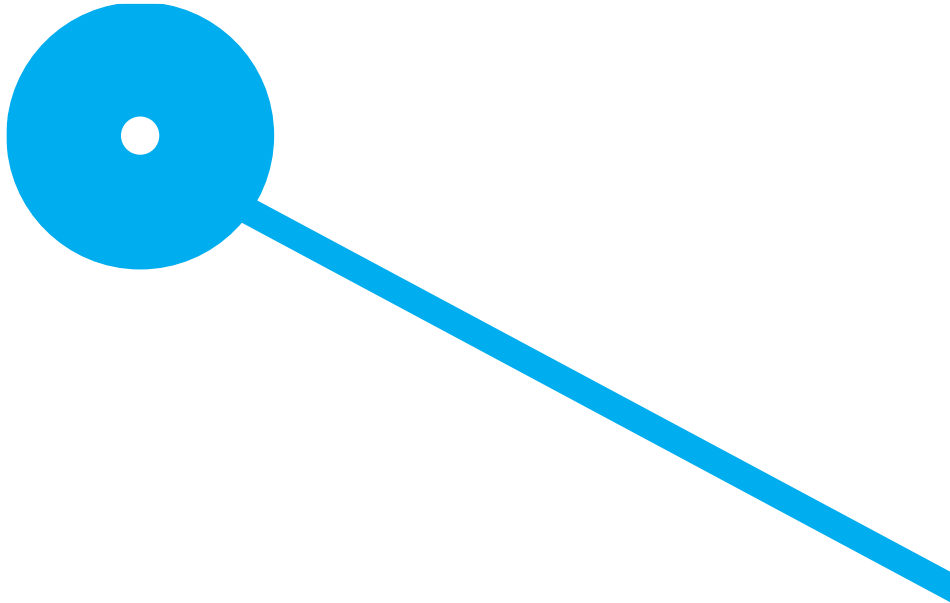


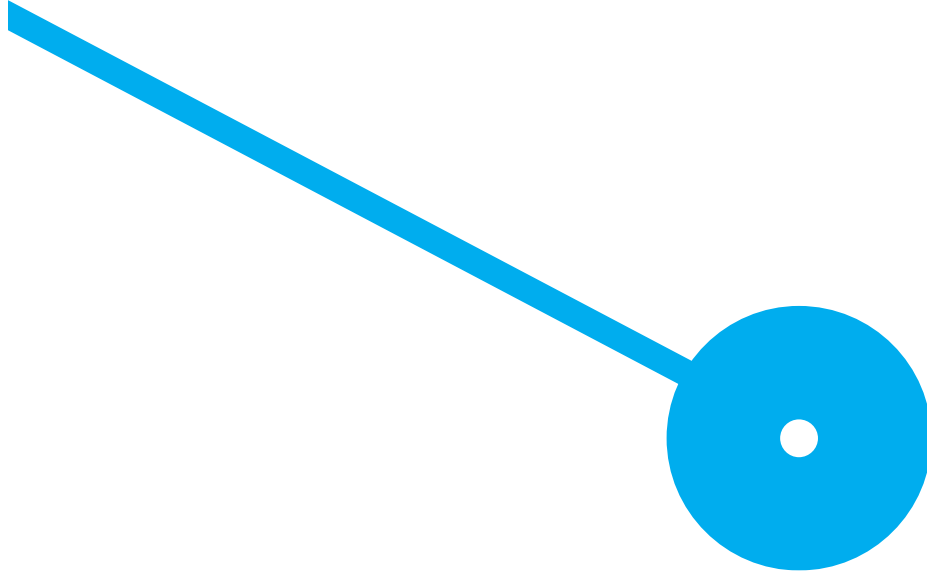
Controlled and Secure Sharing Threat  
Intelligence  
Ricardo Fernandes



# Controlled and Secure Sharing Threat Intelligence

Ricardo Fernandes

12/2021





# Controlled and Secure Sharing Threat Intelligence

Ricardo Manuel de Moura Fernandes

Orientadores: Prof. António Alberto dos Santos Pinto, Prof.  
Pedro Filipe Cruz Pinto



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Background</b>	<b>3</b>
2.1	Encryption . . . . .	3
2.1.1	Block ciphers . . . . .	4
2.1.2	Public-key encryption . . . . .	5
2.2	Secure hash functions . . . . .	6
2.3	Message authentication codes . . . . .	7
<b>3</b>	<b>Searchable Encryption</b>	<b>9</b>
<b>4</b>	<b>Proposed solution</b>	<b>15</b>
4.1	Requirements . . . . .	15
4.2	Specification . . . . .	16
4.2.1	Adopted notation . . . . .	17
4.2.2	Peering Configuration and Validation . . . . .	18
4.2.3	Data synchronisation . . . . .	19
4.2.4	Data searching . . . . .	20
<b>5</b>	<b>Implementation</b>	<b>25</b>
5.1	Initial setup . . . . .	25
5.2	Peering validation . . . . .	27
5.3	Data synchronisation . . . . .	30
5.3.1	Synchronisation configuration . . . . .	30
5.3.2	Data control policies . . . . .	31
5.3.3	Data query . . . . .	33
5.3.4	Synchronisation request . . . . .	34
5.4	Data sharing through a shared index . . . . .	39
5.4.1	Sharing groups . . . . .	39
5.4.2	Symmetric key generation . . . . .	41
5.4.3	Shared Index Update . . . . .	46
5.4.4	Shared Index Searching . . . . .	50
<b>6</b>	<b>Prototype validation</b>	<b>55</b>
6.1	Peering between entities . . . . .	56
6.2	Data synchronisation . . . . .	58
6.3	Shared index . . . . .	61
6.4	Discussion . . . . .	63



# List of Figures

1.1	Overview of a Malware Information Sharing Platform (MISP) community . . . . .	2
3.1	Forward and reverse index . . . . .	10
3.2	SE model . . . . .	11
4.1	Reference scenario . . . . .	15
4.2	Proposed solution . . . . .	16
4.3	Process of peering validation . . . . .	19
4.4	Data synchronisation between entities . . . . .	21
4.5	Creation of symmetric key . . . . .	22
4.6	Shared index update . . . . .	23
4.7	Data search through a shared index . . . . .	24
6.1	Implemented prototype . . . . .	56
6.2	Entity creation route . . . . .	56
6.3	Public key upload route . . . . .	57
6.4	Peer validation route . . . . .	57
6.5	Synchronisation scheduling route . . . . .	58
6.6	Sample input policies . . . . .	59
6.7	Sample output policies . . . . .	60
6.8	Synchronisation output . . . . .	61
6.9	Group creation route . . . . .	62
6.10	Association of an entity to the group . . . . .	62
6.11	Symmetric key generation route . . . . .	62
6.12	Shared index configuration route . . . . .	63
6.13	Index configuration values . . . . .	64
6.14	Visual representation of the shared index . . . . .	65
6.15	Route to search in the shared index . . . . .	66



# List of Tables

4.1	Adopted notation . . . . .	18
-----	----------------------------	----





# Listings

4.1	Example of a data query . . . . .	19
5.1	UUID Generation . . . . .	25
5.2	Key Pair Generation . . . . .	26
5.3	Peering validation - Request . . . . .	27
5.4	Peering validation - Response . . . . .	28
5.5	Synchronisation configuration . . . . .	31
5.6	Policy configuration . . . . .	32
5.7	The queryBuilder Function . . . . .	33
5.8	Function used for input policy filtering . . . . .	34
5.9	Data synchronisation request . . . . .	35
5.10	Output Policy Control Function . . . . .	37
5.11	MISP search function . . . . .	38
5.12	Shared Group Creation . . . . .	39
5.13	Adding Entity to Shared Group . . . . .	40
5.14	Symmetric key Generation - Initial request . . . . .	41
5.15	Symmetric key Generation - Receiving a request . . . . .	43
5.16	Configuration of the Creation of a Shared Index . . . . .	46
5.17	Index Update . . . . .	47
5.18	Searching a term . . . . .	50
5.19	Decryption and verification of a search result . . . . .	50
5.20	Overall search process on a shared index . . . . .	52



# List of Acronyms

<b>AES</b>	Advanced Encryption Standard
<b>API</b>	Application Programming Interface
<b>BF</b>	Bloom Filter
<b>CBC</b>	Cipher Block Chaining
<b>CTI</b>	Cyber-threat Information
<b>DS-PEKS</b>	Dual-server PEKS
<b>DSSE</b>	Dynamic Searchable Symmetric Encryption
<b>ECB</b>	Electronic Codebook
<b>ESE</b>	Efficiently searchable encryption
<b>HMAC</b>	Hash-based Message Authentication Code
<b>IBE</b>	Identity Based Encryption
<b>IND-SCF-CKCA</b>	Indistinguishability of secure channel free PEKS against chosen keyword and ciphertext attack
<b>IND-KGA</b>	Indistinguishability against keyword guessing attack
<b>IoC</b>	Indicators of Compromise
<b>JSON</b>	JavaScript Object Notation
<b>MAC</b>	Message Authentication Code
<b>MISP</b>	Malware Information Sharing Platform
<b>NIST</b>	National Institute of Standards and Technology
<b>NSA</b>	National Security Agency
<b>PAP</b>	Permissible Actions Protocol
<b>PEKS</b>	Public Key Encryption with Keyword Search
<b>PKG</b>	Private Key Generator
<b>SE</b>	Searchable Encryption

<b>SE-EPOM</b>	Searchable Encryption based on Efficient Privacy-preserving Outsourced calculation framework with Multiple keys
<b>SHA</b>	Secure Hash Algorithm
<b>SSE</b>	Symmetric Searchable Encryption
<b>RSA</b>	Rivest-Shamir-Adleman
<b>RSA-DOAEP</b>	RSA Deterministic Optimal Asymmetric Encryption Padding
<b>RSA-OAEP</b>	RSA Optimal Asymmetric Encryption Padding
<b>URL</b>	Uniform Resource Locator
<b>UUID</b>	Universally Unique Identifier

# Acknowledgements

First of all, a mention to the ESTG for providing me with this master's course, allowing me to improve my knowledge in the area of Computer Engineering.

To the professors António Pinto and Pedro Pinto that always demonstrated availability in the resolution of the several problems that were appearing along this journey.

Last but not least, to my parents who always supported me. Without their great efforts, it would not have been possible to accomplish this course.



# Abstract

Cyber threat information sharing platforms have become a useful weapon for dealing with cyberattacks, proactively mitigating them and thus reducing risk exposure. These allow multiple agencies to connect with each other, forming a community, and share that same intrusion information regarding cyberattacks or threats with each other.

The Malware Information Sharing Platform (MISP) is particularly developed to promote the open dissemination of information such as intrusion indicators within a community. This exchange of information related to threats or incidents is treated as a data synchronisation procedure between different MISP instances, which may belong to one or more communities, companies or organisations. However, this platform presents limitations if its information is considered as classified or shared only for a certain period of time. This implies that this information should be treated only in encrypted form. One solution is to use MISP with searchable encryption techniques to impose greater control over information sharing.

In this document, it is present a system that guarantees a controlled synchronisation of information between entities through the use of encrypted search techniques to guarantee the confidentiality of the information present in the MISP platform and also the use of synchronisation policies to control the way information is exchanged.

**Keywords:** Searchable encryption, Classified information, Information sharing.





# Resumo

As plataformas de partilha de informação de ciberameaças tornaram-se uma arma útil para lidar com os ciberataques, mitigando-os proativamente e, assim, reduzindo a exposição ao risco. Estas permitem que vários organismos se liguem entre si, formando uma comunidade, e que partilhem entre si essas mesmas informações de intrusão relativas a ataques ou ameaças cibernéticas.

A plataforma Malware Information Sharing Platform (MISP) está particularmente desenvolvido para promover a disseminação aberta de informação como os indicadores de intrusão no seio de uma comunidade. Esta troca de informação relacionada com ameaças ou incidentes é tratada como um procedimento de sincronização de dados entre diferentes instâncias MISP, que podem pertencer a uma ou mais comunidades, empresas ou organizações. No entanto, esta plataforma apresenta limitações se a sua informação for considerada como classificada ou partilhada apenas por um determinado período de tempo. Isto implica que esta informação deve ser tratada apenas de forma encriptada. Uma solução é utilizar MISP com técnicas de cifragem pesquisáveis para impor um maior controlo sobre a partilha de informação.

Neste documento apresentamos um sistema que garante uma sincronização controlada da informação entre entidades por meio do uso de técnicas de pesquisa cifrada para garantir a confidencialidade das informações presentes na plataforma MISP e ainda o uso de políticas de sincronização para controlar o modo como as informações são trocadas.

**Pavras chave:** Encriptação pesquisável, informação classificada, partilha de informação.



# Chapter 1

## Introduction

Cyberthreat information sharing platforms are an important weapon against cyberattacks, proactively mitigating them and thus, reducing risk exposure of networks, systems, and data. These platforms allow multiple agencies to form a community that connects and shares intrusion information concerning attacks or threats.

The MISP [1] is particularly designed to promote the open dissemination of threat or incident-related information such as Indicators of Compromise (IoC) within a community. This information exchange is treated as a data synchronisation procedure between different MISP instances, which may belong to one or more communities, companies, or organisations.

MISP information sharing can only be controlled to a certain extent<sup>1</sup>, presenting limitations whenever an organisation wishes to impose stricter control on this dissemination. The MISP platform includes features to control the dissemination of information such as the use of sharing groups or the use of labels, by using Permissible Actions Protocol (PAP) [2] to manage the access of specific information. However, it imposes no real control on the dissemination of threat information, e.g. in case an event information tagged with PAP:RED (a tag indicating that the information should not be used externally, such as checking whether if a file hash is present in a malware database) is shared with another organisation, there is no way to enforce it since the information has already been exchanged between MISP instances. In essence, MISP assumes that there is an always present trust relationship between the organisations that exchange information and this may not always be the case. Particularly, if one considers classified information exchange between military entities that may cooperate in an international context for a specific period of time.

In this document a controlled information sharing solution is proposed using searchable encryption techniques to impose greater control over information sharing in MISP. An overview of the proposed solution is presented in Fig. 1.1. It includes a controlled synchronisation process that works as a complement to the already existing synchronisation between MISP instances. The proposed solution uses an asynchronous search engine built as a REST Application Programming Interface (API) to enable secure communication between entities and thus, enables the secure sharing and searching of intrusion related information

---

<sup>1</sup><https://misp-project.org/taxonomies.html>

between these entities without requiring access to cleartext information.

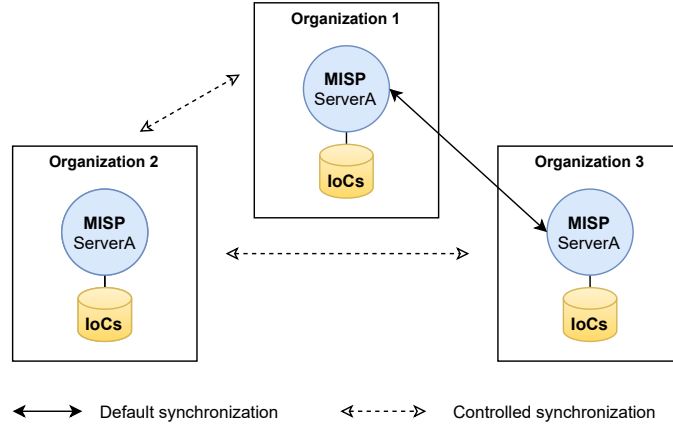


Figure 1.1: Overview of a MISP community

This document is structured as follows. Chapter 2 presents some cryptographic concepts used in this work. Chapter 3 refers to the concept of searchable encryption. Chapter 4 and Chapter 5 present respectively the specification of the proposed solution and its implementation. Chapter 6 describes the validation of a prototype created while Chapter 7 presents the conclusions of this work.

## Chapter 2

# Background

Cryptography is the technique used to keep data inaccessible by third parties in cases where it is necessary to restrict access to the data, for example in a conversation between two parties or to store sensitive data. The earliest accounts of cryptography date back to ancient Egyptian times [3]. The most famous accounts of cryptography point to Julius Caesar in the year 100BC, where messages exchanged by troops during the conquest of the Roman Empire were encrypted by shifting each letter of the message three positions down in the alphabet. Over time this technique became weak as it was enough to do the reverse shift to discover the message that had been encrypted [4]. In the 16<sup>th</sup> Century, an improved version of the Caesar cipher was presented called the Vigenère cipher. The difference was that the number of shifts was no longer a fixed number but was defined by values defined in a key. This key was composed of a set of letters representing the number of shifts to be made based on their position in the alphabet. However, for the same message and the same key, the encrypted result was always the same and, despite being more reliable than Caesar's cipher, the cipher also became impractical because it was easily broken by analysing the frequency of letters, and due to being a deterministic cipher [5]. Later, in the 20<sup>th</sup> Century, technology had great advances that influenced cryptology, the best-known case being the German Enigma machine [6]. This mechanism was a key piece for German troops during World War II for secret messaging. This mechanism was broken by the British, who spent years and many resources to understand its operation, leading to the victory of the allied troops in the war and consequently its end.

More recent cryptographic algorithms are based on mathematical theory and computer science practice and are designed around concepts such as factorisation and the discrete logarithm problem, because they are easy problems to apply but very difficult to solve. And as computational power advances over the years, algorithms continue to evolve, usually by increasing the size of encryption keys in order to prevent ciphers from being broken in a short time [7].

### 2.1 Encryption

Encryption [8] is a central concept of cryptography. Considering the following case: two persons, Alice and Bob, want to speak with each other. In a general

situation, the channels they use to communicate is not secure. A third person, called Eve, is listening to the channel, so she receives every message  $m$  that both Alice and Bob send to each other. In this case, the use of encryption is a good solution to guarantee the security of the channel and consequently avoiding Eve knowing the content of the messages. To do this, firstly, Alice and Bob should agree on a secret key  $K$  in some way that Eve cannot listen to. When, for example, Bob wants to send a message  $m$  to Alice, the message  $m$  should be encrypted with an encryption function  $E(K, m)$  resulting in a ciphertext  $c$ . Receiving the ciphertext  $c$ , Alice only needs to decrypt it using the secret key  $K$  with a decryption function  $D(K, c)$  to get the original message  $m$  from Bob. Eve cannot read the message  $m$  because she does not know the secret key  $K$ , so she cannot decrypt the ciphertext  $c$ . This basic mechanism of encryption can be applied to multiple scenarios like e-mail security and data storage with different approaches.

In the previous scenario, it is mentioned a case of symmetric encryption. Symmetric encryption requires a single secret key to encrypt and decrypt the messages. One example of an algorithm for symmetric encryption is Advanced Encryption Standard (AES).

### 2.1.1 Block ciphers

A block cipher [9] is an encryption function for fixed-size blocks of data. Nowadays, block ciphers have 128 bits of size, encrypt plaintext with 128-bit and generate the ciphertext with 128 bits of size. These blocks are reversible which means that there is a function that decrypts a ciphertext to the original plaintext [10]. Block ciphers encrypt data with a secret key, which consists of a string of bits like the plaintext and ciphertext. Keys sizes vary, examples being 128 or 256 bits. The common representations for encryption are  $E(K, p)$ , considering the key  $K$  and plaintext  $p$ , and representations for decryption are  $D(K, c)$ , given a key  $K$  and a ciphertext  $c$ . To increase the security of block ciphers the use of block cipher modes is recommended, when compared to using a block cipher directly.

AES is a well known block cipher [11] that arose from a National Institute of Standards and Technology (NIST) request for proposals to the cryptographic community, where 15 proposals were submitted [12]. From these proposals, five were finalists [13], and Rijndael became the AES standard in 2001 [14, 15]. In a single round of AES, the plaintext with 16 bytes (128 bits) is XOR'ed with 16 bytes of the round key. After that, each byte is processed by an index table that maps 8-bit inputs to 8-bit outputs. When the indexing is completed, these bytes are rearranged in a specific order and then mixed in 4-byte groups. This last operation uses a linear mixing function which means that each output bit is the result of the XOR operation applied to several input bits. The full encryption can have a different number of rounds. With 128-bit keys, the algorithm uses 10 rounds. For 192-bit keys, 12 rounds are used. For 256-bit keys, 14 rounds are used. One of the advantages of AES is parallelism. The steps consist of several operations performed in parallel which enables high-speed implementations. The rounds used for each key length were defined after the algorithm presentation, where was shown an attack to AES if the number of rounds defined was 6 [16]. Posteriorly the attacks were improved to handle other numbers of rounds [17], less than the final rounds chosen, but these attacks covered only

70% of the cipher. This algorithm was selected based on the assumption that there will no large improvements on future attacks.

