TÉCNICO LISBOA

Programmable Sandbox for Malware Analysis

Diogo Filipe dos Santos Vilela

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisor: Prof. Miguel Nuno Dias Alves Pupo Correia

Examination Committee

Chairperson: Prof. Daniel Jorge Viegas Gonçalves
Supervisor: Prof. Miguel Nuno Dias Alves Pupo Correia
Member of the Committee: Prof. Pedro Ricardo Morais Inácio

September 2021

Acknowledgments

First and foremost I would like to thank my advisor Professor Miguel Pupo Correia for giving me the opportunity to work on this topic with him and for his patience and guidance throughout the course of this dissertation.

I would also like to thank my family who always supported me unconditionally and without whom I wouldn't be where I am today. To my friends Franscisco Barros and Rafael Ribeiro, thank you for a great friendship and great moments since the first year of university that I am certain we will cherish for many years to come. To Andreia Valente who I also had the luck to meet in these academic years and is an invaluable part of my life, I would like to specially thank you for all the revisions of the not-so-good-to-look-at text, for bullying me into power through the most difficult times and for making me a better person.

Last but not least, I would like to thank VirusTotal for granting me access to their folder of malware samples, which was then used for the design of the evaluation experiment.

Abstract

Ransomware has grown to be one of the largest cybersecurity threats in the past few years, so efforts in building better detection mechanisms have also increased. Ransomware's mainstream approach of encrypting a large amount of files, leads to the rationale of using dynamic malware analysis to build detection models as they produce file access patterns that could be used as features for the training of said models.

However, it has been reported that papers published in this research area have seen a lack of scientific rigor due to absence of proper experiment and result reporting that can be attributed to the friction of properly log experiments alongside fast research approach iteration.

In the effort to reduce the mentioned friction, this dissertation presents the Programmable Sandbox for Malware Analysis (PSMA), a scalable dynamic malware analysis system that enables researchers to perform programmable and repeatable experiments, by using a modular approach and storing the modules used and parameters configured in each experiment, which can later be used for the creation of proper reports on new tested approaches.

Keywords

Dynamic Malware Analysis; Ransomware; Sandbox;

Resumo

Nos últimos anos, *Ransomware* tem vindo a crescer para ser uma das maiores ameaças em ciber segurança e como tal os eforços em construir melhores mecanismos de deteção têm também aumentado. A principal tática usada em *Ransomware* de cifrar grandes quantidades de ficheiros, leva ao pensamento de usar técnicas de análise dinâmica de *malware* para desenvolver modelos de deteção, uma vez que esta produz padrões de acesso aos ficheiros que podem ser usados como características no treino de tais modelos.

No entanto, tem vindo a ser reportado que artigos nesta área de investigação têm tido falta de rigor científico devido à falta de relatórios adequados sobre as experiências feitas e resultados obtidos. Esta falta pode ser atribuida à fricção existente entre devidamente registar as experiências ao mesmo tempo que existe uma rápida iteração dos métodos usasdos.

Com o objectivo de reduzir a fricção mencionada, esta dissertação apresenta *Programmable Sandbox for Malware Analysis (PSMA)*, um sistema escalável de análise dinâmica de *malware* que permite aos investigadores executar experiências programáveis e repetíveis, fazendo uso de uma abordagem modular e o armazenando dos módulos usados e parâmetros configurados em todas as experiências, que mais tarde podem ser usados para a criação de relatórios adequados sobre novas abordagens testadas.

Palavras Chave

Análise Dinâmica de *Malware*; *Ransomware*; *Sandbox*

Contents

1 Introduction **1**

1.1 Motivation 3

1.2 Goals and Contributions 4

1.3 Thesis Outline 5

2 Related Work **7**

2.1 Ransomware Taxonomy 9

2.1.1 Scareware 9

2.1.2 Locker-Ransomware 10

2.1.3 Crypto-Ransomware 10

2.2 Malware Analysis 12

2.2.1 Static Malware Analysis 15

2.2.2 Dynamic Malware Analysis 16

2.3 Reproducibility 22

3 Proposed System **25**

3.1 Requirements 27

3.1.1 Enable Reproducibility 27

3.1.2 Modular 27

3.1.3 Scalable 28

3.1.4 Isolated 28

3.2 Architecture 29

3.2.1 High-Level View 29

3.2.2 Host 29

3.2.3 Message Broker 30

3.2.4 Storage 30

3.2.5 Data Collection System 30

3.2.6 Data Processing System 31

3.3 Implementation 32

3.3.1 Storage System 33
3.3.2 Message Broker 34
3.3.3 Data Collection System 35
3.3.4 Data Processing System 36
3.3.5 Host 36
3.4 Usage 38
3.4.1 System Setup 38
3.4.2 Experiment Execution 38
4 Evaluation **45**
4.1 Evaluation Design 47
4.2 Evaluation Results 48
5 Conclusions **53**
5.1 Summarized Contribution Analysis 55
5.2 Future Directions 55

