PriVeil Circle: using Secure Multiparty Computation for sharing threat information

Miguel Gil da Silva Simão

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisor(s): Prof. Miguel Filipe Leitão Pardal
Prof. Carlos Nuno da Cruz Ribeiro

Examination Committee

Chairperson: Prof. Teresa Maria Sá Ferreira Vazão Vasques
Supervisor: Prof. Miguel Filipe Leitão Pardal
Member of the Committee: Prof. Pedro Ricardo Morais Inácio

January 2021

Declaration

I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgments

I want to express my gratitude to my supervisors, Professor Miguel Pardal and Professor Carlos Ribeiro, for letting me do this work and work under their guidance. In particular, a special thanks to professor Miguel Pardal for all the meetings (both in-person and virtual), guidance, support and feedback provided during the development of this work. To my family and friends, I would like to to thank them for their support over these months. Without you all, this work would not have been possible.

Resumo

As organizações são frequentemente afetadas por roubos de dados que provocam prejuízos financeiros e reputacionais. Para eficazmente prevenir este problema, estas necessitam de recolher informação sobre ameaças informáticas quer internamente, quer externamente, através de comunidades de partilha de informação. Contudo, as organizações continuam relutantes em partilhar os seus dados, visto que podem revelar informação sensível sobre as suas infraestruturas. Uma das formas de lidar com informação sensível é a anonimização, apesar de reduzir a utilidade dos dados e poder ser vulnerável a uma reidentificação dos parâmetros anonimizados.

Neste trabalho apresentamos o *Circle*, a segunda componente da plataforma de partilha de ameaças informáticas *PriVeil*. Numa primeira fase, os utilizadores submetem os relatórios de segurança cifrados para o componente *Square*, o qual faz a correspondência entre eles. Os utilizadores cujos relatórios têm correspondência são encaminhados para um *Circle*, onde a operação de Computação Multipartidária Segura (do inglês, *Secure Multiparty Computation*) de Interseção Privada de Conjuntos (do inglês, *Private Set Intersection*) permite encontrar etiquetas comuns nos relatórios através de uma computação que garante a preservação da privacidade. Isto permite aos utilizadores confirmarem que outros podem estar a ser afetados por uma ameaça semelhante.

Nós implementámos e avaliámos o *Circle* através de nove experiências, nas quais medimos a duração, utilização de processador e memória do programa de um participante, durante uma sessão. Os resultados mostram que o desempenho do nosso protótipo é adequado à sua aplicação em situações reais, proporcionando um ambiente onde os utilizadores podem partilhar informação sobre ameaças informáticas.

Palavras-chave: Partilha de Ameaças Informáticas, Relatórios de Segurança, Privacidade, Computação Multipartidária Segura, Interseção Privada de Conjuntos

Abstract

Nowadays, data breaches occur frequently in organizations, causing them economical and reputational losses. To prevent them more effectively, organizations need to gather cyber threat information internally and externally from sharing communities. However, organizations are still reluctant to share their own cyber threat data, since they are afraid to disclose sensitive information about their infrastructure. Anonymization of indicators is a commonly used solution when handling sensitive information, but it reduces the data utility and can be vulnerable to re-identification of the anonymized parameters.

In this work we present *Circle*, the second component of the *PriVeil* cyber threat sharing platform. In a first phase, users submit their encrypted security reports to the *Square* component of the platform, which matches them. Users whose reports matched are forwarded to a *Circle* instance, where the Secure Multiparty Computation operation of Private Set Intersection allows the participants to find the common tags contained in their reports through a privacy-preserving computation. This allows users to confirm that other users may be subject to a similar cyber threat described in their reports.

We implemented and evaluated *Circle* with nine experiments, in which we measured the duration, CPU and memory usage of a participant program during a session. The results showed that the performance of our prototype makes it applicable to real-case scenarios, providing an environment where users can share information about cyber threats.

Keywords: Cyber Threat Sharing, Security Reports, Privacy, Secure Multiparty Computation, Private Set Intersection

Contents

List of Tables

List of Figures

Glossary

2PC Two-party Computation

AEAD Authenticated Encryption with Associated Data

AES Advanced Encryption Standard

API Application Programming Interface

CA Certification Authority

CVE Common Vulnerabilities and Exposures

GC Garbled Circuits

HE Homomorphic Encryption

HMAC Hash-based Message Authentication Code

IP Internet Protocol

IoC Indicators of Compromise

MAC Message Authentication Code

MISP Malware Information Sharing platform

OPPRF Oblivious Programmable Pseudorandom Function

OPRF Oblivious Pseudorandom Function

PRF Pseudorandom Function

PSI Private Set Intersection

RPC Remote Procedure Call

RSA Rivest-Shamir-Adleman

SHA Secure Hash Algorithm

SMC Secure Multiparty Computation

SSL Secure Sockets Layer

SSS Shamir's Secret Sharing

TCP Transport Control Protocol

TLS	Transport Layer Security
TTP	Trusted Third Party
UDP	User Datagram Protocol
USD	United States Dollar
XML	Extensible Markup language

Chapter 1

Introduction

In the Information Age, data is a major asset of every organization. A part of this data is confidential and, consequently, meant to remain private. According to [CLY17], a data breach occurs when confidential information from the victim is disclosed to unauthorized parties. If there is an unauthorized access to confidential data, the organization could suffer financial and reputational losses. In a report conducted by Ponemon Institute and sponsored by IBM Security [IS19], it is estimated that the average cost of a data breach between July of 2018 and April of 2019 was USD3,92 million. The same study also states that the average time to identify and contain a breach is 279 days, i.e. 9 months. Despite the increase in cyber security awareness in employees and in the skill of cyber security teams, hackers are also becoming craftier, finding new vulnerabilities and creating complex and destructive exploits. Furthermore, hacking is made easier by exploits and malware that can be purchased at online anonymous markets. This commoditization is lowering the capabilities needed to engage in cybercrime activities [vWTS+18].

As cooperation increases amongst criminals to discover new exploits, it is almost impossible for a single organization to handle cyber threats. To counter the ever-evolving cybercrime, governments, organizations and individuals, need to adopt new defensive tactics that also increase their cooperation. In particular, sharing cyber threat data externally is a crucial way to find new vulnerabilities being exploited by attackers, new threats that affect organizations and even Indicators of Compromise (IoC) such as known malware hashes and Internet Protocol (IP) addresses related to threat actors.

The Malware Information Sharing Platform (MISP) [WDWI16] is a well-known platform where users can share any relevant threat indicator with each other. In MISP, users deploy MISP instances that can connect with other users MISP instances to allow the sharing of any relevant threat information. The submitted information is called an event and is the responsibility of the event owner to select the information he wants to share and with whom they want to share it with. Through the use of MISP, users can share threat information in a trusted environment.

As we can observe, organizations have been gathering cyber threat data internally, through respective security teams, and externally, from open-source and commercial sources and sharing communities [BGS15]. However, despite the importance of cyber threat information sharing platforms, there are still obstacles to be addressed. In [JBW+16], the authors highlight the safeguarding of sensitive information as one of the major challenges to cyber threat information sharing. In fact, directly sharing sensitive security logs, network information, malware samples and packet captures could expose the infrastructure of an organization and its defensive capabilities, leading to the emergence of new cyber threats. As a result, the authors suggest the anonymization and sanitization of certain parameters when handling and sharing sensitive information: in network indicators, anonymizing or sanitizing IP addresses of the target systems and indicators which might reveal the network infrastructure of the organization; in phishing email samples, anonymizing email addresses belonging to the organization and even removing any sensitive information contained in the message; in system, network and applications logs anonymizing IP addresses, timestamps, ports and protocols which might reveal any information about systems of the organization.

However, in the process of anonymization, relevant information which could be useful when dealing with a cyber threat can be lost. As stated by Fisk et al. [FAP+15], the process of anonymization of data in a cyber threat information sharing system often reduces its utility. Furthermore, the authors also mention that anonymization techniques can be vulnerable to attacks. In particular, Narayanan and Shmatikov [NS08] developed an algorithm that could de-anonymize anonymized records with a small amount of background knowledge about them. They applied their algorithm to the Netflix Prize, a contest which contained anonymized movie ratings from 500 000 Netflix subscribers, and demonstrated that an attacker with very limited knowledge could identify the movie rating record of each subscriber.

As a result, we conclude that, when sharing cyber threat information, anonymization is not an ideal solution to protect organizations sensitive data, since it decreases the shared data usefulness and it can be vulnerable to attacks, rendering its usage ineffective. These facts incited us to find a cyber threat sharing solution that prevented the unwilling disclosure of sensitive information, without the need of anonymization of sensitive information before sharing the data.

1.1 Approach

PriVeil [Gon19] was designed to be a platform that allows the participants to share threat information they possess with each other, to create a cooperation environment which allows them to better deal with cyber threats. *PriVeil* relies on Homomorphic Encryption (HE) and Secure Multiparty Computation (SMC) to create a system where users can share information which can contain sensitive indicators, without having to anonymize them. *PriVeil* is structured to operate in two phases: *Square* and *Circle*[1]. In *Square*, organizations submit their encrypted security reports describing security events to the platform. HE gave the system the ability to match the encrypted reports that describe similar cyber threats. HE schemes allow computations to be performed directly on encrypted data, yielding the same final result as if they were performed on the plaintext data itself [NLV11]. Afterwards, entities whose reports matched, are notified and receive a token that authorizes them to engage in a *Circle* session and are then forwarded to it.

The *Circle* component allows participants whose reports matched in *Square* to engage in a progressive disclosure of information, where they share information in a privacy-preserving way.

1.2 Objectives

One of the main goals of this work was to develop *Circle*, the second component of the *PriVeil* cyber threat information sharing system. To allow privacy-preserving computations to be performed in *Circle*, we resorted to the SMC operation of Private Set Intersection (PSI) to allow the participants to share cyber threat information contained in their security reports. However, at any point during *Circle*, they can leave the system without any penalty. This characteristic ensures that the participating organizations retain control over their data, only sharing the information they are willing to.

The second main objective of this work was to evaluate the performance and effectiveness of *Circle* according to realistic scenarios, to prove that this system is applicable in practice and can be used by real organizations, in order to share cyber threat information.

[1] In [Gon19], the nomenclature for *Square* is *Concourse* and for *Circle* is *Conclave*. We changed the *Concourse* name to *Square* to give a better idea of its functionality: it is a "public square", where users can submit their encrypted security events which are afterwards matched. The name *Conclave* was changed to *Circle* in the sense that this phase is a more "inner circle", where users share more granular threat information contained in their reports.

1.3 Dissertation Outline

The remainder of the dissertation is structured as follows: in chapter 2 we describe the cryptographic primitives, protocols and tools which were used to implement *Circle*; in chapter 3 we explain the requirements of our prototype, how it was designed and implemented, its functionalities and characteristics; in chapter 4 we assess whether the requirements of *Circle* were fulfilled, we detail the methodology used to evaluate *Circle* and the results obtained; in chapter 5 we provide the final remarks on this work and improvements to be addressed in future work.

Chapter 2

Background & Related Work

In this chapter we provide the theoretical foundations necessary to understand the premisses and protocols we used to design and implement *Circle*. We start by providing a summary of modern cryptography, encryption techniques and their security properties and protocols, in section 2.1. Moreover, we give a detailed explanation of the cryptographic primitive of Secure Multiparty Computation (SMC), in section 2.2 and elaborate on its security properties, foundations, adversaries and protocols. Afterwards, in section 2.3, we provide an overview of a specific SMC operation, Private Set Intersection (PSI), which we used to implement *Circle* and give real-life examples of its application. We conclude this chapter by describing three SMC frameworks that implement PSI, in section 2.4.

2.1 Cryptography

Cryptography can be traced back to the Egyptians about 4000 years ago. However, according to [KL14], modern cryptography only emerged around the 1980s. It can be defined as the science that uses mathematical procedures to secure digital information against unauthorized third-parties. Modern cryptography is applied when securing communications, authenticating users and storing information securely.

Menezes et al. [MvOV96] define four main information security goals to be achieved, when using cryptographic protocols: *confidentiality*, *data integrity*, *authentication* and *non-repudiation*. *Confidentiality* ensures that only authorized parties can access the information. *Data integrity* implies that unauthorized parties did not alter the data. *Authentication* guarantees that parties can prove that they are who they claim to be and that information comes from where it is supposed to. Finally, *non-repudiation* implies that a party cannot deny that certain actions were done. In order to achieve these information security objectives we can use three cryptographic primitives: symmetric key cryptography, public key cryptography[1] and cryptographic hash functions.

In symmetric key cryptography, encryption and decryption are performed using the same key. In the encryption algorithm, the original message, the plaintext, is encrypted into the encrypted message, the ciphertext, by using a secret key. In the decryption algorithm, the same key is used to decrypt the ciphertext and recover the plaintext. In subsection 2.1.1 we describe the Advanced Encryption Standard (AES) algorithm, which is based on symmetric key cryptography.

A Message Authentication Code (MAC) is a special case of symmetric key cryptography. A MAC is a code which is computed using a message m and a shared symmetric key K. The sender of the message computes the MAC $M = MAC(K, m)$ and appends it to the respective message. Upon arrival, the receiver recomputes the MAC using the message m and the secret shared key K. If the recomputed MAC coincides with MAC appended to the message, the message was not changed in transit. However, if the recomputed MAC and and the one appended to the message differ, the receiver can conclude that the message was changed and its integrity compromised [Sta10].

In public key cryptography, there is a pair of two different keys: a private (secret) key and a public key. In the encryption algorithm, the public key is used to transform the plaintext into the ciphertext. In the decryption algorithm, the private key of the pair is used to recover the original message from the ciphertext. Furthermore, when using public key cryptography, a party can combine a message with a secret key to create a digital signature. The paired public key can then be used to validate that who generated and signed the message is the owner of the public key. The Rivest-Shamir-Adleman (RSA) [RSA78] algorithm is a widely used public key cryptography system. This algorithm relies on the high complexity of factoring large numbers and computing modular logarithms to make brute force attacks to it computationally infeasible. Nowadays, since RSA is slow, it is not frequently used to encrypt data, but to transmit symmetric keys over an insecure channel.

As defined in [Zú18], a hash function h is a one-way function which accepts as input a block of data of variable size M and outputs its hash value $H = h(M)$, an apparently random fixed size block of data. There are three properties desired for cryptographic hash functions:

1. **Preimage resistance:** It is computationally infeasible for an attacker with the value H to find M, such that $M = h^{-1}(H)$.

2. **Second-Preimage resistance:** It is computationally infeasible for an attacker with the

[1] Another frequent designation is Asymmetric Key Cryptography.

message M and its hash value $H = h(M)$, to obtain a different message M_1 such that $h(M) = h(M_1)$

3. **Collision resistance:** It is very hard for an attacker to find two different messages M and M_1 such that $h(M) = h(M_1)$

In subsection 2.1.2 we describe the Secure Hash Algorithm 2 (SHA-2). We can use cryptographic hash functions to construct a MAC. Such a scheme corresponds to a Hash-based Message Authentication Code (HMAC). In a HMAC, the message and a secret key are combined and subsequently put through a hash function to create an authenticated message. HMAC is proven to be secure as long as the underlying hash function is not broken [BCK96].

2.1.1 Advanced Encryption Standard

In 1997, the National Institute for Standards and Technology (NIST) published an open and international contest to find a symmetric key cipher for the new standard. After 4 years, in 2001, NIST announced that the Rijndael cipher [DR02], created by John Daemen and Vincent Rijmen, would become the cipher to be used in the new AES.

AES is based on substitution-permutation networks, which employs the substitution and permutation operations. In substitution, or S-box, the input, a binary word of x bits is transformed in a binary word of y bits, where x can be different from y. In permutation, or P-box, a binary word has its bits reordered. This algorithm supports an input block length of 128 bits and key lengths of 128 bits, for AES-128, 192 bits for AES-192 and 256 bits for AES-256. The first step in AES is to transform the 128 bits block input into the state array. The state array can be pictured as a 4x4 square matrix. Similarly, the input key is transformed into a state array. This state array can be pictured as a matrix with 4 rows, where the number of columns depends on the key length. For the 128, 192 and 256 bits used in the AES key, the key state array can be pictured as a 4x4, 4x6 and 4x8 matrix, respectively. The five operations defined for AES encryption are:

- *Key Schedule*: The input key state array is, through a series of operations, transformed in an expanded key of size 1408 bits, 1664 bits and 1920 bits for AES-128, AES-192 and AES-256, respectively. The expanded key is subsequently split in 11 round keys of 128 bits for AES-128, in 13 round keys of 128 bits for AES-192 and in 15 round keys of 128 bits for AES-256. Each one of these round keys is transformed in a 4x4 state array and used in each *AddRoundKey* operation.

- *AddRoundKey*: Each byte in the input 4x4 state array is *XORed*[2] with the corresponding byte of the round key 4x4 state array.

- *SubBytes*: Each byte of the state array is substituted by the corresponding value of the S-box.

- *ShiftRows*: Left shifts the rows of the state array over different offsets. The offsets for the first, second, third and fourth row of the state array are 0, 1, 2 and 3, respectively.

- *MixColumns*: Each byte of a column of the state array is transformed into a new value by operating on all the elements of the column of that byte. The operation performed is a matrix multiplication.

In figure 2.1 we represent the operations performed in the encryption and decryption phases of AES.

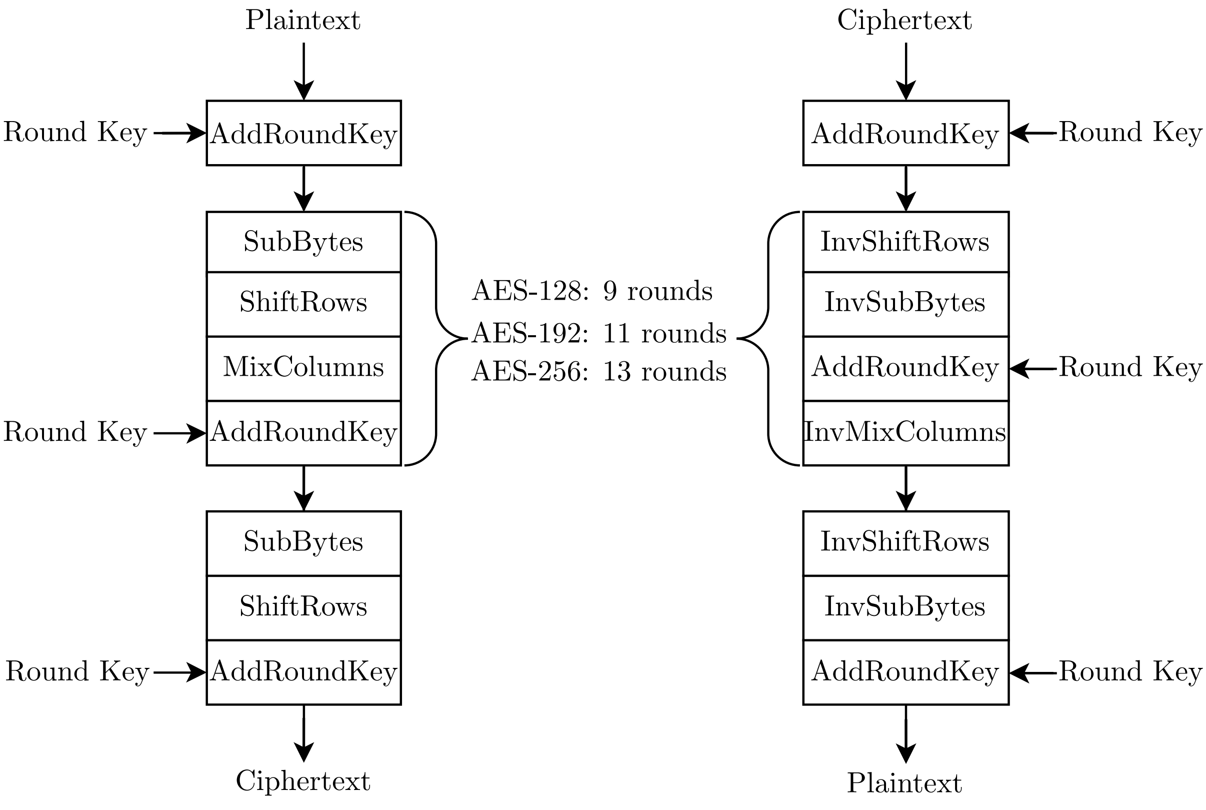


Figure 2.1: AES encryption and decryption phases.

The AES algorithm starts by executing the *Key Schedule* operation, where all the round keys are generated, followed by a *AddRoundKey* operation. Afterwards, for input key lengths of 128, 192 and 256 bits 9, 11 and 13 *Rounds* are executed, where the functions *SubBytes*, *ShiftRows*,

[2]XOR corresponds to the boolean operation of eXclusive OR.

MixColumns and *AddRoundKey* are performed sequentially. The protocol is concluded by a *FinalRound* that includes the *SubBytes*, *ShiftRows* and *AddRoundKey* operations.

A characteristic of *SubBytes*, *ShiftRows* and *MixColumns* is that they are invertible. In [Sta10] we can find an overview of each of the inverse operations. The inverse of *SubBytes*, *InvSubBytes*, uses the inverse S-box. The inverse of *ShiftRows*, *InvShiftRows*, shifts the rows in the opposite direction. Finally, the inverse of *MixColumns*, *InvMixColumns*, performs the inverse operation on the elements of the state. The fact that these functions have an inverse defined allows to perform the decryption of the ciphertext back to the plaintext.

AES is still considered secure, since there is no known attack that can break the full algorithm. In [BKR11], the authors propose a cryptanalysis method on the full AES, the biclique attack, that would allow to recover the secret key used in AES-128 with a time complexity of $2^{126.18}$ and data complexity of 2^{88}. However, as the authors state, their results do not threaten in any way the usage of AES. Since then, in [TW15], the authors improved biclique attack, improving the complexity for breaking the AES-128 secret key to a time complexity of $2^{126.01}$ and data complexity of 2^{72}. Once more, these results do not threaten the wide usage of AES protocol, so it remains the primary symmetric key cryptography protocol for the foreseeable future.

2.1.2 Secure Hash Algorithm 2

The first Secure Hash Algorithm, SHA-0, was published by NIST in 1993 and only after two years, the second Secure Hash Algorithm, SHA-1, was defined. In 2002, the SHA-2 family was standardized by NIST [NIS02] in order to increase SHA-1 message output size of 160 bits and improve the underlying algorithm. This new group of SHA hash functions includes the most commonly used SHA-256 and SHA-512 and the not so frequently used SHA-224 and SHA-384. The SHA-256 and SHA-224 functions accept message inputs with lengths smaller than 2^{64} bits. While SHA-256, subsequently, outputs a message digest of 256 bits (32 bytes), SHA-224 outputs a digest of 224 bits (28 bytes). SHA-512 and SHA-384 accept message inputs with lengths smaller than 2^{128} bits. The SHA-512 function outputs a digest of 512 bits (64 bytes), while SHA-384 outputs a digest of 384 bits (48 bytes).

SHA-2 is employed in a wide range of applications: in Federal departments and agencies in the United States, for the protection of sensitive unclassified information [NIS02], in the mining process employed by the *Bitcoin cryptocurrency*[3] and, in general, to verify the *data integrity* of messages and files.