When there is a need to encrypt data with a size other than the fixed block size, the solution consists on using block cipher modes [9]. Two well known cipher modes are the Electronic Codebook (ECB) [18] and the Cipher Block Chaining (CBC) [19]. In ECB, message blocks are encrypted separately, meaning that two identical blocks will result in the same the encrypted blocks. Depending on the message structure, the attacker may obtain some information about the message. In CBC, the encryption of a message block also will also include the result of the encryption of the previous block, thus avoiding the problem of the ECB mode. In the case of equal message blocks, these will result in different blocks, reducing the information available to an attacker.

A known issue of symmetric encryption related to key management. A secret key  $K$  must be shared between all involved parties. One problem relates to how one should share the key so that it remains known only to relevant parties. A third person can be listening to the channel used to share the key. Another problem relates to group size, or specifically how to manage the keys when we have a group of more than two people that want to communicate with each other. An option could be the generation of a shared group key that comprises contributes from all members. In this case, assuming that  $n$  represents the number of members of the group, each member would have to exchange  $n - 1$  messages. The group would have to exchange  $\sum_{i=1}^{n-1} i$  keys. This number increases with the number of elements of the group, making the key management problem worse.

### 2.1.2 Public-key encryption

The use of public-key encryption [20], also called asymmetric-key encryption, make use of two different, but related, encryption keys. One to encrypt a message and the other to decrypt that message. To do this, firstly, Bob needs to generate a pair of asymmetric keys, one of these is his secret key ( $S_{Bob}$ ), and the other is his public key ( $P_{Bob}$ ). Then he can share his public key with everyone. When someone wants to send a message to Bob, the public key ( $P_{Bob}$ ) is used to encrypt the message  $m$ , resulting in the ciphertext  $c = E(P_{Bob}, m)$ , and send it to Bob. Bob, the only person who knows  $S_{Bob}$ , can decrypt the received message.

Public-key encryption algorithms are based on mathematical problems such as factorisation or discrete algorithm [21]. A very important and obvious requirement of these algorithms is that it should not be possible to generate the secret key from the public key [22]. The problem of key distribution is simpler with this type of encryption. Considering the same group with  $n$  members, and each member only need to share his public key, thus the total number of keys shared is  $n$  keys. However, this schema is not so efficient when compared to symmetric encryption, with regards to CPU usage and storage space. The usual approach to make use of the advantages of the asymmetry encryption, while still being efficient as the symmetric encryption, is to use both in a combined manner. Public-key algorithms are used to exchange a symmetric secret key that is then used to encrypt the data.

Rivest-Shamir-Adleman (RSA) [23] is a well-known asymmetric cipher algorithm, first published in 1978 [24]. This algorithm is based on the factorisation



of large numbers that are the product between two large prime numbers, in the order of a thousand bits in length or more. Its architecture works as a one-way function in which it is easy to compute in one direction but almost impossible to compute in the reverse direction. The difficulty lies in trying to discover the two large prime numbers used. The RSA system [25] starts by defining two large prime numbers  $p$  and  $q$  and then computing  $n = pq$ . The values  $p$  and  $q$  must be of similar size, resulting in a modulus  $n$  with twice the size of the prime values. Then two exponents  $e$  and  $d$  are used and the requirement is  $ed = 1(\text{mod } t)$  where  $t := \text{lcm}(p-1, q-1)$  and  $\text{lcm}()$  means the lowest common multiple, the lowest number that is evenly divisible by  $p$  and  $q$ . To ensure  $ed = 1(\text{mod } t)$ , the Extended Euclidean algorithm [26] is used to compute  $d$  as the inverse of  $e$  modulo  $t$ . For the encryption of a message  $m$ , the ciphertext  $c := m^e$  is computed. For the decryption of a ciphertext  $c$ ,  $c^d(\text{mod } n)$  is computed. The RSA public key consists of the pair  $(n, e)$ , while the RSA private key consists of the values  $(p, q, t, d)$  and must be kept secret by whomever generates the key pair. RSA keys are typically 2048 or 4096 bits long and, in addition to encryption, are used in the process of digital signatures.

Digital signatures [27] are used to guarantee the authenticity and integrity of a message. Consider again the scenario where Bob and Alice are sending messages to each other, and Eve can hear the communication channel they are both using to communicate. Even if messages are sent in encrypted form, Eve can alter the message. In that way, when Bob sends a message  $m$  to Alice, Alice will receive a different message  $m'$ , after Eve changes the original message. To avoid this situation, when Bob is about to send the message, he computes a signature of the message  $s := \sigma(S_{Bob}, m)$  and sends it with the message to Alice. To verify the message integrity, Alice verifies the message  $m$  with the signature  $s$  received  $v(P_{Bob}, m, s)$  using Bob's public key. This way, Alice can be sure that Bob signed the message and that the message was not changed.

## 2.2 Secure hash functions

Hash functions [28] are functions that, given an arbitrarily input, produce a fixed-size result. The result of the hash function can vary between 128 and 1024 bits, regardless of the size of the input. A secure hash function is an hash function that complies with stricter requirements, such as collision resistance. Secure hash functions can also be called message digest functions, producing a result called digest or fingerprint. There are many applications in cryptography for secure hash functions. Many times it is useful to map variable-sized values into fixed-size values. In other situations, these functions can be used in cryptographic pseudo-random number generators to generate several keys to build a single shared secret key.

The simplest property that a secure hash function must have is that it should be a one-way function. In other words, considering a message  $m$  as input of the function and  $x$  the digest, it should be simple to compute  $h(m) = x$  but impossible to find  $m$  given only the value  $x$ . Collision resistance is another key property of a secure hash function [29]. A collision can be described as given two different inputs,  $m_1$  and  $m_2$ , the result of applying the same secure hash function is the same such that  $h(m_1) = h(m_2)$ . Each hash function has an infinite number of collisions because, since these functions have a fixed-size output, the number

of output values is finite, given the infinite number of possible inputs. However, this requirement says that collisions should not be found even though they exist.

Secure Hash Algorithm (SHA) [30] is a family of algorithms designed by the National Security Agency (NSA) and standardised by NIST [31]. The first algorithm that appeared was SHA-0. SHA-0 had weaknesses, described three years later by Chabaud and Joux [32], leading to the appearance of a new version, called SHA-1. The main problem found on this algorithm was the 160-bit result size. It only requires  $2^{80}$  steps to generate collisions, and this value is much lower when compared to modern block ciphers with key sizes between 128 and 256 bits. Research efforts lead to the discovery of new attacks with a complexity lesser than  $2^{80}$  computations [33, 34]. To address the problems of SHA-1, in 2001, three new hash functions were published by NIST. A fourth one was added to the specification in 2004 [35], originating the SHA-2 family of secure hash functions. These functions have outputs with 224, 256, 384, and 512 bits, and they permit the use of AES keys with 128, 192, and 256 bits [36]. Currently, the SHA-2 and SHA-3 family of secure hash functions have no known feasible attack.

## 2.3 Message authentication codes

Message Authentication Code (MAC) [37] is a function used to detect changes in messages. Encryption, by itself, prevents other people from reading the messages but does not prevent third parties from changing the messages. MAC appears to deal with this problem. When someone wants to send a message  $m$ , the sender may also compute a MAC value of the message  $m$ , using a secret key  $K$ . The other person receives the message  $m$  and the MAC value. After receiving the message, he calculates the MAC of the received message and compares it to the MAC received in conjunction with the message. MAC functions receive two arguments: a key  $K$  with fixed size and a message  $m$ . The result is a fixed-size MAC value. The MAC expression can be described as  $MAC(K, m)$ . To guarantee message integrity, the message  $m$  is sent in conjunction with the result of  $MAC(K, m)$ , called tag. Considering the other person receives a message manipulated  $m'$  and a tag  $T$ . In the verification process,  $T$  is verified as a correct MAC value for the message received using  $MAC(K, m')$ . The message was manipulated so the result  $MAC(K, m')$  value will not match with  $T$  received.

Hash-based Message Authentication Code (HMAC) [38, 39, 40] is a function with a random mapping with keys and messages as input. This function can be expressed as  $h(K \oplus a || h(K \oplus b || m))$ , where  $a$  and  $b$  are constants. The message  $m$  is only hashed once, and the output is hashed again using the key  $K$ , as [39, 40] described in more detail. HMAC can work with many hash functions like SHA hash functions (SHA-1, SHA-224, SHA-256, and others), and his design prevents the same collision attacks founded on SHA-1 [41]. In HMAC, the beginning of the message to hash is based on a secret key, and the attacker does not know which means that using HMAC with SHA-1 does not have as many problems as the direct application of SHA-1. However, this approach comes with a huge risk, considering the evolution of the attacks over time, and is not recommended to use.

HMAC was designed to resist attacks, with proof given on security bounds on the resulting construction and avoids key recovery, but is still limited to  $n/2$

bits of security. HMAC ensures that attackers have to do  $2^{n/2}$  iterations with the system under attack, making that more difficult.

## Chapter 3

# Searchable Encryption

In a common search operation, an entity sends a searchable term to another entity to retrieve the corresponding data. However, after this search operation, both the knowledge of the searched term and the possibility that the term is present at the source entity, becomes known by the queried entity. To achieve data confidentiality, the most common solution is to encrypt all information. To search within this information, the trivial approach is to decrypt all the information, thus allowing search operations to be performed over the clear text. Although simple, this solution negatively impacts on confidentiality, scalability and performance. Searchable Encryption (SE) emerges as a technique that preserves the data confidentially while allowing search operations to be performed [42]. Over the years, various types of constructs have been proposed. In some of them, researchers have focused on the efficiency of searchable encryption techniques, while others have focused on security and privacy [43]. The efficiency of an SE scheme is measured by the generated costs associated with the computation produced and the communication performed. Security, on the other hand, is measured through the information that is exchanged with the system, more specifically by checking whether the system learns information from the performed searches, as well as the results obtained from them.

SE solutions either use an encryption algorithm that allows search operations to be performed on the ciphertext, or use an auxiliary index that is created based on keywords to be used in searching operations. Using indexes is more efficient and can increase search performance since they allow queries to be performed using *trapdoors*, generated from the keywords to be searched. A *trapdoor* can be the result of a cryptographic hash function over the text to be queried. These functions are simple to calculate when in possession of all the data, but time-consuming to reverse without knowing the original clear text information. However, index-based solutions typically require additional calculations, which are performed at the information storage stage, to extract keywords, encrypt them, and then add them to the index. To construct a searchable index, there are two main approaches: forward index and inverted index. Figure 3.1 shows the two types of index used in SE schemes. A forward index takes the form of a list of keywords per document [44] and the complexity of the search is  $O(n)$ ,  $n$  being the number of documents. An inverted index takes the form of a list of documents by keywords, making it easier to identify all documents that contain a given keyword [45, 46] and its search complexity is  $O(|D(k)|)$ ,  $|D(k)|$  being the

number of documents containing a given keyword  $k$ .

Forward index		Reverse index	
document	keyword	keyword	document
d1	k1, k3	k1	d1, d4
d2	k2	k2	d2, d3
...	...	...	...
n	k	k	n

Figure 3.1: Forward and reverse index

Figure 3.2 presents the typical SE model which consists of three parts: the data owner, the data user, and the data storage server [47]. The data owner is the person responsible for the data on the storage server. Usually, the information on the storage server is sensitive data that needs to be stored in a way that ensures confidentiality and privacy. To do so, the information is encrypted by the data owner before it is stored on the data server. Along with the encrypted data, the data owner submits an index to allow searching the information on the data server. This index is composed of keywords that reference the existing data on the storage server. These keywords are also encrypted to prevent the storage server from taking away any knowledge about the searches made by users and the results of the searches performed. The data user is the person who performs searches on the data storage server. Searches are performed by creating a query containing the keywords the user wants to search for. This query is then encrypted and sent to the storage server to which the server responds with results in encrypted form, decrypted later by the data user. The storage server is the element responsible for storing the data, performing the search operations taking into account the queries submitted by users, and returning the results obtained.

Searchable ciphers can be divided into two major types depending on the cryptographic primitives used: Symmetric Searchable Encryption (SSE) and Public Key Encryption with Keyword Search (PEKS).

SSE uses symmetric key cryptography where only the owner of the secret key can produce ciphertexts and perform searches. In the first reference to this type of scheme [48], the search process consisted of a sequential scan through the entire ciphertext and the words were encrypted separately and concatenated using a special format hash value. During a search process, the server extracted these stored hash values to make a comparison and check if there was a match with the search performed. However, this scheme showed low efficiency in the search process because, as the server scanned the entire ciphertext for the searched term, the search time increases as the amount of stored data increases due to the linear relationship between the search time and the amount of data.

Later, new solutions were presented to increase the efficiency of the search process, as presented in [44]. In this paper, a proposed index implementation is presented, and the index would be created independently of the plaintext

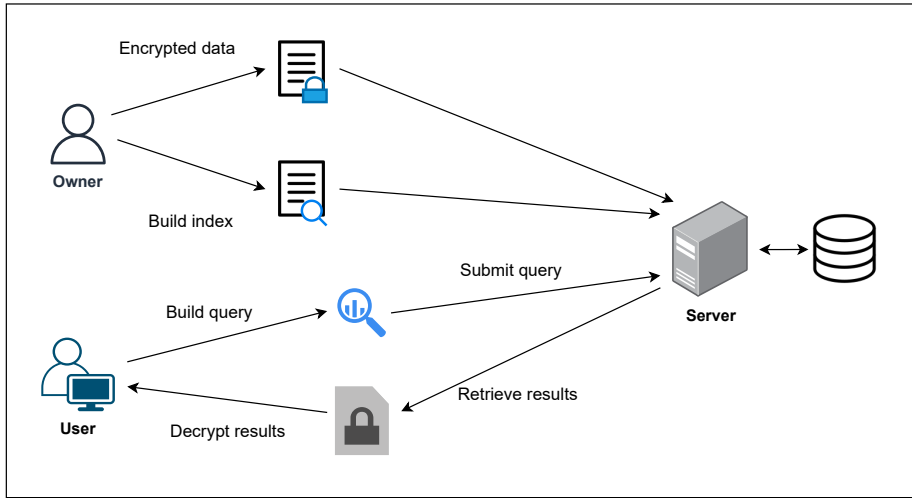


Figure 3.2: SE model

encryption algorithm. The implementation of this index is based on the use of a Bloom Filter (BF) [49] which is a probabilistic data structure to indicate the probability that an element, in this case, a searched keyword, exists in a given dataset. This structure, with  $n$  bits of length initialised to 0, will receive the keywords that belong to the index through  $k$  independent hash functions that map each keyword into  $k$  positions of the matrix structure. The bits in these positions are then changed to 1. During the search process, the  $k$  positions are generated by the same  $k$  hash functions and if any bit of these positions is 0 then the keyword is not found. In case all the bits of the positions are 1 there is a high probability that the keyword belongs to the dataset which means that it is possible to have false positives in the search results. Another solution [50] proposes a scheme based on an inverted index composed of keywords that map the identifiers of the stored data to take advantage of the efficiency of the search process in keyword-based indexes.

Some authors [51] identify problems on using searchable encryption, such as information leakage, the discovery of the cryptographic hashes of keywords contained in an updated document, or the inefficiency in terms of searching and index update times. In their paper [51], the authors introduce the concept of Dynamic Searchable Symmetric Encryption (DSSE). This allows a client to encrypt data in such a way that it can later generate lookup keys and send them as queries to a storage server. Given a *token*, the server can search over the encrypted data and return the appropriate encrypted files, ensuring a significant reduction in information leakage and maintaining efficiency in search and update operations. Authors in [52] introduce a new solution, called *Blind Storage*. This allows clients to store a set of files on remote servers, such as Dropbox, in such a way that the server cannot know how many files are stored there, nor their size. The server only knows of the existence of a file when it is obtained by a client, without ever knowing the name of the file or its content.

PEKS allows anyone to create ciphertexts using a public key, but only the owner of the corresponding private key can perform searches. This makes PEKS

easy to use in multi-user scenarios. An example of PEKS can be found in [53] where a scheme based on the cryptographic primitive Identity Based Encryption (IBE) [54] is presented where the public key is generated based on the information of the entity responsible for that key. Private keys are generated by the Private Key Generator (PKG) which publicly shares a public primary key but keeps the private primary key secret. It is then from the public primary key that identities generate their public key by combining their information with the publicly shared public primary key and the private key is generated through the private primary key. The transmission of the private key is done through a secure and confidential channel, requiring an authentication between an identity and the PKG. However, this scheme uses a deterministic encryption system for the creation of trapdoors indicating that the junction of the same key with the same password entered by the identity will always result in the same trapdoor allowing the server to take knowledge of the searches that are submitted. This problem is mitigated in a solution presented later [55] where trapdoors are now generated based on a time interval. In this way, the server is unable to make a relationship between past and future searches.

The paper [56] presents strong privacy definitions for public key encryption schemes where the encryption algorithm is deterministic and then introduces the proposed RSA Deterministic Optimal Asymmetric Encryption Padding (RSA-DOAEP) solution (based on the existing RSA Optimal Asymmetric Encryption Padding (RSA-OAEP) [57, 58] scheme with the addition of the deterministic factor) which has the additional feature of length preservation, where the length of the ciphertext is equal to the length of the plaintext. Also mentioned in the paper is the use of an SE technique, called Efficiently searchable encryption (ESE), where the encryption is randomised but makes use of a deterministic collision-resistant function of the plaintext that can also be calculated from the ciphertext and produces a searchable *tag*. It enables fast comparison-based searching. This schema allows indexing the data appropriately in standard data structures (e.g. tree-based) and search according to the *tags*.

Security shortcomings in a PEKS scheme are addressed in [59]. In this case, keyword guessing attacks are used to endanger the privacy of the user and the searches they perform, and thus, a new searchable encryption scheme called Searchable Encryption based on Efficient Privacy-preserving Outsourced calculation framework with Multiple keys (SE-EPOM) is presented. This scheme suits distributed environments comprising multiple data writers and readers, and can implement multiple servers designated to assist the main storage server in performing keyword searches, while preserving privacy over the encrypted data. Using multiple parallel servers allows to speed up response and balance the workload, while effectively resisting a password guessing attack from the cloud storage server.

The problem of keyword guessing attacks that traditional PEKS models are subject to is also addressed in the articles [60] and [61]. The first paper addresses the existence of chosen keyword attacks, ciphertext attacks, and keyword guessing attacks by a stranger to the system in the PEKS model without the existence of a secure channel between the receiver and the server when submitting trapdoors to perform searches. Two important security notions are then presented: Indistinguishability of secure channel free PEKS against chosen keyword and ciphertext attack (IND-SCF-CKCA) ensures that the server has

not obtained the *trapdoors* of the keywords and therefore does not create any knowledge about this information; Indistinguishability against keyword guessing attack (IND-KGA) ensures that any outsider who has obtained a *trapdoor* for a particular keyword, cannot observe the relationship between the *trapdoor* and any keywords. The second paper proposes a new framework, called Dual-server PEKS (DS-PEKS), which is divided into two independent servers. In this model, the front-end server receives a search and generates a *trapdoor* using its private key, sends internal test states to the second server with the corresponding *trapdoor* and PEKS ciphertexts. The second server can then decide which documents are queried by the client using its private key and the internal test states received from the front server.

All these solutions presented are based on the application of searchable encryption techniques in an increasingly efficient way, not leaving aside the security regarding the privacy of searches made by users, regardless of the data source. However, none of these solutions consider its applicability to Military information exchange and its requirements. Moreover, none of the solutions considers external sources of data, as is the case of using a MISP platform as an information repository.





## Chapter 4

# Proposed solution

The controlled information sharing functionality herein proposed is to be used to exchange and query information available at MISP instances. It is designed to regulate trust issues and enforce fine-grained control, without losing the capability of using relevant external threat-related information, thus supporting classified information exchange required by specific organisations, governments or military forces.

The reference scenario is presented in Fig. 4.1 and the proposed functionality operates similarly to a communication proxy between MISP instances, which is implemented as a web-based REST API. Also, it comprises a secure search functionality by using searchable encryption to impose required confidentiality, thus forcing all exchanged information to be previously encrypted. The proposal is designed to comply with specific requirements and features presented in the following sections.