List of Figures

2.1 Al-rimy's Ransomware Taxonomy [1] 9
2.2 Windows PE file, the number of sections is variable, shown two common sections [2] . . . 15
2.3 Hypervisor Types [3] 20

3.1 PSMA's High-Level Architecture 29
3.2 PSMA's Storage 30
3.3 Data Collection System Architecture 31
3.4 Data Processing System Architecture 32
3.5 VMI Input YAML File 39
3.6 VMI Output YAML File 39
3.7 DCS Module Input YAML File 40
3.8 DCS Module Output YAML File 40
3.9 DPS Module Input YAML File 41
3.10 DPS Module Output YAML File 41
3.11 Experiment Input YAML File 42
3.12 Experiment Output YAML File 43
3.13 Experiment Status YAML File 44

4.1 DCS Task Execution Time Time-series 49
4.2 DCS Task Execution Time Histogram 50
4.3 Monitoring Dashboard 51

List of Tables

3.1 PSMA's endpoints 37

4.1 Evaluation System's Specifications 47
4.2 Experiment Tasks' Statistics 48

List of Algorithms

2.1 Cipher victim's targeted files 13
2.2 Decipher victim's files 13

Acronyms

ACR Asymmetric Crypto-Ransomware

AES Advanced Encryption Standard

API Application Programming Interface

C2 Command and Control

CPU Central Processing Unit

DCS Data Collection System

DDOS Distributed Denial of Service

DES Data Encryption Standard

DLL Dynamic Loaded Library

DPS Data Processing System

ECC Elliptic Curve Cryptography

HCR Hybrid Key Crypto-Ransomware

IAT Import Address Table

JSON JavaScript Object Notation

MFT Master File Table

NTFS New Technology File System

OS Operating System

OVA Open Virtualization Format

PAAS Platform As A Service

PCAP Packet Capture

PE Portable Executable

PSMA Programmable Sandbox for Malware Analysis

RC	Rivest Cipher
RSA	Rivest-Shamir-Adleman
SCR	Symmetric Crypto-Ransomware
SFTP	SSH File Transfer Protocol
SHA	Secure Hash Algorithm
SMB	Server Message Block
SSDT	System Service Descriptor Table
SSH	Secure Shell
UUID	Universally Unique Identifier
VMDK	Virtual Machine Disk
VMI	Virtual Machine Image
VMM	Virtual Machine Monitor
VM	Virtual Machine
YAML	YAML Ain't Markup Language
YARA	Yet Another Recursive/Ridiculous Acronym

1

Introduction

Contents

1.1 Motivation 3
1.2 Goals and Contributions 4
1.3 Thesis Outline 5

This chapter serves as an introduction to this dissertation, starting by presenting the motivation for the study of ransomware and how malware analysis research can benefit from the implementation of proper research methods in Section 1.1. An overview of the goals of and contributions made by this project is provided in Section 1.2. Lastly, the remainder's structure of the document is presented in Section 1.3.

1.1 Motivation

Over the past few years, the population's increasing dependency on computer systems has created new opportunities for malicious actors to take advantage and disrupt a now vulnerable society. Ranging from cybersecurity enthusiasts to nation sponsored teams, various malicious actors develop and deploy malicious software (also known as malware) with that intent. As a result, the number of distinct malware artifacts has increased significantly, currently being submitted and detected more than 500 thousand different samples daily on Virus Total [4], a platform that aggregates the analysis from multiple antivirus engines.

Ransomware, a specific type of malware, is considered to be one of the biggest cibersecurity threats currently. Its main approach is to encrypt users' files and request a payment in order to unlock said files and although not a new technique (dating as early as 1989 with the AIDS Trojan [5]), only in the recent years has it become a major cybersecurity concern, as new emerging technologies keep providing more ways of further assisting attackers to remain anonymous while, at the same time, facilitating the payment process with the ever-increasing popularity of cryptocurrencies [6].