[3] `https://www.mycryptopedia.com/sha-256-related-bitcoin/`, accessed on 25/12/2020.

2.1.3 Transport Layer Security

The Transport Layer Security Protocol (TLS) is the successor of the Sockets Layer Protocol (SSL). The aim of TLS is to provide two parties, often a client and a server, with a secure communication channel. The TLS protocol is complex and supports a variety of cipher suites and cryptographic protocols that can be used to implement it. The first version, TLS 1.0, emerged in 1999 as the successor of SSL 3.0 and, since the two are connected, the protocol is often referred to as TLS/SSL. The most recent version, TLS 1.3, was defined in RFC 8446 [Res18] in 2018, providing a simpler, faster and more secure protocol than the previous version, TLS 1.2. We can distinguish two different sub-protocols that are used in all TLS implementations:

1. *Handshake protocol*: A TLS communication starts by the client sending a *ClientHello* message to the server with parameters like the supported ciphers and supported TLS version. The server answers with a *ServerHello* message containing the chosen protocol version and cipher suite. We can have three types of *authentication* during the *handshake*: neither party authenticates itself to the other, only the server authenticates itself to the client, or both parties authenticate themselves to the other party. In the Internet, websites that use the HTTPS protocol [Res00], usually require a *server authentication* setting. However, since in our project, when we used TLS, we relied on a *mutual authentication* setting, we will describe in more detail this case. In a *mutual authentication* setting, both parties are required to prove their identity to the other party, so, each party needs to send its certificate to the other. A digital certificate is the document that proves that a certain party owns the public key contained in it [CSF+08]. Among other fields, a certificate contains information about the public key contained in it, the entity that owns it and a signature of a third-party, which confirms that the information contained in the certificate is valid. In order to be valid, the third-party signature must belong to a trustworthy entity, a Certification Authority (CA). After the client and server receive each others certificates, they verify its validity and the validity of the signature of the CA, aborting the *handshake* in case one of them is not valid. When both server and client confirm that the certificate of the other party and signature of the CA contained in it are valid, they use the certificates public keys and corresponding secret keys to compute the symmetric session keys that will be used to encrypt and decrypt communications and the secret keys used to compute the Message Authentication Codes. The computation depends on the cipher suite agreed by the client and server in the beginning of the *handshake*.

2. *Record protocol*: The *TLS record protocol* is responsible for securing the application data.

It is responsible for encrypting outgoing and decrypting incoming messages and verifying the integrity of incoming and applying the MAC to outgoing messages.

When using the TLS protocol to secure communications between two parties, the aim is to achieve three security properties of the ones defined in section 2.1: *authentication*, *confidentiality* and *data integrity* [Aum17]. The property of *confidentiality* is obtained as result of the communications between the parties being encrypted using symmetric encryption. In the *handshake*, the parties compute a symmetric key used to encrypt and decrypt communications, making them unintelligible to those who do not possess the key. TLS 1.3 introduces a construction named authenticated encryption with associated data (AEAD), which is applied when client and server exchange messages. Each message sent contains a MAC computed from a session key and the message itself. Moreover, the message and appended MAC are subsequently encrypted with a session key. AEAD provides TLS application data with the properties of *authentication*, *confidentiality* and *data integrity*.

2.2 Secure Multiparty Computation

SMC is a cryptographic primitive whose theoretical foundations started in the 1980s, although practical applications only emerged in the last 20 years. As defined by [CDN15], the goal of SMC is to allow a group of parties that do not trust each other to collectively compute a common function, without revealing their inputs to the other parties. A simple solution to this problem would be to find a trusted third party (TTP) that all participants trust: all parties would send their private inputs to the TTP, which would then compute the function of those inputs and deliver the output to the parties. The trusted party would afterwards need to forget the private inputs sent to it. However, this solution requires the parties to completely trust the TTP, thus giving it access to their private inputs. SMC is an alternative to this approach overcoming the requirement of a TTP, while still allowing the parties to exchange information and compute a common function of their inputs, without revealing any information about them to each other. To mathematically define the problem that SMC tries to solve, let us suppose we have n parties, $P_1, ..., P_n$, with $n \geq 2$, whose secret inputs are $x_1, ..., x_n$, respectively. The objective of SMC is to let the parties collectively compute a function of either the type represented in equation (2.1), or in equation (2.2):

$$y = f(x_1, ..., x_n) \tag{2.1}$$

$$(y_1, ..., y_n) = f(x_1, ..., x_n). \quad (2.2)$$

In the end of the computation, in the case of equation (2.1), all parties receive the output y, whereas, in the case of equation (2.2), each party P_i receives the output y_i, for $i = 1, ..., n$. When performing these computations, we want to ensure two conditions:

1. **Privacy of inputs:** In the end of a SMC operation, the only new information that each of the participating parties has learnt is the output assigned to it, and what can be inferred from it. In particular, each party should not be able to learn the other parties inputs nor any intermediate calculations that can lead to the discovery of other parties inputs.

2. **Correctness of the output:** In the end of a SMC operation, the output that each of the participating parties receives is correct.

Performing a computation f such that the properties of *Privacy* of inputs and *Correctness* of the output are guaranteed can be defined as *computing f securely.*

In general, to create a SMC protocol, the function which is represented by equation (2.1) or equation (2.2), is defined as a circuit. In [CDN15], Cramer et al. remark that circuits can be seen as graphs where the vertices correspond to the gates and the edges to the wires connecting those gates. Circuits are divided in two types: boolean circuits and arithmetic circuits. On one hand, in boolean circuits, the gates correspond to logic operations like AND, OR and XOR, and their inputs correspond to bits. On the other hand, in arithmetic circuits, the gates correspond to arithmetic operations like addition, multiplication and scalar multiplication and the inputs are elements of a finite field $\mathbb{F}$.

As described in [EKR18], when creating a SMC protocol, one has to differentiate between the two-party computation (2PC) case, where the number of parties is exactly two, and the multiparty case, where the number of parties can be equal or greater than two. In general, it is not trivial to generalize 2PC protocols for the multiparty case, since the assumptions and techniques which are sufficient to secure the communications between two parties are different than those needed to secure the communications between multiple parties [GS18]. As a result, literature usually either focuses on the development of two-party protocols or multiparty protocols. Furthermore, we need to define the adversaries that may target the protocol and those it is capable to withstand. In subsection 2.2.1 we provide an overview of the adversary model usually considered for SMC protocols. Finally, according to [Orl11], the implementation of a specific *secure* SMC protocol usually relies on either one of these two primitives: Garbled Circuits (GC), which is used to evaluate boolean circuits in the 2PC setting and is described in subsection 2.2.2;

Secret Sharing, which is generally used when computing SMC functions as arithmetic circuits and is described in subsection 2.2.3.

2.2.1 Adversary Model

When creating a SMC protocol we have to consider the type of adversaries that may target it. In [Can00], three main distinctions are made when considering the adversary model of a SMC protocol. The first aspect to consider is the ability of the adversary to deviate from the protocol. We consider two types of adversaries:

1. **Semi-honest adversary**: Only gathers whatever information it can from the corrupted parties. Although the corrupted parties can cooperate to gain more information, they still follow the intended protocol.

2. **Malicious adversary**: The corrupted parties can deviate from the protocol, executing actions that are not intended during the normal behaviour of the protocol.

The second aspect to consider is the ability of the adversary to corrupt parties when the protocol is already being executed. With regard to this ability we can distinguish two types of adversaries:

1. **Static adversary**: Can arbitrarily choose which parties to control before the execution of the protocol. Throughout the protocol, the number of corrupted parties remains fixed.

2. **Adaptive adversary**: Can choose the parties to corrupt while the protocol is being executed. This decision can be aided by information obtained during the protocol.

Finally, when developing a SMC protocol, we can have two different of settings in which information is exchanged:

1. **Secure channel setting**: Despite an adversary having unlimited computing power, the point-to-point communication channels are perfectly secure, making it impossible for an adversary to obtain the parties secret inputs.

2. **Computational setting**: An adversary can capture all communications between the parties, although it cannot obtain the parties secret inputs in a feasible amount of time.

2.2.2 Garbled Circuits

In [Yao82], Andrew Yao introduced an illustrative SMC problem, the *Millionaire's problem.* In this problem, we consider two parties, Alice and Bob, who wish to know who is richer, without revealing to the other party how much money they have. In this work, the author presents three solutions to solve this specific problem, guaranteeing the requirements of *Privacy* of inputs and *Correctness* of the output. Later, Yao proposed GC, a more general solution to approach 2PC problems, in the semi-honest adversary setting, in the context of this work and [Yao86]. In GC, two parties, the Garbler, P_0, and the Evaluator, P_1, want to compute the output y of the function $f(x_0, x_1)$. To illustrate how this protocol works, let us use the example from [Yak17]. Let us consider that we want to evaluate the logic AND function between two input bits, as represented in equation (2.3):

$$y = f(x_0, x_1) = x_0 \wedge x_1. \tag{2.3}$$

Initially, the Garbler assigns random strings as labels for the two possible values (0 and 1) of the inputs x_0 and x_1. The label $W_{x_0}^0$ is assigned to the case where $x_0 = 0$ and the label $W_{x_0}^1$ to the case where $x_0 = 1$. Moreover, the label $W_{x_1}^0$ is assigned to the case where $x_1 = 0$ and the label $W_{x_1}^1$ to the case where $x_1 = 1$. The correspondence between these labels and the input bits x_0 and x_1 is represented in table 2.1.

Table 2.1: AND gate truth table.

x_0	x_1	y
0	0	0
0	1	0
1	0	0
1	1	1

x_0	x_1	y
$W_{x_0}^0$	$W_{x_1}^0$	0
$W_{x_0}^0$	$W_{x_1}^1$	0
$W_{x_0}^1$	$W_{x_1}^0$	0
$W_{x_0}^1$	$W_{x_1}^1$	1

Afterwards, the two input labels of each row are used in a key derivation function, H, to derive a symmetric key which is used to encrypt the output y. The result of this encryption can be observed in table 2.2.

Table 2.2: Encrypted AND gate.

x_0	x_1	**Encrypted output**
$W_{x_0}^0$	$W_{x_1}^0$	$Enc_{H(W_{x_0}^0, W_{x_1}^0)}(0)$
$W_{x_0}^0$	$W_{x_1}^1$	$Enc_{H(W_{x_0}^0, W_{x_1}^1)}(0)$
$W_{x_0}^1$	$W_{x_1}^0$	$Enc_{H(W_{x_0}^1, W_{x_1}^0)}(0)$
$W_{x_0}^1$	$W_{x_1}^1$	$Enc_{H(W_{x_0}^1, W_{x_1}^1)}(1)$

The rows of table 2.2 are subsequently randomly permuted, so that the output cannot be determined from the table row. An example of a possible randomly permuted garbled table is represented in table 2.3.

Table 2.3: Garbled AND gate.

Permuted Garbled table
$Enc_{H(W_{x_0}^1, W_{x_1}^0)}(0)$
$Enc_{H(W_{x_0}^0, W_{x_1}^0)}(0)$
$Enc_{H(W_{x_0}^1, W_{x_1}^1)}(1)$
$Enc_{H(W_{x_0}^0, W_{x_1}^1)}(0)$

After computing the garbled AND gate of table 2.3, the Garbler, P_0, sends it to the Evaluator, P_1. In order to compute the output of equation (2.3), the Evaluator needs to decrypt the row that contains the labels corresponding to the real input values. Consequently, the Garbler needs to send to the Evaluator these values. Since it knows the real bit value b_0 of x_0, the Garbler can send the correspondent label, $W_{x_0}^{b_0}$, which the Evaluator cannot use to find b_0, because the labels are random, independent and equally distributed. Furthermore, the Garbler also needs to send the label $W_{x_1}^{b_1}$ corresponding to the real input bit of x_1, b_1, to the Evaluator. Nevertheless, the Garbler does not know whether $W_{x_1}^{b_1}$ corresponds to $W_{x_1}^0$ or $W_{x_1}^1$, although a simple solution would be for the Garbler to send both labels to the Evaluator. However, this would allow the Evaluator to decrypt two ciphertexts in table 2.3. The solution is for the Garbler and Evaluator to use 1-out-of-2 oblivious transfer [EGL82], so that the Evaluator receives the correct $W_{x_1}^{b_1}$ from the Garbler. After obtaining the labels $W_{x_0}^{b_0}$ and $W_{x_1}^{b_1}$ of the real inputs, b_0 and b_1, the Evaluator decrypts the corresponding ciphertext, obtaining the correct output from table 2.3. It afterwards communicates the correct output to the Garbler.

However, real-life applications of GC to perform 2PC would imply more complex functions, represented by not single gates but a whole circuit. In these cases, the Garbler needs to garble the entire function circuit and, for the gates whose output wire corresponds to the input of another gate, instead of directly encrypting the output bits, as in table 2.2, the Garbler encrypts a label W_y^0 when $y = 0$, and W_y^1 when $y = 1$. As an example, in table 2.4, we can observe the encryption of an AND gate, where its output corresponds to the input of another gate.

The original GC protocol proposed by Yao was highly inefficient to be used in practice. As a result, over the years, several works developed optimizations for this protocol. Kolesnikov et al [KS08] proposed an improvement of GC, *free XOR*, that allows the evaluation of XOR gates, without the need to create the respective garbled tables and to use symmetric key cryptography. In [ZRE15], the authors proposed *Half-gates*, a method that allows to garble AND gates with only

Table 2.4: Encrypted AND gate in a circuit, where its output corresponds to the input of another gate.

x_0	x_1	**Encrypted output**
$W_{x_0}^0$	$W_{x_1}^0$	$Enc_{H(W_{x_0}^0,W_{x_1}^0)}(W_y^0)$
$W_{x_0}^0$	$W_{x_1}^1$	$Enc_{H(W_{x_0}^0,W_{x_1}^1)}(W_y^0)$
$W_{x_0}^1$	$W_{x_1}^0$	$Enc_{H(W_{x_0}^1,W_{x_1}^0)}(W_y^0)$
$W_{x_0}^1$	$W_{x_1}^1$	$Enc_{H(W_{x_0}^1,W_{x_1}^1)}(W_y^1)$

two ciphertexts. Furthermore, this work was developed to support compatibility with the *free XOR* technique. The improvements to GC described in these works, as well as in others, make it one of the most used building blocks when constructing SMC protocols, more specifically, in the 2PC setting and when the function to be computed *securely* is specified as a boolean circuit.

2.2.3 Secret Sharing

Secret sharing is a technique through which a secret is split into pieces which are distributed to several parties. When the pieces are combined, they allow the reconstruction of the secret.

In [Sha79], Adi Shamir proposes a way to divide a piece of data D into n shares, such that D can be recovered by knowing k pieces, where $0 \leq k \leq n$. This technique was designated a (k, n) threshold secret-sharing scheme. In this scheme, the original secret, D, can be recovered by assembling any k shares, whereas the possession of less than k shares reveal no information about the secret. In this work, the author details a secret sharing scheme, afterwards named Shamir's Secret Sharing (SSS), which is based on polynomial interpolation over a finite field. Let us assume we have: a finite field $\mathbb{F}_q$ of size q, where q is a prime and $0 \leq k \leq n < q$; a secret number D, where $D < q$ and $D = a_0$; $k-1$ positive, distinct coefficients, $a_1 + a_2 + ... + a_{k-1}x^{k-1}$, which are randomly chosen from an uniform distribution from the integers in $]0, q[$. We can construct a polynomial of degree $k-1$ that satisfies equation (2.4):

$$p(x) = a_0 + a_1x + a_2x^2 + ... + a_{k-1}x^{k-1}. \tag{2.4}$$

Furthermore, for $i = 1, ..., n$ we can construct n points D_i such that equation (2.5) holds:

$$D_i = p(i) \ mod \ q. \tag{2.5}$$

The n points D_i are the shares of the secret D. The points $(1, D_1), ..., (n, D_n)$ and the value of q are, subsequently, distributed to the participants. The participants can reconstruct the original secret D by combining any k of the n points distributed and using interpolation to obtain the coefficients $a_1, ..., a_{k-1}$ of the polynomial. In the end, the secret D can be reconstructed by computing equation (2.6):

$$D_0 = p(0) \ mod \ q = a_0. \tag{2.6}$$

In SSS, the usage of modular arithmetic instead of real arithmetic ensures that an adversary that possesses up to $k-1$ shares of the secret learns nothing more than one with zero shares.

SSS can be applied to SMC problems due to its homomorphic property [Ben87]: operations, like addition and multiplication, that were to be performed on the secret inputs, can instead be performed on the secrets shares. The output of these operations is equal whether they were performed on the shares of the secrets or the secret itself.

Besides SSS, other authors developed different secret sharing schemes. Blakley [Bla79] developed his own secret sharing scheme in parallel with SSS, focusing on a geometric approach to also create a (k, n) threshold secret sharing scheme. In this work, the secret is a point in a k-dimensional plane, and the shares are the coefficients of the hyperplanes whose intersection is the point that corresponds to the secret. The work of Chor et al. [CGMA85] introduces the notion of Verifiable Secret Sharing, a secret sharing scheme that gives the parties the ability to confirm that every party received a valid share of the secret, without needing to know the secret itself.

The wide application of secret sharing schemes in SMC emerged with the work of Ben-Or et al. [BOGW88]. The authors use a Secret Sharing scheme similar to SSS to compute a SMC function of the type of equation (2.2): n parties perform a *secure* computation, where the $i-th$ party, with secret input x_i, obtains, in the end of the computation, the $i-th$ output y_i. Since this work, Secret Sharing has been greatly adopted in SMC, both in the two-party and multiparty computation settings.

2.2.4 TinyTable Protocol

In [DNNR17][4], Damgård et al. propose TinyTable, a *secure* 2PC protocol where the SMC function is specified as a boolean circuit. This protocol consists of two phases: a *preprocessing* phase, where knowledge of the parties inputs is not required, and an *online* phase, that uses the parties inputs as well as the output of computations performed in the *preprocessing* phase. The usage of public key cryptography primitives is restricted to the *preprocessing* phase, which

[4]For a further explanation on the TinyTable protocol, the presentation of this paper at Crypto 2017 conference can be watched at `https://www.youtube.com/watch?v=K5MUbtHFqH0`.

makes it very slow. Furthermore, this phase can be executed any time prior to the *online* phase since it does not require the inputs of the parties. The *online* phase is where the computations on the inputs of the parties is performed. Due to the usage of public key cryptography being restricted to the *preprocessing* phase, the *online* phase is much faster than the previous one.

With regard to the content of section 2.4.1, we will show the algorithm of the TinyTable protocol for the semi-honest and static adversary setting, where exactly one of the parties can be corrupt. The following assumptions are made:

1. We have a boolean circuit C with gates $G_1, ..., G_N$ and wires $w_1, ..., w_M$.

2. The circuit contains arbitrary gates with two input wires and one output wire.

3. Arbitrary fan-out is allowed: a wire leaving a gate can have an arbitrary number of copies which are all assigned the same wire index.

4. Both parties will learn the circuit evaluation output.

5. The order in which the circuit is evaluated gate by gate is arbitrary and fixed, such that output gates come last. When we are about to evaluate gate i, its inputs have already been computed.

6. The function $G_i(.,.)$ is the result of computation of gate G_i.

The *preprocessing* phase in the TinyTable protocol is based in the *preprocessing* of the TinyOT protocol described in [NNOB12]. In this work, the authors use Oblivious Transfer to implement a 2PC protocol which is secure against a malicious adversary. In the *preprocessing* phase of TinyTable, the two parties, A and B, respectively get a hold of scrambled versions of truth-tables, A_i and B_i, for each gate G_i and uniform random mask bits r_i, which will be required, afterwards, in the *online* phase. The *preprocessing* phase algorithm for semi-honest adversary setting is represented in algorithm 1.

In the *online* phase of the TinyTable protocol, the actual computation involving the parties inputs is performed. In this phase, the parties perform look-ups on the scrambled tables by using the bits masked by the previously chosen uniformly random bits. The *online* phase algorithm for semi-honest adversary setting security is represented in algorithm 2.

The final result holds because the scrambled tables A_i and B_i are set up such that $A_i[e_u, e_v]$ and $B_i[e_u, e_v]$ are an additive secret sharing of e_o: $A_i[b_u \oplus r_u, b_v \oplus r_v] \oplus B_i[b_u \oplus r_u, b_v \oplus r_v] = b_o \oplus r_o$. The algorithm described allows the computation of the circuit C securely assuming a semi-honest adversary.

Algorithm 1: TinyTable protocol *preprocessing* phase.

```
foreach wire w_j do
    Select a random mask bit r_j;
    if w_j is an input wire then
        Give r_j to the player that owns it;
    else if w_j is an output wire then
        Send r_j to both players;
    end
end
foreach gate G_i with input wires w_u and w_v and output wire w_o do
    Construct two tables A_i and B_i, each with 4 entries, indexed by bits (c, d);
    foreach of the four possible values of (c, d) do
        Chose a random bit s_{c,d};
        if both parties are honest or A is corrupt then
            Set A_i[c, d] = s_{c,d};
            Set B_i[c, d] = s_{c,d} ⊕ (r_o ⊕ G_i(c ⊕ r_u, d ⊕ r_v));
        else if B is corrupt then
            Set B_i[c, d] = s_{c,d};
            Set A_i[c, d] = s_{c,d} ⊕ (r_o ⊕ G_i(c ⊕ r_u, d ⊕ r_v));
        end
    end
    Hand A_i to player A;
    Hand B_i to player B;
end
```

In order to evaluate TinyTable, the authors used this protocol to compute a boolean circuit corresponding to the encryption process of AES-128. In this setting, one of the party inputs the plaintext and the other the expanded key[5], such that, in the end of the computation, both parties learn the ciphertext. In a LAN setting consisting of two machines, this computation yielded a total execution time of 3.94 ms.

2.3 Private Set Intersection

In the context of SMC, PSI is a widely researched operation. In this operation, two or more parties, $P_1, ..., P_n | n \geq 2$, with the respective private sets, $S_1, ..., S_n$ want to compute the intersection

[5]As mentioned in subsection 2.1.1, for AES-128, the expanded key is a 1408 bits key which is generated during the *KeySchedule* operation.

Algorithm 2: TinyTable protocol *online* phase.

```
foreach input wire w_j do
    if A holds this wire with input bit b_j then
        Send e_j = r_j ⊕ b_j to B;
    else if B holds this wire with input bit b_j then
        Send e_j = r_j ⊕ b_j to A;
    end
end
for i = 1, ..., N do
    Let G_i have input wires w_u and w_v and output wire w_o (so that e_u and e_v have been
      computed);
    A sends A_i[e_u, e_v] to B;
    B sends B_i[e_u, e_v] to A;
    The parties obtain e_o = A_i[e_u, e_v] ⊕ B_i[e_u, e_v];
    foreach output wire w_o do
        Both parties output the bits b_o = e_o ⊕ r_o = G_i[b_u, b_v]
    end
end
```

of those same sets, as represented in equation (2.7):

$$f(S_1, ..., S_n) = S_1 \cap ... \cap S_n. \tag{2.7}$$

In the end of this computation, either all parties or a subset of them learn the elements which are common to all sets, without anything being disclosed about the different ones. In general, implementations of PSI can be divided into two classes: generic protocols, which specify this operation as a circuit and rely on a general SMC technique to solve it, and custom protocols, which are created for the specific structure of the PSI operation [PSTY19].