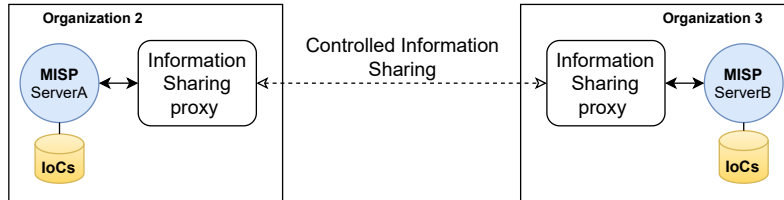


Figure 4.1: Reference scenario

### 4.1 Requirements

The proposed solution aims to impose greater control over the sharing of intrusion information between organisations that use the MISP platform. The solution now presented is based on the use of SE techniques to impose greater control on that sharing. One of the main problems to be solved by our solution consists in the creation of a mechanism for extracting existing intrusion information from a MISP server. After such information extraction, a searchable inverted index must be created with IoCs associated with that information.

This mechanism is essential because it allows confidentiality in the exchange of information between entities and in carrying out searches. The search results, when they return information, will then be inserted in the MISP server that originated the search. The following functional requirements were identified for this solution proposal:

- R1 The existence of a bidirectional data synchronisation mechanism between the MISP server and the encrypted search service.
- R2 The system must guarantee the confidentiality of the searches performed.
- R3 The system must allow the specification of the entities with which controlled synchronisation of information will be performed.
- R4 The system must allow the definition of synchronisation policies between the various entities participating in the network.
- R5 The existence of a Rest API that supports the operation of the general system.

## 4.2 Specification

Fig. 4.2 presents an overview of the proposed solution, consisting of a data synchronisation module, a REST API, a database, a backend, and a frontend. A remote index shared between entities is also envisioned to allow for information searching. Next, the main features that should be included in the proposed solution are detailed.

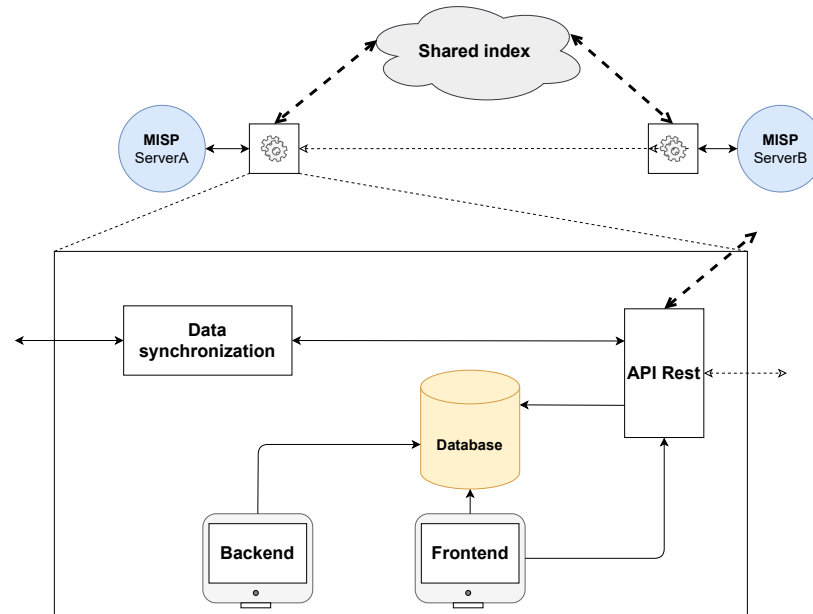


Figure 4.2: Proposed solution

The proposed solution comprehends two main components: controlled sharing and secure sharing. Controlled sharing is ensured through the application of data control policies, indicating which information can be accessed by external entities. The search functionality will operate in an asynchronous mode, similar to that used in peer-to-peer networks. Whenever a user issues a query, the query will be relayed to other systems and will finish without waiting for any response. After that, the systems that have positive results to answer the query will apply control policies that have been defined for the query system, though acting as an information filter. Information that successfully passes through these filters will then be communicated to the query system. This mode of operation is well suited for asynchronous programming languages such as JavaScript, improving the scalability of the system, and minimising user interface blockages.

Secure sharing is guaranteed through the use of a shared search index. This index should be a reverse index, it will be composed of trapdoors created using the IoCs associated with the information existing on the MISP platform, and it will be shared with entities that have formed a trust group. This mechanism is essential because it enables the confidentiality of the information exchanged between entities, including the performed queries. The confidentiality of the searching functionality will operate as follows. When a user of a system in a given organisation queries systems of other organisations, the query data is not sent in cleartext. Instead, the system will send trapdoors representing those queries and, as result, the system will return the set of entities that have information corresponding to the submitted query. The final step consists of creating the data request to the entities comprised in the search response. The secure searching functionality will require a MISP import/export capability to either export or import results of a query. Having such import/export capability, controlled information sharing can be enforced. Moreover, greater control can also be imposed over the exchanged threat information between MISP instances, acting as a proxy. The proxy will extract existing threat information from a MISP server. The search results will then be inserted into the MISP server that originated the search.

The proposed solution also assumes the existence of backend and frontend components for system administrators and users. However, the API is designed to work independently of the existence of these modules, i.e. the API is completely functional for the internal and external modules.

### 4.2.1 Adopted notation

Table 4.1 presents the notation adopted throughout this work. Letters A, B and C represent example entities.  $ID_A$  represents the unique identifier of entity A.  $T_s$  represents a timestamp.  $N_A$  represents a nonce generated by entity A.  $K_{ABC}$  represents a symmetric key shared between entities A, B and C.  $SEK$  represents a session encryption key.  $PrivK_A$  represents the private key of entity A.  $PubK_A$  represents the public key of entity A.  $\{M\}_K$  represents the encryption of message  $M$  with the encryption key  $K$ .  $H(M)$  represents the result of a secure hash function with input  $M$ .  $HMAC(K, M)$  represents the result of a secure HMAC function with input  $M$  and the key  $K$ .  $Q$  represents a query.  $X; Y$  represents value X concatenated with value Y.

The processes for peering configuration and validation, data synchronisation, and data searching between multiple entities are detailed as follows.

Table 4.1: Adopted notation

A, B, C	Entities
$ID_A$	ID of entity A
$T_s$	Timestamp
$N_A$	Nonce generated by A
$K_{ABC}$	Symmetric shared key between entities
$SEK$	Session encryption key
$PrivK_A$	Private key of entity A
$PubK_A$	Public key of entity A
$\{M\}_K$	M encrypted with key K
$H(M)$	Hash function of M
$HMAC(K, M)$	HMAC function of M with key K
$Q$	Query
$X; Y$	X concatenated with Y

#### 4.2.2 Peering Configuration and Validation

Before exchanging data between two or more entities, a peering process must be initialised. The peering configuration between two distinct entities is done by importing a file with the required data, or through a form within the front-end, similarly to what happens in the MISP platform. The entity (or peer) description includes information such as its: Uniform Resource Locator (URL), unique identifier, public key, and symmetric key. The public key will be used to ensure the integrity of the exchanged messages, and the symmetric key will be used to ensure confidentiality. The file-based entity description is assumed to be securely exchange between the entities that wish to establish their peering.

Once the peering configuration is done, the validation process is performed, according to Fig. 4.3. The validation consists of exchanging encrypted messages to ensure that the information exchange and the requests validation can be done correctly. Considering the existence of a peering configuration between two entities, A and B, the first entity makes a peering validation request using the following message:

$$ID_A; T_s; \{ID_A; T_s; N_A\}_{PubK_B} \quad (4.1)$$

This message comprises: the unique identifier of the sender entity ( $ID_A$ ), a timestamp of the moment that this request was made ( $T_s$ ), and an encrypted value. This encrypted value is the result of encrypting the previous two values, concatenated with a fresh nonce, with the public key of the receiving entity ( $\{ID_A; T_s; N_A\}_{PubK_B}$ ).

Upon receiving this message, the receiving entity checks whether the unique identifier of the sender matches a previously configured peer and decrypts the cipher value with its private key, to validate the timestamp and the integrity of the message. The timestamp combined with a fresh nonce will ensure resistance against replay attacks. After this verification, a message is returned, with the same structure, to the entity that made the configuration validation request, changing the sender entity identifier, the timestamp, and the public key used

for the encryption. Upon reception of this message, the validation of the message is performed by verifying that the correct nonce was used ( $N_A$ ). If the values match, it means that there were no problems exchanging the messages and therefore the status of the peering configuration is updated to valid.

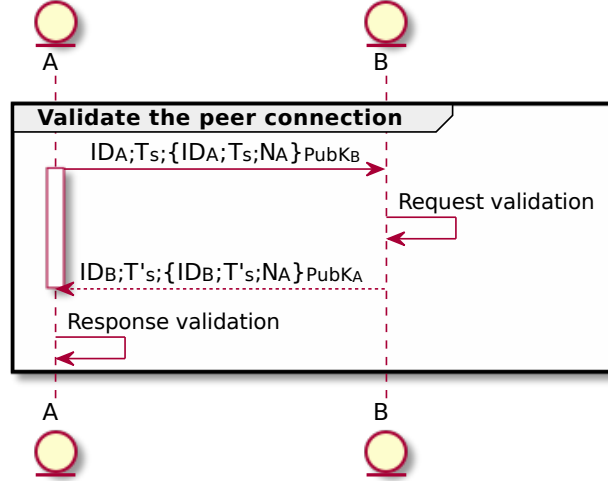


Figure 4.3: Process of peering validation

### 4.2.3 Data synchronisation

The synchronisation process, shown in Fig. 4.4, consists of exchanging data between two entities after their peering is configured and validated. This is done through a REST API and, before any synchronisation, a user must configure its settings (e.g. to synchronise only a specific type of data or enforce a specific period of time). A session key ( $SEK$ ) is generated and added to the request. This session key is used to encrypt the response that will be received and has a lifetime of one request, i.e. a new session key will be generated with each request made. The chosen type of data is expressed through a query ( $Q$ ), as shown in Listing 4.1, and the data synchronisation request is made with a message format similar to the format used in the peering validation ( $ID_A; T_s; \{T_s; ID_A; Q; SEK\}_{PubK_B}$ ).

Upon receiving such a request, the receiving entity validates it, reads the query, and queries its MISP instance for matching information. After obtaining the matching information, the response is encrypted with the session key ( $\{T'_s; ID_B; Data\}_{SEK}$ ), and sent back to the requesting entity.

```

1 {
2   "created_by": "afa1c113-6363-4960-bf83-cd5ae5272dd7",
3   "created_at": "10-05-2021T17:30:35",
4   "query_groups": [
5     {
6       "condition": "AND",
7       "values": [
8         {

```

```

9           "type": "ip-src",
10          "value": "192.168.5.1"
11        },
12        {
13          "type": "domain",
14          "value": "domain.com"
15        }
16      ]
17    },
18    {
19      "condition": "OR",
20      "values": [
21        {
22          "type": "ip",
23          "value": "192.168.0.33"
24        },
25        {
26          "type": "domain",
27          "value": "host.net"
28        }
29      ]
30    }
31  ]
32 }

```

Listing 4.1: Example of a data query

#### 4.2.4 Data searching

The data search process between two entities can be performed either directly to the database of an entity's MISP instance, or through a shared, centralised index. In the first case, the search is done using the synchronisation process mentioned in subsection 4.2.3. The second case, depicted in Fig. 4.7, is targeted for entities that require the exchange of classified information or that only wish to share specific information pertaining a joint military operation, for instance. In this case, a shared encrypted index on a remote server should be used. All entities belonging to this trusted group can upload indexing information of the data they consider relevant to share. Searching through the remote and encrypted index comprises three major steps: (1) Symmetric key construction, (2) Index update, and (3) Searching. These are described as next.

1. Symmetric key construction - The symmetric key ( $K_{ABC}$ ) consists of the key that will be used by the entities belonging to a given trust group for the process of updating the shared index and performing searches on it. This key is generated with the contribution of all the entities in the group, as shown in Fig. 4.5. Each entity calculates a secure hash having as input a freshly generated nonce value ( $N_A$ ) concatenated with the current timestamp ( $H(N_A.T_s)$ ). This hash is sent to all entities belonging to the group. When all entities have, in their possession, the hashes of all entities, each entity will calculate the same final key. To do so, all entities' hashes must be sorted alphabetically and concatenated into one ( $H(N_A.T_s).H(N_B.T'_s).H(N_C.T'_s)$ ), after which the SHA-512-256 hash

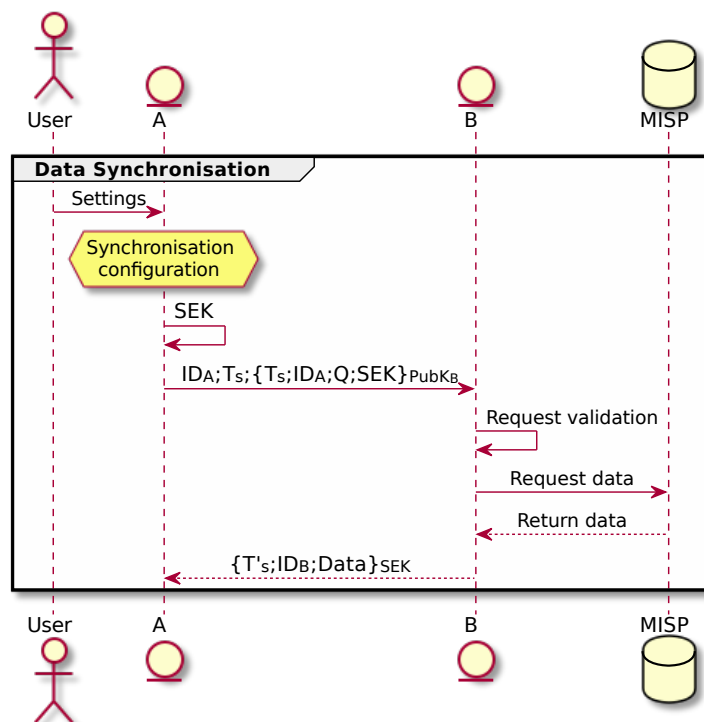


Figure 4.4: Data synchronisation between entities



function is applied, generating a key with 256 bits. This key will then be used in trapdoor generations and also to enforce the confidentiality of all trapdoors shared within the group. The total number of messages required in this procedure will be of  $n(n-1)$ , for  $n$  entities. This number of messages may limit the size of the sharing group, however, it was assumed that this will not pose as a limitation to our solution since the number of existing military organisations that cooperate is small. It is assumed a maximum group size of ten.

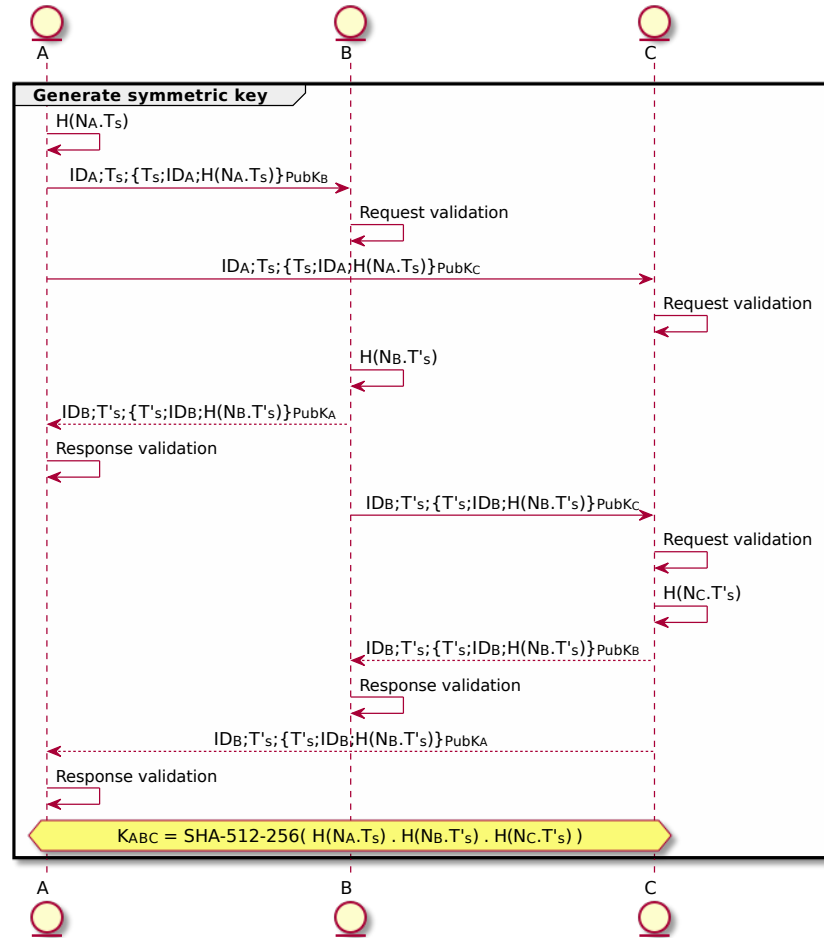


Figure 4.5: Creation of symmetric key

2. Index update - Updating the shared index, shown in Fig. 4.6, consists of sharing specific information that the entity decides to share with all remaining entities in a trusted group. To do this, the user defines which data will be indexed. The retrieval of matching data from its MISP instance is then performed. For each obtained value, a trapdoor is computed using an HMAC function ( $HMAC(K_{ABC}, Value)$ ) using the shared key of the group. In addition to the trapdoor, the unique identifier of the entity that updated the index is sent, as well as a signature of the created trapdoor

$$(\{ID_B; \{HMAC(K_{ABC}, Value)\}_{PrivK_B}\}_{K_{ABC}}).$$

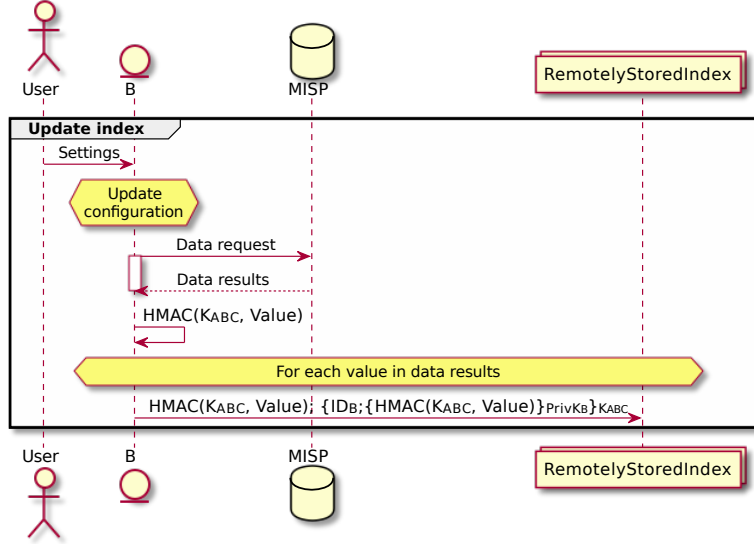


Figure 4.6: Shared index update

3. Searching - The search process is presented in Fig. 4.7 and it is similar to the search made directly to a MISP instance. The user specifies a value to be searched. Then, the related trapdoor is generated ( $HMAC(K_{ABC}, Value)$ ) and used to query the shared index for a match. If the trapdoor exists in the index, the index will reply with a list of entities that reported the presence of the trapdoor. If the list is not null, a data request is made to the reported entities. The request includes the query being made, but also a session key created for encrypting the response ( $ID_A; Ts; \{Ts; ID_A; Q; SEK\}_{PubK_B}$ ).

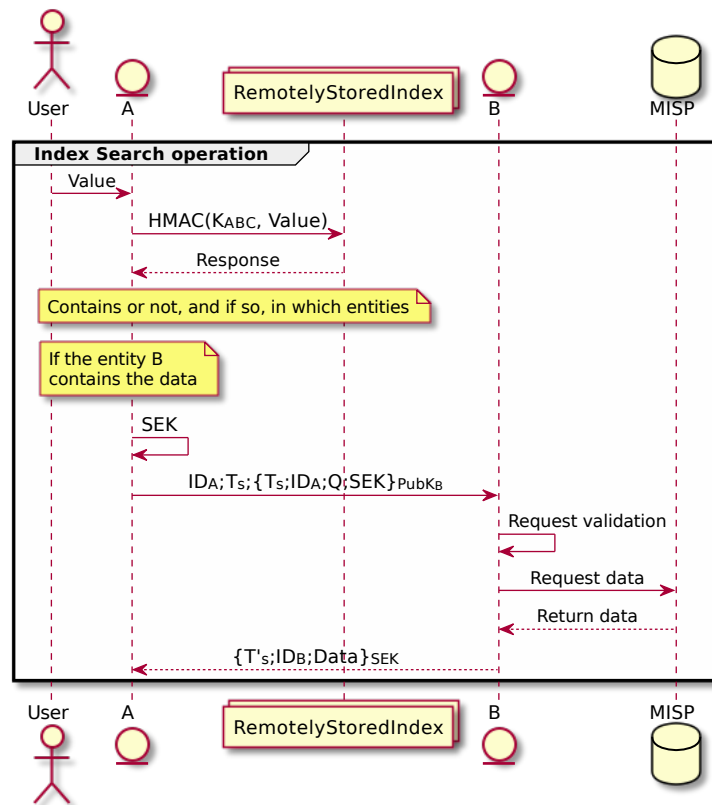


Figure 4.7: Data search through a shared index

## Chapter 5

# Implementation

This chapter details the implementation of a prototype comprehending the main functions of the proposed solution. The main functions are assumed to be the implementation of the initial setup, the peering configuration, the data synchronisation, and data sharing while using a remote index. Each of these functions will be detailed in the following sections.