A major example of a ransomware attack, is the infamous WannaCry Ransomware Attack which appeared in May 2017 and that brought up the public's attention and interest to ransomware's damaging capabilities [7] as it is estimated to have imposed around 4 billion USD in losses. This attack, which is considered to be the biggest ransomware attack to date, profiting the attackers around 54 Bitcoins (or ∼1.900 million USD at current exchange rate), was responsible for disrupting the United Kingdom's National Health Service by infecting both staff computers and medical devices. Many of these devices were found to be running outdated operating systems, lacking the patches necessary for stopping the attack which were already deployed in newer versions. By analysing the ransomware samples found in the wild, researchers identified two main modules inside WannaCry: one, which was responsible for the encryption, used a hybrid key encryption method based on Rivest-Shamir-Adleman (RSA) 2048-bit key pairs and Advanced Encryption Standard (AES) 128-bit keys. The other module was responsible for the propagation of the samples to systems in the same network as the victim. The module operates by starting a search for the DoublePulsar backdoor on the system, and if not present, it then attempts to leverage the Server Message Block (SMB) v1 service using an exploit called EternalBlue, which, like

the DoublePulsar exploit, was developed by the National Security Agency and leaked in April 2017 by the Shadow Brokers threat group.

Due to ransomware usually performing encryption on a large portion of files in a system, patterns of increased disk access rates and high volume file modifications can be used to identify the actions of an infection by ransomware. This suggests the use of dynamic malware analysis techniques where one executes a large set of ransomware samples in a controlled and monitored environment in order to extract these patterns and build detection models by using machine learning approaches.

Although the importance and use of dynamic malware analysis, not exclusively while studying ransomware, has increased in the last years and many works have been proposed in this research area, many of them fail to properly report their findings. This leads to lack of scientific rigor by hindering the efforts of validating such works and building new approaches based upon them, which overall introduces delays in the evolution of this research area. Rossow et al. [8] reviewed 36 academic publications and found frequent shortcomings in terms of insufficient description of the experimental setup and presented guidelines regarding transparency, realism, correctness, and safety that research in the malware analysis area should follow. The shortcomings presented can be traced back to the urgent and rapid development of new approaches combined with the lack of tools that streamlines the reporting process without causing low friction in the process of experimenting new approaches.

1.2 Goals and Contributions

The main goal of this dissertation is to present the Programmable Sandbox for Malware Analysis (PSMA), a scalable dynamic malware analysis system that enables researchers to perform programmable and repeatable experiments. Additionally, this system aims to promote collaboration between members of the research community by having a modular design at the core of the procedures run in each experiment.

Building upon existing works, the approach taken in the design of the system was to build a pipeline-like architecture where the experiment starts by sending the samples to be analysed to the Data Collection System (DCS) where the samples are executed in Virtual Machines (VMs), which are defined by Virtual Machine Images (VMIs) provided by the researcher, and their behaviour logged. Then the data collected is sent to the Data Processing System (DPS), where a processing module, also provided by the researcher, processes the collected data inside a container, allowing for dependencies for the module passed to be met without altering the software/configurations in the system. Finally, the results and parameters used in the experiment are properly stored so that proper reports of the experiments run and approaches used can be published, promoting better practices in the research community.

Additionally in order to make PSMA useful for both low and high scale malware analysis, both the DCS and the DPS were designed to be scalable, since they are the most resource-intense parts of the

system. Finally, and in light of the goal of contribution in the research community, it was also developed a versioned module and virtual machine image storage solution that allows researchers with access to the system to create and update modules that then can be used by other researchers whilst maintaining a previous version of those modules that have possibly been used in experiments reported or worth reporting.

1.3 Thesis Outline

The remainder of the document is structured as follows. Chapter 2 presents the background necessary on the problem at hand and discusses the developed work in the area. Chapter 3 presents the requirements, architecture, technological implementation and how to use the proposed system. Chapter 4 describes the evaluation performed and its results. Finally, Chapter 5 concludes the dissertation by summarizing the work done and discussing future work.