We will give examples of three protocols that can be used to implement the PSI operation in the semi-honest adversary setting. Firstly, in subsection 2.3.1, we describe three generic GC protocols that implement PSI. Secondly, in subsection 2.3.2, we give an overview of a 2PC specific PSI protocol based on public key encryption. Finally, we describe a multiparty custom protocol for implementing the PSI operation in subsection 2.3.3

2.3.1 Yao's Garbled Circuits 2PC PSI

In [HEK12], Huang et al. use Yao's GC to implement the 2PC PSI operation. The authors develop three different boolean circuits to compute PSI: *Bitwise-AND*, which consists on performing the AND operation on the bitwise representations of the parties sets; *Pairwise Comparisons*, which performs equality tests for each pair of items in the parties sets; *Sort-Compare-Shuffle*, that starts by sorting the parties sets and saving them into a single list, then performing equality tests on adjacent elements of the list and finally shuffling the resulting list of common elements.

2.3.2 Public Key Encryption 2PC PSI

In [FNP04], the authors propose a protocol for performing 2PC PSI based on public key encryption. In this protocol the authors use a public key encryption scheme with homomorphic properties[6] together with polynomial interpolation to allow the computation of PSI. In this protocol, two parties, client and server, want to compute the intersection of their sets, but while the client learns the intersection of the sets, the server learns nothing. The client defines a polynomial whose roots are the elements of its set. It, subsequently, sends the homomorphic encryptions of the coefficients of the polynomial to the server, which uses the homomorphic properties of the encryption scheme to evaluate the encrypted polynomial at its own inputs. Afterwards, the server multiplies each result by a random number to get an intermediate result, which is then added to the encryption of its own result and then sends this result to the client. For each of the elements in the intersection of the two parties sets, the result of the previous computation is the value of the common element, while for all other values the result is random. The server then sends the resulting intersection ciphertexts to the client, that decrypts all the ciphertexts received and outputs the elements of its set for which there is a corresponding decrypted value. These elements are the ones in the intersection of the parties sets.

2.3.3 Multiparty PSI

In [KMP$^+$17], Kolesnikov et al. present the first implementation of a multiparty PSI protocol. The authors develop an approach which is based on oblivious programmable pseudorandom function (OPPRF). An oblivious pseudorandom function (OPRF) [FIPR05] is a two-party protocol in which the sender learns a pseudorandom function (PRF) key k and the receiver learns $F(k, r)$ where F is a PRF and r is the input of the receiver. In OPPRF, the PRF F allows the sender to choose the output of F on a limited number of inputs. In a high-level overview, the authors protocol is divided into two main stages: *conditional zero-sharing* and *conditional reconstruction*. In *condition zero-sharing* the n parties collectively and securely generate additive sharings of zero, with each party P_i obtaining, for each of its items x_j, a share s_j^i, where equation (2.8) holds:

[6] The Paillier cryptosystem [Pai99] is an example of public key encryption scheme with homomorphic properties

$$\sum_{i=1}^{n} s_j^i = 0. \tag{2.8}$$

If all parties have the element x_j in their sets, the sum of their shares is zero. Otherwise, the sum of the shares is different from zero. In *conditional reconstruction* each party P_i programs an instance of OPPRF to output its share s_j^i when evaluated on x_j. If all parties evaluate the OPPRF on the same value x_j, the sum that the OPPRF outputs is equal to zero. This result indicates that all parties have the element x_j in their sets. If the sum corresponding to the output of the OPPRF is a random value, not all parties have the input x_j in their sets.

2.3.4 PSI Practical Applications

In [HCE11], the authors develop a prototype of privacy-preserving mobile application for Android devices to demonstrate a practical application of 2PC PSI. The application was based on the approach to PSI of Yao's GC, previously described in subsection 2.3.1, more specifically the *Sort-Compare-Shuffle* circuit. The developed application, *CommonContacts*, allows two users running it to discover contacts they have in common while keeping the contacts which are not in common private to each party. For a set size of 256 contacts, the total execution time of *CommonContacts* was of 9,97 minutes.

In [GKF$^+$06] the authors propose *Reliable Email*, *RE:*, a whitelisting system to lower the rate of false positives in email spam detection. In particular, one characteristic of this system is that it uses a *friend-of-friend* query to allow the recipient of an email to determine if one of its friends has attested to the email sender. To achieve this goal, *RE:* relies on the two-party public key based PSI protocol described in subsection 2.3.2. In this context, the recipient of the email acts as the client and the sender as the server. Both sender and receiver of the email have as input a set of friends and in the end of the operation, the recipient learns the intersection of the two sets of friends while the sender learns nothing. The friends in the intersection correspond to the friends of the recipient who have attested to the sender. From a set of 20 million corporate emails which did not contain spam messages, normal email spam detection flagged 172 emails as being spam (false positives). The system developed by the authors, *RE:*, would have whitelisted 84% of these false positives, thus reducing the number of false positives to 28 emails.

2.4 SMC Frameworks

In this section, we describe three SMC frameworks implementing the PSI operation which was required for our project. Although a more complete list of SMC frameworks has been compiled, we will provide an overview of the three most promising frameworks that implement PSI: FRESCO, ABY and Swanky.

2.4.1 FRESCO

FRESCO[7] is the FRamework for Efficient Secure COmputation. It is an open-source project developed and maintained by the Alexandra Institute in Denmark, an organization whose main focus is IT research. The FRESCO framework is developed in Java and supports computations in both the two-party and multiparty computation setting. FRESCO is aimed to allow an easy integration with applications, providing four demonstrations (demos) of secure computations which are common in SMC: AES Encryption, Sum of Parties Inputs, Distance between Two Parties Points and PSI. More specifically, the PSI demo provided by FRESCO uses the TinyTable protocol, which was described in subsection 2.2.4. In this 2PC demo, each party inputs several values: its own identifier (id), IP address and port, the id, IP address and port of the other party, the protocol to be used[8], the input integer set and the AES-128 secret key. The parties then use the TinyTable protocol to perform the XOR of their secret keys and afterwards use it to compute the AES-128 encryption of their inputs. In the end of the computation both parties obtain the AES-128 encryption of a concatenated list of their inputs. In this list, the first half of the elements corresponds to the encryption of the input set of the first party and the second half of the elements corresponds to the encryption of the set of the second party. The parties are afterwards responsible for identifying the intersecting elements, by looking which encryption of elements in the first half of the concatenated list match the encryption of elements in the second half. Despite this demo being called PSI, it does not correspond to the definition of the PSI operation provided in section 2.3, since the elements in the intersection are not a direct output of the 2PC computation. It is still necessary to perform a comparison of the elements in each half.

In figure 2.2, we can observe the result of running the FRESCO PSI demo with the inputs provided by the developers. From figure 2.2 we can see that the input set of party 1 contains the integers 2, 3, 4, 5, 8, 9 and 14, while the input set of party 2 contains the integers 2, 3, 4, 6, 7, 12, 14. The intersecting elements of these two sets are the integers 2, 3, 4 and 14. In fact, in the end of the FRESCO PSI computation, both parties obtain the concatenated list of encrypted input sets, where the AES-128 encryption of the items in indexes 0 and 7, 1 and 8, 2 and 9 and 6 and 13 match. As a result, both parties know that they have the elements 2 (indexes 0 and

[7]The source code of FRESCO can be found at `https://github.com/aicis/fresco` and its documentation at `https://fresco.readthedocs.io/en/latest/`.

[8]In FRESCO, TinyTable is subdivided into two phases: its *preprocessing* phase, `tinytablesprepro`, and *online* phase, `tinytables`.

```
option: e : SEQUENTIAL_BATCHED
option: i : 1
option: l : null
option: p : 1:localhost:8081
option: p : 2:localhost:8082
option: s : tinytables
option: in : 2,3,4,5,8,9,14
option: key : abc123abc123abc123abc123abc123ab

(...)

The resulting ciphertexts are:
result(0): 0388dace60b6a392f328c2b971b2fe78
result(1): f795aaab494b5923f7fd89ff948bc1e0
result(2): 200211214e7394da2089b6acd093abe0
result(3): c94da219118e297d7b7ebcbcc9c388f2
result(4): 0253786e126504f0dab90c48a30321de
result(5): 3345e6b0461e7c9e6c6b7afedde83f40
result(6): 6002496db63fa4b91bee387fa3030c95
result(7): 0388dace60b6a392f328c2b971b2fe78
result(8): f795aaab494b5923f7fd89ff948bc1e0
result(9): 200211214e7394da2089b6acd093abe0
result(10): 8ade7d85a8ee35616f7124a9d5270291
result(11): 95b84d1b96c690ff2f2de30bf2ec89e0
result(12): 5d11452b58ac50aa2eb3a195b61b87e5
result(13): 6002496db63fa4b91bee387fa3030c95
```

(a) Output of the PSI demo for party 1.

```
option: e : SEQUENTIAL_BATCHED
option: i : 2
option: l : null
option: p : 1:localhost:8081
option: p : 2:localhost:8082
option: s : tinytables
option: in : 2,3,4,6,7,12,14
option: key : abc123abc123abc123abc123abc123ab

(...)

The resulting ciphertexts are:
result(0): 0388dace60b6a392f328c2b971b2fe78
result(1): f795aaab494b5923f7fd89ff948bc1e0
result(2): 200211214e7394da2089b6acd093abe0
result(3): c94da219118e297d7b7ebcbcc9c388f2
result(4): 0253786e126504f0dab90c48a30321de
result(5): 3345e6b0461e7c9e6c6b7afedde83f40
result(6): 6002496db63fa4b91bee387fa3030c95
result(7): 0388dace60b6a392f328c2b971b2fe78
result(8): f795aaab494b5923f7fd89ff948bc1e0
result(9): 200211214e7394da2089b6acd093abe0
result(10): 8ade7d85a8ee35616f7124a9d5270291
result(11): 95b84d1b96c690ff2f2de30bf2ec89e0
result(12): 5d11452b58ac50aa2eb3a195b61b87e5
result(13): 6002496db63fa4b91bee387fa3030c95
```

(b) Output of the PSI demo for party 2.

Figure 2.2: FRESCO PSI demo with example input sets.

7), 3 (indexes 1 and 8), 4 (indexes 2 and 9) and 14 (indexes 6 and 13) in common with the other party, since the first half of the result (indexes 0 to 6) corresponds to the encryption of the input set of the first party while the second half (indexes 7 to 13) corresponds to the encryption of the input set of the second party.

2.4.2 ABY

ABY[9] [DSZ15] is an open-source framework developed by Daniel Demmler, Thomas Schneider and Michael Zohner from the ENCRYPTO group, in the TU Darmstadt University in Germany. This framework is developed in C++ and only supports operations in the 2PC setting. It uses and combines secret sharing schemes and Yao's GC to implement a variety of SMC operations like the *Millionaire's problem*, AES Encryption, Euclidean Distance between Two points and PSI. The PSI example provided by ABY uses the implementation of the *Sort-Compare-Shuffle* circuit, which was briefly described in subsection 2.3.1.

2.4.3 Swanky

Swanky is an open-source suite of libraries for SMC operations developed by Galois, an IT research and consulting company in the United States of America. Swanky is developed in Rust and provides SMC operations in both the two and multiparty computation settings. It is

[9] The source code of ABY can be found at `https://github.com/encryptogroup/ABY` and its documentation at `http://encryptogroup.github.io/ABY/docs/index.html`.

divided in four libraries: *fancy-garbling*, which implements boolean and and arithmetic circuits; *scuttlebutt*, which implements basic SMC primitives like GC; *ocelot*, which implements OT and OPRF protocols; *popsicle*, which implements PSI protocols. The PSI library, *popsicle*[10], contains PSI implementations of three different protocols in the semi-honest adversary setting: the 2PC PSI protocol based on OPRF described in [PSZ18] the two-party PSI protocol based on OPPRF presented in [PSTY19] and the multiparty PSI protocol described in subsection 2.3.3.

2.5 Summary

In this chapter we provided an overview of cryptography, described the SMC cryptographic primitive, defined the PSI operation and, finally, introduced three SMC frameworks that implement the PSI operation. All the topics discussed are relevant for the implementation of PriVeil Circle since a variety of the constructions and protocols described were required in the development of our prototype.

[10]The source code of *Popsicle* can be found at `https://github.com/GaloisInc/swanky/tree/master/popsicle` and its documentation at `https://galoisinc.github.io/swanky/popsicle`.

Chapter 3

PriVeil Circle

In this chapter we describe the design and implementation details of *Circle*, the second phase of the privacy-preserving threat information sharing platform *PriVeil*. This chapter is divided in the following sections: in section 3.1 we provide an overview of the *PriVeil* cyber threat sharing system; in section 3.2 we describe the functional requirements intended for *Circle*; in section 3.3 we explain the *Circle* component; in section 3.4 we give an overview of the technologies used to implement *Circle*; in section 3.5 we provide a description of the main functionalities of the *Dealer* and its implementation details; in section 3.6 we detail the *Players* functionalities and implementation details; in section 3.7, we conclude this chapter by showing an example of a *Circle* session.

3.1 *PriVeil* Overview

In *PriVeil*, the information which is shared to allow cooperation when dealing with cyber threats is contained in security reports. Security reports are generated by organizations when they identify a cyber threat and contain information about the threat: a small description of it, its severity, the data when it was detected and tags which contain keywords associated with the threat.

PriVeil is structured in two phases, supported by two components: *Square* and *Circle*. Figure 3.1 represents the architecture of the *PriVeil* platform. In the first phase of *PriVeil*, users start by submitting their encrypted security reports describing a cyber threat to the *Square* component (step 1 in figure 3.1). The system, subsequently, matches the encrypted security reports that describe similar cyber threats (2) and notifies the users whose reports matched (3). This is possible because *Square* uses a Homomorphic Encryption cryptographic scheme to allow computations to be performed on ciphered events. Afterwards, these participants receive a token which authorizes them to engage in a *Circle* session and are redirected to it (4).

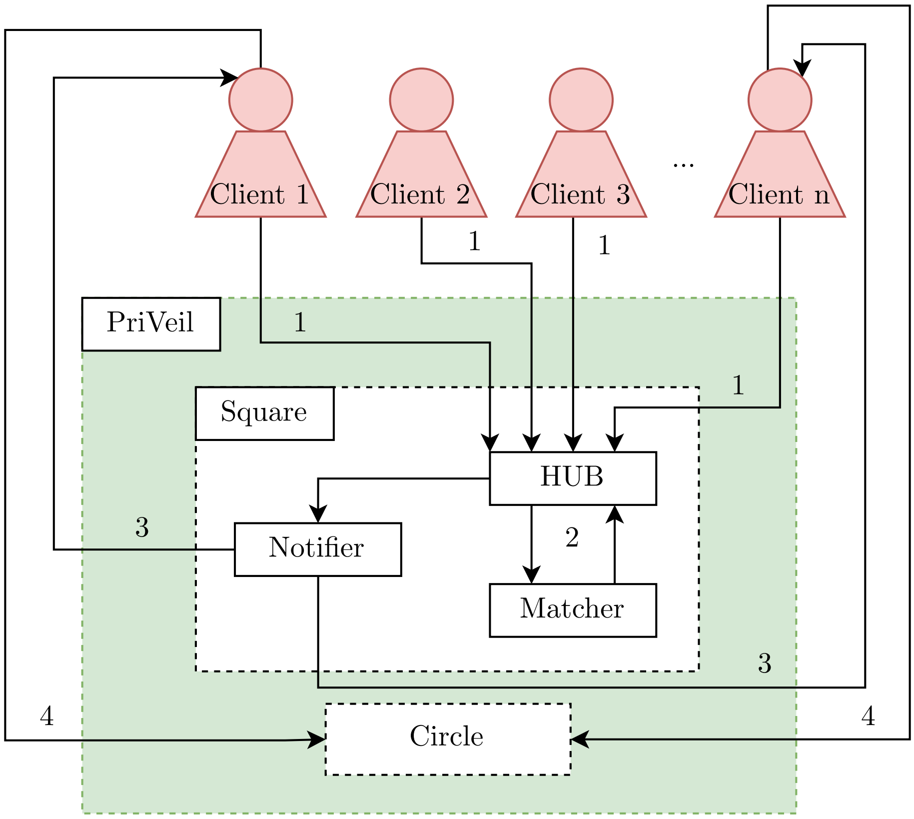


Figure 3.1: *PriVeil* Architecture, adapted from Gonçalves [Gon19].

In *Circle*, the participants can progressively perform a controlled disclosure of information about the cyber threats described in the security reports.

3.2 Requirements

In order to develop the second phase of *PriVeil* we determined that we required two types of parties in *Circle*: the *Dealer* and the *Players*. The *Dealer* is the coordinator of a *Circle* session and acts as a communications facilitator, asking the *Players* for necessary information as the session advances. Although it puts the *Players* in touch with each other to share threat data, the *Dealer* does not have access to the cyber threat information that is being shared. The *Players* represent organizations willing to share cyber threat information. They were forwarded to *Circle* by *Square* and act as the clients of the *Circle* session. They send to the *Dealer* the information it requires and communicate with other *Players* in specific communication rounds, to share cyber threat data.

We now present the functional requirements for *Circle*, the Attacker Model considered and the security requirements.

3.2.1 Functional Requirements

After discussing the functionalities that we desired *Circle* to have and how it would fit the overall *PriVeil* project, we defined the following functional requirements:

- FR_1: *Players* can at any time leave the *Circle* session they are participating in, thus retaining full control over what cyber threat information is shared, and how it is shared.
- FR_2: Only parties that received a token from *Square* to participate in the *Circle* session are authorized to participate in it.
- FR_3: Allow *Players* to share cyber threat data contained in their security reports, in a privacy-preserving manner.

3.2.2 Attacker Model

Due to the fact that our system is a part of a cyber threat information sharing platform, we need to prevent leakage of data, as well as an unauthorized access to it. To fulfill these goals we had to define the attackers that could possibly target *Circle* and those that the system would be able to withstand:

1. **Eavesdropper**: Passively listens to all communications.
2. ***Player* Impersonator**: Impersonates a *Player* of *Circle*.
3. ***Dealer* Impersonator**: Impersonates the *Dealer* of *Circle*.
4. **Tamperer**: Modifies the data exchanged between source and destination.

Firstly, an attacker can gain eavesdropping capabilities by positioning itself anywhere in the route between the data source and destination, reading all the information exchanged. Moreover, on one hand, an attacker becomes a *Player* Impersonator if it can pose as a *Player*, deceiving other *Players* and the *Dealer* into thinking they are communicating with a legitimate *Player*. On the other hand, an attacker becomes a *Dealer* Impersonator if it can trick the *Players* into sending it information, as they would in the case of the legitimate *Dealer*. Finally, a Tamperer is the type of attacker that can position itself anywhere in the route between source and destination, modifying the data that is exchanged or even preventing it from reaching its destination.

In *PriVeil Circle* we had to consider two distinct types of communication to which the attacker model above defined could be applied to:

- C_{PD}: Bidirectional *Player* to *Dealer* communication.

- C_{PP}: Bidirectional *Player* to *Player* communication.

This distinction is necessary because the attackers involved are different for these two types of communication. In the case of C_{PD}, attackers of all types can target the system, since this is a *Player* to *Dealer* communication, where both the *Player* and the *Dealer* can be impersonated, an Eavesdropper can listen to communications between them, and a Tamperer can alter data in-transit. However, when we consider a communication of type C_{PP}, the *Dealer* is not involved in it, and, as a result, a *Dealer* Impersonator is not relevant in this situation. Nevertheless, all the remaining attackers are still viable, for the same reasons described for C_{PD}.

3.2.3 Security Requirements

We defined two security requirements for *Circle* based on the two types of communications considered and the attackers that could target them:

- SR_1: An attacker with eavesdropping, tampering and *Player* and *Dealer* impersonating capabilities is unsuccessful when targeting a communication of type C_{PD}.
- SR_2: An attacker with eavesdropping, tampering and *Player* impersonating capabilities is unsuccessful when targeting a communication of type C_{PP}.

3.3 *Circle* Prototype

When designing our prototype we assumed that each *Player* had already described a cyber threat in a security report with the same format as the example illustrated in figure 3.2. As we can observe, each report is divided in five primary entities: `summary`, `additionalinformation`, `metrics`, `description` and another `summary`. All these, with the exception of the first `summary` accept text as values. The first `summary` is further divided in `title`, `severity`, `tlp`, `pap`, `assignee`, `date`, `tags` and `closedate`. Among these entities, all fields with the exception of `tags` entity accept text as values. The `tags` entity is divided in an arbitrary number of `tag` fields which accept text as their values.

Our main focus in this work was to allow *Players* to share the tags contained in security reports, while guaranteeing the properties of *Privacy* of Inputs and *Correctness* of the output[1]. Since the tags contain keywords which can be Indicators of Compromise, like known malware hash signatures, malicious IP addresses, e-mail addresses, phone numbers or even Common vulnerability and Exposures (CVE) entries, it is important to share this information in the context of a cyber threat information sharing platform.

[1] These two security properties were formally defined in section 2.2

```xml
<?xml version="1.0" encoding="UTF-8"?>
<report>
    <summary>
        <title>Example report</title>
        <severity>M</severity>
        <tlp>TLP:GREEN</tlp>
        <pap>PAP:GREEN</pap>
        <assignee>IST</assignee>
        <date>Wed,Sep 9th,2020 10:00 +01:00</date>
        <tags>
            <tag>CVE-2010-1219</tag>
            <tag>CVE-2020-13955</tag>
            <tag>10.10.10.10</tag>
        </tags>
        <closedate>Thu,Sep 10th,2020 10:00 +00:00</closedate>
    </summary>
    <additionalinformation>This is an example Securiy report</additionalinformation>
    <metrics>severity</metrics>
    <description>This security report illustrates how a security report can be described in XML format</description>
    <summary>Example XML description of security report</summary>
</report>

```

Figure 3.2: Example of a security report describing a cyber threat.

During the development of our work we used a dataset consisting of images of real cyber threat security reports from the monitoring systems of NAV. NAV[2] is the portuguese company which is responsible for monitoring the air traffic control in Portugal. This company provided us the cyber threat security reports, which we first had to strip of the personal data contained in them, since we did not want to unwillingly leak any personal information during the development of our prototype.

3.3.1 *Circle* Private Set Intersection

In *Circle* we wanted to allow *Players* (which represent organizations) to share cyber threat information contained in their security reports. However, as described in chapter 1, organizations may not be willing to share information that may contain sensitive indicators. Furthermore, anonymization of these indicators is not a sufficient solution, because it reduces the shared data utility and may be susceptible to attacks. These reasons encouraged us to search for a solution that allows a computation to be performed *securely* on cyber threat data, despite the fact that it might contain sensitive information.