### 5.1 Initial setup

The initial setup is a process executed only once and is responsible for setting up the two basic elements of the system: the Universally Unique Identifier (UUID) and the public-private key pair. The UUID is the unique identifier of the entity and is crucial for the communication between entities. This value enables an entity to identify the sender of each message and thus enables the verification of the integrity of each message. The public-private key pair guarantees that an entity can send messages to other entities in a confidential way. The asymmetric encryption schema is used on almost all processes of communication used by this system except for data exchange where usually the size of the message is expected to be larger and the symmetric encryption approach is used for its effectiveness in encrypting larger amounts of bits. Section 5.3.4 details the use of both symmetric and asymmetric encryption in order to achieve confidentiality and integrity within the proposed solution.

Both elements are generated and stored in files and the reference saved on the environment variables of the system. In the case of the public-private key pair, the environment variable store the file path of the keys. By being stored in the file system, this information enables system recovery from reboots or other transient failures. In order to accomplish this, before the generation process, the system verifies if there are any files with this information and tries to load the data, keeping the original information. As an additional verification, if the system detects the existence files with information, it will then verify the integrity of the information. If any of the previous conditions fails, the system generates new values to be used.

```
1 const filePath = BASE_DIR + '/uuid.txt'
2
3 if (!fs.existsSync(filePath)) {
4   const entity_uuid = uuidv4()
```

```

5   fs.writeFileSync(filePath, entity_uuid, { encoding: 'utf-8', flag
   : 'w+' })
6 }
7
8   const uuidFromFile = fs.readFileSync(filePath, 'utf-8')
9
10  if (!validate(uuidFromFile)) {
11    const entity_uuid = uuidv4()
12
13    fs.writeFileSync(filePath, entity_uuid, { encoding: 'utf-8', flag
   : 'w+' })
14
15    process.env.UUID = entity_uuid
16  } else {
17    process.env.UUID = uuidFromFile
18  }

```

Listing 5.1: UUID Generation

Listing 5.1 and Listing 5.2 show an excerpt that comprises the initial setup. Listing 5.1 shows the configuration of the UUID. The first step (lines 1-6) consists of verifying if there is a file *uuid.txt* containing the UUID value. If not, a new value is generated (line 4) and stored within the file. The second step (lines 8-18) consists of loading of the UUID value into the system. At this moment, there is a file *uuid.txt* with one of the two possible values: the UUID value generated previously by the system or the value given by the administrator of the system. Therefore, it is necessary to validate the format of the value. If the value is not valid (lines 10-16), the system generates a new one, stores it within the file, and then the value is ready to be used. Otherwise, if the value read from the file is valid, the system will use it.

```

1  function generateKeyPair() {
2    const nodeRSA = new NodeRSA({})
3    const keys = nodeRSA.generateKeyPair(2048)
4
5    return {
6      private: keys.exportKey('private'),
7      public: keys.exportKey('public')
8    }
9  }
10
11  const pubKeyFile = BASE_DIR + '/public.pem'
12  const privKeyFile = BASE_DIR + '/private.pem'
13
14  if (!fs.existsSync(pubKeyFile) || !fs.existsSync(privKeyFile)) {
15    const keys = generateKeyPair()
16
17    fs.writeFileSync(pubKeyFile, keys.public, { encoding: 'utf-8',
   flag: 'w+' })
18    fs.writeFileSync(privKeyFile, keys.private, { encoding: 'utf-8',
   flag: 'w+' })
19  }
20
21  process.env.PRIV_KEY_FILE = privKeyFile
22  process.env.PUB_KEY_FILE = pubKeyFile

```

Listing 5.2: Key Pair Generation

Listing 5.2 is related to the configuration of the public-private key pair. The process is very similar to the previous one. The system verifies if both the public

and the private key exist and, if not, generates a new pair of keys (line 15). The key generation (lines 1-9) makes use of an RSA module and generates a new key pair with 2048 bits. If a new key pair needs to be generated, the system will store them on the file system. Two environment variables are defined comprising the path to these files.

## 5.2 Peering validation

The peering validation, described in section 4.2.2, is the process that enables the secure communication between two entities. This process is performed after the entity setup. An entity can be described by its UUID, URL, and the public key. The validation of an entity is made by exchanging a nonce value and verifying if this value remains the same at the end of the process. To do so, this value is encrypted along with the UUID of the entity making the validation request and the timestamp of when this request is made. The Listing 5.3 shows a segment of the algorithm of a peering validation from the point of view of the entity making the request for validation.

```

1 (...
2
3 const toEncrypt = `${process.env.UUID};${currentTime};${nonce}`
4
5 const publicKeyFileDir = entity.pub_key
6 const publicKeyFile = fs.readFileSync(publicKeyFileDir, 'utf-8')
7
8 const encryptedContent = encryptMessage(toEncrypt, publicKeyFile)
9
10 const messageToSend = `${process.env.UUID};${currentTime};${
    encryptedContent}`
11
12 const response = await postRequest(`${entity.url}/entities/${
    process.env.UUID}/peer-receiver`, { message: messageToSend })
13
14 /**
15  * responseParts[0] : UUID of entity who response the request
16  * responseParts[1] : Timestamp of the response
17  * responseParts[2] : Encrypted content of the response
18  */
19 const responseParts = response.message.trim().split(';')
20
21 if (responseParts.length !== 3) return res.status(400).send({
22   message: 'Peer validation failed! Encrypted content of the
    response does not match the encrypted content of the request'
23 })
24
25 const myPrivateKeyFile = fs.readFileSync(process.env.PRIV_KEY_FILE,
    'utf-8')
26
27 const decryptedContent = decryptMessage(responseParts[2],
    myPrivateKeyFile)
28
29 // UUID;Ts;Nonce
30 const decryptedContentParts = decryptedContent.trim().split(';')
31
32 if (decryptedContentParts.length !== 3) return res.status(400).send
    ({
33   message: 'Peer validation failed! Decrypted content of the
    response does not have the correct format: UUID;Ts;Nonce'

```

```

34 })
35
36 if (responseParts[0] !== decryptedContentParts[0]) return res.
    status(400).send({
37     message: 'Peer validation failed! UUID is different from the
        UUID encrypted'
38 })
39
40 const nonceMatch = decryptedContentParts[2] === nonce
41
42 if (!nonceMatch) return res.status(400).send({
43     message: 'Peer validation failed! Nonce value generated does
        not match the nonce value from the response'
44 })
45
46 const okStatus = await Status.findOne({ where: { value: 'OK' } })
47
48 if (entity.status_id !== okStatus.id) await Entity.update(
49     { status_id: okStatus.id },
50     { where: { id: uuid } }
51 )
52
53 (...)

```

Listing 5.3: Peering validation - Request

In the beginning of the process, the system generates a string (line 3) containing the values to be encrypted. The entities use asymmetric encryption to make requests, so the string is encrypted using the public key of the entity that will receive the request (lines 5-8). The request is made to a specific route of the entity and the request body is the message with the encrypted string (lines 10-12). The response message has the same format as the request message. The message can be divided into three parts (entity UUID, timestamp, and the encrypted content). After the decryption of the last part of the message (lines 25-34), the system will make two validations. The first one is to verify if the UUID of the response matches with the UUID within the encrypted content. Due to the possibility of external interference in the communication between entities, it is necessary to check that the message has not been changed during the transit (lines 36-38). The last verification consists of verifying if the nonce value received matches the initial nonce value (lines 40-44). The peering between the two entities is considered valid if the previous two validations are successful. In this case, the entity status is updated to success (lines 46-51).

```

1  (...)
2
3  /**
4   * messageParts[0] : UUID of entity who made the request
5   * messageParts[1] : Timestamp of the request
6   * messageParts[2] : Encrypted content of the request
7   */
8  const messageParts = req.body.message.trim().split(',')
9
10 if (messageParts.length !== 3) return res.status(400).send({
11     message: 'Bad format message. Expected: UUID;Ts;EncryptedString
12 },
13 })
14 if (req.params.uuid !== messageParts[0]) return res.status(400).
    send({

```

```

15     message: 'Entity UUID do not match with UUID from the message
16     request!'
17 })
18 const fromEntity = await Entity.findOne({ where: { id: messageParts
19     [0] } })
20 if (!fromEntity) return res.status(400).send({
21     message: 'Entity ${messageParts[0]} not found! Peer validation
22     is invalid!'
23 })
24 const myPrivateKeyFile = fs.readFileSync(process.env.PRIV_KEY_FILE,
25     'utf-8')
26 const decryptedContent = decryptMessage(messageParts[2],
27     myPrivateKeyFile)
28 /**
29  * Extract nonce value
30  * Decrypted content format: UUID;Ts;Nonce
31  */
32 const decryptedContentParts = decryptedContent.trim().split(';')
33
34 if (decryptedContentParts.length !== 3) return res.status(400).send
35     ({
36     message: 'Bad format content decrypted. Expected: UUID;Ts;Nonce'
37 })
38 const toEncrypt = `${process.env.UUID};${currentTime};${
39     decryptedContentParts[2]}
40
41 const fromEntityPubKeyFileDir = fromEntity.pub_key
42 if (!fs.existsSync(fromEntityPubKeyFileDir)) return res.status(500)
43     .send({
44     message: 'The entity ${uuid} doesn't have any public key
45     associated!'
46 })
47
48 const fromEntityPublicKeyFile = fs.readFileSync(
49     fromEntityPubKeyFileDir, 'utf-8')
50 const encryptedContent = encryptMessage(toEncrypt,
51     fromEntityPublicKeyFile)
52 return res.send({ message: `${process.env.UUID};${currentTime};${
53     encryptedContent}` })
54 (...)

```

Listing 5.4: Peering validation - Response

Listing 5.4 shows the perspective of the entity that receives a new request for peer validation. The main goal of this entity is to extract the nonce value from the encrypted content of the message and send it back, encrypted with the public key of the entity that made the request. To do so, firstly, it is necessary to verify what entity is making the request. The entity can be found using the UUID in the first section of the message (lines 14-22). Without the correct UUID, the system will not be able to find the right public key to encrypt the response, so the system will verify if the entity that makes the request is already



configured in its database. The next steps are to decrypt the last section of the message and extract the nonce value (lines 24-36). The system will encrypt the nonce value with the UUID of the entity that corresponds to the request and add a new timestamp (lines 38-48). The process ends with the response to the request (line 50).

The peer validation process has one particularity: it is a unidirectional process, which means that after the validation, the peering will be valid only in the entity that requests the validation. On the other hand, the opposite peer needs to request a peering validation if it wants its configuration to be valid. In case of validation of only one of the two entities, only that one will be able to make requests (still limited, since some processes require the validation of both parties to work). However, it guarantees that the communication is carried out correctly, according to the configurations previously made. Even if the validation is correct over time, this peer validation can be repeated manually at will. This allows the configuration status to be evaluated over time, enabling administrators to correct configuration errors that may occur on one of the entities belonging to the peer.

## 5.3 Data synchronisation

Data synchronisation enables two entities to exchange data with each other. This process can be divided into several sub-processes, such as: synchronisation configuration, data control policies, data queries, and synchronisation requests. The synchronisation configuration allows for defining the synchronisation periodicity and works together with the data control policies to enable entities with control over the data that will be exchanged with other entities. The query consists of a structure that indicates the requirements for obtaining data. Finally, a request must be formulated, including the query, so that the data exchange is confidential and the data policies defined by both entities are complied with.

### 5.3.1 Synchronisation configuration

Synchronisation configuration is one of the key elements of the synchronisation process. This is where one defines when the synchronisation is performed. The relationship between the synchronisation configurations and entities is defined as '0..1:1', i.e. for each entity only one configuration can be created, although being optional.

A configuration is created through four properties: time, period, start date, and end date. The time corresponds to the hour of the day in which the synchronisation will be carried out. The period represents the synchronisation periodicity (for example, daily, weekly or monthly). Start and end date represent the time interval during which this synchronisation will be in operation. Once the configuration has been created, it has a status parameter that indicates whether the synchronisation is active or not.

This configuration takes advantage of the `node-schedule`<sup>1</sup> module that allows for task scheduling mimicking the `crontab` command available on Linux systems. The Listing 5.5 shows an excerpt of the function used to setup a configuration and scheduling a synchronisation.

<sup>1</sup><https://www.npmjs.com/package/node-schedule>

```

1  (...)
2
3  const newDataInConfiguration = await DataInConfiguration.create({
4      entity_id: uuid,
5      configuration_period_id,
6      configuration_time,
7      configuration_start,
8      configuration_end,
9      is_active: true
10 })
11
12 const cronjobExpression = getCronScheduleExpression(
13     configuration_time,
14     period.value,
15     [configuration_start, configuration_end]
16 )
17
18 const jobName = `data_in_configuration_${uuid}`
19
20 if (scheduledJobs[jobName]) scheduledJobs[jobName].cancel()
21
22 scheduleJob(jobName, cronjobExpression, (fireDate) => {
23     console.log(`${jobName}: Data synchronization executed at ${
24         fireDate}`)
25     mispSyncData(entity)
26 })
27
28 (...)

```

Listing 5.5: Synchronisation configuration

To set the timing of the synchronisation a *crontab* expression is required. This expression is generated (lines 12-16) based on the four parameters entered. An expression consists of a string that has five elements: minute, hour, day of the month, month, and day of the week, in that order. For example, the expression `'0 7 * 7-12 *'` indicates that the task will be executed at 07:00 every day of the month between July and December in every day of the week. Once the expression has been generated, it is already possible to schedule the synchronisation. To avoid conflicts between schedules, a unique name composed of the name of the task and the UUID of the entity (line 18) is created and any existing old task with the same name is deleted (line 20). This configuration can be updated or eliminated. The list of schedules is always updated according to the user's options.

### 5.3.2 Data control policies

The data control policies include two major groups: input policies and output policies. Input policies control the data that enters the system and output policies control the data that a system provides to external systems.

As described at the beginning of Section 5.3, data control policies work in conjunction with the synchronisation configuration defined for an entity. More precisely the input policies indicate which data are to be obtained in the synchronisation. On the other side is the entity that receives the synchronisation request and this is where the data output policies act, as they control the data that will be shared, regardless of the request made.

A policy has two parameters to be defined: value and application. The application parameter indicates the type of policy, in this case, the value IN, for input policies, or the value OUT, for output policies. The value parameter corresponds assumes the form of a JavaScript Object Notation (JSON) object where the key is the IoC type. For example, an input policy with the value { 'ip-src' : '192.168.1.1' } indicates that during the synchronisation process a request will be made for all events mentioning this IoC. On the other hand, an output policy with this same value indicates that the entity will reject any search that contains this IoC.

The policies have a 1..\*:1..\* relationship with the entities since an entity can have several policies associated and a policy can be associated with several entities. This way, there are no duplicated policies, making the policy definition process clearer and more efficient.

```

1  const uuid = req.params.uuid
2  const { value, in_applied } = req.body
3
4  try {
5
6    const entity = await Entity.findByPk(uuid)
7
8    if (!entity) return res.status(404).send({
9      message: 'Entity ${uuid} not found!'
10   })
11
12   const policyKey = Object.keys(value)[0]
13
14   const dataSelector = await DataSelector.findOne({
15     where: { value: policyKey }
16   })
17
18   if (!dataSelector) return res.status(400).send({
19     message: `${policyKey} key not recognized!`
20   })
21
22   const valueStr = JSON.stringify(value)
23   const policy_application = in_applied ? 'IN' : 'OUT'
24
25   const existingPolicy = await DataPolicy.findOne({
26     where: {
27       value: valueStr,
28       policy_application
29     }
30   })
31
32   if (existingPolicy) {
33     const entityHavePolicy = await entity.hasPolicy(existingPolicy)
34
35     if (entityHavePolicy) return res.status(400).send({
36       message: 'Entity ${uuid} already have this policy',
37       data: existingPolicy
38     })
39
40     await entity.addPolicy(existingPolicy)
41
42     return res.send({
43       message: 'Policy added to entity ${uuid}',
44       data: existingPolicy
45     })
46   }

```

```

47
48   const newPolicy = await DataPolicy.create({
49     value: valueStr,
50     policy_application,
51     is_active: true
52   })
53
54   await entity.addPolicy(newPolicy)
55
56   return res.send({
57     message: 'Policy added to entity ${uuid}',
58     data: newPolicy
59   })
60
61 } catch (error) {
62   return res.status(500).send({ message: error.toString(), error:
63     error.response ? error.response.data : null })
64 }

```

Listing 5.6: Policy configuration

Listing 5.6 describes the code responsible for configuring a data policy for an entity (line 1). After verifying the existence of the entity that will receive the policy (lines 6-10), the first step consists in verifying that the type of IoC received is recognised by the MISP platform (lines 12-20). The list of possible values was previously added to the system database considering the list provided by the official documentation<sup>2</sup> of the MISP platform. After this process, the policy is converted into a string and receives the value IN or OUT depending on the application type (lines 22-23). The last step consists in creating the policy. The system firstly, checks for the existence of a policy with the same application (lines 25-30). If the policy already exists, it is also necessary to verify if that policy is already associated with the entity, and in this case, it is only necessary to create the policy association (lines 32-46). If it does not exist, then it is only necessary to create the policy and its association with the entity (lines 48-54). It should also be mentioned that the policy maintains its active status. This value may be changed according to the system administrator's preference.

### 5.3.3 Data query

A query consists of a JSON object that describes the information to be searched for, as shown in Listing 5.7. Three elements are included in this object: the entity that creates the query, the timestamp, and the data query groups. This last element is the one that comprises the values to be searched for. In the case of a normal query, the user indicates which data will constitute the query in an immediate search. In the case of data synchronisation, the query's values correspond to the policies previously configured for each entity. The synchronisation process is performed by the system periodically according to the existing configuration, therefore, the system filters the input policies for a specific entity, builds the query, and sends the request to the entity.

```

1 function queryBuilder(values) {
2   return {
3     created_by: process.env.UUID,
4     created_at: `${Date.now()}`,

```

<sup>2</sup><https://www.misp-project.org/datamodels/#types>

```

5     query_groups: [
6       {
7         condition: "OR",
8         values: values.map(v => {
9           return {
10            type: v.type,
11            value: v.value
12          }
13        })
14      }
15    ]
16  }
17 }

```

Listing 5.7: The queryBuilder Function

Listing 5.7 shows the function to build the query. This function is used for any search process performed by the system, either in data synchronisation processes or even in data search with the use of a shared index. The query respects a specific schema. Since the system allows a user to create queries and execute them (for example, manually or through an external frontend application), this control must exist. This way, the system can detect irregularities in the query format before any query execution, avoiding failures during this process. In the case of data synchronisation, the query is created by the system itself, already respecting the correct structure, meaning that the schema validation is not applied.