2

Related Work

Contents

2.1 Ransomware Taxonomy 9
2.2 Malware Analysis 12
2.3 Reproducibility 22

This chapter presents an overview of some major contributions in the areas of ransomware analysis and reproducibility of malware experiments. The remaining chapter is divided as follows: Section 2.1 provides a study of the different types of ransomware, presenting their characteristics and relative usage nowadays. Section 2.2 proceeds to compare and present the two major types of malware analysis techniques (static and dynamic analysis). At last, Section 2.3 focuses on the problem of reproducibility of malware analysis experiments and provides the requirements needed to build a system that enables the adoption of good practices.

2.1 Ransomware Taxonomy

As it naturally happens with every family of malware, with the crescent use and development of new types of ransomware, the need to formally categorize the multiple samples found every day also increased. In response to this need, Al-rimy et al. [1] gathered the work done in literature and proposed a combined ransomware taxonomy categorizing ransomware based on its severity, platform and target as presented in Figure 2.1.

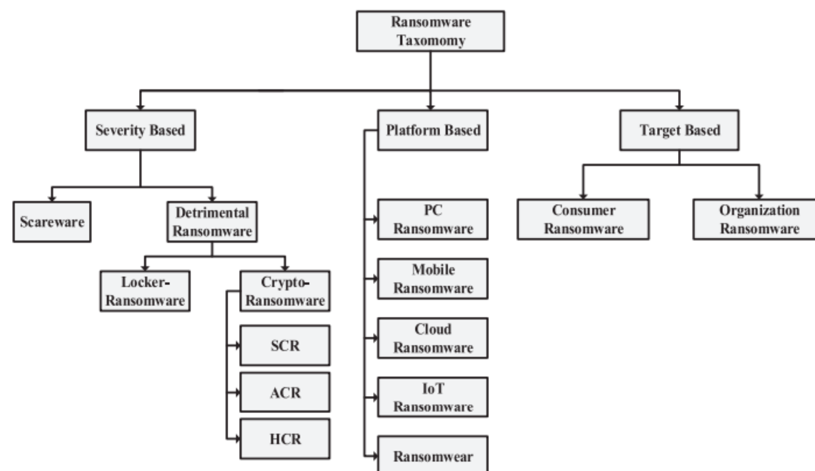

Figure 2.1: Al-rimy's Ransomware Taxonomy [1]

For the remaining of this Section, the focus will be to identify the different techniques applied by ransomware which classifies it into the different severity-based classes presented.

2.1.1 Scareware

Starting with Scareware, this type of ransomware deceits victims into paying for a service that will remove a fake threat from their system. Due to the fact that this type of ransomware does not directly affect the availability of the system, it is not considered to be detrimental ransomware.

Many samples of Scareware can also be identified as Adware, as their main approach to trick the victim is by displaying an ad on their computer with technical jargon that less tech-savvy users may not fully understand and then prompting the victim to install an antivirus to solve the fake presented problem.

Apart from the money spent on a fake service, a successful attack of this kind of ransomware can lead to more serious security problems as by the end of the attack and in order to solve the advertised problems, the user expects to install/run software, usually with administrative rights, which then can be converted to an entrypoint for other types of malware.

2.1.2 Locker-Ransomware

Now considering detrimental ransomware classes and starting with Locker-Ransomware, this class of ransomware forces the victim to pay the ransom by, as the name suggests, locking them out of the system. This is usually achieved by displaying, in a loop, the ransom message and only allowing actions related to the payment of the ransom to be performed.

A commonly adopted technique by Locker-Ransomware on Windows Operating Systems (OSs) is to create a new virtual desktop where a full-screen window is displayed with the ransom message. In order to prevent the user from terminating the locker process, it's a common approach to spawn background threads to monitor and terminate processes that could interfere with the attack, such as the `Taskmgr.exe` (i.e. Task Manager).

Another common – but way less effective – approach is to use the browser as the locker mechanism. This technique uses JavaScript to ”lock” the victim on the locker page by registering hooks to events that indicate the user is trying to leave the page, such as the `onbeforeunload` event that is triggered when the page is about to be unloaded. As stated above, this technique is much less effective as the process has little to no control over the system outside of the browser. However, due to its simplicity and overall cross-platform capabilities, it is still used by some attackers.

2.1.3 Crypto-Ransomware

Focusing now on the most prominent group of ransomware nowadays, Crypto-Ransomware pressures the victim to pay the ransom by declining them access to their files. In order to achieve this objective, the ransomware makes use of cryptographic methods to render the information in the targeted files intelligible. These targeted files usually don't interfere with the system's operation so that the same system can be used to pay the ransom. Regarding the methods used to cipher the files, they can make use of primitives offered by the operating system such as those given by the `CryptoAPI` on the Windows operating systems or can be custom made to hinder the detection process.