The solution we found to guarantee a cyber threat sharing environment between parties who do not trust each other, yet wish to share data that might contain sensitive IoC, was to resort to the Private Set Intersection (PSI) operation, which was described in section 2.3. PSI allows distrusting parties to *securely* compute the intersection of their input sets, without any information being revealed about the non-intersecting elements. By relying on this Secure Multiparty Computation (SMC) operation, we decided to design *PriVeil Circle* to allow *Players* to compute the PSI operation on the sets of tags in their security reports. In the end of this computation, the tags which are equal in the security reports are revealed, without any

[2]The web page of NAV can be visited at `https://www.nav.pt/en/nav-portugal-newhp_en`.

information being leaked about the ones that are different. Since the tags that are not in the intersection are different, they are not relevant in finding similar cyber threats among parties and, because they might contain sensitive IoC, we concluded that they needed to remain private to their respective owners.

Our system is meant to have more than two *Players* in each session, so we could have relied on a multiparty PSI solution. However, the output of this solution would be the intersection of the sets of all parties. If, for example, nine parties had a common element in their set, but the tenth party did not have it, the output of PSI of these sets would not contain this element. By applying this example to the sets of tags in *Players* security reports, only tags which were equal in all *Players* security reports would be in the intersection. This is not our objective for *Circle*, considering that, as an example, two *Players* may contain several tags in the intersection of their sets, which are not equal to those in the remaining *Players* reports. These two parties should still be able to know that they have equal tags in their security reports and, because of that, they might be affected by a similar cyber threat and, consequently, they would benefit from additional cyber threat information sharing. As a result, we relied on a two-party computation (2PC) PSI approach, which allows each *Player* to compute PSI with all the remaining *Players* in *Circle*, over the course of a determined number of one-on-one PSI computations.

Besides the fact that equal tags in the security reports might already indicate the same or a similar cyber threat, the final goal of performing the PSI operation between all *Players* in a *Circle* session is that, once this phase is terminated, each *Player* can make a better informed decision whether it wants to share more information about the cyber threat described in its security report. For instance, if during a *Circle* instance, a *Player* found that it had an acceptable number of tags in its security report equal to the tags in the security report of other *Player*, those two reports could be describing the same or a similar cyber threat. Therefore, these *Players* could, afterwards, make a more informed decision whether they wanted to share more information about the threat.

3.3.2 *Circle* Operation Overview

In figure 3.3 we can observe the flowchart for a *Circle* session, where on the left of the dotted line we have the functionalities executed by the *Dealer* and on the right the functionalities executed by each *Player*. Initially, the *Dealer* listens for *Players* looking to connect to *Circle*, through the `connectionEstablishment` function call. During a certain timeframe, *Players* that were authorized by *Square* can join *Circle* by communicating with the *Dealer*. The end of this timeframe is marked by the value of `joincircletimeout`, while the minimum number of *Players*

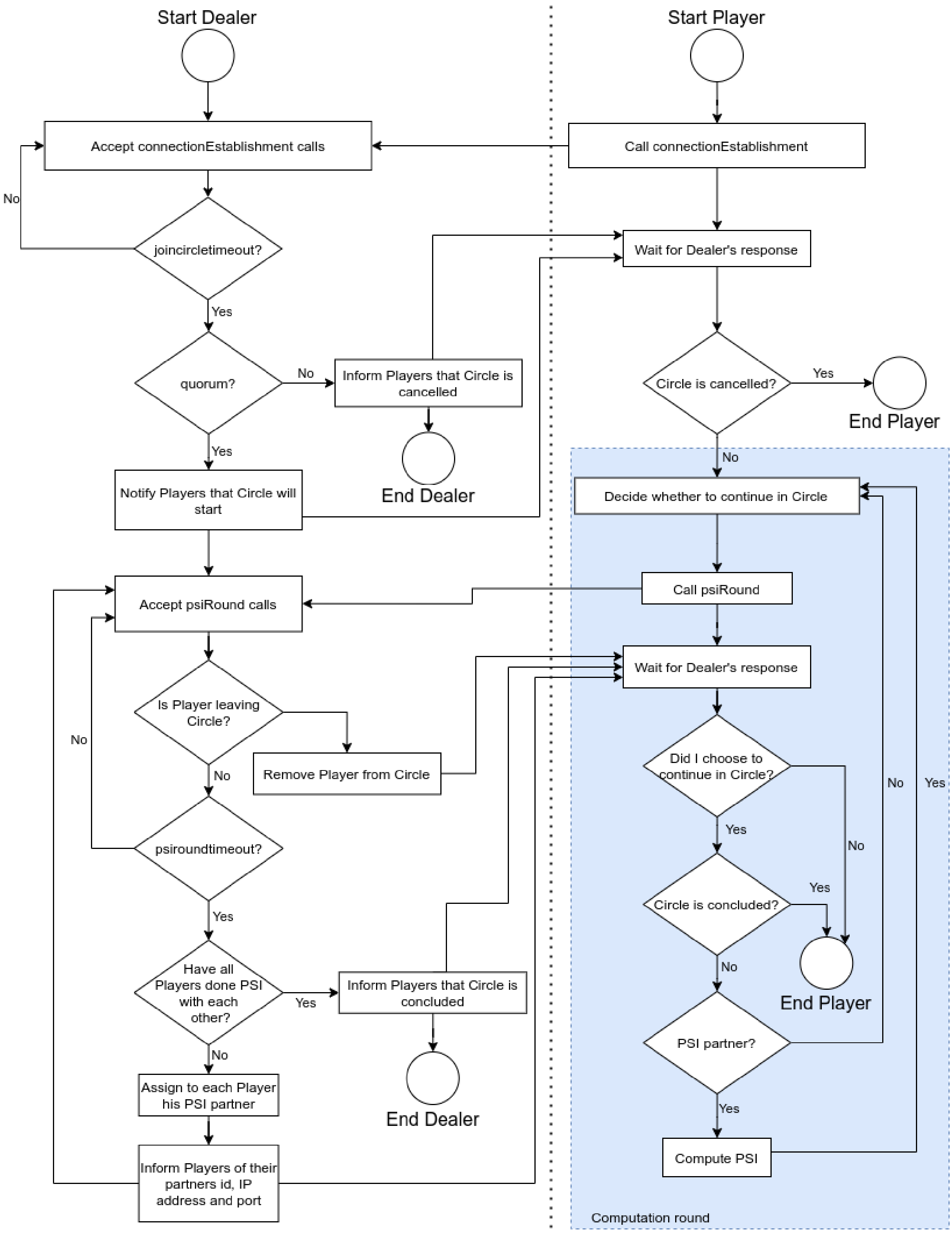


Figure 3.3: Flowchart of a *Circle* session.

required for a *Circle* session to start is the `quorum`. When the `joincircletimeout` occurs and less *Players* than the `quorum` have connected, the *Circle* session is cancelled, the *Dealer* informs the *Players* and then terminates. Otherwise, if the `quorum` is achieved, the *Circle* session can start, since the minimum number of required *Players* to engage in this cyber threat information sharing system was assembled.

For the next step in the session, we defined an abstract event for each *Player*, the *Computation round*, which occurs several times in the same *Circle* session. The number of *Computation rounds* that each *Player* performs depends not only on the number of *Players* in *Circle* but also on the algorithm that assigns the pairs of *Players* that will perform PSI in each *Computation round*, and is described in subsection 3.5.3. When one of its *Computation rounds* starts, each *Player* has the option to leave the *Circle*, giving them full control over what security reports data is shared. If they choose to continue in *Circle*, they make a call to the `psiRound` function of the *Dealer* and receive the identifier (id) and address of a *Player*, with whom they are to perform the two-party PSI operation with, in that *Computation round*. Nevertheless, a *Player* may not have a partner to compute PSI with in certain *Computation rounds*. This situation arises if all other *Players* that have not yet performed the PSI operation with it have already been assigned as PSI partners to other *Players* in this *Computation round*, or if this *Player* has already executed the PSI computation with all other *Players*. In this case, the *Dealer* sends it a message indicating that it will not have a partner for this round. Each *Player* repeats this *Computation round* process until it decides to leave the *Circle* session, the *Dealer* cancels the session because there are less than two remaining *Players* connected or *Circle* is concluded, because there are no more PSI operations to be performed.

3.4 Technical Architecture

The architecture of the *Circle* entities, the *Dealer* and the *Player*, is represented side-by-side in figure 3.4.

3.4.1 Platform

The development of both *Circle* parties was done with Java, because it is a mature programming language, it has an extensive online documentation and it has a large collection of libraries and modules available. In addition, we used the Maven tool to manage the program builds of both the *Dealer* and the *Players*, as it provides an easy way to manage the dependencies of a project and configure individual phases of the build process, by using plugins.

To describe the *Circle* session configuration file and the *Players* security reports we used Extensible Markup Language (XML). We chose this markup language because it provides a format that is both human-readable, to allow its easy edition by humans, and machine-readable, to allow its easier processing by machines. Moreover, as security reports are written in text and

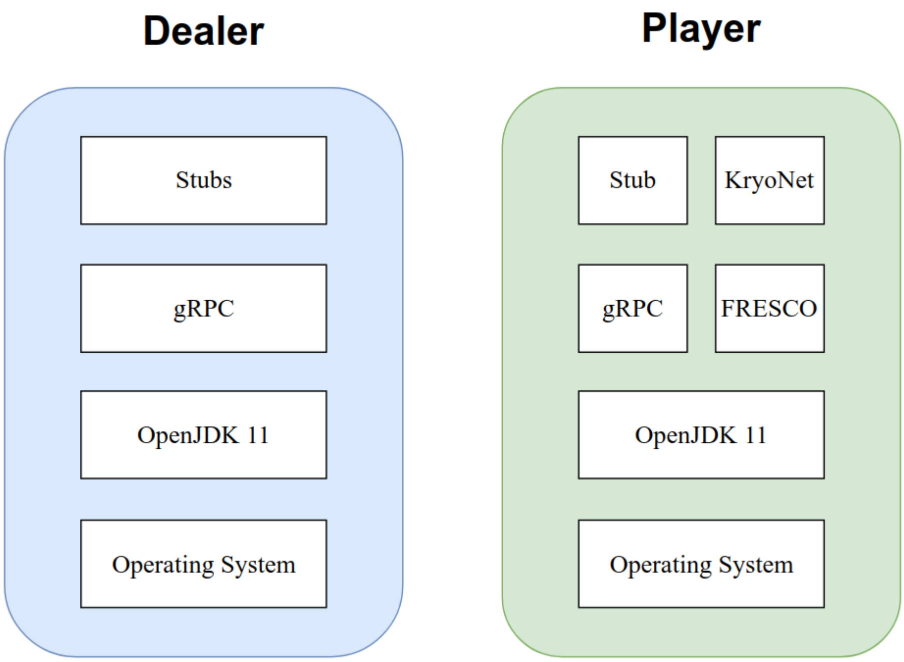


Figure 3.4: *Dealer* and *Player* architectures.

XML is a text-based format, it is fairly trivial to translate security reports into XML format and back from XML to text.

3.4.2 Remote Communication

To implement the functions `connectionEstablishment` and `psiRound`, called by the *Players* when communicating with the *Dealer*, we relied on gRPC, a Remote Procedure Call (RPC) framework which has support for Java, it supports synchronous and asynchronous calls and it is open-source and free to use. Furthermore, we used protocol buffers (Protobuf), developed by Google, as a serializing mechanism for these communications because it is the default serializing mechanism for gRPC, it is also open-source and provides a very efficient binary encoding format. Protobuf requires defining the structure of the data to be serialized in a *proto* file, which corresponds to a message definition language expressed in a text file with a .proto extension. Protocol buffer data is structured as messages, where each one is a small logical record of information containing fields in name-value pairs. To define gRPC services in a *proto* file, we specify a service in the file, with RPC method parameters and return types defined as protocol buffer messages. The gRPC server and client interfaces, for the *Dealer* and *Player*, respectively, were generated from the proto file by using the protocol buffer compiler, `protoc`. Since Maven was used to compile and execute the *Dealer* and *Player* codes, we used a plugin to generate these interfaces. The *Circle.proto* file that we used to specify the RPC services available in *Circle* is represented in figure 3.5. From figure 3.5 we can observe that the two RPC func-

```
// Protocol Buffers definition for PriVeil Circle

syntax = "proto3";
package priveil.grpc;

// Client connect request payload
message ConnectRequest {
  string message = 1;
  int32 code = 2;
  string address = 3;
}

// Server response to client connection request
message ConnectResponse {
  string message = 1;
  int32 code = 2;
  int32 id = 3;
}

message PsiRequest {
  string message = 1;
  int32 code = 2;
  int32 id = 3;
  string address = 4;
}

message PsiResponse {
  string message = 1;
  int32 code = 2;
  int32 id = 3;
  string address = 4;
}

// Defining a Service, a Service can have multiple RPC operations
service CircleService {
  // Define a RPC operation
  rpc connectionEstablishment(ConnectRequest) returns (stream ConnectResponse) {}
  rpc psiRound(PsiRequest) returns (PsiResponse) {}
}
```

Figure 3.5: Protocol buffers definition for the *Dealer* (*proto* file)

tions `connectionEstablishment` and `psiRound`, implemented by the *Dealer* and called by the *Players*, were defined in the `CircleService` service. In all defined messages, `ConnectRequest`, `ConnectResponse`, `PsiRequest` and `PsiResponse`, we have two common fields: the message and the code. The message and code are used in all four messages and are a way for the *Dealer* and the *Players* to decide what their next actions will be. For example, when the `quorum` is not achieved and the *Circle* session is cancelled, the *Dealer* sends the `ConnectResponse` with this information contained in the message and code fields, such that, when each *Player* receive this `ConnectResponse`, it knows that the *Circle* was cancelled because the `quorum` was not achieved. Another example is when a *Player* decides to leave *Circle* at the beginning of its *Computation round*. In this case, when this *Player* makes a call to `psiRound`, it sends this information in the message and code fields, such that, when the *Dealer* receives it, it removes the *Player* from *Circle*. The address field sent by the *Player* to the *Dealer*, in `ConnectRequest` and `PsiRequest`, contains the IP address and port it will use to compute PSI with the other *Players* during the *Computation rounds*, whereas the address sent by the *Dealer* to the *Players*, in `PsiResponse`, contains the IP address and port of their PSI partners for that *Computation round*, in case there is one available. Finally, the id field in `ConnectResponse`, `PsiRequest` and `PsiResponse` messages contains the *Circle* identifier (id) which was assigned by the *Dealer* to the current *Player* that is making a call to `connectionEstablishment` or `psiRound`.

3.4.3 SMC Framework

As previously explained in subsection 3.3.1, we used the 2PC PSI operation to allow the *Players* to *securely* compute the common tags in their cyber threat security reports. However, in order to *securely* implement this SMC operation, we did not possess the necessary theoretical background to design and develop our own PSI protocol. As a result, we concluded that we would need a SMC framework that provided a *secure* implementation of PSI and, thus, we extensively researched frameworks to find those that fulfilled this requirement. The three most promising frameworks we discovered were: ABY, Swanky and FRESCO. As described in section 2.4, all three of them provide implementations of PSI but rely on different protocols and techniques to achieve it.

The ABY framework relies on the *Sort-Compare-Shuffle* circuit protocol to implement 2PC PSI based on Garbled Circuits. However, the authors themselves state that this work is an experimental project, that should not be used in real-world applications, and with no guarantees that its SMC operations provide the desired properties of *Privacy* of inputs and *Correctness* of the output.

Swanky provides implementations of two different state-of-the-art 2PC PSI protocols based on OPRF functions, through its *popsicle* library. Nevertheless, the current version of this framework is considered unstable by the authors and its documentation is very limited, which makes it very hard to integrate with other applications.

At last, FRESCO is a well-known framework that has been used in real-life projects, it has an extensive documentation and has great interoperability. FRESCO is the only one of the above frameworks that does not use a 2PC PSI protocol to implement its PSI demo. It relies on the TinyTable protocol, described in subsection 2.2.4, to provide to both parties, in the end of the operation, AES-128 encryptions of both input sets. The two parties are, subsequently, responsible for computing the intersecting elements themselves, by comparing both input sets encryptions and finding those that are equal. This computation is still *secure* since the parties cannot decrypt the AES-128 encryption of the output because they do not possess the secret key and, as discussed in subsection 2.1.1, AES-128 has no known attacks that can break the protocol and allow to recover the plaintext from the ciphertext without the secret key.

Due to the pros and cons we presented above for each framework, we chose FRESCO as the framework to implement 2PC PSI in *Circle*. FRESCO is written in Java, which allows an easier integration with the *Players* programs and, to allow communication between parties during computations, it uses KryoNet as its default network communication supplier. KryoNet is a Java library that provides an API for both TCP and UDP network communication.