### 5.3.4 Synchronisation request

The message format of a data synchronisation request is similarly to a peer validation request, as mentioned in section 4.2.2. All requests made between two entities consist of a message that includes the UUID of the entity making the request, the timestamp of the request, and an encrypted content. The difference between the types of requests is in the encrypted content. In the case of a synchronisation request, the encrypted content will comprise the query to be conducted. The data synchronisation process uses both asymmetric and symmetric encryption. Since symmetric encryption is more efficient for large amounts of data, the security and confidentiality of the secret key must be guaranteed. Therefore, the key is attached to the encrypted content (only known to the entity making the request and the entity receiving the request) and is valid for only one request, which means it is not reusable and a new secret key is created for each data request made.

```

1  async function getValidPolicies(entity) {
2    const inPolicies = await entity.getPolicies({
3      where: { policy_application: 'IN' }
4    })
5
6    return inPolicies.filter(p => p.is_active === true).map(p => {
7      const policy = JSON.parse(p.value)
8      const keyPolicy = Object.keys(policy)[0]
9
10     return {
11       type: keyPolicy,
12       value: policy[keyPolicy]
13     }
14   })
15 }

```

```

14 | })
15 | }

```

Listing 5.8: Function used for input policy filtering

As mentioned in section 5.3.2, the data that one entity requests from another entity in a synchronisation process corresponds to the input policies that a user configures. Listing 5.8 shows the function that, given an entity as input, returns all valid configured input policies. In this case, a valid input policy has type IN (lines 2-4) and is active at the time of synchronisation (line 6). The response of this function consists of a list of objects with two parameters: the IoC type and the IoC value (lines 10-13). In this way, this structure fits with the query construction function, presented in listing 5.7.

```

1 | (...)
2 |
3 | const currentTime = Date.now()
4 |
5 | const queryValues = inputQueryValues ? inputQueryValues : await
  |   getValidPolicies(entity)
6 | const query = queryBuilder(queryValues)
7 |
8 | const sek = generate256BitKeyLength(`${process.env.UUID};${
  |   currentTime}`)
9 |
10 | const toEncrypt = `${process.env.UUID};${currentTime};${JSON.
  |   stringify(query)};${sek}`
11 |
12 | const publicKeyFileDir = entity.pub_key
13 | const publicKeyFile = fs.readFileSync(publicKeyFileDir, 'utf-8')
14 |
15 | const encryptedContent = encryptMessage(toEncrypt, publicKeyFile)
16 | const messageToSend = `${process.env.UUID};${currentTime};${
  |   encryptedContent}`
17 |
18 | const response = await postRequest(`${entity.url}/entities/${
  |   process.env.UUID}/search-receiver`, { message: messageToSend })
19 |
20 | const messageReceived = symmetricDecrypt(sek, response.message)
21 |
22 | /**
23 |  * messageParts[0] : UUID of entity who made the response
24 |  * messageParts[1] : Timestamp of the response
25 |  * messageParts[2] : Data attached to the response
26 |  */
27 | const messageParts = messageReceived.trim().split(';')
28 |
29 | const eventData = JSON.parse(messageParts[2])
30 |
31 | if (eventData.query_groups[0].length === 0) {
32 |   console.log(`Entity ${entity.id} did not return any results. The
  |     synchronization is completed!`)
33 |   return
34 | }
35 |
36 | for (let eventObject of eventData.query_groups[0].results) {
37 |
38 |   console.log(`Verifying the existence of event: ${eventObject.info
  |     }`)
39 |
40 |   const exists = await getEvent(eventObject.info)

```

```

41
42   if (exists) {
43       console.log('> Event: ${eventObject.info} - already exists on $
44           {process.env.MISP_API}')
45       await addAttributesToEvent(eventObject.attributes, exists.id)
46   } else {
47       console.log('> Adding event: ${eventObject.info}')
48       await addEvent(eventObject)
49       const eventCreated = await getEvent(eventObject.info)
50       await addAttributesToEvent(eventObject.attributes, eventCreated
51           .id)
52   }
53 }
54 (...)
```

Listing 5.9: Data synchronisation request

Listing 5.9 shows the process of data synchronisation by the entity sending the request. The first step consists in forming the data query based on the valid policies defined for it (lines 5-6). Next, a session key is created, which will be used to encrypt the response of the request (line 8). The function receives, as input, an arbitrary message and, to avoid key repetition, the hash function receives the UUID of the entity and the timestamp of the moment, since the timestamp allows greater unpredictability in the result of the final key. From this moment on, the process is similar to the peer validation process (lines 10-18), where both the query and the session key are encrypted with the public key of the target entity resulting in the message:

$$ID_A; T_s; \{ID_A; T_s; N_A; Q; SEK\}_{PubK_B} \quad (5.1)$$

The response to the synchronisation request is encrypted with the session key embedded in the request. The final step consists of inserting the obtained information into the MISP platform. After decrypting the response, the system verifies if the result has search groups with data (lines 31-34). The non-existence of data in the response has one of two possible causes: (1) the entity that received the request did not find information in its MISP instance that satisfies the conditions described in the query, or (2) the entity that receives the request also applies policies, in this case, data output policies to control the information that leaves its system. This control process will be described on Listing 5.10 related to the entity that received the synchronisation request.

If there is data in the response, the synchronisation process continues. To avoid duplication of information, it is necessary to check if there is already any MISP event similar to the events that are in the results list (lines 36-58). This is because there may be cases where there is an update of some previously synchronised event. In this case, it is also verified which IoCs are associated with that event. A comparison of the existing IoCs with the IoCs present in the

search results is then made and the system updates the existing event with the new IoCs found in the response.

On the other side of the synchronisation process is the entity that receives the request. On this side, three steps are performed: policy enforcement, data search in the MISP instance, and response to the request. The first step, previously mentioned, consists of applying the policies that control the entity's data output, and Listing 5.10 shows the function that performs this control based on the received query. This may result in the final search query being different from the query received, because the system will remove all values that do not exist in its data output policies from the query. Please note that a data output policy corresponds to a value that can be searched by other entities.

```

1 async function queryValidator(entity, query) {
2   const entityPolicies = await entity.getPolicies({ where: {
3     policy_application: 'OUT' } })
4
5   let validatedGroups = []
6
7   for (let group of query.query_groups) {
8     const validValues = group.values.filter((value, index, arr) =>
9       {
10        const valueStr = `${value.type}:${value.value}`
11        const policyMatch = entityPolicies.find(p => p.value ===
12          valueStr)
13        return policyMatch !== undefined
14      })
15
16    group.values = validValues
17
18    validatedGroups.push(group)
19  }
20
21  const finalGroups = validatedGroups.filter(g => g.values.length
22    !== 0)
23
24  query.query_groups = finalGroups
25
26  return query
27 }
```

Listing 5.10: Output Policy Control Function

This control function receives two parameters: the query received and the entity that made the synchronisation request. The entity is needed to identify which policies should be applied (line 2). Having in its possession the policies of the entity, the system makes a comparison of the policies with the existing values in the query's search groups (lines 6-20). For each search group, a filtering of the values that respect the existing policies is performed (lines 8-14). For each value of the search group, a string representation is created (line 9). Those values whose string representation coincides with the string representation of one of the entity's policies are considered valid (lines 11-13). At the end of this filtering process, there may be search groups with fewer values than the initial amount of values or even empty groups, indicating that the group was blocked



by the configured policies. In this case, it does not make sense for these groups to continue in the query and therefore they are removed (lines 22-24).

After the query is validated, it is submitted to the MISP instance for data searching. The search result will have the format based on the query inserted, that is, the base structure remains with the addition of the results found. The number of searches carried out will be equal to the number of search groups available in the query, which means that for each group a new field, called "results" will be added and will contain the search results of that group of values.

```
1  async function mispSearch(query) {  
2  
3    query.created_by = process.env.UUID  
4  
5    for (let group of query.query_groups) {  
6  
7      const tags = group.values.map(v => v.type)  
8      const values = group.values.map(v => v.value)  
9  
10     const eventAttributes = await getMISPAttributes(tags, values)  
11  
12     if (!eventAttributes) {  
13       group.results = []  
14       continue  
15     }  
16  
17     const events = await getMISPEvents(Object.keys(eventAttributes)  
18     )  
19     events.forEach(event => {  
20       event.attributes = eventAttributes[event.id]  
21     })  
22  
23     group.results = events  
24  
25   }  
26  
27   query.created_at = Date.now()  
28  
29   return query  
30 }
```

Listing 5.11: MISP search function

Listing 5.11 shows the search function to query the MISP platform and is based on the API provided in the official documentation<sup>3</sup>. The first part consists of searching for IoCs that match the query values (lines 7-10). The result of the search will be a list of MISP event IDs that have one of these IoCs associated. If the list is empty then there are no events in the MISP database with association to the entered IoCs. Knowing which event IDs satisfy the query, the system makes a new request to the MISP instance, this time in order to obtain all the information about these events (line 17). The IoCs found in the previous request are added to the corresponding MISP event and the final results are added to the group in the query (lines 19-23). To finish the process, the timestamp is updated to the time the query was performed (line 27). The

<sup>3</sup><https://www.misp-project.org/documentation/openapi.html>

search results are then encrypted with the session key of this specific request and the response will have the following structure:

$$\{ID_B; T_s; Data\}_{SEK} \quad (5.2)$$

Any data request made will be interpreted as a data search and the system will save each search in a search history. Each entity will have its search history divided into two types of searches: incoming and outgoing searches. Incoming searches correspond to data requests made by external entities and are classified as "IN". Outgoing queries are data queries made to external entities, that is, the sending of a query to another entity, and are classified with the value "OUT". This data set can be interesting for repeating old queries or even for statistical interests.

## 5.4 Data sharing through a shared index

The sharing of data using a shared index is the most relevant functionality of this implementation. Its purpose is to allow multiple entities to have a common means of sharing information. A shared index is used in situations where multiple entities agree to form an information sharing group. This means that the entities will have permission to update and search the shared index. This index will contain references to the information existing in the MISP instances of each one of the entities forming the group. When an entity performs a search over the indexed data, the result of the search will be the identification of the entity that has relevant information about that search. Consequently, the entity can make data synchronisation requests to the entities that reported having related information.

The information sharing functionality with shared index can be divided into four main processes: the creation of sharing groups, symmetric key construction, shared index creation and update, and shared index search.

### 5.4.1 Sharing groups

A sharing group refers to a mutual connection between two or more entities that decide to share information. However, it is important to mention that one entity can be part of multiple groups. The creation of a sharing group starts by defining the name of the group. The name of the group will be used to guarantee its uniqueness. Subsequently, a relation is made with the entities which will be part of the group, with all agreeing on a symmetric key to be used by all.

```

1 (...
2
3 const existedSharedGroup = await SharedGroup.findOne({ where: {
4   name } })
5
6 if (existedSharedGroup) return res.status(400).send({
7   message: 'Shared group ${name} already exists!'
8 })
9
10 const initialStatus = await Status.findOne({ where: { value: '
11   PENDING' } })
12
13 if (!initialStatus) return res.status(400).json({

```

```

12   message: 'PENDING status not found! Insert, please insert first.'
13 })
14
15 const newSharedGroup = await SharedGroup.create({
16   name,
17   status_id: initialStatus.id
18 })
19
20 (...)
```

Listing 5.12: Shared Group Creation

Listing 5.12 shows the partial code for the function to create a new shared group. The entered group's name is checked against the existing groups to avoid conflict (lines 3-7). The name must be unique. If there is no conflict, the new group is created having an initial status of PENDING (lines 9-18). This status informs the entity's users that the sharing group is not fully configured. The configuration is completed after adding the entities to the group and building the symmetric key, discussed in Section 5.4.2. At that moment, the status will be changed to OK or ERROR depending on the result of the configuration of the new group.

```

1  (...)
2
3  const sharedGroup = await SharedGroup.findByPk(id)
4
5  if (!sharedGroup) return res.status(404).send({
6    message: 'Shared group ${id} not found!'
7  })
8
9  const entity = await Entity.findByPk(uuid)
10
11 if (!entity) return res.status(404).send({
12   message: 'Entity ${uuid} not found!'
13 })
14
15 const hasEntity = await sharedGroup.hasEntity(entity)
16
17 if (hasEntity) return res.status(400).send({
18   message: 'Shared group ${sharedGroup.name} already have entity ${
19     entity.id} associated!'
20 })
21 await sharedGroup.addEntity(entity)
22
23 (...)
```

Listing 5.13: Adding Entity to Shared Group

The next step in setting up a sharing group is to add the entities that will make up that group and Listing 5.13 shows the partial function of this process. It is important to mention that a prior configuration of the entity in the system is required, as the system needs to recognise that the entity exists in its database (lines 9-13). The relationship between sharing groups and entities is many-to-many, which means that one group can have several entities associated with it and one entity can be associated with several groups. The last step, described in the listing (lines 15-21), has the purpose of verifying if the association between an entity and a group already exists, thus avoiding duplication.

### 5.4.2 Symmetric key generation

Each shared group's symmetric key is generated with contributions from all entities in the group. The resulting key will be used by all the members of the group to manipulate the data existing in the shared index. The key has two main uses: in the creation of the trapdoors (for both index update and search); and to encrypt of the digital signature of the trapdoor that will be placed in the index. The symmetric key is built as described in Section 4.2.4. This implies that the algorithm shall produce the same key in all entities of a given group. As shown in Figure 4.5, the key construction process produces a chain reaction, since the entity that decides to create the key first will trigger a reaction in the other entities in the group, causing them to also perform the key construction process. Before the final moment of key construction, each entity must contain a number of contributions equal to the number of entities present in the group, i.e., its own contribution together with the contribution of the other group entities. Having all these ingredients, the system sorts the contributions (hashes) and applies a hash function to obtain the final symmetric key.

Two preconditions are required for the key construction process to work properly on all entities in the group. The first condition is that all entities create the group with the same name since the group name will always be unique. The second condition is that the associations between groups and entities must be completed in each member of the group in order to avoid contradictory results in the different entities that compose the group. The algorithm was also prepared to verify that the set of UUIDs of the entities are equal in any entity present in the group.

```

1 (...
2
3 const groupEntities = await sharedGroup.getEntities()
4
5 // The system cannot generate the key without entities associated
  to group
6 if (groupEntities.length === 0) return res.status(400).send({
7   message: 'Keygen failed! Group "${sharedGroup.name}" does not
  have any entity associated!'
8 })
9
10 /**
11  * Get the group hash or generate a new one
12  * This hash is the contribute to generate the symmetric key for
  the group
13  */
14 const groupHash = await getEntityCurrentHashGroup(id, currentTime)
15
16 // Array to store the hashes from other entities of the group
17 addHashToGroup(sharedGroup.name, process.env.UUID, groupHash)
18
19 // Request to other group entities their hashes to build the final
  key
20 for (entity of groupEntities) {
21
22   // Build the message to send to the entity
23   const messageToSend = buildSecretKeyEndpointMessage(entity,
    currentTime, groupHash, res, sharedGroup.name, groupEntities)
24
25   const response = await postRequest(`${entity.url}/entities/${

```

```

26     process.env.UUID}/keygen-receiver', { message: messageToSend })
27
28     /**
29     * responseParts[0] : UUID of entity who made the request
30     * responseParts[1] : Timestamp of the request
31     * responseParts[2] : Encrypted content of the request
32     */
33     const responseParts = response.message.trim().split(';')
34
35     if (responseParts.length !== 3) return res.status(400).send({
36         message: 'Bad format message. Expected: UUID;Ts;EncryptedString'
37     })
38
39     // Load my private key to decrypt the content
40     const myPrivateKeyFile = fs.readFileSync(process.env.PRIV_KEY_FILE, 'utf-8')
41
42     // Decrypt the request
43     const decryptedContent = decryptMessage(responseParts[2], myPrivateKeyFile)
44
45     /**
46     * decryptedContentParts[0] : UUID of entity who made the request
47     * decryptedContentParts[1] : Timestamp of the request
48     * decryptedContentParts[2] : Entity hash contribute
49     */
50     const decryptedContentParts = decryptedContent.trim().split(';')
51
52     if (decryptedContentParts.length !== 3) return res.status(400).send({
53         message: 'Bad format content decrypted. Expected: UUID;Ts;GroupName;Entities;Hash'
54     })
55
56     // Checks if the UUID of the response matches with the encrypted UUID
57     if (responseParts[0] !== decryptedContentParts[0]) return res.status(500).send({
58         message: 'Decrypt error: UUID of the response does not match with the encrypted UUID'
59     })
60
61     // Add the hash received to the group
62     addHashToGroup(sharedGroup.name, decryptedContentParts[0], decryptedContentParts[2])
63
64 }
65
66 // Final key, using all the contributes collected
67 await generateFinalSecretKey(sharedGroup)
68
69 res.send({ message: 'Secret key generated with success!' })
70
71 (...)

```

Listing 5.14: Symmetric key Generation - Initial request

Listing 5.14 shows a section of code that processes the initial key generation request. The first step is to check if there are entities associated with the group (lines 3-8). If there are no entities in the group, it is not possible to

send requests and, in this case, the process is cancelled. The next step of the process is the generation of the contribution hash that will be used in the final symmetric key (line 14). The function *getEntityCurrentHashGroup* receives the group identifier in question and the timestamp of the moment when the request is made. Before generating a new hash, the function checks whether the system has previously generated a hash for this group, this is because the generated hash is saved for reuse, but only for the designated group. Different groups have different hash contributions. If there is no hash saved, a new hash contribution is generated. The resulting contribution be stored in a temporary list (line 17). This temporary list is responsible for storing all contributions of all entities. After the final key generation, there is no need to store the hashes contributions received from the other entities, so the list is deleted. From this moment on, it is already possible to send the hash contribution request for each entity in the group (lines 20-64). The first step is to build the request (line 23). This request has the same format as any other request made between entities which consist of an entity identifier, a timestamp, and an encrypted content. The encrypted content has some differences compared to the encrypted content used in the peering validation process described in Section 5.2. In this case, three main elements are added: the group name, the list of identifiers of the entities that belong to the group, and the contribution hash. The group name and the list of entity identifiers will be used to guarantee the integrity of all the entities that belong to the group. The contribution hash will enable the other entities to build the final symmetric key as well. The response of the request made to each entity is decrypted (lines 25-59) where integrity checks are also made. After this verification, the hash contribution sent by the entity is extracted and added to the temporary list (line 62). When the requests to all entities are finalised, the system builds the final symmetric key.

```

1 (...
2
3 /**
4  * decryptedContentParts[0] : UUID of entity who made the request
5  * decryptedContentParts[1] : Timestamp of the request
6  * decryptedContentParts[2] : Group name that the entity belongs
7  * decryptedContentParts[3] : List of entities who belongs to the
   group
8  * decryptedContentParts[4] : Hash shared by entity to build the
   shared key
9  */
10 const decryptedContentParts = decryptedContent.trim().split(';')
11
12 if (decryptedContentParts.length !== 5) return res.status(400).send
   ({
13   message: 'Bad format content decrypted. Expected: UUID;Ts;
   GroupName;Entities;Hash'
14 })
15
16 // Checks if the UUID of the request matches with the encrypted
   UUID
17 if (messageParts[0] !== decryptedContentParts[0]) return res.status
   (500).send({
18   message: 'Decrypt error: UUID of the request does not match with
   the encrypted UUID'
19 })
20
21 // Checks if the entity have a shared group with the name received

```

```

22 const sharedGroup = await SharedGroup.findOne({ where: { name:
    decryptedContentParts[2] } })
23
24 if (!sharedGroup) return res.status(500).send({
25   message: 'Entity ${process.env.UUID} does not have the group "${
    decryptedContentParts[2]}"'
26 })
27
28 const groupEntitiesFromDatabase = await sharedGroup.getEntities()
29 const groupEntitiesFromRequest = JSON.parse(decryptedContentParts
    [3])
30
31 // Check if the entity groups matches
32 const groupEntitiesMatch =
    groupEntitiesMatchWithGroupEntitiesDatabase(
        groupEntitiesFromRequest, groupEntitiesFromDatabase)
33
34 if (!groupEntitiesMatch) return res.status(500).send({
35   message: 'Error: Group ${sharedGroup.name} does not match on both
    entities. The group must have the same configuration on each
    entity'
36 })
37
38 const myHash = await getEntityCurrentHashGroup(sharedGroup.id,
    currentTime)
39
40 // Add my hash to group and the hash from the request
41 addHashToGroup(sharedGroup.name, process.env.UUID, myHash)
42 addHashToGroup(sharedGroup.name, decryptedContentParts[0],
    decryptedContentParts[4])
43
44 // Request the remaining hashes to the other entities of the group
45 for (let entityDB of groupEntitiesFromDatabase) {
46
47   // Checks if already have the have of the current entity
48   // If exists, go to next entity
49   if (entityHashExists(entityDB.id, sharedGroup.name)) continue
50
51   // Build the message to send to the entity
52   const messageToSend = buildSecretKeyEndpointMessage(entityDB,
    currentTime, myHash, res, sharedGroup.name,
    groupEntitiesFromDatabase)
53
54   const response = await postRequest(`${entity.url}/entities/${
    process.env.UUID}/keygen-receiver`, { message: messageToSend })
55
56   /**
57    * responseParts[0] : UUID of entity who made the request
58    * responseParts[1] : Timestamp of the request
59    * responseParts[2] : Encrypted content of the request
60    */
61   const responseParts = response.message.trim().split(';')
62
63   if (responseParts.length !== 3) return res.status(400).send({
64     message: 'Bad format message. Expected: UUID;Ts;EncryptedString
    ,
65   })
66
67   // Load my private key to decrypt the content
68   const myPrivateKeyFile = fs.readFileSync(process.env.
    PRIV_KEY_FILE, 'utf-8')
69

```

```

70 // Decrypt the request
71 const decryptedContent = decryptMessage(responseParts[2],
    myPrivateKeyFile)
72
73 /**
74  * decryptedContentParts[0] : UUID of entity who made the request
75  * decryptedContentParts[1] : Timestamp of the request
76  * decryptedContentParts[2] : Entity hash contribute
77  */
78 const decryptedContentParts = decryptedContent.trim().split(';')
79
80 if (decryptedContentParts.length != 3) return res.status(400).
    send({
81     message: 'Bad format content decrypted. Expected: UUID;Ts;
        GroupName;Entities;Hash'
82 })
83
84 // Checks if the UUID of the response matches with the encrypted
    UUID
85 if (responseParts[0] !== decryptedContentParts[0]) return res.
    status(500).send({
86     message: 'Decrypt error: UUID of the response does not match
        with the encrypted UUID'
87 })
88
89 // Add the hash received to the group
90 addHashToGroup(sharedGroup.name, decryptedContentParts[0],
    decryptedContentParts[2])
91
92 }
93
94 // Response message
95 const entityResponse = buildSecretKeyEndpointMessage(entity,
    currentTime, myHash)
96
97 // Final key, using all the contributes collected
98 await generateFinalSecretKey(sharedGroup)
99
100 return res.send({ message: entityResponse })
101
102 (...)