The ever-rising popularity of this ransomware group can be explained by its damaging capabilities

and irreversibility when properly implemented, as the data on the targeted files can be of major value and be irrecoverably lost if payment for the decryption key does not occur. This allows attackers to change their target audience to high-value systems, from normal users' computers to enterprises' infrastructures which allows attackers to demand much higher ransom values.

Al-rimy et al. [1] categorized further down this ransomware's class by the type of key used in the encryption process.

2.1.3.A Symmetric Crypto-Ransomware (SCR)

Taking a look into the resulting classes from a chronological point of view, we start by analyzing Symmetric Crypto-Ransomware which uses, as the name implies, symmetric key algorithms to perform the encryption and decryption process, such as Advanced Encryption Standard (AES), Data Encryption Standard (DES) and Rivest Cipher (RC)4.

The use of symmetric key algorithms poses two main advantages and also a challenge which were, respectively, responsible for its early adoption and its abandonment by the early ransomware authors. The two main advantages of these algorithms are their performance, which turns detection of the attack in time to stop it rather difficult, and their simplicity in comparison with their asymmetric key counterparts. However, the fact that only one key is used in this type of algorithm poses the challenge of keeping this key accessible to the attacker's process while keeping it secret from the victim. Many approaches to tackle this problem were proposed, such as creating the key prior to the creation of the ransomware and storing it obfuscated inside the ransomware itself or generating it in runtime and sending it to a Command and Control (C2) server. However, researchers have eventually found methods to undermine these techniques forcing attackers to change their approach. A notable example of this type of ransomware is GPCode.

2.1.3.B Asymmetric Crypto-Ransomware (ACR)

Adapting to the advances made in defeating SCR techniques, malware authors ended up developing a new type of ransomware called Asymmetric Crypto-Ransomware which was based on asymmetric key algorithms that instead of only using one secret key for the encryption and decryption processes, used a pair of private/public keys where the public key was used to encrypt the targeted files whereas the private key was used in the decryption process. These algorithms, such as Rivest-Shamir-Adleman (RSA) or Elliptic Curve Cryptography (ECC), offer the advantage that only the public key is needed in the victim's system before the payment of the ransom.

However, the disclosure of the private key, that could be obtained when paying the ransom, would render future attacks ineffective. Rapidly ransomware authors created countermeasures to tackle this challenge, such as creating a pool of ransomware samples, each with a unique private/public key pair,

where the public key would reside inside the sample itself, ensuring that at least one payment would exist for each sample in the pool. Another very common technique is to make the victim's system request the attacker's server for a public key, making it possible to generate a unique private/public key pair for each victim. However this technique has the disadvantage of requiring the victim's computer to be able to communicate with the attacker's server at infection time, otherwise, the encryption process would not start.

Even with the ability to keep the private key secret and hence making the attack irreversible without the payment of the ransom, this technique suffers from a major drawback when compared against symmetric techniques: asymmetric encryption algorithms perform much slower, leading to a higher probability of detection before the encryption process ends.

A notable example of this type of ransomware is the GPCoder.F, an evolution of the previously mentioned GPCode, being one of the first ACR samples to make use of an RSA-1024 bit public key.

2.1.3.C Hybrid Key Crypto-Ransomware (HCR)

Analyzing the advantages and drawbacks of the previously mentioned classes, ransomware authors combined them, creating a hybrid approach, the Hybrid Key Crypto-Ransomware which is the most commonly developed and deployed type of crypto-ransomware nowadays.

The approach used in HCR, aims to have the robustness of ACR brought by the use of public/private key pairs whilst having the speed of symmetric key algorithms present in SCR. To this end, and as presented in Algorithm 2.1, for each file to be encrypted it is generated a secret key with which the file will be encrypted. What distinguishes this approach from SCR is the fact that those keys are then encrypted by a public key generated on the client. Regarding the correspondent private key, it is encrypted using the Server's public key at generation time, hence closing the chain.

With the system infected and assuming no flaws in the ransomware's implementation, the only way to recover the files is to decipher the Client's private key, which is the ”service” offered by the attackers in exchange for the ransom. By paying the ransom and having proof of said payment, the victim is then instructed to use a tool, previously installed by the ransomware, that begins by deciphering the Client's private key, which allows for deciphering the secrets keys that are finally used to recover the targeted files as can be seen in Algorithm 2.2.

