Mulamba, and M. Rajaram, “Secure logging as a service—delegating log management to the cloud,” *IEEE systems journal*, vol. 7, no. 2, pp. 323–334, 2013.

[107] A. Herzberg, S. Jarecki, H. Krawczyk, and M. Yung, “Proactive secret sharing or: How to cope with perpetual leakage,” in *Annual International Cryptology Conference*. Springer, 1995, pp. 339–352.

[108] I. Teranishi, J. Furukawa, and K. Sako, “K-times anonymous authentication,” in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2004, pp. 308–322.

[109] S. Zawoad, A. K. Dutta, and R. Hasan, “Seclaas: secure logging-as-a-service for cloud forensics,” in *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*. ACM, 2013, pp. 219–230.

[110] S. Zawoad, A. Dutta, and R. Hasan, “Towards building forensics enabled cloud through secure logging-as-a-service,” *IEEE Transactions on Dependable and Secure Computing*, no. 1, pp. 1–1, 2016.

[111] T. Grance, S. Chevalier, K. K. Scarfone, and H. Dang, “Guide to integrating forensic techniques into incident response,” National Institute of Standards & Technology, Tech. Rep., 2006.

[112] N. C. C. F. S. W. Group *et al.*, “Nist cloud computing forensic science challenges,” National Institute of Standards and Technology, Tech. Rep., 2014.

[113] J. Benaloh and M. De Mare, “One-way accumulators: A decentralized alternative to digital signatures,” in *Workshop on the Theory and Application of of Cryptographic Techniques*. Springer, 1993, pp. 274–285.

[114] B. R. Waters, D. Balfanz, G. Durfee, and D. K. Smetters, “Building an encrypted and searchable audit log.” in *NDSS*, vol. 4, 2004, pp. 5–6.

[115] B. P. Gopularam, S. Dara, and N. Niranjan, “Experiments in encrypted and searchable network audit logs,” in *2015 International Conference on Emerging Information Technology and Engineering Solutions*. IEEE, 2015, pp. 18–22.

[116] Y. Ohtaki, “Constructing a searchable encrypted log using encrypted inverted indexes,” in *Cyberworlds, 2005. International Conference on*. IEEE, 2005, pp. 7–pp.

[117] Y. Ohtaki, M. Kamada, and K. Kurosawa, “A scheme for partial disclosure of transaction log,” *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. 88, no. 1, pp. 222–229, 2005.

[118] Y. Ohtaki, “Partial disclosure of searchable encrypted data with support for boolean queries,” in *Availability, Reliability and Security, 2008. ARES 08. Third International Conference on*. IEEE, 2008, pp. 1083–1090.

[119] A. Sabbaghi and F. Mahmoudi, “Establishing an efficient and searchable encrypted log using record authenticator,” in *Computer Technology and Development, 2009. ICCTD’09. International Conference on*, vol. 2. IEEE, 2009, pp. 206–211.

[120] R. Accorsi, “On the relationship of privacy and secure remote logging in dynamic systems,” in *IFIP International Information Security Conference*. Springer, 2006, pp. 329–339.

[121] R. Accorsi and A. Hohl, “Delegating secure logging in pervasive computing systems,” in *International Conference on Security in Pervasive Computing*. Springer, 2006, pp. 58–72.

[122] R. Accorsi, “Bbox: A distributed secure log architecture,” in *European Public Key Infrastructure Workshop*. Springer, 2010, pp. 109–124.

[123] L. C. Savade and S. Chavan, “A technique to search log records using system of linear equations,” in *2012 CSI Sixth International Conference on Software Engineering (CONSEG)*. IEEE, 2012, pp. 1–4.

[124] P. N. Sabes, “Linear algebraic equations, svd, and the pseudo-inverse,” *San Francisco, Oct*, 2001.

[125] W. Zhao, L. Qiang, H. Zou, A. Zhang, and J. Li, “Privacy-preserving and unforgeable searchable encrypted audit logs for cloud storage,” in *2018 5th IEEE International Conference on Cyber Security and Cloud Computing (CSCloud)/2018 4th IEEE International Conference on Edge Computing and Scalable Cloud (EdgeCom)*. IEEE, 2018, pp. 29–34.

[126] J. W. Byun, H. S. Rhee, H.-A. Park, and D. H. Lee, “Off-line keyword guessing attacks on recent keyword search schemes over encrypted data,” in *Workshop on Secure Data Management*. Springer, 2006, pp. 75–83.