```

Listing 5.15: Symmetric key Generation - Receiving a request

Listing 5.15 shows the code section of an entity that receives a request for the construction of a shared symmetric key. After obtaining the decrypted content of the request (line 10-14), the system performs some checks to ensure the integrity of the process. The identifier of the entity that made the request is checked against the identifier extracted from the encrypted content (lines 17-19), then existence of a sharing group with the same name is verified (lines 22-26), and, finally, the entity that made the request is checked to against the list of the entities that form the group. This verification is done by comparing the list of identifiers of entities extracted from the encrypted content of the request with the list of identifiers of entities obtained from its own database (lines 28-36). After these checks, the process is very similar to the one shown in Listing 5.14. The system starts by getting its contribution hash (line 38), creates the temporary list with the existing contribution hashes (lines 41-42), and finally will perform the remaining requests for contribution hashes (lines 45-92). However, there is one difference compared to the process described in



Listing 5.14. The system will only send requests to those entities for which it does not yet have the corresponding contribution hash (line 49). This condition is controlled by its presence in the temporary list. At this point, the entity represented by Listing 5.15 already has two contribution hashes: it is own and the one received by the incoming request. Therefore, it will only be necessary to make requests to the remaining entities of the group. If this entity is the end of the chain, then there will be no more requests, because the temporary list of the group's contribution hashes is already complete, it is only necessary to send its contribution hash to the requesting entities.

### 5.4.3 Shared Index Update

The process of updating a shared index was developed so that an entity shares part of its information with a restricted group of entities that make up a sharing group in a confidential manner and guaranteeing the integrity of all the information exchanged. The confidentiality of the information existing in the shared index is guaranteed by the use of the final symmetric key, described in the previous section. The integrity of this information is guaranteed through the use of the public-private key pairs of the entities belonging to the group.

The shared index was implemented using a remote Mongo database. This type of database does not require a fixed structure and works with JSON structures, which is ideal for allocating the created indexes. The location of the index also requires agreement between all participating entities. After agreeing on the index, all entities update their group by creating a configuration. Of note is the fact that all information stored within this database is fully encrypted.

The configuration of an index is very similar to the existing configuration of normal data synchronisation, described in Section 5.3.1. The configuration defines the periodicity of the execution of the index update process and, in this case, the user defines the period and the moment of its execution. The relation between sharing groups and index configurations is one-to-one, that is, for each sharing group only one index can be defined, and therefore only one configuration exists.

```

1 (...
2
3 const sharedGroup = await SharedGroup.findByPk(id)
4
5 if (!sharedGroup) return res.status(400).send({
6   message: 'Group ${id} not found!'
7 })
8
9 const existingConfiguration = await sharedGroup.
   getIndexConfiguration()
10
11 if (existingConfiguration) return res.status(400).send({
12   message: 'There is already a configuration for group ${
   sharedGroup.name}!'
13 })
14
15 const period = await Period.findByPk(update_period_id)
16
17 if (!period) return res.status(400).send({
18   message: 'Period with ID ${update_period_id} not found!'
19 })
20

```

```

21 const newIndexConfiguration = await sharedGroup.
    createIndexConfiguration({
22   index_url,
23   update_period_id,
24   update_time,
25   is_active: true
26 })
27
28 const cronjobExpression = getCronScheduleExpression(
29   update_time,
30   period.value
31 )
32
33 const jobName = `index_configuration_${id}`
34
35 if (scheduledJobs[jobName]) scheduledJobs[jobName].cancel()
36
37 scheduleJob(jobName, cronjobExpression, (fireDate) => {
38   console.log(`${jobName}: Index update executed at ${fireDate}`)
39
40   updateSharedGroupIndex(id)
41 })
42
43 (...)

```

Listing 5.16: Configuration of the Creation of a Shared Index

Listing 5.16 shows the partial function of creating a new index configuration for a sharing group. Three parameters are required to create a new configuration: the address of the Mongo database where the index is to be located, the index update period, and the update time. The system starts by checking whether the indicated share group exists (lines 3-7) and, if so, whether the group already has an index configuration (lines 9-13). For a new configuration to be created, the group must be previously created and cannot have any index configuration associated with it. After this verification, the process is similar to the creation of a synchronisation configuration. The index update task is scheduled based on the update period and time specified by the user (lines 15-41). The name of the scheduled task includes the task type concatenated with the entity identifier (line 33), thus avoiding name conflicts between the various tasks scheduled by the system. For the index configuration to work, it is also necessary to configure which information will be shared on the index. This information corresponds to the IoCs that the user wants to share with the other entities in the group.

```

1  async function updateSharedGroupIndex(sharedGroupId) {
2
3    const sharedGroup = await SharedGroup.findByPk(sharedGroupId)
4    const indexConfiguration = await sharedGroup.
        getIndexConfiguration()
5
6    // Execute the update only if the configuration is enabled
7    if (indexConfiguration.is_active) {
8
9      const mongoClient = await mongoose.connect(indexConfiguration.
        index_url)
10
11     // Get the values associated to the configuration and extract
        her value

```

```

12  const configurationValues = await indexConfiguration.
13  getConfigurationValues()
14  const jsonValues = configurationValues.map(v => v.value)
15
16  console.log(`${jsonValues.length} values founded...\n`)
17
18  // Key used to ensure the data confidentiality
19  const secretKey = sharedGroup.shared_index_key
20
21  for (value of jsonValues) {
22
23      // MISP request to verify if the value exists in our MISP
24      database
25      const iocExists = await iocExistsInMISP(value)
26
27      console.log(`${value} ${iocExists ? '' : 'does not'} exist in
28      MISP!`)
29
30      // If the MISP don't have the IoC, ignore it
31      if (!iocExists) continue
32
33      // Trapdoor using HMAC function
34      const trapdoor = createHMAC(secretKey, value)
35
36      // Check if the trapdoor already exists in the index. In that
37      case, the existing index entry will be updated.
38      // Otherwise, it will be created a new index entry
39      const ioc = await IoCReference.exists({ trapdoor })
40
41      /**
42       * Message and signature associated to the trapdoor
43       * Index will store:
44       *   trapdoor: HMAC(ioc) using the group secret key
45       *   entities: who entities have the information about this
46       ioc + signed trapdoor
47       */
48
49      // Load private key to sign the content
50      const myPrivateKeyFile = fs.readFileSync(process.env.
51      PRIV_KEY_FILE, 'utf-8')
52
53      // Sign trapdoor to ensure the integrity of the trapdoor
54      const signedTrapdoor = signMessage(trapdoor, myPrivateKeyFile
55      )
56
57      // Content to store in the index: Entity UUID + signed
58      trapdoor
59      const trapdoorInformation = symmetricEncrypt(secretKey, `${
60      process.env.UUID};${signedTrapdoor}`)
61
62      if (ioc) {
63
64          await IoCReference.findOneAndUpdate(
65              { trapdoor },
66              {
67                  $addToSet: { entities: trapdoorInformation }
68              }
69          )
70      } else {
71
72          const newIoC = new IoCReference({

```

```
65         _id: new mongoose.Types.ObjectId(),
66         trapdoor: trapdoor,
67         entities: [trapdoorInformation]
68     })
69
70     await newIoC.save()
71 }
72
73 }
74
75 console.log('Process complete!')
76
77 mongoClient.disconnect()
78 }
79 }
```

Listing 5.17: Index Update

Listing 5.17 shows the function used for updating the index of a shared index. The system starts by identifying the group and its index configuration (lines 3-4) and verifies if the configuration is enabled (line 7). If so, the connection to the Mongo database is established (line 9). After the connection is established, the system obtains the information to be uploaded to the index (lines 12-13) and the group symmetric key, which will be used to guarantee the confidentiality of the information in the index (line 18). After the system has all the data to perform the update process, a last check is made on the values that will be introduced in the index. The system will verify if all values listed in the configuration are present in their MISP instance (lines 23-28). The user is free to add any IoC to the configuration of the index, but these IoCs might not be present in the MISP database.

The structure of the chosen shared index is based on the concept of a reverse index. This means that each key of the index has a list of values. The key corresponds to the trapdoor created over the indexed value and that will be used to perform the search. The list of values corresponds to the result of the encryption of the identifier of the entity that performs the update process concatenated with the signature of the generated trapdoor. This structure indicates that the result of a search process for a value (represented by a trapdoor in the index) will be a list of entities that have information about that same value. Initially, the system creates the trapdoor using the symmetric key of the share group (line 31) and queries the index for any existing reference to this trapdoor (line 35). The existence of a reference indicates that some entity of the group has information about the trapdoor. Next, the trapdoor is signed (lines 45-48) using the private key of the entity that is updating the index. This signature will allow for the verification of the integrity of the trapdoors, through the signing with the private key of the entity that performed the signature. The signature is encrypted together with the identifier of the entity performing the update (line 51). Finally, this encrypted content is inserted into the index (lines 53-78). If the trapdoor does not exist in the index, a new entry is created with the trapdoor and the encrypted content. If the trapdoor already exists, the encrypted content is added to the existing trapdoor.

The shared index does not have any clear text information and therefore only those who know the symmetric key of the sharing group have access to the indexed data. In an eventual case where this key is exposed, the solution is to delete the index entries and generate a new key for the group. The index does

not hold the information present in the MISP instances only the references to the information, maintaining the security of this information.

#### 5.4.4 Shared Index Searching

The search function of the shared index completes the process of information sharing in the system. The purpose of the search is to identify which entities hold information related to a searched term, and then a direct data synchronisation request is made to each one of the resulting entities. As described in Section 5.4.3, the index of each sharing group is a reverse index. The following two listings show the two main moments related to the process of searching over a shared index: performing the search for a term in the index (Listing 5.18), and the decryption and verification of the obtained answer (Listing 5.19).

```

1 async function getIoCFromIndex(ioc, sharedGroupId) {
2   const sharedGroup = await SharedGroup.findByPk(sharedGroupId)
3   const indexConfiguration = await sharedGroup.
     getIndexConfiguration()
4
5   const mongoClient = await mongoose.connect(indexConfiguration.
     index_url)
6
7   // Build trapdoor to search it on index
8   const trapdoor = createHMAC(sharedGroup.shared_index_key, JSON.
     stringify(ioc))
9
10  // Find a mongo object with the trapdoor information
11  const iocReference = await IoCReference.findOne({ trapdoor })
12
13  mongoClient.disconnect()
14
15  return iocReference
16 }
```

Listing 5.18: Searching a term

```

1 async function getEntityUUIDFromIndexIoC(trapdoor, sharedGroupKey,
   entityDigest, res) {
2   // Digest generated by entity
3   const decryptedContent = symmetricDecrypt(sharedGroupKey,
     entityDigest)
4
5   /**
6    * decryptedContentParts[0] : entity UUID
7    * decryptedContentParts[1] : trapdoor signature
8    */
9   const decryptedContentParts = decryptedContent.split(';')
10
11   // Bad format
12   if (decryptedContentParts.length !== 2) return {
13     entityUUID: decryptedContentParts[0],
14     match: false
15   }
16
17   // If exists, ignore my digest
18   if (decryptedContentParts[0] === process.env.UUID) return {
19     entityUUID: decryptedContentParts[0],
20     match: false
21   }
```

```

22
23 // Check entity from UUID received
24 const entity = await Entity.findByPk(decryptedContentParts[0])
25
26 if (!entity) return res.status(500).send({
27   message: 'Message digest from trapdoor ${trapdoor}: entity <${
28     decryptedContentParts[0]}> not found!'
29 })
30
31 // Public key to verify the signature
32 const publicKeyFile = fs.readFileSync(entity.pub_key, 'utf-8')
33
34 // Verify trapdoor signature
35 const signatureMatch = verifySignature(trapdoor,
36   decryptedContentParts[1], publicKeyFile)
37
38 return {
39   entityUUID: decryptedContentParts[0],
40   match: signatureMatch
41 }

```

Listing 5.19: Decryption and verification of a search result

The search for a term in the index is described in Listing 5.18. The system starts by getting the shared index settings from the group and by making the connection (lines 2-5). Then, the search is performed over the index. The term entered by the user is not searched in clear text, instead, a trapdoor is created over that value and using the symmetric key of the group (line 8). The system checks whether the created trapdoor exists in the index. Since, in an HMAC function, the same value and the same key always result in the same trapdoor, the system performs a comparison of the created trapdoor with the existing trapdoors in the shared index (line 11). The result of the search will be an object containing the list of the encrypted contents of the trapdoor entered by the entities of the group at the time of the update of the shared index.

Listing 5.19 describes the processing of the responses resulting from searching the shared index. This function is performed for each encrypted content that comprises the list returned by the search. The first step consists in decrypting the content (line 3) using the symmetric key of the sharing group. This gives access to the entity identifier and the trapdoor signature (line 9). These values appear concatenated and the system performs the first check on the format of the content (lines 12-15). The next step is to verify the extracted entity identifier. For a given entity, only the identifiers of other entities that have information about a search term are of interest. If the identifier of the entity that performs the search is comprised in encrypted portion of the response, it means that itself has information and has updated the trapdoor in the index. In this case, the entity is ignored to avoid the execution of a synchronisation process with itself (lines 18-21). After this verification, and after obtaining the information of the entity that made this update, the system then verifies the integrity of the trapdoor using the signature created by each entity (lines 31-34). The signature verification is done based on a public key and the function for this verification receives 3 parameters: the value that was signed, the signature, and the public key of the entity that created the signature. If the signature is invalid, the system ignores the entity, otherwise, the entity is considered valid, and the system can proceed to the data synchronisation request.

```

1  (...)
2
3  const sharedGroup = await SharedGroup.findByPk(req.params.id)
4
5  if (!sharedGroup) return res.status(404).send({
6    message: 'Shared group ${id} not found!'
7  })
8
9  // Checks the IoC data selector
10 const dataSelector = await DataSelector.findOne({ where: { value:
    Object.keys(ioc)[0] } })
11
12 if (!dataSelector) return res.status(400).send({
13   message: 'IoC Key <${Object.keys(ioc)[0]}> not found!'
14 })
15
16 // Find the IoC in the index
17 // Get the list of digests from the entities who have information
    about the IoC
18 const iocFromIndex = await getIoCFromIndex(ioc, sharedGroup.id)
19
20 if (!iocFromIndex) return res.status(404).send({
21   message: 'IoC ${JSON.stringify(ioc)} not found in index!'
22 })
23
24 let validEntities = []
25
26 // Get the entities UUID from digests
27 for (let entityDigest of iocFromIndex.entities) {
28
29   // Decrypt the entity content
30   const { entityUUID, match } = await getEntityUUIDFromIndexIoC(
    iocFromIndex.trapdoor, sharedGroup.shared_index_key,
    entityDigest, res)
31
32   // Get only the valid entities, if the signature match is true
33   if (match) validEntities.push(entityUUID)
34
35 }
36
37 await dataSyncAfterIndexSearch(validEntities, ioc)
38
39 (...)

```

Listing 5.20: Overall search process on a shared index

Listing 5.20 shows the overall process of a search over a shared index. After checking that the sharing group exists (lines 3-7), the system checks the validity of the IoC type to be searched (lines 10-14). Recall that the entered IoC type must be a valid type according to the the MISP platform. If the searched value passes this analysis, the index search is performed (line 18) to check if there are entities in the group that may have information about this search. The list of encrypted contents of the search result is then analysed by the system (lines 27-25) and for each verified content, the response will be the entity identifier and the validity of the signature (line 30). Once the search is completed and the integrity of the results obtained is verified, the system moves on to the data synchronisation process and therefore the system performs the same steps as the normal data synchronisation. However, the data output policies are always applied by the entity that receives the synchronisation request. Thus, although

an entity is part of a sharing group and makes information available through the shared index, it still keeps control of the direct sharing of that information with other entities.





## Chapter 6

# Prototype validation

This chapter presents the validation of the functionalities described in Chapter 5 and aims to show the fulfilment of the requirements defined in Section 4.1, by using two instances of the proposed solution, communicating with each other. The scenario adopted for the validation of the prototype resembles the reference scenario described in Figure 4.1 and it simulates the exchange of information between two distinct organisations. The components used were the following:

- A computer running Windows 10 with an Intel(R) Core(TM) i7-1065G7 CPU@1.30GHz-1.50 GHz and 16GB of RAM;
- Two instances of the MySQL 5.7 database;
- 2 VirtualBox (version 6.1.18) machines, each one running a MISP instance (version 2.4);
- A VirtualBox machine running a Linux server (Ubuntu 18.04.5 LTS) and running a Docker environment (version 20.10.7);
- A remotely allocated MongoDB (version 4.4.9) database instance.

The prototype was divided into two instances. In Instance 1, the developed REST API is running on the computer as localhost. This API (requirement R5) is connected to two components: a MySQL database used in the general operation of the system and a virtual machine running an instance of the MISP platform. In Instance 2, there is a virtual machine running a Linux server running Docker. The developed Rest API and another instance of the MySQL database are executed within the Docker environment. Finally, as in Instance 1, the Rest API of Instance 2 is connected to an instance of the MISP platform. To enable the secure sharing of confidential information through a shared index, a remote MongoDB database was also used to store the indexes created for each sharing group.

For the purpose of the demonstration described next, Instance 1 is assumed to be the key instance (our instance), while Instance 2 is assumed to be the one that belongs to an external organisation to whom we want to establish communication. The main features shown are the initial peering configuration and validation, a data synchronisation request, and finally, the functionalities related to the use of a sharing group.

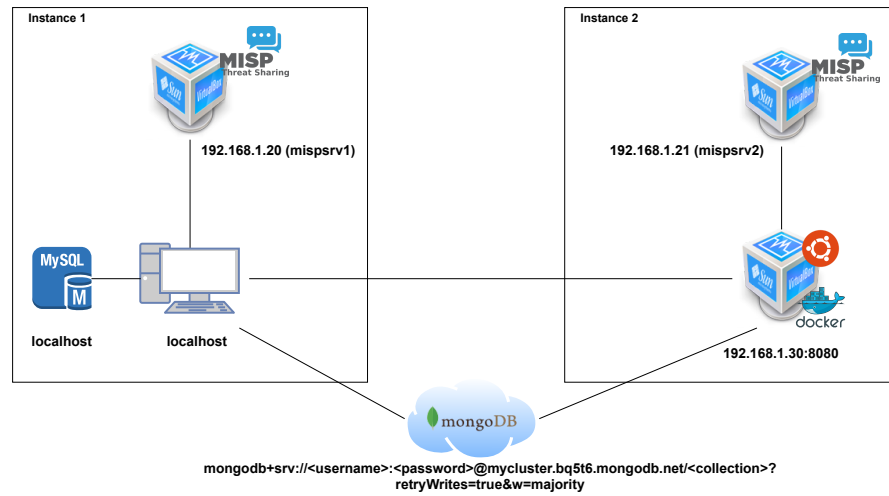


Figure 6.1: Implemented prototype

## 6.1 Peering between entities

The first step in establishing communication between two instances is the peering configuration. This configuration is done by creating a new entity in the system, which will represent the instance in question. The entity description comprises three parameters: an identifier, the instance URL, and its public key. These parameters are configured using the API routes (or endpoints) and are shown in Figure 6.2 and Figure 6.3.