2.2 Malware Analysis

As stated in Chapter 1, so that proper protection mechanisms can be built against the threats posed by malware, it is essential for proper malware analysis to be developed to both help in the study of malware samples by malware researchers and assist on malware automatic detection.

Algorithm 2.1: Cipher victim's targeted files

```
Cipher (S_pub, F)
    inputs: The server's public key S_pub
            The targeted files F
    C_priv, C_pub ← GenerateKeyPair();
    E_C_priv ← AsymCipher(S_pub, C_priv);
    WriteToDisk(E_C_priv);
    List_E_CS ← {};
    foreach file f ∈ F do
        C_S ← GenerateSecretKey();
        E_f ← SymCipher(C_S, f);
        E_CS ← AsymCipher(C_pub, C_S);
        List_E_CS ← List_E_CS + E_CS;
        WriteToDisk(E_f);
        DeleteFromDisk(f);
    WriteToDisk(List_E_CS);
```

Algorithm 2.2: Decipher victim's files

```
Decipher (E_C_priv, E_F, List_E_CS, Proof)
    inputs: The ciphered client private key E_C_priv
            The ciphered files E_F
            The ciphered secret keys List_E_CS
            The ransom payment proof Proof
    C_priv ← DecipherPrivateKey(E_C_priv, Proof);
    foreach ciphered file E_f ∈ E_F do
        E_CS ← GetCipheredSecretKey(List_E_CS, E_f);
        C_S ← AsymDecipher(C_priv, E_CS);
        f ← SymDecipher(C_S, E_f);
        WriteToDisk(f);
        DeleteFromDisk(E_f);
    DeleteFromDisk(E_C_priv);
    DeleteFromDisk(List_E_CS);
```

Some of these malware analysis techniques allow for the extraction of a malware sample's goal when infecting a system and the steps it takes to accomplish its objectives, which is information that then can be used to build rules for early detection of malicious operations being performed on infected systems and, ultimately, assist on the removal of such threats for those systems. Others, by studying the code structure/style, libraries/toolkits used and behaviours/techniques, enable researchers to relate samples to previous and/or current attacks, assisting on the categorization and determination of authorship of those samples allowing for legal actions to be taken against their authors when required.

Finally and probably most importantly, data extracted from malware analysis allows for the development of models able to properly identify never-seen samples as malware, creating a layer of protection for systems vulnerable to such new threats. These models can be built automatically, for example, based on previously observed behaviour [9, 10], but also manually by malware researchers when finding common sections in many positive samples and then creating, for example, rules for Yet Another Recursive/Ridiculous Acronym (YARA) which is a tool created in 2008, designed for malware analysis/detection by using pattern matching of the rules defined against code and memory sections of the samples [9].

In order to build these models, be it manually or automatically, and for them to perform reasonably well, many malware samples are needed. Depending on the entity that performs the analysis and further collects the data, those samples can come from various sources. If an organization is collecting samples from attacks against their network, those samples can come from network sensors that can detect data coming from low-reputation sites or in e-mail attachments. Not exclusively to, but in the case of such analysis being done in the academic research scenario, the samples can be obtained from the community. This community includes organizations, other academic research studies and companies specializing in malware analysis. When dealing/sharing such samples, it is necessary to take precautions so that confidential information is not shared and that whoever receives those samples can perform the analysis safely.

Due to the huge volume of malware being released every day, analysis techniques often have to deal with the trade-off between accuracy and resources spent, which can be observed when looking at manual malware, that although their results are usually very good, it is not feasible to apply it at large scale. It is then mandatory for large scale analysis that the techniques used are, most importantly, automated but also fast and scalable.

In the rest of this Section we will study the two main categories of automated malware analysis. Section 2.2.1 presents an overview of Static Malware Analysis techniques and the main challenges faced by these type of approaches. Section 2.2.2 presents common approaches for Dynamic Malware Analysis regarding the techniques used for behavioural data collection and possible execution environments, finalizing with the main challenges faced when performing this type of analysis.

2.2.1 Static Malware Analysis

Static analysis is the study and prediction of a program's behaviour by analysing its code be it binary or source. When applied in the field of Malware Analysis these techniques fall into the category of Static Malware Analysis and mostly deal with the binary representation of malware.