3.5 *Dealer* Implementation

In this section we describe the implementation of the *Dealer* of *Circle*. The *Dealer* is the entity that organizes a *Circle* session and communicates with the *Players*, asking them for the required information for the progression of the session. It takes as input to its program a XML configuration file, *configuration.xml*, which is constructed with the results of a security reports match that occurred in *PriVeil Square*. This file contains parameters like the maximum number of *Players* allowed in the *Circle* session, the minimum number of *Players* required for it to start and the timeouts to join *Circle*. An example of a configuration file that was used in a *Circle* session is shown in figure 3.6.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <quorum>3</quorum>
    <timeouts>
        <joincircletimeout>120</joincircletimeout>
        <psiroundtimeout>300</psiroundtimeout>
    </timeouts>
    <maxnumberofplayers>3</maxnumberofplayers>
    <address>
        <ip>127.0.0.1</ip>
        <port>9999</port>
    </address>
</configuration>
```

Figure 3.6: Example of *Dealer* *configuration.xml* file.

In the file of figure 3.6, the `quorum` represents the minimum number of *Players* required to start the *Circle* session, otherwise it will not occur. The value of this variable is always bigger than two, since we need at least two *Players* in order to create a cyber threat sharing environment. The entities `joincircletimeout` and `psiroundtimeout` are the values of the timeouts for the *Players* to join a *Circle* session and answer to the *Dealer* after making the `psiRound` RPC call, respectively. The `maxnumberofplayers` represents the maximum number of *Players* allowed in the *Circle* session. This value is equal to the number of users whose security reports matched in the *Square* and, consequently, were offered access to the *Circle* session. It is important to notice that a *Circle* session can occur when the number of *Players* is equal or greater than the `quorum`, although it only allows up to a number of *Players* equal to `maxnumberofplayers`. At last, the values of the *ip* and *port* entities correspond to the IP address and port where the *Dealer* will be running the remote functions `connectionEstablishment` and `psiRound`, which will be called by the *Players* during the session.

As described in section 3.4, we implemented the *Dealer* in Java and used gRPC to allow the communication between the *Dealer* and the *Players*. When the server interface is generated from the *Circle.proto* file by the protobuf compiler, `protoc`, called by the Maven tool, a base class for `CircleService` is offered, the `CircleServiceGrpc.CircleServiceImplBase` class. This class contains the two RPC functions defined in the `CircleService` service, `connectionEstablishment` and `psiRound`. As required by gRPC, we created the class `CircleServiceImpl` to extend the `CircleServiceGrpc.CircleServiceImplBase` class and override its two methods. The calls to the `connectionEstablishment` and `psiRound` functions are concurrent, as different *Players* can simultaneously make a call to the same function. As a result, to avoid the concurrent modification of an object shared by different threads, we resorted to the `synchronized` keyword, provided by Java, to create `synchronized blocks` and `synchronized methods`. These were mainly used when threads were modifying the *Players* records and pairs of *Players* which are assigned to perform the PSI operation, during the `psiRound` function call. In these cases, we did not want other threads to modify the same variables, at the same time, since race conditions would inevitably occur. Moreover, to allow simple coordination between the threads handling the RPC calls of each *Player*, we used the `Java.lang.Object.wait()` and `Java.lang.Object.notifyAll()` methods. In the remainder of the document, we will simply refer to these two methods as `wait()` and `notifyAll()`.

In the next subsections we will provide a more detailed explanation of the important functions performed by the *Dealer*. In subsection 3.5.1 we will describe the function `connectionEstablishment`; in subsection 3.5.2 we will explain the execution of the `psiRound` function; in subsection 3.5.3 we will describe how we designed and developed the algorithm that assigned the pairs of *Players* that would perform 2PC PSI in their *Computation rounds*; finally, in subsection 3.5.4 we detail how we implemented the TLS protocol to secure communications between the *Dealer* and the *Players*.

3.5.1 Connection Establishment

The `connectionEstablishment` method implemented by the *Dealer* was defined in the *Circle.proto* file, as represented in figure 3.5. This RPC function uses the `stream` keyword to enable a server-side streaming RPC, as the client sends a request to the server, the `ConnectRequest` message, and gets a stream to read a sequence of `ConnectResponse` messages back. In figure 3.7 we can observe the flowchart of the call of a *Player* to the `connectionEstablishment` function. When a *Player* makes a call to this function, the *Dealer* makes a few verifications on the message it received: whether the `message` and `code` fields have the expected values, if the timeout to join

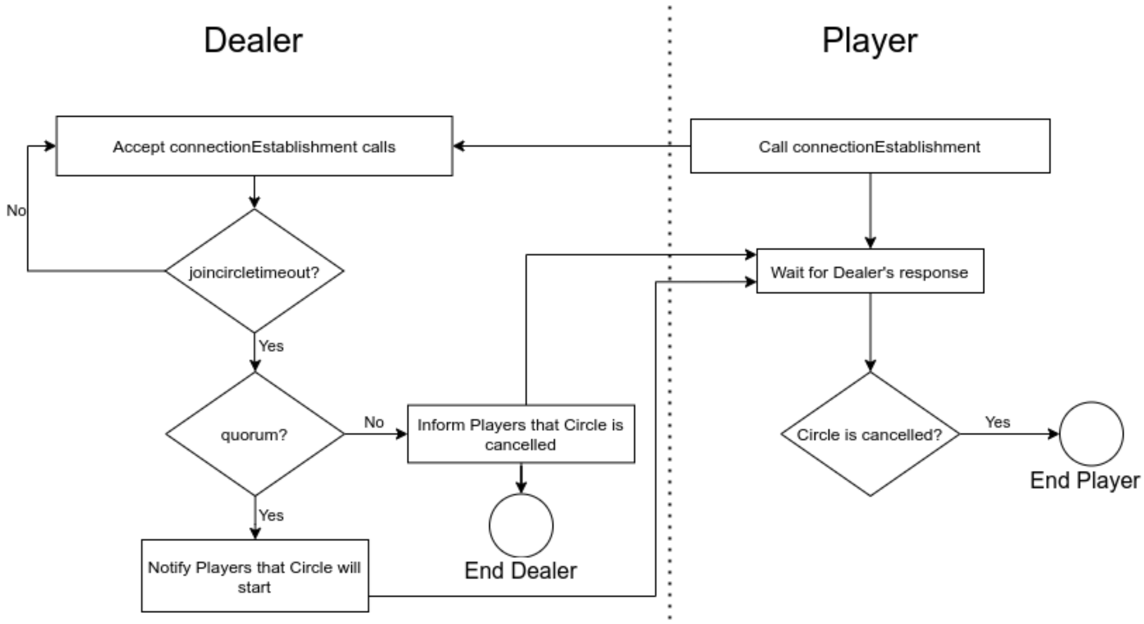


Figure 3.7: Flowchart of the call of a *Player* to `connectionEstablishment`.

Circle, `joincircletimeout`, has not occurred and if the maximum number of *Players* allowed in the *Circle* session, `maxnumberofplayers`, was not already reached. These verifications act as a basic defensive mechanism to avoid invalid messages from *Players* and attempts to call this RPC function at invalid time instants. When the `ConnectRequest` message sent by the *Player* passes all this basic checks, the *Dealer* sends a `ConnectResponse`, where the message and code fields indicate that the *Player* has to wait for another notification from the *Dealer* and, afterwards, the thread serving this *Player* blocks, with the `wait()` method, until the time-out to join *Circle* occurs or until a thread serving other *Player* wakes it. The only case when a thread wakes all other threads with `notifyAll()` is when the maximum number of *Players* for this session is reached, because, in this situation, it is pointless to wait more time, as no more *Players* are allowed to join. When the timeout occurs, either the `quorum` was achieved, and we have a sufficient number of *Players* to proceed with the *Circle* session, or not enough *Players* have joined, and the *Dealer* has to inform the *Players* about this situation. In the first case, the *Dealer* sends a `ConnectResponse` message to the *Players*, where the message and code fields indicate that they can now make calls to the `psiRound` and with the respective *Circle* identifiers assigned to each one, while in the later, it sends a `ConnectResponse` to the *Players* indicating that *Circle* was cancelled. In the last situation both parties, subsequently, terminate their programs.

3.5.2 PSI Round

As shown in figure 3.5, `psiRound` is a simple RPC function, where the *Player* sends the `PsiRequest` message using the stub and waits for a single `PsiResponse` to come back. In figure 3.8 we can observe the flowchart of the call of a *Player* to the `psiRound` function. Similarly to the `connectionEstablishment` method, the *Dealer* makes a few verifications on the message sent by a *Player*. First of all, since `psiRound` is supposed to be called by the *Players* only after `connectionEstablishment` and if the timeout to join the *Circle* session has occurred and the `quorum` was achieved, the *Dealer* checks if the *Player* can make a call to this method at this instant. Since the *Dealer* simultaneously serves these two RPC methods, it has to guarantee that the *Players* cannot make arbitrary calls to these functions, so it verifies whether the `PsiRequest` received has the expected `message` and `code` field values. Furthermore, the *Dealer* verifies if the IP address and port sent by the *Player*, in the `PsiRequest` message, are in the records of *Players* that have joined *Circle*, confirming that this *Player* belongs to the current

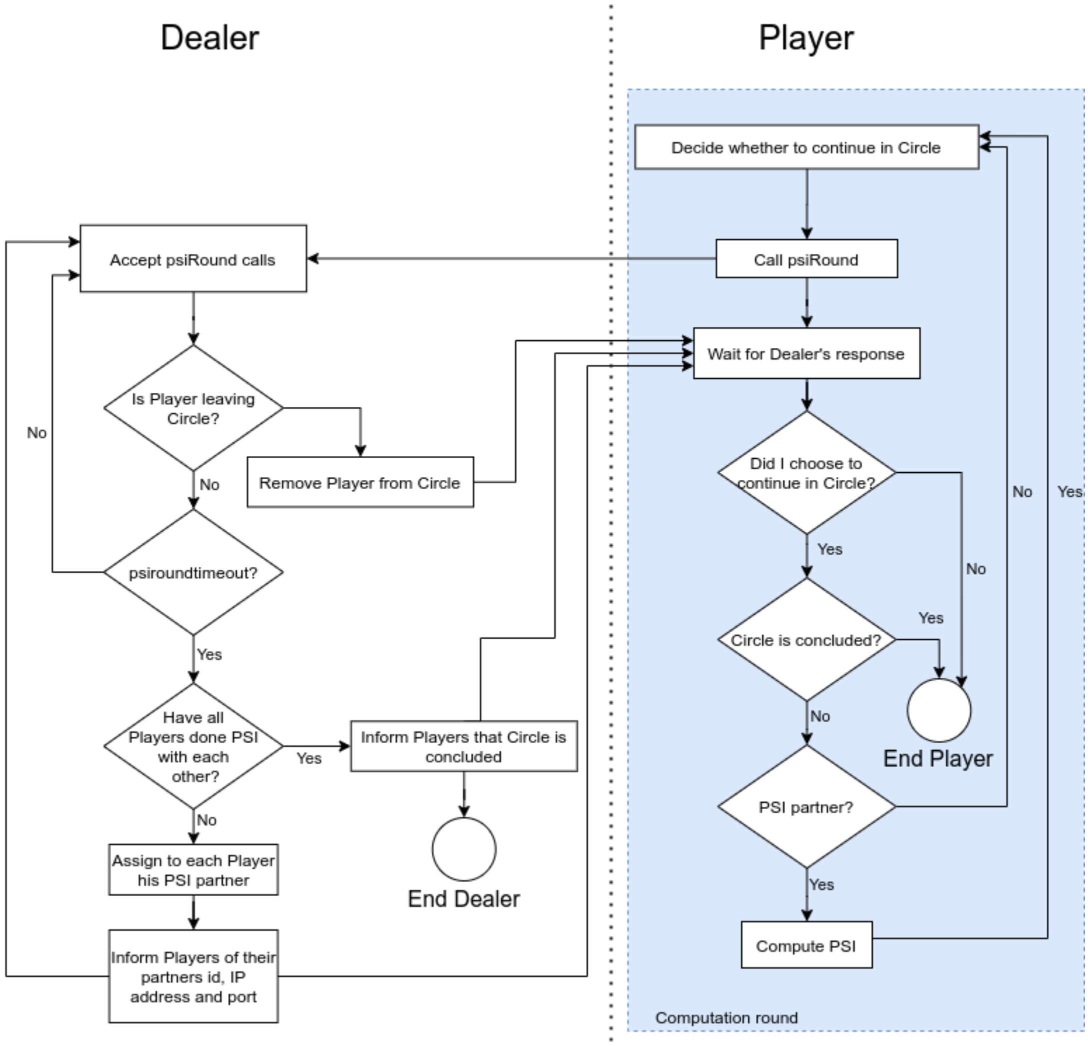


Figure 3.8: Flowchart of the call of a *Player* to `psiRound`.

Circle session. Finally, the `PsiRequest` sent by the *Player* can also contain the information, in the message and code fields, that it wants to leave the session. If this is the case, the *Dealer* removes it from the *Circle Players* records and decrements the number of *Players* connected.

After all the above verifications, the thread serving the *Player* calls the function `hasPsiResult`[3], which is responsible for making the necessary preparations to assign the partners to do PSI in these *Players Computation rounds*, before blocking, with `wait()`. Afterwards, two situations can occur: either all other *Players* make the call to `psiRound` and the thread assigned to the last *Player* wakes all others, with the `notifyAll()` method, or one or more *Players* have not made the call to `psiRound` until the timeout, `psiroundtimeout`, occurs, in which case, all the blocking threads also awake. In the last case, to make the *Circle* session more robust, we considered that the *Players*, that did not send the `PsiRequest` in time, did not want to continue sharing and, as a result, are removed from the *Circle* session. The *Dealer*, subsequently, removes these *Players* from the records, proceeding with the session. Moreover, when there is only one remaining *Player* connected to the *Circle* session, *Circle* is cancelled by the *Dealer*, as there is the need for at least two *Players* to allow the 2PC PSI operation to be performed.

Afterwards, the threads assigned to each of the remaining *Players* use the `getOtherPlayerId`[4] method to compute the IP address, port and *Circle* id of the *Players* with whom they will perform PSI with in this *Computation round*. The values of the IP address, port and *Circle* identifier of the PSI partners are sent by the threads to the respective *Players*, in the `PsiResponse` message. Nevertheless, if a thread serving a certain *Player* does not find an available PSI partner for its *Computation round*, it sends the `PsiResponse`, where the message and code fields indicate that that *Player* will have no pair to do 2PC PSI with.

After *Players* finish their *Computation rounds*, whether they had a partner to perform 2PC PSI with, or not, they make a new call to `psiRound` function and, consequently, all the process described above is repeated, in order to find *Players* that have not yet performed two-party PSI with each other. This successive calls to the `psiRound` function is repeated until, as previously stated, all *Players* have performed 2PC PSI with every other *Player*, allowing each one to discover others that have the same tags in their cyber threat security reports. When this condition is verified, the *Dealer* terminates *Circle* and informs the *Players* that the session is concluded.

[3]The functionality of this method will be discussed in the next subsection

[4]The procedure used in `getOtherPlayerId` to allow *Players* to determine their PSI partners will be discussed in detail in the next subsection.

3.5.3 *Player* Assignment Algorithm

Since we wanted to allow each *Player* of *Circle* to perform PSI with every other *Player*, we had to create an algorithm that decided which pairs of *Players* were going to perform PSI in each *Computation round*. To avoid the large time execution penalty a *Circle* session would incur in, if only a pair of *Players* was computing PSI at each *Computation* round, we created the algorithm described bellow to parallelize PSI operations between different pairs of *Players*, in each *Computation round*.

When the `quorum` is achieved, the *Circle* session starts, but, before the *Dealer* accepts calls to the `psiRound` function, three linked lists are defined: `psiPlayersEdges`, which initially contains all possible combinations of pairs of *Circle Players* identifiers and, during the *Circle* session keeps track of the pairs of *Players* ids that have not yet computed PSI; `currentPsiPlayersEdges`, that keeps track of the pairs of *Players* that are performing the two-party PSI operation during their *Computation rounds*; `notAllowedNumbers`, which keeps track of the *Circle* ids of *Players* chosen to compute PSI. In `psiPlayersEdges` and `currentPsiPlayersEdges`, the pairs of *Players* are pairs of two `Integers`, the respective *Circle* ids, with three restrictions: the two `Integers` in each pair cannot be equal, since it is impossible for a *Player* to compute PSI with itself; each pair of `Integers` has to be unique, since a pair of *Players* only needs to compute 2PC PSI once during *Circle*; each pair of the form (i, j), where i and j are *Players* ids, cannot occur with its `Integers` in reversed positions, (j, i), since when a pair of *Players* performs the PSI operation both obtain the its result. As a consequence, for a total of n *Players* in a *Circle* session, the initial number of pairs in `psiPlayersEdges` is given by equation (3.1):

$$\#pairs = n + (n-1) + (n-2) + ... + (n-(n-1)). \tag{3.1}$$

We identified equation (3.1) as being an arithmetic series whose convergence value is represented in equation (3.2):

$$\sum_{i=0}^{n-1} n - i = \frac{n(n-1)}{2}. \tag{3.2}$$

As a result, by applying the result of equation (3.2) to equation (3.1), we can calculate the initial number of pairs that the `psiPlayersEdges` list will contain from equation (3.3):

$$\#pairs = \frac{n(n-1)}{2}. \tag{3.3}$$

Besides the initial number of *Players* ids pairs that the list `psiPlayersEdges` contains, the result of equation (3.3) also corresponds to the the total number of distinct PSI operations that will occur in a *Circle* session.

The assignment of a partner for a *Player* for the PSI operation in its *Computation round* is performed in the function `getOtherPlayerId`, which is called by each thread serving a *Player*, during the `psiRound` function call. This is a `synchronized method` because the three linked lists described above will be modified and we have to ensure that only one thread can modify them at a time, to avoid race conditions. A thread that calls this method, first checks if the *Player* that made this `psiRound` call has already its id in the `notAllowedNumbers` list. In affirmative case, it implies that a thread serving another *Player* has already removed a pair containing the id of this *Player* from `psiPlayersEdges` and determined that this *Player* will be performing PSI with it. Therefore, the thread serving this *Player* only finds the id of its partner by looking in the `currentPsiPlayersEdges` list and exits this method. However, if the id of this *Player* is not in the `notAllowedNumbers` list, it has not yet been assigned a partner to perform PSI with, in this *Computation round*. Consequently, the thread searches the `psiPlayersEdges` list for a pair that contains the id of this *Player*, and whose the id of the other *Player* is not in `notAllowedNumbers`. The first pair that fulfills this requirement is chosen and removed from `psiPlayersEdges`. This thread then adds both the id of the current *Player* and the id of its new partner to the `notAllowedNumbers` list and their pair to the `currentPsiPlayersEdges`. However, in some cases, a given *Player* may not have a partner available to do two-party PSI with, in its *Computation round*. This occurs if the id of this *Player* is not in the `notAllowedNumbers` list, nor in a pair in `currentPsiPlayersEdges`, and there are no pairs in `psiPlayersEdges` whose one of the ids matches the id of this *Player* and the other id matches the id of a *Player*, which was not already chosen for a PSI operation with a different *Player*. In this case, the function `getOtherPlayerId` outputs -1 as the id of the partner of this *Player*, which is an invalid id number. As a result of the the output of `getOtherPlayerId` being -1, the thread serving this *Player* sends a `PsiResponse`, where the message and code fields indicate that it will not be computing PSI in that *Computation round*.

When *Players* finish each *Computation round*, they begin a new one by deciding whether to remain in *Circle* and, subsequently, making a new call to the `psiRound` function, through a `PsiRequest` message. In order to update the three lists for this set of *Players Computation rounds*, we created the `synchronized method hasPsiResult`. When a thread serving the `psiRound` function call of a *Player* reaches this method, it checks if the id of this *Player* is contained in the `notAllowedNumbers` list, implying this *Player* had performed a PSI operation in its previous *Computation round*. In affirmative case, it afterwards checks if there is still a pair with its id in the `currentPsiPlayersEdges` list. If this is the case, the thread serving this *Player* is the first one, between the thread handling the call of this *Player* and the one handling the call of its partner, to reach this function, so it removes the pair from the `currentPsiPlayersEdges` list and the id of this *Player* from `notAllowedNumbers`, making it available for the *Computation round*. When the thread serving the partner of this *Player* partner reaches the `hasPsiResult` function, it confirms that its id is in `notAllowedNumbers` but not in `currentPsiPlayersEdges`. As a result, the thread then removes its id from the `notAllowedNumbers` list. Furthermore, when a thread serving a *Player* reaches this function and the id of the *Player* is not contained in `notAllowedNumbers` nor there is a pair in `currentPsiPlayersEdges` with its id, it means this *Player* did not have a PSI partner in the previous *Computation round*, so the thread changes neither list. After threads exit the `hasPsiResult` function, they continue executing the `psiRound` function, as it was described in subsection 3.5.2.

The algorithm described in this subsection was created to allow a parallelization of the two-party PSI operations performed by the *Players*. If we did not use it, operations involving different pairs of *Players* would need to be executed sequentially, which would be very inefficient because it would imply that, while a pair of *Players* was performing 2PC PSI, the remaining *Players* in *Circle* would be idle, waiting for those two *Players* to finish. Therefore, by allowing different pairs of *Players* to perform two-party PSI simultaneously, we greatly reduce the total execution time of a *Circle* session.

3.5.4 Secure Communications

In *Circle*, we resorted to the TLS protocol, described in subsection 2.1.3, to secure the communications between the *Dealer* and the *Players*. This protocol was easily integrated in our application since gRPC provides native support for TLS.

When using TLS, we used a model of *mutual authentication* between the server, the *Dealer*, and the clients, the *Players*. In this model, both parties need to authenticate themselves to the other party, thus proving their identity. In the *TLS handshake*, the proof of identity if provided by one party to the other in the form of a digital certificate, which contains the name of the party, the trusted Certification Authority (CA) that vouches for the authenticity of the certificate, and the public encryption key of the party. In our project, since it is a prototype, we created our own CA and used it to sign the digital certificates of both the *Players* and the *Dealer*. The *PriVeil* system can operate with its own CA or an external one, so, when we designed *Circle* as the sequence of *Square*, we concluded that these certificates could have been delivered to the *Dealer* and the *Players* by the system. Besides the *authentication* provided by TLS in this situation, the fact that its application data has a MAC appended to it also guarantees the properties of *authentication* and *data integrity* of the messages exchanged between the *Dealer* and the *Players*. Furthermore, when using TLS, communications are encrypted with symmetric key encryption, providing *confidentiality* to communications of type C_{PD}.

3.6 *Player* Implementation

In this section we describe the implementation of the clients of *Circle*, the *Players*. The *Players* make calls to the RPC functions implemented by the *Dealer*, `connectionEstablishment` and `psiRound`, sending it information as requested. They take as input to their programs the IP address and port they will use to communicate with other *Players* during the PSI operation, their security reports in XML format, as the example of figure 3.2, the IP address and port of the *Dealer* and, finally, the AES-128 secret key they will use in the 2PC PSI computation implemented by FRESCO. When developing our program we used images of ten security reports from NAV for testing, as described in section 3.3.

Similarly to the *Dealer*, we implemented the *Players* programs in Java and used gRPC to allow communications with the *Dealer*. When Maven uses `protoc` to compile the *Circle.proto* file through its plugins, it generates the `CircleServiceGrpc` class. This class contains the stub class, `blockingStub`, which we used to create stubs that allowed the communication between the *Players* and the *Dealer* in calls to the RPC functions `connectionEstablishment` and `psiRound`.

When *Players* call the `connectionEstablishment` method to let the *Dealer* know they wish to participate in the *Circle session*, they send a `ConnectRequest` request which contains a message with a code and their IP address and port to use in the two PSI computations with the other *Players*.

In the case of the `psiRound` function calls, the *Players* send a `PsiRequest` request with a text message and a code, their IP address and port and the *Circle* id assigned to them by *Dealer* during the `connectionEstablishment` function call.

As previously explained, we defined abstract events for the *Players*, the *Computation rounds*. Each *Player* performs a series of actions in each one of its *Computation rounds*: it decides whether to continue in *Circle* or leave, makes a call to the `psiRound` function served by the *Dealer*, if the *Player* has a partner for that round, performs the PSI computation with another *Player*. The *Computation rounds* of each *Player* continue until it leaves the *Circle* session, it is removed from *Circle* because it did not send a `PsiRequest` to the *Dealer* until the `psiroundtimeout` occurred, or *Circle* terminates successfully since all *Players* have computed 2PC PSI with every other *Player*.

In the next subsection we will provide a description of an adjustment we had to make when we integrated the 2PC PSI operation provided by FRESCO with our system.

3.6.1 Integration with FRESCO

As previously mentioned, in *Circle*, we delegated the PSI operation to the FRESCO framework. However, two adaptations were required to integrate the implementation of the 2PC PSI operation by FRESCO with *Circle*.

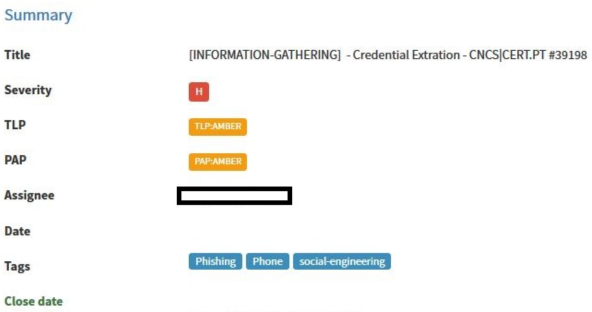
Since FRESCO only accepts sets of integers as the parties inputs for the intersection operation and we required to compute this operation between sets of tags, which contain text values and hence are represented as `Strings`, we had to find a solution which allowed us to make a conversion between the Java `String` tags to Java `Integers` values. To this end, we resorted to the hash function SHA-256, which was discussed in subsection 2.1.2. After we compute SHA-256 on the byte representation of the text value of each tag, we obtain a 32 bytes output. Afterwards, since each `Integer` in Java occupies four bytes of space, we decided to truncate the SHA-256 hash value of each tag to its first four bytes and used them to construct an `Integer` value. From these first four bytes we have a total of 2^{32} `Integers` that can be generated, from the value of -2 147 483 648 to 2 147 483 647. This solution allowed us to be able to compute the two-party PSI operation between sets of text tags, while still using FRESCO.

The other restriction of the PSI operation implemented by FRESCO is that the input sets of both parties are required to have the same size. This is a problem because cyber threat security reports, which belong to different organizations, are very unlikely to contain the the exact same number of tags in them. However, in the ten different cyber threat security reports from NAV that we used when developing our platform, the number of tags was never bigger than five. As a consequence, we considered that, for each *Circle session*, we would need to establish a maximum number of tags allowed in *Players* security reports, which should be always equal or bigger than ten. When performing 2PC PSI, the first solution we found to address this restriction was for each *Player* to provide the actual tags contained in its security report together with empty tags, until the maximum number of security tags allowed was reached. This solution was not ideal since, when both parties received the output of the 2PC PSI, they would be able to identify which was the AES-128 encryption of the empty tags and, consequently, would be able to know what was the number of actual tags in the security report of the other *Player* .

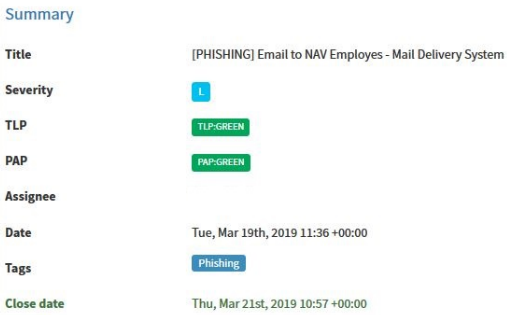
A better way to address this restriction of the implementation of PSI provided by FRESCO is for each *Player* to provide the actual tags contained in its security report, together with random tags, until the maximum number of security tags allowed is reached. When *Players* receive the output of the 2PC PSI computation they are not able to distinguish the random tags from the actual tags in the security report of the other *Player*. Unlike the empty tag solution, this solution prevents each *Player* from knowing the actual number of tags in the report of the other *Player*. In *Circle*, the latter solution was implemented over the previous one, although the evaluation and results of chapter 4 were obtained for the empty tags implementation. This change has no significant impact on the performance.

3.7 *Circle* Example

In this section, to conclude the chapter, we demonstrate a session of *Circle* with two *Players*. In this situation, we assumed that the *Players* had the security reports of figure 3.9. Both



(a) *Player* 1 security report.



(b) *Player* 2 security report.

Figure 3.9: *Players* security reports.

these reports are from the dataset that we obtained from NAV. As described in section 3.1, the security reports in the NAV dataset were images so, before we could use them as input to *Players* in a *Circle* session, we first had to convert them to XML format. The result of the conversion of the security reports to XML is represented in figure 3.10.


```xml
<?xml version="1.0" encoding="UTF-8"?>
<report>
    <summary>
        <title>[INFORMATION-GATHERING] -Credential Extraction-CNCS|CERT.PT#39198</title>
        <severity>H</severity>
        <tlp>TLP:AMBER</tlp>
        <pap>PAP:AMBER</pap>
        <assignee></assignee>
        <date></date>
        <tags>
            <tag>Phishing</tag>
            <tag>Phone</tag>
            <tag>social-engineering</tag>
        </tags>
        <closedate></closedate>
    </summary>
    <additionalinformation></additionalinformation>
    <metrics>Duration 5</metrics>
    <description>Entre as ... as ... de hoje foram recebidas num número de telefone
    operacional do ..., três chamadas telefónicas, de alguém que se identificou como
    sendo da Microsoft informando que os nossos computadores estariam contaminados com
    vírus. 10h50-Sotaque de leste,em inglês,originada de 0038... (não foi possível
    identificar os restantes números) 11h15-idem, originada de 962 633 875 11h42-idem,
    originada de 00385 989 656 377</description>
    <summary>Caso fechado</summary>
</report>
```


(a) *Player* 1 security report described in XML.