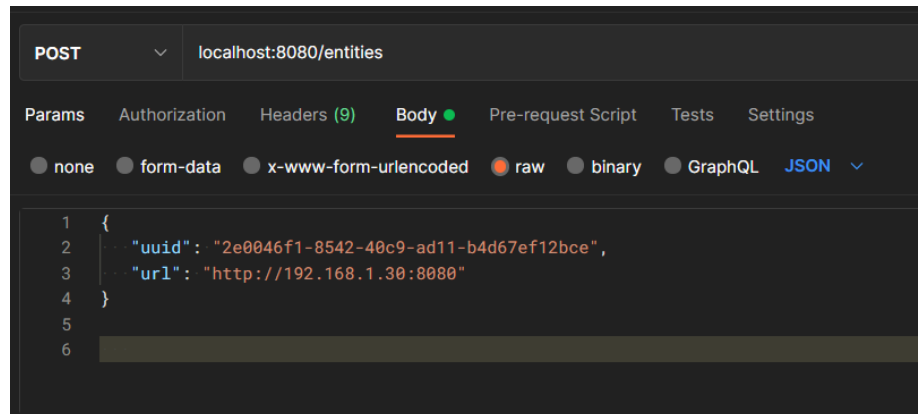


Figure 6.2: Entity creation route

Figure 6.2 shows the route used to pass the parameters entity identifier and its URL. In the shown example, the entity identifier is **2e0046f1-8542-40c9-ad11-b4d67ef12bce**, and its URL is **http://192.168.1.30:8080**. Figure 6.3 shows the route used to upload a file containing the public key of the entity being configured. The saved file will have the entity identifier as its name.

POST localhost:8080/entities/2e0046f1-8542-40c9-ad11-b4d67ef12bce/pub-key/upload

Params Authorization Headers (8) **Body** Pre-request Script Tests Settings

☐ none ☒ form-data ☐ x-www-form-urlencoded ☐ raw ☐ binary ☐ GraphQL

KEY	VALUE
<input checked="" type="checkbox"/> file	public.pem ×
Key	Value

Figure 6.3: Public key upload route

GET localhost:8080/entities/2e0046f1-8542-40c9-ad11-b4d67ef12bce/peer-sender

Params Authorization Headers (6) **Body** Pre-request Script Tests Settings

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary ☐ GraphQL Text

1

Body Cookies Headers (10) Test Results

Pretty Raw Preview Visualize JSON

```
1 {
2   "nonce_match": true,
3   "updated_entity": {
4     "id": "2e0046f1-8542-40c9-ad11-b4d67ef12bce",
5     "name": null,
6     "url": "http://192.168.1.30:8080",
7     "pub_key": "./pub-key-uploads/2e0046f1-8542-40c9-ad11-b4d67ef12bce.pem",
8     "status_id": 3,
9     "createdAt": "2021-06-17T16:00:27.000Z",
10    "updatedAt": "2021-08-09T09:46:54.000Z",
11    "status": {
12      "id": 3,
13      "value": "OK",
14      "color": "#40de48",
15      "createdAt": "2021-06-16T14:47:33.000Z",
16      "updatedAt": "2021-06-16T14:47:33.000Z"
17    }
18  }
19 }
```

Figure 6.4: Peer validation route

The final step consists in performing a validation of the peering configuration in order to verify that the communication is performed as expected. Figure 6.4 shows the route used to trigger this validation. This route does not accept any parameters, it consists of a GET request to the route that will trigger the process described in Section 5.2. The outcome expected of this route is the update of the entity status to OK, indicating that the communication happened as expected (requirement R3). The request had a total duration of 93 ms.

## 6.2 Data synchronisation

To carry out a synchronisation of data with another entity, it is necessary to schedule the synchronisation requests to be made. The synchronisation requests can be carry out on a daily basis. Figure 6.5 shows the scheduling of a synchronisation period of “1”, which means **DAILY**, the hour at which the synchronisation should start (**13:00**), its start date (**8-05-2021**), and end date (**9-23-2021**). After the creation of this configuration, it is expected that the system will execute the data synchronisation request respecting the defined limits.

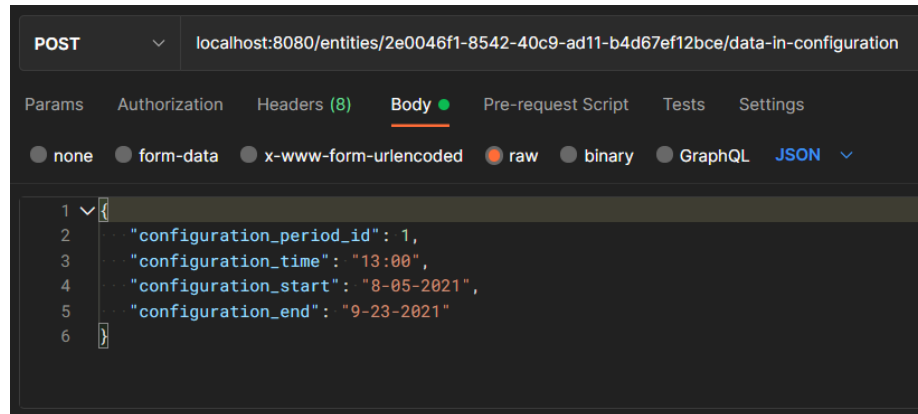
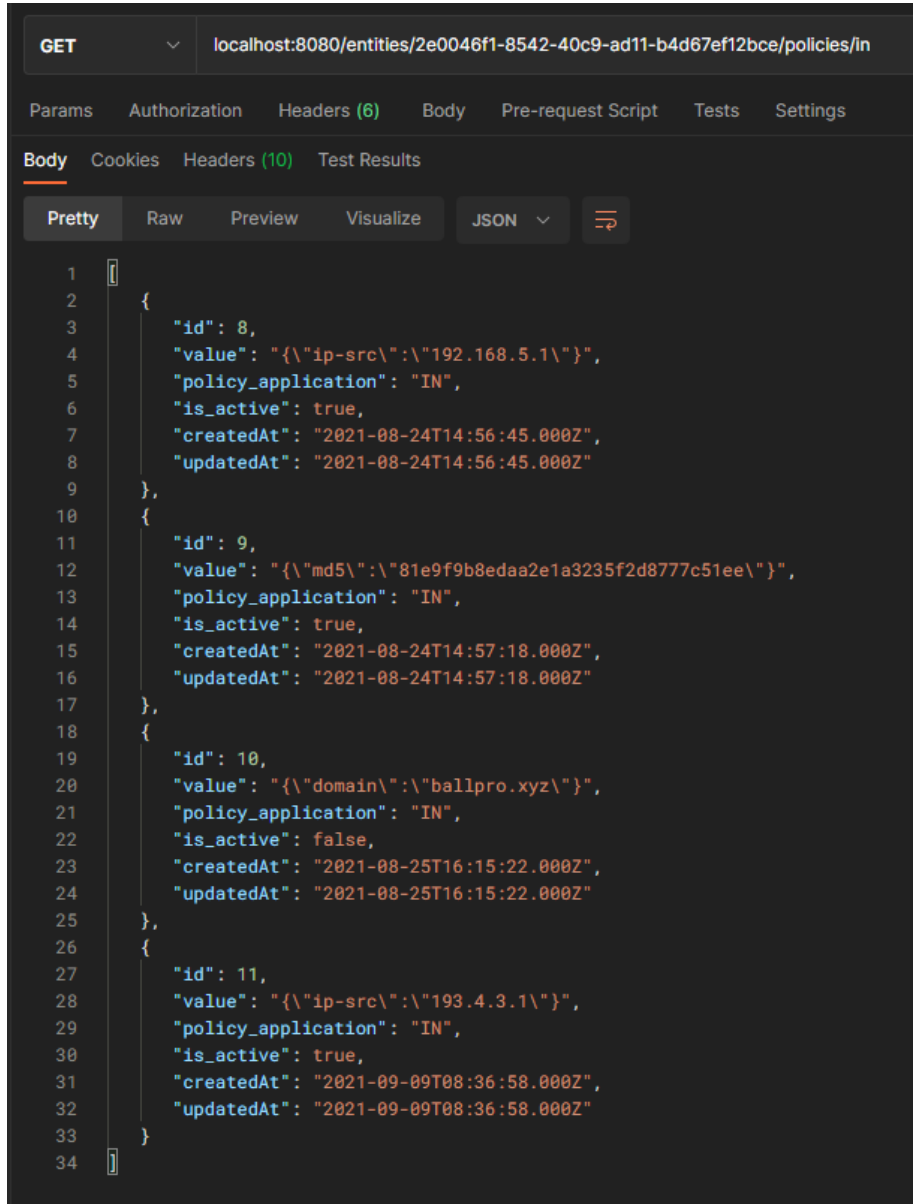


Figure 6.5: Synchronisation scheduling route

In the data synchronisation process, policies can be applied in order better to control the sent/received information. On one hand, Entity 1 (owner of Instance 1) defines the terms to be requested to Entity 2 (owner of Instance 2) by synchronisation, thus defining its data input policies. On the other hand, Entity 2 defines which terms are prohibited from being obtained by the Entity 1, thus defining its data output policies. Figure 6.6 shows the data input policies defined on Entity 1 and for Entity 2, whereas Figure 6.7 shows the data output policies defined on Entity 2 to control Entity 1. Cross analysing both policies, it is possible to conclude that the only the term permitted in both is the value `{"ip-src": "192.168.5.1"}`, indicating that the remaining values of the input data policy will be ignored by Entity 2. Such results in Entity 1 only being able to synchronise information related to the value `{"ip-src": "192.168.5.1"}` (requirement R4).

Once the policies have been defined and the configuration created, the data



```
GET localhost:8080/entities/2e0046f1-8542-40c9-ad11-b4d67ef12bce/policies/in

Params Authorization Headers (6) Body Pre-request Script Tests Settings

Body Cookies Headers (10) Test Results

Pretty Raw Preview Visualize JSON

1 [
2   {
3     "id": 8,
4     "value": "{\\"ip-src\\":\\"192.168.5.1\\"}",
5     "policy_application": "IN",
6     "is_active": true,
7     "createdAt": "2021-08-24T14:56:45.000Z",
8     "updatedAt": "2021-08-24T14:56:45.000Z"
9   },
10  {
11    "id": 9,
12    "value": "{\\"md5\\":\\"81e9f9b8edaa2e1a3235f2d8777c51ee\\"}",
13    "policy_application": "IN",
14    "is_active": true,
15    "createdAt": "2021-08-24T14:57:18.000Z",
16    "updatedAt": "2021-08-24T14:57:18.000Z"
17  },
18  {
19    "id": 10,
20    "value": "{\\"domain\\":\\"ballpro.xyz\\"}",
21    "policy_application": "IN",
22    "is_active": false,
23    "createdAt": "2021-08-25T16:15:22.000Z",
24    "updatedAt": "2021-08-25T16:15:22.000Z"
25  },
26  {
27    "id": 11,
28    "value": "{\\"ip-src\\":\\"193.4.3.1\\"}",
29    "policy_application": "IN",
30    "is_active": true,
31    "createdAt": "2021-09-09T08:36:58.000Z",
32    "updatedAt": "2021-09-09T08:36:58.000Z"
33  }
34 ]
```

Figure 6.6: Sample input policies

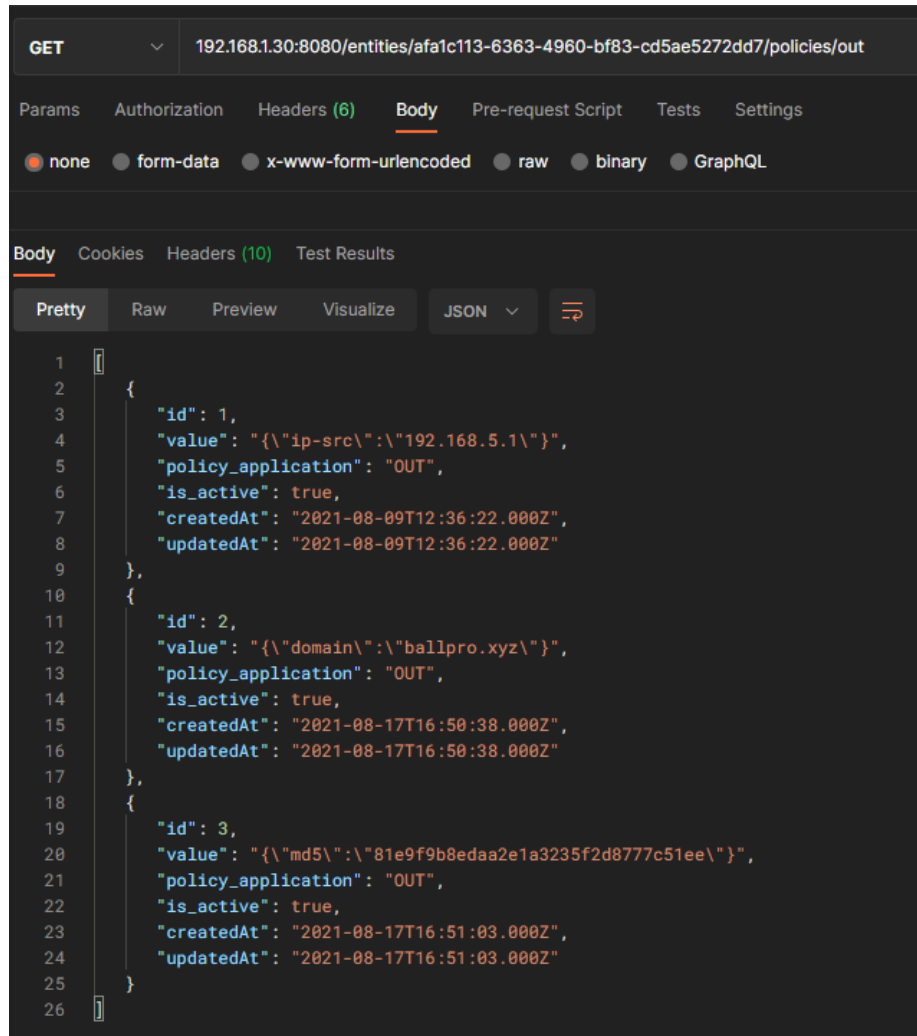


Figure 6.7: Sample output policies

synchronisation process is carried out. Figure 6.8 shows the logs of the execution of one synchronisation process (requirement R1). It can be observed that the synchronisation process lasted 626ms and that three events were found. However, none of the events were added in the MISP instance because the previous synchronisation process obtained the same results, meaning that the information was already present.

```

Org: {
  id: '1',
  name: 'ORGNAME',
  uuid: '82e405d8-83f0-4571-9b0e-58c05c5f47a9'
},
Orgc: {
  id: '1',
  name: 'ORGNAME',
  uuid: '82e405d8-83f0-4571-9b0e-58c05c5f47a9'
},
EventTag: [],
attributes: [ [Object], [Object] ]
}
]
Verifying the existence of event: Event for synchronisation test
> Event: Event for synchronisation test - already exists on https://mispshr1
Verifying the existence of event: Event for synchronisation test #2
> Event: Event for synchronisation test #2 - already exists on https://mispshr1
Verifying the existence of event: Event for synchronisation test #3
> Event: Event for synchronisation test #3 - already exists on https://mispshr1
Synchronization completed in 626 ms!

```

Figure 6.8: Synchronisation output

## 6.3 Shared index

The shared index enables secure, policy-based, information exchange within a sharing group that is formed by two or more entities. For demonstration purposes, Entities 1 and 2 will constitute a sharing group to verify the operation of the search and update functionalities when using a shared index.

The first step is to define a name for the new sharing group to ensure that both entities have the same group created, as described in section 5.4.1. After the group creation, each entity associates the remaining entities that will be part of it. Note that these steps must be performed in all entities. In the adopted scenario, Entity 1 will associate Entity 2, and Entity2 will associate Entity 1. Figure 6.9 shows the route used for the creation of the sharing group. Figure 6.10 shows the route used to associate an entity to an existing group, using its identifier.

Upon completing all associations by all involved entities, a shared symmetric key is generated for this specific group. Recall that this key comprises contributions from all entities within a sharing group. The route used for the key generation is shown in Figure 6.11.

The process of updating the index requires, as with the synchronisation process, the creation of a configuration. This configuration is similar to the synchronisation scheduling, this time using the shared index. Figure 6.12 shows



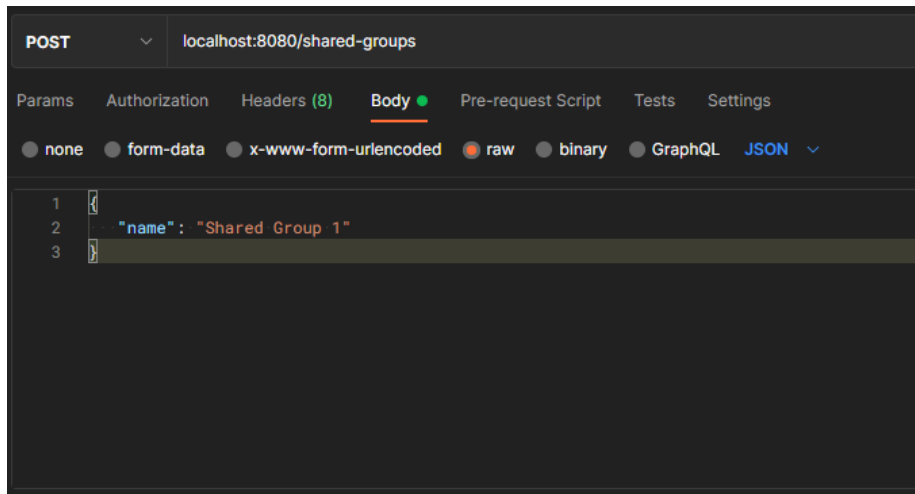


Figure 6.9: Group creation route

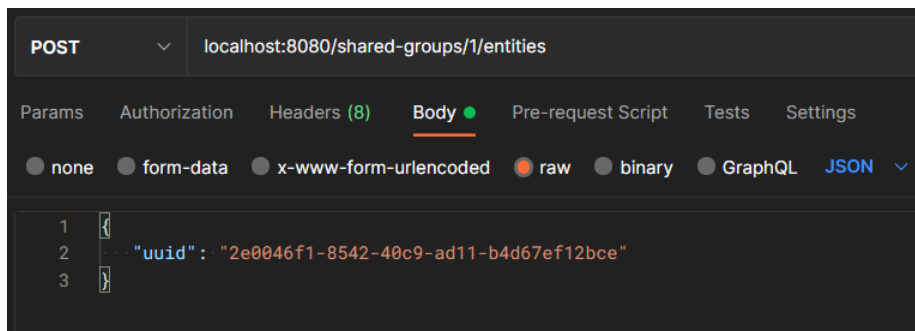


Figure 6.10: Association of an entity to the group

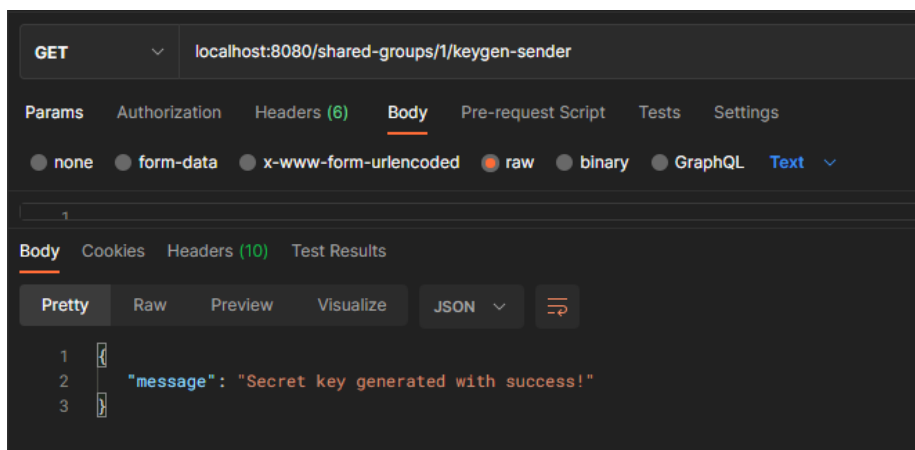


Figure 6.11: Symmetric key generation route

the route used to configure a new shared index and it receives 3 parameters: the URL of the MongoDB database to store the index, the update scheduling (which works similarly to the data synchronisation scheduling), and the time at which the update should be performed.