Depending on the type of post-collection data processing done, various types of data can be extracted from the samples. For example, the command-line tool `strings`, which extracts from any file the sequence formed by three or more ASCII and Unicode characters, can be used to identify which functions and libraries are imported in a malware sample or help the identification of C2 servers as they commonly are stored inside the samples as strings. Another common approach, is generating logical code blocks by analysing `CALL`, `JMP`, etc. assembly instructions. When paired with data tainting analysis approaches, these techniques allow for the production of data flow graphs.

Focusing on the Portable Executable (PE) file format, which is used on Windows in `.exe` and `.dll` files, and has the structure depicted in Figure 2.2, Wicherski [11] proposed an hash function to be applied to files that follow the PE file format, that can be useful for clustering malware by basing it on the structural data found in the file headers and the structural information about the executable's section data. This function produces a main hash value by considering the Windows Specific Fields that are present on the Optional Header and for each Section defined in the Section Table it is produced a sub hash value. This approach allows for high level clustering by using the produced main hash and for finer grained clustering by employing the sub hashes calculated.

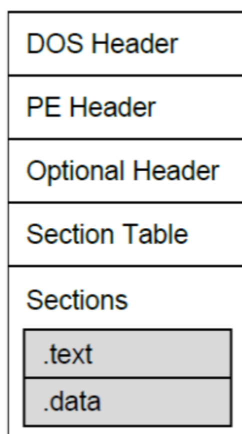

Figure 2.2: Windows PE file, the number of sections is variable, shown two common sections [2]

Also using data stored in the file header, ImpHash [12] looks into the Import Address Table (IAT) located in the Optional Header and computes the hash of its contents. This enables clustering of samples that import from the same libraries/files, which usually happens in malware samples related to the same family. However, a major disadvantage of this method is that the order by which the imports are done in the source code matters for the order presented in IAT, which allows for the production of different hash

values for two similar samples.

Although these approaches have the advantage of not requiring the malware sample to be executed, this exact fact is the source of some disadvantages. For example, due to the lack of run-time data, the behavior analysis result can possibly differ from the actual behaviors exhibited by the sample when executed. Another limitation – and the one with the most serious consequences – is the impact of obfuscating and packing techniques have on static analysis. When packing is performed, parts of code are compressed or encrypted in the form of data inside the malware sample and when the sample is executed it may perform some checks in order to decide whether it's going to proceed with its activities or not. If those checks are passed then it proceeds to unpack/decipher the data, load it into the process and finally execute it. Because the data – before unpacking – is scrambled and does not easily resemble code, most static analysis approaches aren't able to provide good results when such techniques are applied.

2.2.2 Dynamic Malware Analysis

On the other hand, one could "simply" execute a malware sample and monitor its actions in order to extract its behaviour. To this type of procedure is called Dynamic Malware Analysis. This approach, when compared with Static Malware Analysis has the advantage of allowing to record the behaviour given a certain input/system data, defeating the code obfuscation and packing techniques that are a major concern in Static Malware Analysis.

Although dynamic analysis provides big advantages when compared with static analysis, some challenges arise from the fact that a malware sample has to be executed in order to perform the analysis. Two of those challenges, safety and evasion, will be presented in Section 2.2.2.C. Another challenge, that largely depends on the resources available and number of samples to analyse, is deciding how long should the execution be as there is no way of knowing how long will a sample wait until it starts its malicious behaviour or if it will even terminate its actions [13], and how detailed the collected data should be as too little data could mean only a high view of the behaviour is extracted which may not be detailed enough for the analysis task at hand, but at the same time if too much data is collected, the parsing of it could make the process of extracting useful information rather difficult and resource intensive.

2.2.2.A Data Collection

Common to every type of analysis, data collection is at the core of it. In the case of Dynamic Malware Analysis, the data collected is, most of the time, a representation of the actions performed by the sample being analysed.

A – Function Call Monitoring In most operating systems, two modes exist in which applications can be run: user-mode and kernel-mode. Most applications are run in user-mode, however only code that runs in kernel-mode has direct access to the system's state, such as the filesystem or the network interface. In order for applications to interact with the system's state, operating systems often provide well defined Application Programming Interfaces (APIs) that can be called from user-mode in order to instruct the kernel of the operating system to execute actions that requires kernel-level privilege. To the collection of APIs provided by an operating system is given the name of system call interface and the functions available in these APIs are called system calls.