```xml
<?xml version="1.0" encoding="UTF-8"?>
<report>
    <summary>
        <title>[PHISHING] Email to NAV Employes-Mail Delivery System</title>
        <severity>L</severity>
        <tlp>TLP:GREEN</tlp>
        <pap>PAP:GREEN</pap>
        <assignee></assignee>
        <date>Tue,Mar 19th,2019 11:36 +00:00</date>
        <tags>
            <tag>Phishing</tag>
        </tags>
        <closedate>Thu,Mar 21st,2019 10:57 +00:00</closedate>
    </summary>
    <additionalinformation></additionalinformation>
    <metrics></metrics>
    <description>Mensagem de Phishing a simular mail delivery</description>
    <summary></summary>
</report>
```


(b) *Player* 2 security report described in XML.

Figure 3.10: *Players* security reports described in XML.

In figure 3.11 we have the most relevant part of the output obtained for each *Player*, in the end of the *Circle* session. We can observe that the *Players* take as inputs to their programs

```
miguel@Desktop:~/git/PriVeilCircle/Circle/client/player1$../target
/appassembler/bin/circle-grpc-client 127.0.0.1:8881 report3.xml
127.0.0.1:9999 abc123abc123abc123abc123abc123ab
CIRCLE PLAYER STARTED
Do you wish to start connecting to Circle? [Y/N]
y

(...)

message: "Please wait for Dealer notification"
code: 5

message: "All players addreses received, proceeding to PSI round"
code: 7
id: 1

Do you wish to continue in PSI Round? [Y/N]
y
message: "Sending other player address"
code: 11
id: 2
address: "127.0.0.1:8882"

(...)

option: s : tinytables
option: e : SEQUENTIAL_BATCHED
option: i : 1
option: p : 1:127.0.0.1:8881
option: p : 2:127.0.0.1:8882
option: key : abc123abc123abc123abc123abc123ab
option: in : Phishing,Phone,social-engineering
The resulting ciphertexts are:
result(0): 700f58d21feb31d7b271fce8f63e33f2
result(1): 37e9a7eb1a06108f7dd6dfb6379cfd4b
result(2): 3d4fe0593520479126b882849f5b31f0
result(3): 18680599965975d96df663ce40f6f166
result(4): 459639eba61ed7f557598d1e7138eacd
result(5): 57e21e5f87de3257a779b25ddbf02e2a
result(6): 27f20696029737544f2780452125e001
result(7): ac8df4260c787cf8297a0c4fc60bf460
result(8): c2d4e9628a5fadd8c31db0508084b7d5
result(9): 20aa9c9a2922bda635707ebc5374428b
result(10): 700f58d21feb31d7b271fce8f63e33f2
result(11): 8333838055309541c93a582444bf1d10
result(12): 43f923f42e10f49ee949d5b361de351d
result(13): 02b865875941da0971b6ac5c8623f056
result(14): 3c9ccfde92bb9a4ee195bfd8987f6352
result(15): 6dd7fc9c8347c7c80180b6fd312acb93
result(16): 945f8cac49d266d02dd48695caa27d08
result(17): bed5d9f8e8b85561966df37bd398ea36
result(18): cda0c8365ffa5243598f88a9aa2fe74a
result(19): 448b0f899288e4cb6ae8299f68d4e04a
Do you wish to continue in PSI Round? [Y/N]
y
message: "Psi Round is over"
code: 9

PSI Round is over. Proceeding to next round.
CIRCLE PLAYER TERMINATED
```

(a) Output of the *Circle* session for *Player* 1.

```
miguel@Desktop:~/git/PriVeilCircle/Circle/client/player2$../target
/appassembler/bin/circle-grpc-client 127.0.0.1:8882 report8.xml
127.0.0.1:9999 abc123abc123abc123abc123abc123ab
CIRCLE PLAYER STARTED
Do you wish to start connecting to Circle? [Y/N]
y

(...)

message: "Please wait for Dealer notification"
code: 5

message: "All players addreses received, proceeding to PSI round"
code: 7
id: 2

Do you wish to continue in PSI Round? [Y/N]
y
message: "Sending other player address"
code: 11
id: 1
address: "127.0.0.1:8881"

(...)

option: s : tinytables
option: e : SEQUENTIAL_BATCHED
option: i : 2
option: p : 2:127.0.0.1:8882
option: p : 1:127.0.0.1:8881
option: key : abc123abc123abc123abc123abc123ab
option: in : Phishing
The resulting ciphertexts are:
result(0): 700f58d21feb31d7b271fce8f63e33f2
result(1): 37e9a7eb1a06108f7dd6dfb6379cfd4b
result(2): 3d4fe0593520479126b882849f5b31f0
result(3): 18680599965975d96df663ce40f6f166
result(4): 459639eba61ed7f557598d1e7138eacd
result(5): 57e21e5f87de3257a779b25ddbf02e2a
result(6): 27f20696029737544f2780452125e001
result(7): ac8df4260c787cf8297a0c4fc60bf460
result(8): c2d4e9628a5fadd8c31db0508084b7d5
result(9): 20aa9c9a2922bda635707ebc5374428b
result(10): 700f58d21feb31d7b271fce8f63e33f2
result(11): 8333838055309541c93a582444bf1d10
result(12): 43f923f42e10f49ee949d5b361de351d
result(13): 02b865875941da0971b6ac5c8623f056
result(14): 3c9ccfde92bb9a4ee195bfd8987f6352
result(15): 6dd7fc9c8347c7c80180b6fd312acb93
result(16): 945f8cac49d266d02dd48695caa27d08
result(17): bed5d9f8e8b85561966df37bd398ea36
result(18): cda0c8365ffa5243598f88a9aa2fe74a
result(19): 448b0f899288e4cb6ae8299f68d4e04a
Do you wish to continue in PSI Round? [Y/N]
y
message: "Psi Round is over"
code: 9

PSI Round is over. Proceeding to next round.
CIRCLE PLAYER TERMINATED
```

(b) Output of the *Circle* session for *Player* 2.

Figure 3.11: Output of the *Circle* session for both *Players*.

the IP address and port that they want to use when computing PSI with other *Players*, the XML security report file, the IP address and port of the *Dealer* and, finally, the AES-128 secret key which is used in the implementation of PSI provided by FRESCO. Furthermore, we can then see the two `ConnectResponse` messages received from the *Dealer*, after the *Players* call the `connectionEstablishment` function. The first one is informing the *Players* to wait for a further notification and the next one informing the *Players* that the PSI phase will start[5]. After these we can see the first *Computation round* of the *Players*, where they are given the choice to continue in *Circle* or leave the session. In this example, we instructed both *Players* to continue in *Circle* so, the next message from the *Dealer* is the `PsiResponse` containing the address of the respective *Players* PSI partners for that *Computation round*. Since we only have two participants in this session, they will inevitably have to compute PSI with each other in this *Computation round*. We can confirm that in this `PsiResponse`, each *Player* receives the IP address, port and *Circle*

[5]In this example we set `quorum`=2 and `maxnumberofplayers`=2 in the *configuration.xml* file of the *Dealer*.

identifier of the other. The *Players* then compute the PSI operation implemented by FRESCO. From comparison with figure 2.2 we can see that the inputs (option "in") are not `Integers` anymore, but rather `Strings` which correspond to the tags of the reports in figure 3.10. These `Strings` are, afterwards converted in `Integers`, inside the *Players* program, which the FRESCO uses in its PSI implementation, as described in subsection 3.6.1. We can also observe that the concatenated list obtained as the output of the PSI computation contains 20 elements, despite the first *Player* only containing one tag in its security report and the second only containing three tags. This corresponds to the second situation described in section 3.6.1, where the two parties need to have input sets of the same size so that FRESCO can compute PSI. In this example, we fixed a maximum number of ten tags allowed in *Players* security reports. Since the first half of the output list corresponds to the encryption of the input set of the first *Player*, the AES-128 encryption of its three security report tags, *Phishing*, *Phone* and *social-engineering*, are, respectively, *700f58d21feb31d7b271fce8f63e33f2*, *37e9a7eb1a06108f7dd6dfb6379cfd4b* and *3d4fe05935204791 26b882849f5b31f0* (indexes 0, 1 and 2). Since the second half of the output list corresponds to the encryption of the input set of the second *Player*, the AES-128 encryption of its security report tag, *Phishing*, is *700f58d21feb31d7b271fce8f63e33f2* (index 0). The AES-128 encryptions at the remaining indexes correspond to the encryption of random tags. From these results, the two *Players* know that they both contain the *Phishing* tag in their security reports, because the values at indexes 0 and 10 are equal.

After this *Computation round*, the *Players* proceed to the next one, where they have again to decide whether to continue in *Circle*. We instructed them to continue but, since they were the only two participants of this *Circle* session, they receive a `PsiResponse` from the *Dealer* indicating that the session is over. Both *Players* subsequently terminate their programs.

The simple example provided in this section allows to better understand how a *Circle* session works and what is the information that each *Player* receives in the end of each PSI computation.

3.8 Summary

In this chapter, we described the solution we developed, *Circle*, the sequence of *Square* and the second part of the *PriVeil* system. We wanted to enable users whose reports matched in *Square* to be able to share cyber threat information in a controlled setting. In particular, the tags in the *Players* cyber threat security reports can contain information like IoC, malicious IP addresses and malware hashes. It would be useful to share this information in the context of a cyber threat sharing platform, to allow organizations to collectively search for a solution. However, this data also needs to be handled very carefully, since it can contain sensitive information, which, if disclosed, could cause enormous financial and reputational losses to its owners. As a result, we determined that it would be useful to allow *Players* to compute the tags in their cyber threat security reports which are equal to other *Players* security reports tags, by using a *secure* computation that guarantees the properties of *Privacy* of Inputs and *Correctness* of the Output. We relied on the two-party PSI operation to achieve this goal: two *Players* can compute the tags which are equal in their security reports, without any information being disclosed about the different ones.

Chapter 4

Evaluation

In this chapter we present the evaluation of *PriVeil Circle*, the performed experiments and their respective results.

We start this chapter by addressing whether the requirements defined for *Circle* were fulfilled in section 4.1. In section 4.2 we describe how we designed the set of experiments which we used to evaluate *Circle*, the testbed we used to perform the experiments and the metrics we decided to measure in each test. Furthermore, in section 4.3 we present the results obtained for the metrics measured during the experiments. We conclude this chapter by providing a discussion of the experimental results of *Circle* and how they affect our work, in section 4.4.

4.1 Qualitative Evaluation

In this section we discuss which requirements were fulfilled for *Circle*, from those that were defined in section 3.2.

4.1.1 Functional Requirements Assessment

In table 4.1, we can observe the functional requirements from section 3.2.1 that were addressed by our *Circle* prototype.

Table 4.1: Assessment of the functional requirements fulfilled.

Requirement	**Status**
FR_1	Fulfilled
FR_2	Fulfilled
FR_3	Fulfilled

The functional requirement FR_1 is fulfilled by the fact that the *Players* can choose whether to remain in *Circle* at the beginning of each one of their *Computation rounds*. Furthermore, the *Players* can also leave the *Circle* session at any time, without having to notify the *Dealer*, since this party is prepared to handle this situation and continue the session. The functional requirement FR_2 states that only parties that received a token to access *Circle* from the *Square* component are authorized to participate in it. Due to the *authentication* mechanisms provided by the Transport Layer Protocol, if the *Dealer* detects that the certificate or message of a *Player* is not valid, the *Dealer* immediately aborts communication with this *Player*. Furthermore, if a *Player* detects that the certificate or message of the *Dealer* is not valid, the *Player* also immediately ceases communication with the *Dealer*. These two situations fulfill the functional requirement FR_2, because both the *Dealer* and the *Players* can detect if they are communicating with a party which is authorized to participate in the session and, in case they are not, they can leave the session. The functional requirement FR_3 is addressed by the choice of the Private Set Intersection (PSI) operation to allow *Players* to share cyber threat information. This operation allows a privacy-preserving computation to be performed on the sets of tags in the *Players* security reports. In the end, the tags which are equal in the security reports are revealed, without any information being leaked about the ones that are different.

4.1.2 Security Requirements Assessment

In table 4.2, we can observe the security requirements from section 3.2.3 that were addressed by our prototype.

Table 4.2: Assessment of the security requirements fulfilled.

Requirement	**Status**
SR_1	Fulfilled
SR_2	Not fulfilled

To fulfill SR_1, we used the TLS Protocol to prevent all attackers defined in subsection 3.2.2, in communications of type C_{PD}. The fact that the messages between the *Dealer* and *Players* are encrypted with symmetric key encryption prevents an Eavesdropper, that is unable to distinguish the data exchanged. The *Player* Impersonator and *Dealer* Impersonator are defeated by the *authentication* mechanisms used in the *TLS Handshake* and in the *TLS Records* exchanged, because the *Player* and the *Dealer* are able to understand if they are communicating with each other. Finally, a Tamperer is prevented by the *data integrity* mechanisms that TLS employs, in particular, the MAC appended to each *TLS record*, since both parties can tell whether data was changed in transit. In this work, the security requirement SR_2 was not addressed. FRESCO does not natively use TLS to secure communications in its implementation of 2PC PSI. We were also not able to implement it although, since FRESCO uses KryoNet to create TCP sockets to allow communications during PSI, we would probably need to resort to classes of the `javax.net.ssl` Java module to be able to implement TLS on top of TCP sockets.

4.2 Experimental Design

To evaluate our system, we performed a total of nine experiments, for nine different *Circle* sessions. In each session, we measured: the processing time of the most relevant functions executed by a *Player*, to assure that our system provided answers in a timely way; the system CPU and memory usage of the program of the same *Player*, to show that the required resources to participate in *Circle* are reasonable in commodity hardware. Between distinct experiments, we examined the impact on the aforementioned metrics of different combinations of two parameters: the number of *Players* participating in the *Circle* session and the maximum number of tags allowed in the *Players* cyber threat security reports.

4.2.1 Testbed

We used a setup consisting of two machines to conduct the experiments and measure the metrics mentioned. Their purpose was to execute the program of the *Dealer* and the *Players* programs during the nine *Circle* sessions. The software and hardware specifications of the machines used are the following:

1. The first machine was running Ubuntu 20.04.1 LTS OS, with an Hexa Core Intel i5-8400 processor, 16 GB RAM, 1 TB Hard Disk Drive (7200 rpm), 128 GB Solid State Drive and an internet cable connection with bandwidth of 500 Mbit/s.

2. The second machine was running Ubuntu 20.04.01 LTS OS, with an Octa Core Intel i7-8565U, 16 GB RAM, 500 GB Hard Disk Drive (7200 rpm), 128 GB Solid State Drive and an internet cable connection with bandwidth of 500 Mbit/s.

4.2.2 Time Measurements

The measurement of the processing time of the most relevant functions executed by a *Player* was performed with the help of the Java method `System.nanoTime()`. This function can only be used to measure elapsed time and, as stated in the Java documentation[1], its return value

[1] The documentation for this method can be found at `https://docs.oracle.com/javase/7/docs/api/java/lang/System.html`.

represents the current value of the high-resolution time source of the running Java Virtual Machine, in nanoseconds. Although it provides nanosecond precision, it does not guarantee nanosecond resolution. The code below was used to measure how long a specific part of code takes to execute:

```
long startTime = System.nanoTime();
\\ Code being evaluated
long endTime = System.nanoTime();
long executionTime = endTime-startTime;
```

The first time measurement checkpoint starts counting the time since the start of the program, in the `main` function of the `Main` class, and stops counting when the program is terminated without errors, in the `main` function of the `Main` class. This measurement corresponds to the total execution time of the program of the *Player* . The second measurement corresponds to the time that a *Player* takes to store information about its security report in variables to be used in the program at a later moment. This time checkpoint is called *Parse report*. The third measurement corresponds to the time measured for the call to the remote function `connectionEstablishment`, from the moment the *Player* sends the *ConnectRequest* to the *Dealer*, until the moment the *Player* receives the *ConnectResponse* indicating whether the *Circle* session will occur, from the *Dealer*. This checkpoint is called *Circle connection*. The fourth time measurement corresponds to the duration of a *Computation round* of the *Player*. This checkpoint measures the time since a *Player* is asked whether it wants to continue in *Circle*, makes a call to `psiRound` and receives the `ConnectResponse` from the *Dealer*, until the *Player* ends the PSI operation with its partner, if there was one available. This checkpoint is called *Computation round*. Since this checkpoint measures *Players Computation rounds*, it can occur several times in the same session. Finally, the last execution time measurement corresponds only to the PSI operation. This execution time is a subset of the time measured in the *Computation round* checkpoint, in the case there is a PSI operation involved. This checkpoint is called *PSI operation*.

We applied the time measurement code above to all time checkpoints discussed. Table 4.3 provides a detailed description of all the time measurement checkpoints used to evaluate our solution.

4.2.3 System Resources Measurements

We created the Python script `setup.py` to measure the system CPU and memory usage of the program of a *Player*. This script can take three different commands and be executed from the Linux command-line with the following command:

Table 4.3: Description of time checkpoints introduced in the code of the *Player*.

Total measured time	Time measured from the beginning of the program of the *Player* until its successful termination.
Parse report	Time taken for the program of the *Player* to parse the input security report and store the values in variables to be used afterwards.
***Circle* connection**	Time taken for the *Player* to send its address to the *Circle Dealer* and the response of the *Dealer* indicating if *Circle* will occur or will be cancelled.
Computation round	Time taken for the *Player* to decide whether to remain in *Circle*, send the `ConnectRequest` to the *Dealer*, receive the `ConnectResponse` from the *Dealer* and perform the PSI operation, in case there was a partner available.
PSI operation	Time duration of the PSI operation during the *Computation round*.
Other	All the time that is not included in *Parse report*, *Circle connection* and *Computation round* checkpoints.

```
./setup.py <COMMAND>
```

If this script is executed with the `benchmark` option, it will need two additional commands: the keyword to search for a specific process and the sampling period to measure system resources usage by that process. By using the process keyword provided, the PID of the intended process can be obtained.

In the case of *Circle*, the Python script was executed with:

```
./setup.py benchmark Circle/client 0.5
```

By using this command, the first process that has "Circle/client" in its name is found and its PID is returned, and the measurement sampling period is set to 0.5 s.

The Linux command line command `top` provides a real-time view of the system. It can list processes and threads being managed by the Linux Kernel at each time instant and the system resource usage of each process or thread running. When running the `setup.py` script to benchmark a *Circle* session, a single measurement of the system CPU usage and system memory usage of process with identifier PID was done by using the following command:

```
top -b -n 1 -p <pid> | tail -n 1 | awk {'print $9 \"\t\" $10'}
```

By running this command periodically, with the period specified by the user, various samples of the resources usage of the program of the *Player* are obtained. Therefore, to benchmark the whole program of the *Player*, the `setup.py` Python script was running periodically while the process of the *Player* was running. When the program finished execution, the Python script saved all the timestamps and the system CPU and memory usage at the respective timestamp in a text file.

4.2.4 Experiments Definition

In each one of the nine experiments conducted, we measured the processing time of the most relevant functions and system CPU and memory usage of a single process of a *Player*, which was running on the first machine of the testbed. In all of these *Circle* sessions we used as input to the program of this *Player* the same XML report corresponding to a real-case cyber threat security report from NAV, with 5 tags contained in it. Since in each test we also had to execute the program of the *Dealer* and the remaining *Players*, we resorted to the second machine in the testbed to accomplish this objective.

In the first set of three experiments, the number of *Players* was fixed to 2 and the maximum number of tags was increasingly set to 10, 25 and 50, between different *Circle* sessions. In these group of tests, the number of *Players* was equal to 2, because this corresponds to the minimum number of *Players* required for a two party computation PSI operation to be performed in a *Circle* session. Furthermore, this situation also corresponds to the simplest setting possible for Circle, since a session cannot occur if the number of participants is lesser than 2[2]. For the first experiment we set the maximum number of tags allowed in *Players* security reports to 10, since in all the ten reports we obtained from NAV, the maximum number of tags we observed on a report was 5. Furthermore, in the work that addresses the *Square* component of *PriVeil*, it is estimated that the security reports have on average 10 tags contained in them. As a result, the value of 10 seemed like a reasonable value for the lower bound for the maximum number of tags allowed in the *Players* reports. On the other hand, to try to include all the cases of security reports that have a number of tags bigger than 10, we established the value of 50 tags as the upper bound to the maximum number of tags allowed in security reports. Finally, we concluded that only performing tests for these two values of this parameter would be insufficient, so, we also performed experiments for the intermediate value of 25 tags.

In the next set of three experiments, the number of *Players* was set to 3 and the maximum number of tags allowed in their reports was also increasingly set to 10, 25 and 50. In this group of tests, we fixed the number of *Players* to 3, so that we could observe how the measured metrics would be affected when the *Circle* session involved more than one 2PC PSI operation. In particular, we know from equation (3.3) that, in this case, a total of three PSI computations would be performed.

[2]As mentioned in section 3.5, we always require at least 2 *Players* for a *Circle* session to occur.

Finally, in the last group of experiments, the number of *Players* was fixed to 10, while the maximum number of tags was also increasingly set to 10, 25 and 50. In this final set of experiments, we fixed the number of *Players* to 10, since this is the average number of *Players* we expect to have in each *Circle* session, if we were to employ this project in a real-life cyber threat information sharing scenario. Furthermore, this value seemed reasonable considering that, in *Circle*, we wanted to allow threat information sharing between a more restricted group of organizations, when compared to the type of sharing that occurs in *Square*.

The fact that between the three groups of experiments, we measured the mentioned metrics for 2, 3 and 10 *Players* in the *Circle* sessions, while increasing the maximum number of tags allowed in the reports to 10, 25 and 50 in each individual experiment of each group, allowed us to obtain results for the processing time and CPU and memory usage of the program of the *Player* as a function of the number of tags, for a fixed number of *Players*, as well as a function of the number of *Players*, for a fixed number of tags.

4.3 Experimental Results

In this section we present the results obtained for the experiments defined in the previous section. In subsection 4.3.1 we describe the results obtained for the set of experiments performed in the 2 *Players* setting. In subsection 4.3.2 we present the results obtained in the 3 *Players* setting experiments. In subsection 4.3.3 we discuss the results obtained in the 10 *Players* setting. Finally, in subsections 4.3.5 and 4.3.4 we present the average system resources usage measurements and total execution time measurements as a function of the number *Players* in the *Circle* session.

4.3.1 2 *Players* Experiments

In the 2 *Players* setting, one *Player* was running on the first machine in the testbed, where the performance metrics were measured, and the *Dealer* and the other *Player* were running on the second machine. With the number of *Players* fixed to 2, we increasingly set the maximum number of allowed tags in *Players* security reports to 10, 25 and 50. For each of these three experiments, we obtained measurements for all the time checkpoints described in subsection 4.2.2. The results are presented in figure 4.1.

For the 2 *Players* setting, the total execution time measured for the program of the *Player*, for a maximum of 10, 25 and 50 tags allowed in the security reports was 51,195 s, 96,605 s and 179,07 s, respectively. From these results we can conclude that an increase in the number of tags leads to an increase in the duration of a *Circle* session. In table 4.4 we present the execution

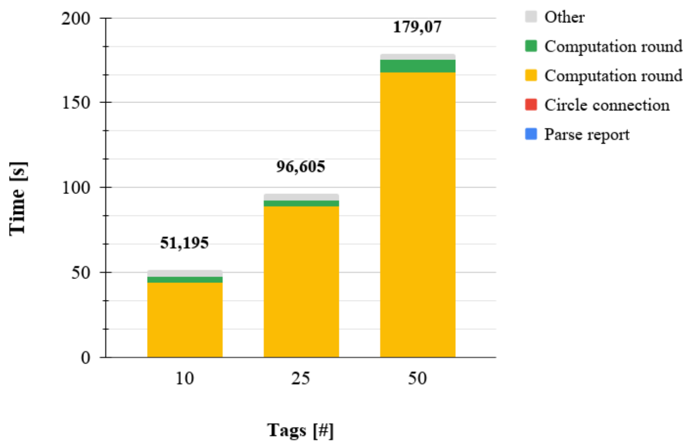


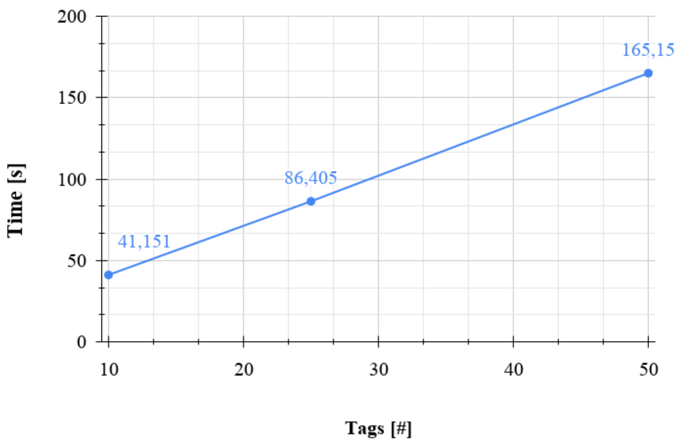
Figure 4.1: Time breakdown as a function of the maximum number of tags allowed in the security reports for 2 *Players*.

times measured in each checkpoint.

Table 4.4: Time breakdown for 10, 25 and 50 maximum tags allowed in *Players* security reports for 2 *Players* (seconds).

Tags	*Parse report*	*Circle connection*	*Computation round*	*Computation round*	*Other*
10	0,039889	0,23237	43,889	3,0548	3,9789
25	0,040341	0,23457	88,616	3,4184	4,2958
50	0,039889	0,28408	167,42	7,4269	3,8909

From the observation of table 4.4 we can conclude that the time measured on checkpoints *Circle connection* and *Other* does not depend on the maximum number of tags allowed in *Players* security reports. This is expectable since the functions measured by these checkpoints include communications with the *Dealer*, interacting with local files and local processing. These are not dependent on the number of tags allowed in the security report of the *Player*. Although the time measured in the *Parse report* checkpoint is dependent on the number of tags in the security report of the *Player*, it does not depend on the maximum number of tags allowed. Therefore, the duration of the *Parse report* is similar in all experiments. The second *Computation round* also has similar durations as the number of tags increases. This is explained by the fact that in the second *Computation round* there are no Private Set Intersections operations left to be computed, as the only PSI operation involved in this *Circle* session was already performed in the first *Computation round*. As a result, in the second *Computation round*, the *Dealer* only informs the *Players* that there are no more computations left and, consequently, the session will be terminated. The first *Computation round* is the checkpoint for which the measured execution time significantly depends on the maximum number of tags allowed in *Players* security reports. In table 4.2 we can see the time breakdown of the first *Computation round*, separated in the time measured in the *PSI operation* checkpoint and other tasks, which include communication with the *Dealer* and local processing.



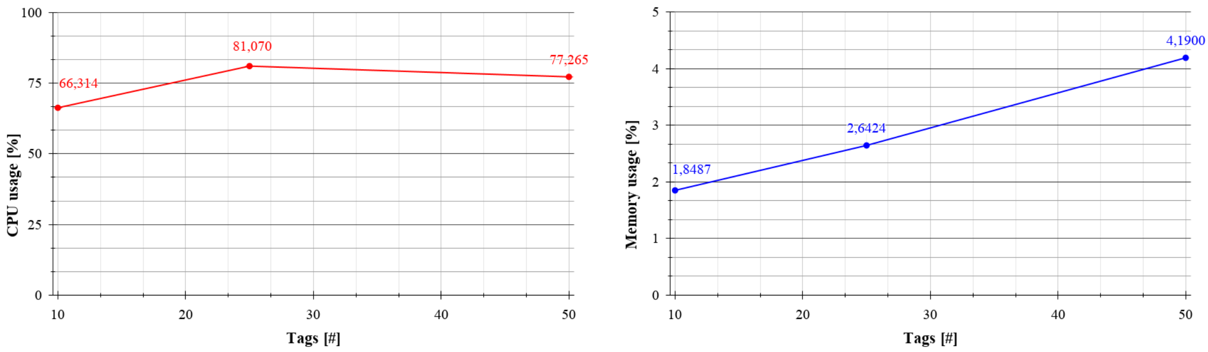
	Computation round	
Tags	*PSI operation*	*Other*
10	41,151	2,7386
25	86,405	2,2106
50	165,15	2,2740

Figure 4.2: *PSI operation* time plot and time breakdown of the first *Computation round* for 10, 25 and 50 maximum tags allowed in *Players* security reports for 2 *Players* (seconds).

From table 4.2 we can observe that, in all three experiments, the part of the *Computation round* that refers to communications and local processing does not vary significantly. However, the PSI operation duration increases significantly. This indicates that the increase in the maximum number of tags allowed in *Players* security reports only directly affects the PSI operation. This is in fact true, because in the context of PSI, the more items in the sets of the parties, the more time it takes to compute this operation between those sets. Furthermore, from the observation of figure 4.2 we can conclude that the duration of the PSI operation increases linearly with an increase in the maximum number of tags allowed in *Players* security reports.

For the 2 *Players* setting, the average CPU and memory usage as a function of the number of tags is represented in figure 4.3.

The results in figure 4.3 concerning the system average CPU usage show an increase of 14,756 % in the average CPU usage when the maximum number of allowed tags in *Players* security reports increases from 10 to 25. However, when the number of tags increases from 25 and 50 there is a 3,805 % decrease in the average system CPU usage of the program of the *Player* . As observed in figure 4.3, the average system memory usage increases linearly with an increase in the maximum number of tags allowed in *Players* security reports.



(a) Average CPU usage.

(b) Average Memory usage.

Figure 4.3: System average CPU and memory usage as a function of the maximum number of tags allowed in the security reports for 2 *Players*.

4.3.2 3 *Players* Experiments

In the 3 *Players* experiments, 1 *Player* was running on the first machine of the testbed, while the remaining 2 *Players* and the *Dealer* were running on the second machine of the testbed. With this configuration, the process of the *Player* running on the first machine was evaluated, with concern to the time metrics described in subsection 4.2.2 and the system resource metrics described in subsection 4.2.3. We performed these evaluations for 10, 25 and 50 maximum tags allowed in *Players* security reports. The time breakdown for these three experiments is represented in figure 4.4. From the results of figure 4.4 we can see that there are four

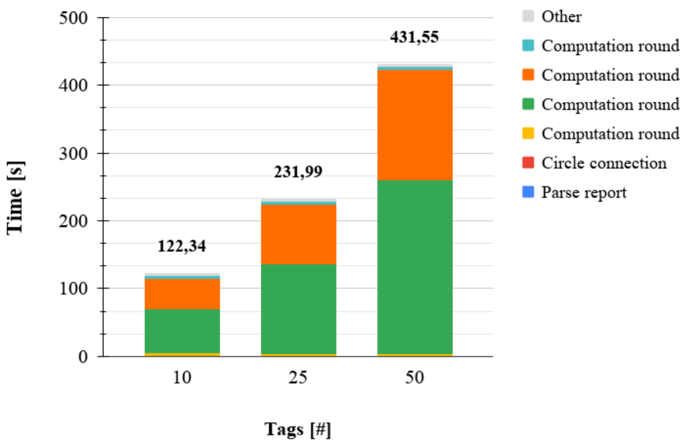


Figure 4.4: Time breakdown as a function of the maximum number of tags allowed in the security reports for 3 *Players*.

Computation rounds involved in this *Circle* session. The first and last *Computation rounds*, the *Parse report*, *Circle connection* and *Other* checkpoints measurements do not depend on the maximum number of tags allowed in *Players* security reports because their values are similar in all these three experiments. In table 4.5 we represented the time spent in each *Computation* *round*, for the 10, 25 and 50 maximum tags allowed in *Players* security reports. From the

Table 4.5: *Computation rounds* duration for 10, 25 and 50 maximum tags allowed in *Players* security reports for 3 *Players* (seconds).

Tags	*Computation round*	*Computation round*	*Computation round*	*Computation round*
10	3,8536	65,356	44,849	4,1041
25	3,6666	131,14	88,190	4,8115
50	3,4605	256,58	162,61	4,5420

observation of the results in table 4.5 we can see that the duration of the first *Computation round* is very short when compared to the following two *Computation rounds*, for all values of the tags. This is justified by the fact that in the first *Computation round*, the *Player* where the measurements were being made did not have a partner to perform the PSI operation with. In this first *Computation round*, the 2 *Players* running on the second machine of the testbed were performing the PSI operation among them. Therefore, the time accounted by the first *Computation round* in table 4.5 is just the time it takes for *Player* to decide to remain in *Circle*, call the *psiRound* function and receive a `psiResponse` indicating that it does not have a partner for that round. In the last *Computation round*, all the *Players* in the *Circle* session had already performed PSI with every other *Player*, so there were no more operations to be performed. Consequently, in this last *Computation round*, the *Dealer* simply informed all the *Players* that the *Circle* session was concluded. Finally, in the second and third *Computation rounds*, the *Player* where the measurements were made was doing PSI with the other 2 *Players* in the session, which justifies the much higher durations of these rounds when compared to the first and last *Computation rounds*.

To provide a more detailed insight on the PSI operation, in figure 4.5 we can see the duration of the two PSI computations for a varying maximum number of tags allowed in the *Players* reports. From the observation of figure 4.5 we can conclude that the PSI operation duration increases linearly with an increase in the maximum number of tags allowed in *Players* security reports. From the comparison of table 4.5 and figure 4.5 we can determine that the first PSI operation only occupies 60,019 % of the correspondent *Computation round*, while the second PSI operation occupies 87,712 % of the respective round. This difference can be justified by the fact that the other 2 *Players* were performing the PSI operation. While the other *Players* were computing PSI, the *Player* where the measurements were made was idle, waiting for the other 2 *Players* to finish. This waiting time is included in the second *Computation round* checkpoint.

The system resource metrics measured during the execution of the program of the *Player*

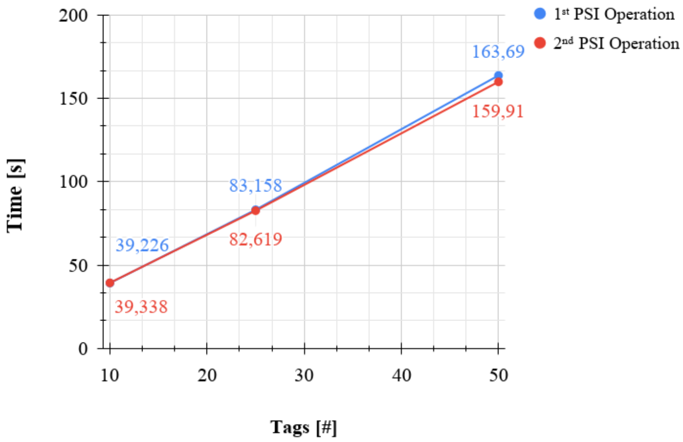
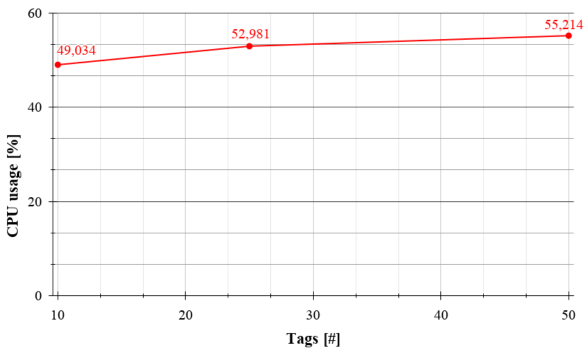
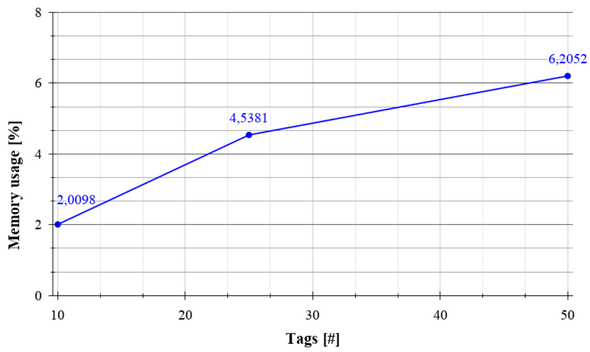


Figure 4.5: PSI operation as a function of the maximum number of tags allowed in the security reports for 3 *Players*.

, system average CPU usage and memory usage, obtained for the 3 *Players* experiments are represented in figure 4.6.



(a) Average CPU usage.



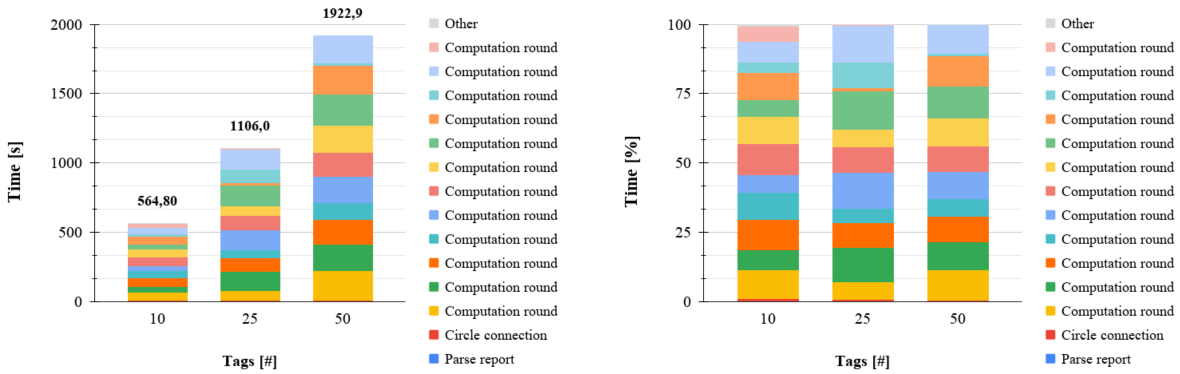
(b) Average Memory usage.

Figure 4.6: System average CPU and memory usage as a function of the maximum number of tags allowed in the security reports for 3 *Players*.

From the results of figure 4.6 we can observe that there is a 3,9470 % increase in average system CPU usage from 10 to 25 maximum tags allowed in *Players* security reports and a 2,2330 % increase from 25 to 50 tags. Similarly to the 2 *Players* experiments, in this group of experiments the average system memory usage also increases as a function of the maximum number of tags allowed in *Players* reports. We verify that there is an increase of 2,5283 % from 10 to 25 tags and an increase of 1,6671 % from 25 to 50 tags.

4.3.3 10 *Players* Experiments

In the 10 *Players* experiments, 5 *Players* were being executed on the first machine in the testbed while the remaining 5 *Players* and the *Dealer* were running on the second machine in the testbed. This distribution of *Players* was performed to split the load among the two machines in the testbed, as a *Circle* session can consume a lot of system resources. With this setting, we increasingly set the maximum number of tags allowed in *Players* security reports to 10, 25 and 50. The time breakdown for the 10 *Players* experiments is represented in figure 4.7.



(a) Time breakdown in seconds. (b) Time breakdown in percentage.

Figure 4.7: Time breakdown as a function of the maximum number of tags allowed in the security reports for 10 *Players*.

From the results of figure 4.7 we can notice some differences on the time breakdowns for each value of the maximum number of tags allowed in *Players* security reports. The same *Computation round* can have very different durations for different number of tags. As described in subsection 3.5.3, we created an algorithm that searches for pairs of *Players* that are available to perform the PSI operation in each round. The order on which the pairs are chosen can change in each *Circle* session, leading to a different order in which the PSI operations between *Players* are performed. This is confirmed by figure 4.7 where almost all *Computation rounds* occupy a very different percentage of the total time for a different number tags. This means that the PSI operation can be executed in different *Computation rounds* for different numbers of tags. However, despite the changing order in which the PSI operations are performed, each *Player* still has to do the same number of PSI computations. In fact, for the 10 *Players* setting there are nine PSI operations to be performed by each *Player* during the *Circle* session.

In table 4.6 we detail the time spent on each PSI operation for 10, 25 and 50 maximum tags allowed in *Players* security reports.

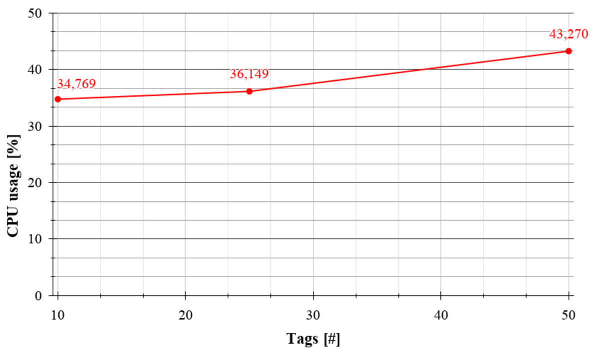
From the results of table 4.6 we can observe that for all three values of the maximum number of tags allowed in *Players* security reports there are four PSI operations whose times are closer to each other and another five PSI operations whose times are closer to each other. For example, in the 10 tags case, the 2nd, 5th, 8th and 9th PSI operations have durations between 22 and 28 s while the remaining five PSI operations have durations between 38 and 47 s. This discrepancy in

Table 4.6: Time spent on the PSI operation for 10, 25 and 50 tags for 10 *Players* (seconds).

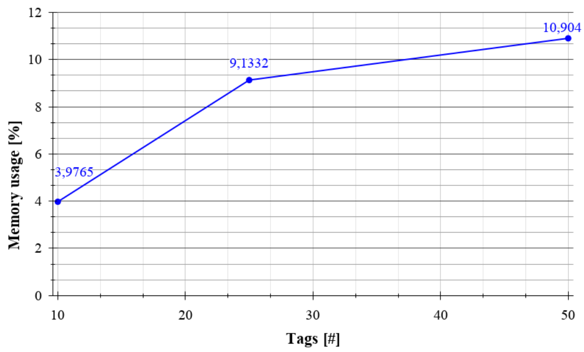
Tags	1st PSI	2nd PSI	3rd PSI	4th PSI	5th PSI	6th PSI	7th PSI	8th PSI	9th PSI	$Mean \pm \sigma$
10	47,38	26,52	41,43	40,10	25,94	38,99	44,43	22,15	27,34	34,92±8,840
25	59,57	86,72	88,49	46,39	90,69	86,26	51,38	92,77	45,35	71,96±19,49
50	199,2	176,3	172,9	104,6	106,2	102,7	109,8	166,3	190,9	147,7±38,52

the durations of the PSI operations is related to which *Players* are performing the PSI operation. During all tests we noticed that *Players* performing PSI in the same machine, like in the case of the 2nd, 5th, 8th and 9th PSI operations, took lesser time than when *Players* performing PSI were in different machines. This is due to the fact that when performing PSI between different machines there are network delays that slow the computation, as in the case of the 1st, 3rd, 4th, 6th and 7th PSI operations. We can verify in table 4.6 that this also happens for the 25 and 50 tags cases.

As described in subsection 4.2.3 we measured the system CPU and memory usage for the 10, 25 and 50 maximum number of tags allowed in *Players* security reports for the 10 *Players* setting. The average system CPU and memory usage for these number of tags is represented in figure 4.8. From the results of figure 4.8 we observe an increase of 1,3800 % in the average CPU



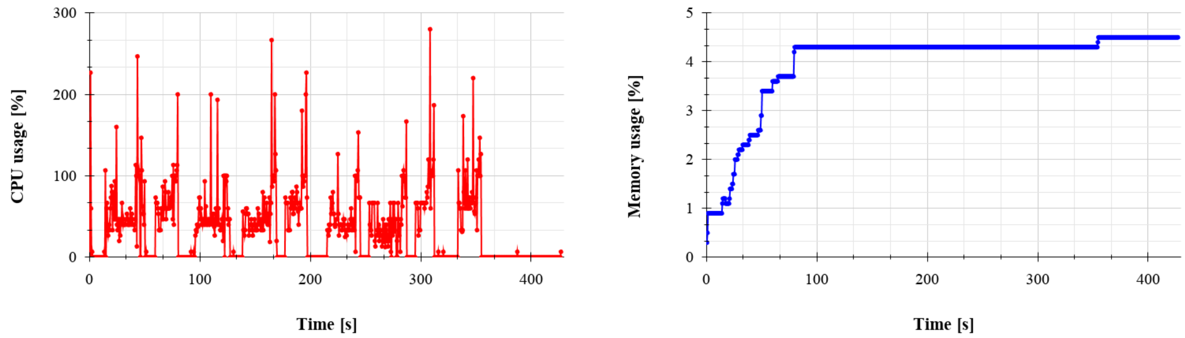
(a) Average CPU usage.



(b) Average Memory usage.

Figure 4.8: System average CPU and memory usage as a function of the maximum number of tags allowed in the security reports for 10 *Players*.

usage from 10 to 25 maximum tags allowed in *Players* security reports. There is an increase of 7,1210 % in average system CPU usage from 25 to 50 tags. With respect to the average system memory usage, we also verify an increase in this system resource metric with the increase of the maximum number of tags allowed in *Players* security reports. There is an increase of 5,1567 % from 10 to 25 tags and an increase of 1,7708 % from 25 to 50 tags. To give an insight on the system CPU and memory usage during the execution of the program of the *Player*, for a fixed number of tags, we present in figure 4.9, the system CPU and memory usage for the 10 *Players* and 10 tags setting.



(a) CPU usage.

(b) Memory usage.

Figure 4.9: System CPU and memory usage for 10 tags and 10 *Players*.

By observing the CPU usage plot in figure 4.9 we can clearly distinguish the periods when the PSI operation is being performed by the *Player* where the measurements were made. In these time intervals there is an increase from near null CPU usage to above 40 %, with several spikes. These nine time intervals, which correspond to the nine PSI operations this *Player* performed, show that PSI is in fact a resource-consuming intensive task, where the CPU usage greatly increases. Concerning the memory usage in this experiment, it increases until 4,3 %, approximately in the 75 s mark, and then it remains constant until the 350 s mark, where it increases to 4,5 %, remaining constant until the end of the end of the program.

4.3.4 Total Measured Time for a Varying Number of *Players*

In figure 4.10 we represent the total execution time of the program of the *Player*, for a varying number of *Players*, for 10, 25 and 50 maximum tags allowed in *Players* security reports. From

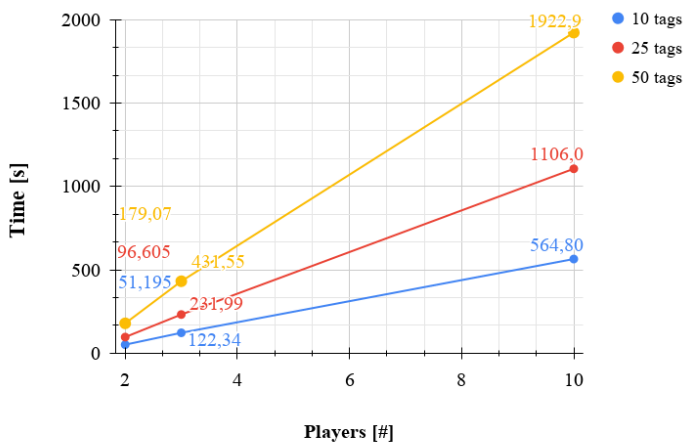
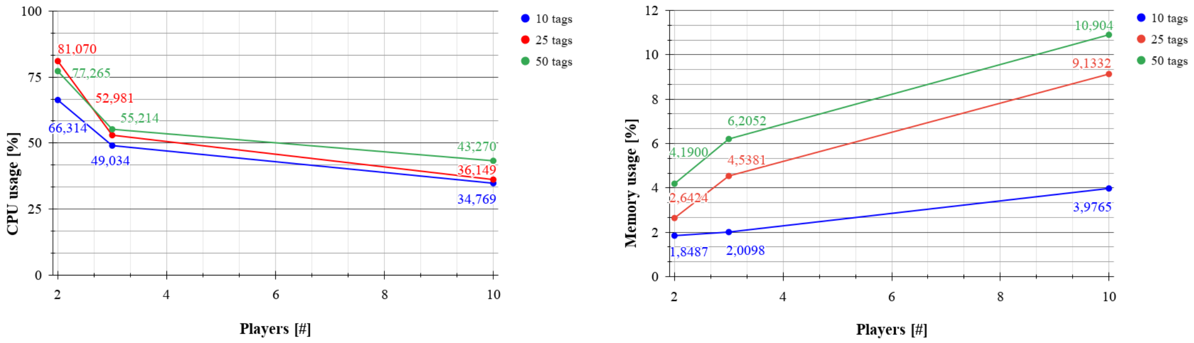


Figure 4.10: Total time execution as a function of the number of *Players*.

the results of figure 4.10 we can conclude that for a fixed number of tags, the total execution time of the program of a *Player* scales linearly with the number of *Players* in the *Circle* session. The shortest duration of a *Circle* session was of 51,195 s, obtained for the experiment with 2 *Players* and a maximum of 10 tags allowed in *Players* reports. For the experiment with 10 *Players* and a maximum of 50 tags allowed in the security reports, the total execution time of the program of the *Player* was of 1922,9 s.