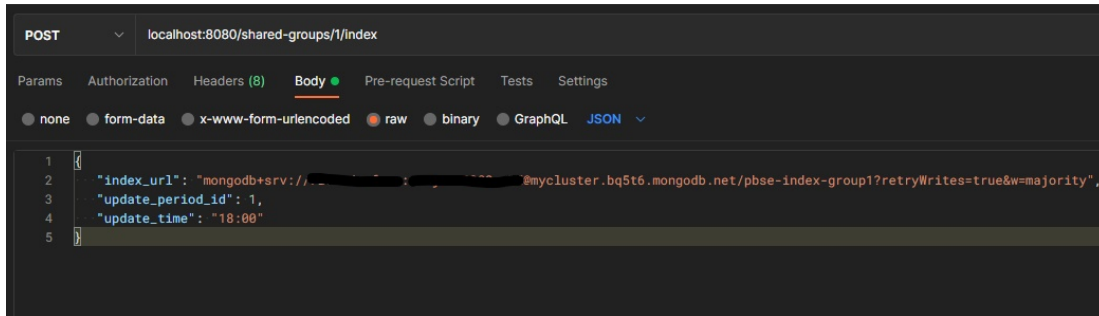


Figure 6.12: Shared index configuration route

After the configuration of the shared index, the user can then define which terms will be shared with the index. Figure 6.13 shows an example list of values allowed to be placed in the index, in the form of trapdoors. The result of the update is presented in Figure 6.14 that shows the MongoDB database with the created trapdoors and the encrypted list of entity identifiers. This list is encrypted and comprises the identifiers of the entities that have information related to the trapdoor, or searchable term (requirement R2). The values entered in the configuration are only uploaded to the shared index if these are present in the MISP platform of the uploading entity, as described in Section 5.4.3.

The search is performed as follows. To search a value through the index it is only necessary to enter that value in a specific route of the API as shown in figure 17 searching for the value `{"ip-src": "192.168.5.1"}`. As described in section 5.4.4, the result of the search in the shared index will be the list of entities that have information about the searched term. Internally, a data synchronisation request is made with the entities obtained from the search in the shared index.

## 6.4 Discussion

The implemented prototype allowed for the verification that the proposed solution is feasible and satisfies the identified requirements. The peering configuration, an essential element of the proposed system, operated as expected and, when concluded, can be verified by each entity. This indicates that the configuration of the identifiers and the public-private key pairs of each entity was done correctly. Data exchange between different MISP instances using the proposed solution was also demonstrated, including the application of data policies in such data exchange. The adoption of a non-reusable symmetric session key also proved to be feasible for small groups and, when using a shared index, the system can guarantee the confidentiality and integrity of all the information made available in a shared. Confidentiality is assured by the use of a symmetric encryption algorithm, and integrity is assured by the use of digital signatures.

```
GET localhost:8080/shared-groups/1/index/values

Params Authorization Headers (6) Body Pre-request Script Tests Settings

Body Cookies Headers (10) Test Results

Pretty Raw Preview Visualize JSON

1 [
2   {
3     "id": 1,
4     "index_configuration_id": 7,
5     "value": "{\"ip-src\":\"193.1.1.1\"}",
6     "createdAt": "2021-07-14T11:43:20.000Z",
7     "updatedAt": "2021-07-14T11:43:20.000Z"
8   },
9   {
10    "id": 3,
11    "index_configuration_id": 7,
12    "value": "{\"ip-src\":\"193.2.5.4\"}",
13    "createdAt": "2021-07-14T11:45:32.000Z",
14    "updatedAt": "2021-07-14T11:45:32.000Z"
15  },
16  {
17    "id": 4,
18    "index_configuration_id": 7,
19    "value": "{\"ip-src\":\"193.100.5.4\"}",
20    "createdAt": "2021-07-14T13:44:55.000Z",
21    "updatedAt": "2021-07-14T13:44:55.000Z"
22  },
23  {
24    "id": 5,
25    "index_configuration_id": 7,
26    "value": "{\"url\":\"fkkfap.com/lc/index.php\"}",
27    "createdAt": "2021-07-20T15:15:23.000Z",
28    "updatedAt": "2021-07-20T15:15:23.000Z"
29  },
30  {
31    "id": 6,
32    "index_configuration_id": 7,
33    "value": "{\"md5\":\"d27e2e5039cc62ca865c8090548c1552\"}",
34    "createdAt": "2021-07-20T15:44:10.000Z",
35    "updatedAt": "2021-07-20T15:44:10.000Z"
36  }
37 ]
```

Figure 6.13: Index configuration values

**pbse-index-group1.iocreferences**

COLLECTION SIZE: 2.64KB TOTAL DOCUMENTS: 3 INDEXES TOTAL SIZE: 36KB

**Find** Indexes Schema Anti-Patterns 0 Aggregation Search Indexes ●

**FILTER** { field: 'value' }

**QUERY RESULTS 1-3 OF 3**

```
{
  "_id": ObjectId("60ffe89b0a455d463cb8a602"),
  "entities": Array
    0: "81061ed6a71dac75712992197ff4b043dbe72c2d88d90411a0e50f7912f302baedeeea..."
  "trapdoor": "a58e3c4db5f3d1c7ed2d5c7052e8ddc43233d0cae14c349b2de24fed3571639b"
  "__v": 0
}
```

```
{
  "_id": ObjectId("612e5fd45b02d25a586a5bb6"),
  "entities": Array
    0: "81061ed6a71dac75712992197ff4b043dbe72c2d88d90411a0e50f7912f302ba9d201f..."
  "trapdoor": "821a24f16ea031abd077674156ae8aab63bdde2bceae0284ef6ce5b1ad299318"
  "__v": 0
}
```

```
{
  "_id": ObjectId("612e69354ec6cc00572f9d6f"),
  "entities": Array
    0: "803eb449c143ab23e791e393a72e0aef97db8bc60ec8951e3f45ac005ad551f8864528..."
  "trapdoor": "13daf3b81a74beb095747897bdc51a3a3b7077c9d588bc5c9dcb4ab78f91e0bd"
  "__v": 0
}
```

Figure 6.14: Visual representation of the shared index

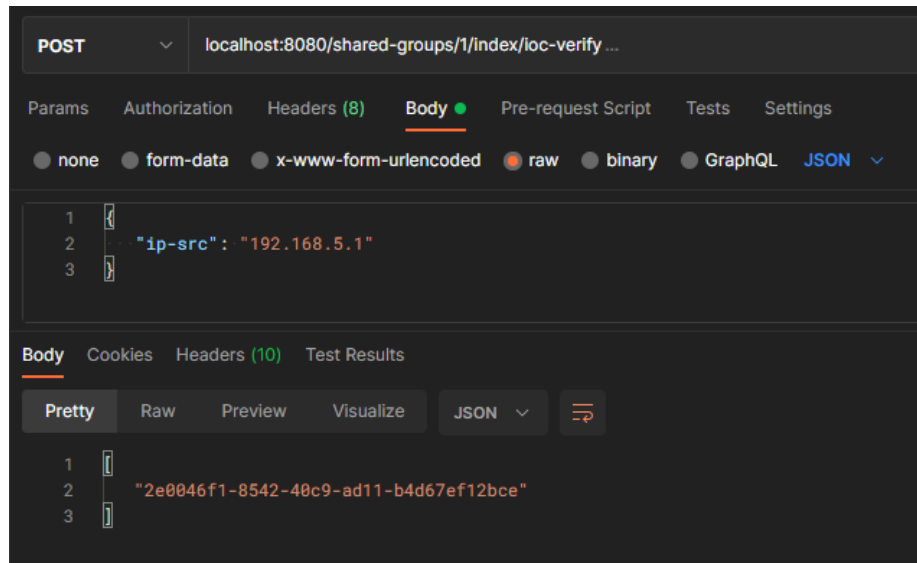


Figure 6.15: Route to search in the shared index

The use of the shared index was also demonstrated.

## Chapter 7

# Conclusions

Using MISP to share threat information among multiple entities presents limitations whenever these entities wish to have tighter control over this disclosure, either by changing trust groups or by restricting the availability of the information for a period of time.

In this document it was proposed a controlled information sharing functionality that enables secure sharing of cyber threat classified information stored in multiple MISP instances. The proposed solution acts as a proxy between MISP instances and makes use of searchable encryption techniques so that greater control over the exchanged information can be enforced. This proposed solution was presented in Chapter 4 which specifies the requirements and the elementary processes of the system through multiple message sequence tables, such as the peering validation process that will allow confidential communication between entities, the synchronisation process used for confidential data exchange between entities, and also the data search process that makes use of the main element of our solution which is the shared searchable index ensuring the confidentiality of searches using searchable encryption techniques and the integrity of the search terms that compose the index.

Chapter 5 presented the implementation of a prototype of our proposed solution where the main processes of our system are presented in more detail to give a better understanding of its operation. The first step of the implementation consisted of the initial setup of the system, composed of the generation of the unique identifier and the RSA key pair with 2048 bit size that will be used in the communication between different entities. As for the peering validation, this consists in validating the configuration of an external entity in our system through its unique identifier, its public key for the encryption of the messages sent, and also its URL to allow communication between both entities.

The data synchronisation process comprises several elements, both on the part of the entity that makes the synchronisation request and on the part of the entity that receives the request. On the side of the entity that makes the synchronisation request, the user schedules the synchronisation that will be automatically executed by the system and also defines the terms to be searched through the creation of input policies. These policies will constitute the search query that will be sent together with a secret key for the encryption of the results. The entity that receives the synchronisation request can control the searches made to its system. This control is then done by the output policies

defined for the entity that made the synchronisation request and indicate which terms are allowed to a certain entity to search. Finally, data sharing through a shared index allows the secure sharing of information between several entities forming a sharing group. The index is managed by all members of the group and is formed by the search terms that each entity makes available together with the unique identifier and the signature of the search term by the entity. The confidentiality of the index is guaranteed through the symmetric key generated with the contribution of all the members of the group which is used to generate the search *trapdoors* as well as encrypt the remaining information present in the index, avoiding the existence of any information in cleartext. Integrity is guaranteed during the search process where the signatures present in the shared index are verified using the public keys of each entity that updated the index.

However, the developed system presents some limitations that can be improved in future work. One of the limitations is in the shared index. As shown in Chapter 6, the index update process only takes into account the insertion of individual search terms and there is no possibility of creating *trapdoors* based on two or more search terms. This results in the search process being performed only by searching the index for individual search terms. On the one hand, the search process becomes more efficient as the complexity of the search for a single term is reduced. However, there might be cases where there is a need to search several terms in a grouped manner. The current solution would be to search for each term individually and then join all the information together, which would not be as efficient as expected since the number of requests to be made would increase according to the number of terms present in the query, leading to an increase in the computational load in the management of several simultaneous requests. Therefore, the solution would be to adapt the search and update processes in the shared index. The system would allow the submission of more elaborate queries while the index update process would take into account all the search terms in the query at the time of generating the *trapdoor* to be stored in the index.

Another future work to be mentioned is the performance of scalability tests on the system among them the performance of tests on the algorithm for generating the symmetric key in the sharing groups. As described in Section 5.4.2, the final symmetric key is generated with the contribution of all the entities present in the sharing group, where each entity generates its key after obtaining all the contributions. Therefore, the test would be important to verify the feasibility of the current architecture of the algorithm in cases where there are sharing groups with a high number of entities present, analysing if the total number of messages exchanged between all entities for the production of the final key has an impact on the computational load of the implemented prototype.

# Bibliography

- [1] Wagner, Cynthia, Dulaunoy, Alexandre, Wagener, Gérard, Iklody, and Andras, “Misp: The design and implementation of a collaborative threat intelligence sharing platform,” in *Proceedings of the 2016 ACM on Workshop on Information Sharing and Collaborative Security*, 2016, pp. 49–56.
- [2] Dulaunoy, Alexandre, Wagener, Gérard, Iklody, Andras, Mokaddem, Sami, Wagner, and Cynthia, “An indicator scoring method for misp platforms,” in *The Networking Conference TNC*, vol. 18, 2018.
- [3] D. Das, U. Lanjewar, and S. Sharma, “The art of cryptology: From ancient number system to strange number system,” in *International Journal of Application or Innovation in Engineering & Management (IIAIEM)*, vol. 2, no. 4, 2013.
- [4] D. Luciano and G. Prichett, “Cryptology: From caesar ciphers to public-key cryptosystems,” in *The College Mathematics Journal*, vol. 18, no. 1, 1987, pp. 2–17.
- [5] J.-P. Aumasson, “Serious Cryptography: A Practical Introduction to Modern Encryption,” 2017.
- [6] W. Stallings, “Network and internetwork security: principles and practice,” in *Prentice Hall Englewood Cliffs, NJ*, vol. 1, 1995.
- [7] J. Katz and Y. Lindell, “Introduction to modern cryptography.” Chapman and Hall/CRC, 2014.
- [8] N. Ferguson, B. Schneier, and T. Kohno, *Cryptography Engineering, Design Principles and Practical Applications*. Wiley Publishing, Inc., 2010, ch. 2.1.
- [9] —, *Cryptography Engineering, Design Principles and Practical Applications*. Wiley Publishing, Inc., 2010, ch. 3.
- [10] M. Agrawal and P. Mishra, “A comparative survey on symmetric key encryption techniques,” in *International Journal on Computer Science and Engineering*, vol. 4, no. 5, 2012, p. 877.
- [11] N. Ferguson, B. Schneier, and T. Kohno, *Cryptography Engineering, Design Principles and Practical Applications*. Wiley Publishing, Inc., 2010, ch. 3.5.2.



- [12] N. I. of Standards and Technology, “AES Round 1 Technical Evaluation, CD-1: Documentation,” August 1998, pp. 54, 327, 329, 337.
- [13] —, “Proc. 3rd AES candidate conference,” April 2000, p. 54.
- [14] A. E. Standard, “Federal information processing standards publication 197,” in *FIPS PUB*, 2001, pp. 46–3.
- [15] J. Daemen and V. Rijmen, “The design of Rijndael: AES-the advanced encryption standard.” Springer Science & Business Media, 2013.
- [16] —, “AES Proposal: Rijndael. In National Institute of Standards and Technology [12],” p. 55.
- [17] N. Ferguson, J. Kelsey, S. Lucks, B. Schneier, M. Stay, D. Wagner, and D. Whiting, “Improved Cryptanalysis of Rijndael,” in *Lecture Notes in Computer Science*, vol. 1978, Springer-Verlag, 2000, pp. 213–230.
- [18] N. Ferguson, B. Schneier, and T. Kohno, *Cryptography Engineering, Design Principles and Practical Applications*. Wiley Publishing, Inc., 2010, ch. 4.2.
- [19] —, *Cryptography Engineering, Design Principles and Practical Applications*, 2010.
- [20] —, *Cryptography Engineering, Design Principles and Practical Applications*. Wiley Publishing, Inc., 2010, ch. 2.3.
- [21] 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.
- [22] 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.
- [23] N. Ferguson, B. Schneier, and T. Kohno, *Cryptography Engineering, Design Principles and Practical Applications*. Wiley Publishing, Inc., 2010, ch. 12.1.
- [24] R. Rivest, A. Shamir, and L. Adleman, “A Method for Obtaining Digital Signatures and Public-Key Cryptosystems,” in *Communications of the ACM*, vol. 21, February 1978, pp. 120–126.
- [25] N. Ferguson, B. Schneier, and T. Kohno, *Cryptography Engineering, Design Principles and Practical Applications*. Wiley Publishing, Inc., 2010, ch. 12.4.
- [26] —, *Cryptography Engineering, Design Principles and Practical Applications*. Wiley Publishing, Inc., 2010, ch. 10.3.5.
- [27] —, *Cryptography Engineering, Design Principles and Practical Applications*. Wiley Publishing, Inc., 2010, ch. 2.4.

- [28] —, *Cryptography Engineering, Design Principles and Practical Applications*. Wiley Publishing, Inc., 2010, ch. 5.
- [29] S. Bakhtiari, R. Safavi-Naini, and J. Pieprzyk, “Cryptographic hash functions: A survey.” Citeseer, Tech. Rep., 1995.
- [30] N. Ferguson, B. Schneier, and T. Kohno, *Cryptography Engineering, Design Principles and Practical Applications*. Wiley Publishing, Inc., 2010, ch. 5.2.
- [31] N. I. of Standards and Technology, “Secure Hash Standard,” in *FIPS PUB 180-1*, April 1995, p. 82.
- [32] F. Chabaud and A. Joux, “Differential Collisions in SHA-0,” in *Lecture Notes in Computer Science*, H. Krawczyk, Ed., vol. 1462, Springer-Verlag, 1998, pp. 56–71.
- [33] X. Wang, Y. L. Yin, and H. Yu, “Finding Collisions in the Full SHA-1. In Shoup [62],” in *FIPS PUB 180-1*, pp. 17–36.
- [34] B. Schneier, “Schneier on security: cryptanalysis of sha-1.” Schneier.com, 2005.
- [35] N. I. of Standards and Technology, “Secure Hash Standard (draft),” in *FIPS PUB 180-3*, 2008.
- [36] N. Ferguson, B. Schneier, and T. Kohno, *Cryptography Engineering, Design Principles and Practical Applications*. Wiley Publishing, Inc., 2010, ch. 5.2.4.
- [37] —, *Cryptography Engineering, Design Principles and Practical Applications*. Wiley Publishing, Inc., 2010, ch. 6.
- [38] —, *Cryptography Engineering, Design Principles and Practical Applications*. Wiley Publishing, Inc., 2010, ch. 6.4.
- [39] M. Bellare, R. Canetti, and H. Krawczyk, “Keying Hash Functions for Message Authentication,” *In Koblitz*, pp. 1–15.
- [40] H. Krawczyk, M. Bellare, , and R. Canetti, “HMAC: Keyed-Hashing for Message Authentication.” RFC 2104, February 1997.
- [41] M. Bellare, “New Proofs for NMAC and HMAC: Security Without Collision-Resistance,” in *Lecture Notes in Computer Science*, vol. 4117, Springer-Verlag, 2006, pp. 602–619.
- [42] 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.
- [43] 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.
- [44] E. J. Goh, “Secure indexes,” *IACR Cryptol. ePrint Arch.*, vol. 216, 2003.

- [45] 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.
- [46] C. Bösch, P. Hartel, W. Jonker, and A. Peter, "A survey of provably secure searchable encryption," *ACM Computing Surveys (CSUR)*, vol. 47, no. 2, pp. 1–51, 2014.
- [47] 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.
- [48] D. X. Song, r. D. Wagne, and A. Perrig, "Practical techniques for searches on encrypted data," In *Proceeding 2000 IEEE Symposium on Security and Privacy. S&P 2000 (pp. 44-55)*, May 2000.
- [49] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," in *Communications of the ACM*, vol. 13, no. 7, 1970, pp. 422–426.
- [50] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky, "Searchable symmetric encryption: improved definitions and efficient constructions," in *Journal of Computer Security*, vol. 19, no. 5, 2011, pp. 895–934.
- [51] S. Emil, P. Charalampos, and S. Elaine, "Practical Dynamic Searchable Encryption with Small Leakage," in *NDSS*, vol. 71, 2014, pp. 72–75.
- [52] N. Muhammad, P. Manoj, and G. C. A, "Dynamic searchable encryption via blind storage," in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 639–654.
- [53] D. Boneh, G. Di Crescenzo, R. Ostrovsky, and G. Persiano, "Public key encryption with keyword search," Springer, Berlin, Heidelberg, May 2004, pp. 506–522.
- [54] D. Boneh and M. Franklin, "Identity-based encryption from the weil pairing," in *Annual international cryptology conference*. Springer, 2001, pp. 213–229.
- [55] 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.
- [56] B. Mihir, B. Alexandra, and O. Adam, "Deterministic and efficiently searchable encryption," in *Annual International Cryptology Conference*. Springer, 2007, pp. 535–552.
- [57] M. Bellare and P. Rogaway, "Optimal asymmetric encryption – how to encrypt with RSA," in *De Santis, A. (ed.) EUROCRYPT 1994. LNCS*, vol. 950. Heidelberg: Springer, 1995.

- [58] E. Fujisaki, T. Okamoto, D. Pointcheval, and J. Stern, “RSA-OAEP is secure under the RSA assumption,” in *Kilian, J. (ed.) CRYPTO 2001. LNCS*, vol. 2139. Heidelberg: Springer, 2001.
- [59] L. Xueqiao, Y. Guomin, S. Willy, T. Joseph, L. Ximeng, and S. Jian, “Privacy-preserving multi-keyword searchable encryption for distributed systems,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 3, pp. 561–574, 2020.
- [60] F. Liming, S. Willy, G. Chunpeng, and W. Jiandong, “Public key encryption with keyword search secure against keyword guessing attacks without random oracle,” in *Information Sciences*, vol. 238. Elsevier, 2013, pp. 221–241.
- [61] C. Rongmao, M. Yi, Y. Guomin, G. Fuchun, and W. Xiaofen, “Dual-server public-key encryption with keyword search for secure cloud storage,” in *IEEE transactions on information forensics and security*, vol. 11, no. 4. IEEE, 2015, pp. 789–798.
- [62] “Advances in Cryptology—CRYPTO 2005,” in *Lecture Notes in Computer Science*, e. Victor Shoup, Ed., vol. 3621, Springer-Verlag, 2005.