4.3.5 System Resources Metrics for a Varying Number of *Players*

In figure 4.11 we represent the average system CPU and memory usage as a function of the number of *Players* for 10, 25 and 50 maximum tags allowed in *Players* security reports. From



(a) Average CPU usage. (b) Average Memory usage.

Figure 4.11: System average CPU and memory usage as a function of the number of *Players*.

the results of figure 4.11 we can draw several conclusions: in general, the average CPU usage decreases as a function of the number of the *Players* in the *Circle* session. For all three values of the maximum number of tags allowed in *Players* security reports, there is a linear decrease from 2 to 3 *Players* and then a softer, but also linear decrease from 3 to 10 *Players*. This can be explained by the fact that although the number of PSI operations increases with the increase in the number of *Players*, there is more idle time, when the *Player* is not computing the PSI operation with any other *Player*, which significantly lowers the CPU usage during those periods of time and, consequently, the average CPU usage of the program. In the 2 *Players* setting there is almost no idle time, as the *Player* immediately computes the PSI operation with the other *Player*. As the number of *Players* increases so does the time that the *Player* has to wait for other *Players* to finish their respective *Computation rounds*. With respect to the average memory usage, we have seen in subsections 4.3.1, 4.3.2 and 4.3.3 that the memory increases as the maximum number of tags allowed in *Players* security reports increase. Furthermore it also increases as a function of the number of *Players* in the *Circle* session, being the highest average memory usage for the 10 *Players* with 50 maximum tags allowed in *Players* security reports experiment, with 10,904 % and the lowest for the 2 *Players* with 10 tags experiment, with 1,8487 %.

4.4 Discussion

As we have explained in subsection 4.2.4, our experiments were elaborated to address a possible application of the *Circle* component of *PriVeil* in a real-case scenario. Since in the work concerning the *Square* component the expected number of tags per report is approximately 10, we considered that this value would be reasonable for the lower-bound of the maximum number of tags allowed in the *Players* security reports. The upper-bound of 50 tags was defined to try to include all the cases where the number of tags in the reports is bigger than the average. In our experiments, the maximum number of *Players* we considered for a possible *Circle* session was 10. This value seemed reasonable to allow cyber threat information sharing between a restricted group of organizations.

The results obtained for the experiments performed allow us to draw several conclusions. First of all, we verified that in both the 2 and 3 *Players* experiments, the time spent on the PSI operation increases linearly with the maximum number of tags allowed in *Players* security reports. Although the plot for the 10 *Players* case is not presented, table 4.6 shows durations for the PSI operation very similar to those obtained for the 2 and 3 *Players* experiments, shown in figures 4.2 and 4.5. Furthermore, in all experiments we verified that the most intensive task, both in terms of time consumption and system resources usage is the *Computation round* which can include PSI operations, as verified by figures 4.1, 4.4 and 4.7. Secondly, the total time spent on the program of a *Player* increases linearly with the number of *Players* in the *Circle* session, for a fixed maximum number of tags allowed in *Players* security reports. These results allow to estimate how an increasing number of tags and *Players* will affect a session duration in future work.

With regard to the CPU usage, the average system CPU usage, for the 3 and 10 *Players* setting increases linearly with the number of tags. However, for the 2 *Players* setting this is not verified. The average system CPU usage does not represent the CPU usage over time, as the standard deviations are rather high, meaning that during time, the CPU usage greatly deviates from the average value. The result of figure 4.9 confirms this. The results of figure 4.11 show that the average system CPU usage decreases linearly as a function of the number of *Players*. This is justified by the fact that there is more idle time for each individual *Player*, as the number of *Players* increases. This idle time is due to the fact that when the *Player* in question is not performing PSI, it is waiting for the other *Players* to finish, remaining idle.

Finally, as verified by figures 4.3, 4.6 and 4.8, for all experiments, the average system memory usage increases with the increase in the maximum number of tags allowed in *Players* security reports. This is to be expected, since, as shown in figure 4.9, the memory usage of the process of a *Player* generally increases during its duration, remaining constant during periods of time. Furthermore, in figure 4.11 we verified that the average system memory usage increases with the number of *Circle Players*, for a fixed number of tags.

The experiments performed and respective results show that *Circle* is applicable in a real-case scenario. The results of these experiments showed a total execution time of 32,05 minutes for the most intensive experiment with 10 *Players* and a maximum of 50 tags allowed in the security reports. Despite this result being high for a cyber threat information sharing system, we conclude that the results are still acceptable, since we perform a *secure* computation that preserves the *Privacy* of the parties inputs. Regarding the CPU usage of the program of a *Player*, the results obtained show that, although the CPU usage heavily increases during the computation of PSI, it is almost null in the remaining tasks, which allows the program of a *Player* to be easily executed in commodity hardware. Finally, we have observed that the average memory usage increases as a function of both the maximum number of tags allowed in *Players* reports and the number of *Players* in the session. Nevertheless, in all experiments, these results did not have a noticeable impact on the system of the *Player*.

4.5 Summary

In this chapter we analyzed the *Circle* session from the point of view of a *Player*. A *Player* represents a possible organization wanting to share cyber threat information contained in the security report. The experiments and results obtained are important to assess how the *PriVeil Circle* session affects the machine of a *Player*, both from the point of view of system resources consumed and time spent on a *Circle* session.

Chapter 5

Conclusion

In this work, we designed and implemented a prototype of *Circle*, the second component of the cyber threat information sharing system *PriVeil*. By using cryptographic techniques like Secure Multiparty Computation (SMC) and Homomorphic Encryption, *PriVeil* allows participating organizations to completely control how their information is shared and disclosed to other participants. More concretely, in *Circle*, users whose security reports matched in *Square* can share cyber threat information in detail. In order to develop *Circle* we relied on the SMC operation of Private Set Intersection (PSI). In PSI, two or more parties want to compute the intersection of their input sets while guaranteeing the properties of *Privacy* of inputs and *Correctness* of the output. In *Circle*, each *Player* progressively computes the PSI operation on the tags on its security report with the tags in the security reports of every other *Player*. By the end of this round all users know the tags in their security reports which are common to other users security reports, without any information being revealed about the different tags. This information is useful in the context of a cyber threat sharing platform because each *Player* will know the *Players* whose security reports tags had more matches with its own, as well as what the matched tags are, and thus those *Players* might contain the same or a similar cyber threat described in the security report. Despite not implemented, if they wish to, these parties could afterwards engage in another sharing stage, where they disclose the remaining fields in the security report to each other, to facilitate cooperation and find a solution for the described cyber threat(s).

We evaluated *Circle* by performing nine different experiments with a varying number of *Players* and maximum number of tags allowed in the security reports, in which the processing time, CPU and memory usage of a *Player* program were measured during a *Circle* session. The results show that this prototype can be used in real-life scenarios, since it creates an environment where participants can *securely* share the cyber threat information contained in their security reports.

5.1 Achievements

In this work, the following contributions were achieved:

- Design and implementation of *Circle*, the second part of the threat information sharing platform *PriVeil* and where users can share cyber threat data in detail, while retaining full control over their data, only sharing what they are willing to.
- Evaluation of *Circle* according to possible real use case scenarios, which were based on the results obtained for the *Square* component of *PriVeil*.

5.2 Future Work

The current *Circle* prototype can be improved in several aspects to increase both its functionality and performance:

- Improve the *Player* assignment algorithm (described in subsection 3.5.3). Although this algorithm is an effective solution to parallelize PSI operations between different pairs of *Players* instead of having to do PSI one pair at a time, an heuristic approach could be used to select the pairs of *Players* that will compute PSI simultaneously.
- Implement TLS on top of the TCP communications used by FRESCO during the computation of PSI. This would allow the fulfillment of the security requirement SR_2.
- Deploy and test in a real-case scenario with organizations who are willing to participate in a cyber threat information sharing environment. This feedback is essential to improve *Circle* and *PriVeil* since our project is meant to be used by real organizations who want to share cyber threat information on their actual security reports.
- Implement additional steps of cyber threat information sharing. After the PSI stage is terminated, the remaining *Players* in *Circle* can jointly perform a progressive disclosure of the remaining fields in the security reports. This could be achieved by a Layered Encryption scheme, allowing all *Players* to concurrently reveal the same field in the security reports, one by one.
- Integrate the *Square* and *Circle* components of *PriVeil*. Since this work and the work in which *Square* was developed were done separately, we have yet to integrate these two prototypes, in order to conclude the cyber threat information sharing platform *PriVeil*.

Bibliography

[Aum17] Jean-Philippe Aumasson. *Serious Cryptography: A Practical Introduction to Modern Encryption*. No Starch Press, 2017. ISBN:978-1593278267.

[BCK96] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying Hash Functions for Message Authentication. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO '96)*, pages 1–15. Springer-Verlag, 1996.

[Ben87] Josh Cohen Benaloh. Secret Sharing Homomorphisms: Keeping Shares of a Secret Secret. In *Proceedings of Advances in Cryptology (CRYPTO '86)*, volume 263 of *Lecture Notes in Computer Science*, pages 251–260. Springer, Berlin, Heidelberg, 1987.

[BGS15] Sarah Brown, Joep Gommers, and Oscar Serrano. From Cyber Security Information Sharing to Threat Management. In *Proceedings of the 2nd ACM Workshop on Information Sharing and Collaborative Security (WISCS '15)*, pages 43–49. Association for Computing Machinery, 2015. DOI:10.1145/2808128.2808133.

[BKR11] Andrey Bogdanov, Dmitry Khovratovich, and Christian Rechberger. Biclique Cryptanalysis of the Full AES. In *Proceedings of Advances in Cryptology (ASIACRYPT 2011)*, volume 7073 of *Lecture Notes in Computer Science*, pages 344–371. Springer, Berlin, Heidelberg, 2011.

[Bla79] G. R. Blakley. Safeguarding cryptographic keys. In *Proceedings of the International Workshop on Managing Requirements Knowledge (MARK)*, pages 313–318. IEEE, 1979. DOI:10.1109/MARK.1979.8817296.

[BOGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation. In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing (STOC '88)*, pages 1–10. Association for Computing Machinery, 1988. DOI:10.1145/62212.62213.

[Can00] Ran Canetti. Security and Composition of Multiparty Cryptographic Protocols. *Journal of Cryptology*, 13:143–202, 2000.

[CDN15] Ronald Cramer, Ivan Bjerre Damgård, and Jesper Buus Nielsen. *Secure Multiparty Computation and Secret Sharing*. Cambridge University Press, 2015. ISBN:978-1107337756.

[CGMA85] Benny Chor, Shafi Goldwasser, Silvio Micali, and Baruch Awerbuch. Verifiable Secret Sharing and Achieving Simultaneity in the Presence of Faults. In *Proceedings of the 26th Annual Symposium on Foundations of Computer Science (sfcs 1985)*, pages 383–395. IEEE, 1985. DOI:10.1109/SFCS.1985.64.

[CLY17] Long Cheng, Fang Liu, and Danfeng Yao. Enterprise data breach: causes, challenges, prevention, and future directions. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 7, 2017.

[CSF$^+$08] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280, Internet Engineering Task Force, May 2008.

[DNNR17] Ivan Damgård, Jesper Buus Nielsen, Michael Nielsen, and Samuel Ranellucci. The Tinytable Protocol for 2-Party Secure Computation, or: Gate-Scrambling Revisited. In *Proceedings of Advances in Cryptology (CRYPTO 2017)*, volume 10401 of *Lecture Notes in Computer Science*, pages 167–187. Springer, Cham, 2017.

[DR02] Joan Daemen and Vincent Rijmen. *The Design of Rijndael*. Springer, 5^{th} edition, 2002. ISBN:978-3540425809.

[DSZ15] Daniel Demmler, Thomas Schneider, and Michael Zohner. ABY - A Framework for Efficient Mixed-Protocol Secure Two-Party Computation. In *Proceedings of the Network and Distributed System Security Symposium (NDSS '15)*, 2015. DOI:10.14722/ndss.2015.23113.

[EGL82] Shimon Even, Oded Goldreich, and Abraham Lempel. A Randomized Protocol for Signing Contracts. *Communications of the ACM*, 28(6):205–210, 1982. DOI:10.1145/3812.3818.

[EKR18] David Evans, Vladimir Kolesnikov, and Mike Rosulek. A Pragmatic Introduction to Secure Multi-Party Computation. *Foundations and Trends in Privacy and Security*, 2, 2018. DOI:10.1561/3300000019.

[FAP+15] Gina Fisk, Calvin Ardi, Neale Pickett, John Heidemann, Mike Fisk, and Christos Papadopoulos. Privacy Principles for Sharing Cyber Security Data. In *Proceedings of the IEEE Symposium of Security and Privacy Workshops (SPW)*, pages 193–197. IEEE, 2015. DOI:10.1109/SPW.2015.23.

[FIPR05] Michael J. Freedman, Yuval Ishai, Benny Pinkas, and Omer Reingold. Keyword Search and Oblivious Pseudorandom Functions. In *Proceedings of Theory of Cryptography (TCC 2005)*, volume 3378 of *Lecture Notes in Computer Science*, pages 303–324. Springer, Berlin, Heidelberg, 2005.

[FNP04] Michael J. Freedman, Kobbi Nissim, and Benny Pinkas. Efficient Private Matching and Set Intersection. In *Proceedings of Advances in Cryptology (EUROCRYPT 2004)*, volume 3027 of *Lecture Notes in Computer Science*, pages 1–19. Springer, Berlin, Heidelberg, 2004.

[GKF+06] Scott Garriss, Michael Kaminsky, Michael J. Freedman, Brad Karp, David Mazières, and Haifeng Yu. RE: Reliable Email. In *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation (NSDI 2006)*, 2006.

[Gon19] Tiago Gonçalves. Priveil: Privacy-preserving threat information sharing. Master's thesis, Instituto Superior Técnico, December 2019.

[GS18] Sanjam Garg and Akshayaram Srinivasan. Two-Round Multiparty Secure Computation from Minimal Assumptions. In *Proceedings of Advances in Cryptology (EUROCRYPT 2018)*, volume 10821 of *Lecture Notes in Computer Science*, pages 468–499. Springer, Cham, 2018.

[HCE11] Yan Huang, Peter Chapman, and David Evans. Privacy-Preserving Applications on Smartphones. In *Proceedings of the 6th USENIX Workshop on Hot topics in security (HotSec '11)*, volume 8, 2011.

[HEK12] Yan Huang, David Evans, and Jonathan Katz. Private Set Intersection: Are Garbled Circuits Better than Custom Protocols? In *Proceedings of the 19th Network and Distributed Security Symposium (NDSS 2012)*, 2012.

[IS19] Ponemon Institute and IBM Security. Cost of a Data Breach Report. Technical report, IBM Security, 2019.

[JBW+16] Christopher S. Johnson, Mark L. Badger, David A. Waltermire, Julie Snyder, and

Clem Skorupka. Guide to Cyber Threat Information Sharing. Special Publication 800-150, National Institute of Standards and Technology, 2016.

[KL14] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. CRC Press, 2^{nd} edition, 2014. ISBN:978-1466570269.

[KMP+17] Vladimir Kolesnikov, Naor Matania, Benny Pinkas, Mike Rosulek, and Ni Trieu. Practical Multi-party Private Set Intersection from Symmetric-Key Techniques. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*, pages 1257–1272. Association for Computing Machinery, 2017. DOI:10.1145/3133956.3134065.

[KS08] Vladimir Kolesnikov and Thomas Schneider. Improved Garbled Circuit: Free XOR Gates and Applications. In *Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP 2008)*, volume 5126 of *Lecture Notes in Computer Science*, pages 486–498. Springer, Berlin, Heidelberg, 2008.

[MvOV96] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1^{st} edition, 1996. ISBN:978-0849385230.

[NIS02] NIST. Secure hash standard (SHS). FIPS Publication 180-2, National Institute of Standards and Technology, August 2002.

[NLV11] Michael Naehrig, Kristin Lauter, and Vinod Vaikuntanathan. Can Homomorphic Encryption Be Practical? In *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop (CCSW '11)*, pages 113–124. Association for Computing Machinery, 2011. DOI:10.1145/2046660.2046682.

[NNOB12] Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A New Approach to Practical Active-Secure Two-Party Computation. In *Proceedings of Advances in Cryptology (CRYPTO 2012)*, volume 7417 of *Lecture Notes in Computer Science*, pages 681–700. Springer, Berlin, Heidelberg, 2012.

[NS08] Arvind Narayanan and Vitaly Shmatikov. Robust De-anonymization of Large Sparse Datasets. In *Proceedings of the IEEE Symposium on Security and Privacy (sp 2008)*, pages 111–125. IEEE, 2008. DOI:10.1109/SP.2008.33.

[Orl11] Claudio Orlandi. Is Multiparty Computation Any Good In Practice? In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5848–5851. IEEE, 2011. DOI:10.1109/ICASSP.2011.5947691.

[Pai99] Pascal Paillier. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In *Proceedings of Advances in Cryptology (EUROCRYPT '99)*, volume 1592 of *Lecture Notes in Computer Science*, pages 223–238. Springer, Berlin, Heidelberg, 1999.

[PSTY19] Benny Pinkas, Thomas Schneider, Oleksandr Tkachenko, and Avishay Yanai. Efficient Circuit-Based PSI with Linear Communication. In *Proceedings of Advances in Cryptology (EUROCRYPT 2019)*, volume 11478 of *Lecture Notes in Computer Science*, pages 122–153. Springer, Cham, 2019.

[PSZ18] Benny Pinkas, Thomas Schneider, and Michael Zohner. Scalable Private Set Intersection Based on OT Extension. *ACM Transactions on Privacy and Security*, 21(2), 2018. DOI:10.1145/3154794.

[Res00] E. Rescorla. HTTP Over TLS. RFC 2818, Internet Engineering Task Force, May 2000.

[Res18] E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, Internet Engineering Task Force, August 2018.

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978. DOI:10.1145/359340.359342.

[Sha79] Adi Shamir. How to Share a Secret. *Communications of the ACM*, 22(11):612–613, 1979. DOI:10.1145/359168.359176.

[Sta10] William Stallings. *Cryptography and Network Security*. Pearson, 5^{th} edition, 2010. ISBN:978-0137056323.

[TW15] Biaoshuai Tao and Hongjun Wu. Improving the Biclique Cryptanalysis of AES. In *Proceedings of the Australasian Conference on Information Security and Privacy (ACISP 2015)*, volume 9144 of *Lecture Notes in Computer Science*, pages 39–56. Springer,Cham, 2015.

[vWTS$^+$18] Rolf van Wegberg, Samaneh Tajalizadehkhoob, Kyle Soska, Ugur Akyazi, Carlos Gañán, Bram Klievink, Nicolas Christin, and Michel van Eeten. Plug and Prey? Measuring the Commoditization of Cybercrime via Online Anonymous Markets. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security 18)*, pages 1009–1026. USENIX Association, 2018.

[WDWI16] Cynthia Wagner, Alexandre Dulaunoy, Gérard Wagener, and Andras Iklody. MISP - The Design and Implementation of a Collaborative Threat Intelligence Sharing Platform. In *Proceedings of the 3rd ACM Workshop on Information Sharing and Collaborative Security (WISCS 2016)*, pages 49–56. Association for Computing Machinery, 2016. DOI:10.1145/2994539.2994542.

[Yak17] Sophia Yakoubov. A Gentle Introduction to Yao's Garbled Circuits. Massachusetts Institute of Technology, 2017.

[Yao82] Andrew C. Yao. Protocols for Secure Computations. In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science (sfcs 1982)*, pages 160–164. IEEE, 1982. DOI:10.1109/SFCS.1982.38.

[Yao86] Andrew C. Yao. How to generate and exchange secrets. In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*, pages 162–167. IEEE, 1986. DOI:10.1109/SFCS.1986.25.

[ZRE15] Samee Zahur, Mike Rosulek, and David Evans. Two Halves Make a Whole. In *Proceedings of Advances in Cryptology (EUROCRYPT 2015)*, volume 9057 of *Lecture Notes in Computer Science*, pages 220–250. Springer, Berlin, Heidelberg, 2015.

[Zú18] André Zúquete. *Segurança em Redes Informáticas.* FCA, 5^{th} edition, 2018. ISBN:978-9727228577.