Upon a system call, the operating system runs validation procedures so it can verify that the application making the call has sufficient rights to perform the desired action (e.g. create a file in a specific directory). As malware usually needs to interact with the system's state so that it can perform the nefarious actions, by monitoring the system call interface it is possible to obtain the high-level behavior desired from the sample. The main idea behind monitoring the system call interface is similar to the verification process executed by the operating system, where before a system call is pushed down to the kernel, the monitoring process logs the system call executed, its parameters and result. To these actions that are executed before/after the desired system call is given the name of hook function and to the process of intercepting the system calls themselves is called hooking.

One problem that arises from this technique is the fact that malware running in kernel-mode does not require the use of those high-level APIs and therefore can bypass the efforts put into this approach. However, writing malware that skips the use of such APIs is difficult since it requires deep knowledge of the low layers of the operating system. Additionally, due to updates to the operating system, its internals may suffer modifications that can lead to malware written in kernel-mode to stop working as their are highly depended of the operating systems internals.

The two main approaches for performing hooking of functions of interest are binary rewriting and replacing dynamic shared libraries that will now be presented.

Binary Rewriting consists of the idea of changing the malware's binary code, be it on disk before executing it or in its memory image at run-time. These changes that can be performed at the spots where a certain function of interest is called or at the function pointer itself are usually done in the form of an insertion of an unconditional jump to the hook function that will perform the required monitoring using the data on the stack (i.e. the parameters and identification of the function itself), which in the end, restores that data to the stack and by placing another unconditional jump to where the execution was interrupted, restores to normal flow of the system call.

This is the technique used by CWSandbox [14] which creates the target application in suspended mode: mode in which Windows operating systems initializes the application and loads all implicitly linked Dynamic Loaded Librarys (DLLs) but doesn't allow the application to perform any activities. Then

it proceeds to save the code entry points for every function in the DLLs export address table so that the hook function can reconstruct the original function and at that point, it overwrites the first instructions with a `JMP` or `CALL` instruction to the hook function. Additionally, it also hooks the `LoadLibrary` and `LoadLibraryEx` API functions so that the in-runtime binding of DLLs is also monitored.

Replacing dynamic shared libraries works by creating stubs of the functions of interest where they initially perform the monitoring actions and then call the original functions. In order to divert the code from calling the original functions, the original library is often renamed and replaced by one created with the stubs in it.

B – Filesystem and Network Activity Opposing to get a high level detail of the behaviour of a sample by monitoring what system calls the sample invokes, other common approach is to monitor permanent changes and communication to the outside of the analysis environment occurred during the sample's execution. These permanent changes are usually manifested in the analysis environment's filesystem, either with the creation/deletion/modification of files (e.g. as observed in ransomware) or the modification of the registry keys.

Both AMAL [9] and Barecloud [15] record the changes applied to the filesystem after sample execution and use them alongside with knowledge of Windows OSs internals to get the changes applied to the registry. Focusing on filesystem monitoring when analyzing ransomware samples, Kharraz et al. [16] propose monitoring the filesystem activity by monitoring the Master File Table (MFT) on the New Technology File Systems (NTFSs) because, when an attack is occurring, it is expected that many changes are performed on its entries. To monitor the filesystem UNVEIL [17] used the Windows Filesystem Minifilter Driver. They argued that using a minifilter driver enabled UNVEIL's monitor component to be positioned at the closest possible layer to the filesystem, making it harder for the malware sample to bypass the monitoring when compared with using the System Service Descriptor Table (SSDT) to hook filesystem API functions or relevant system calls, for example.

Regarding the communication to the outside, it can be an indicator of communication with C2 servers and/or attempts of propagation of the malware sample. Due to the value of such information, especially when dealing with botnets, it is a common approach in literature to perform network logging, such as in [9, 14, 15, 18].

2.2.2.B Execution Environments

As already mentioned, executing malware samples is at the core of dynamic malware analysis, which typically requires the existence of a dedicated environment. The dedicated environment can be thought of as a sandbox, a term firstly introduced in the literature in the context of confining the actions of untrusted applications in [19].

The techniques and methods used in malware analysis depend on the environment chosen, which will then influence the results and the limitations of such analysis. The main difference in the environments is the virtualization techniques used (if any), as they can range from being a bare-metal machine, where no virtualization is in place, to full machine emulators that simulate Central Processing Unit (CPU) and memory operations.

An important concept highly related to the virtualization in the context of malware analysis is transparency as it can be defined as the property of making analysis systems indistinguishable from non-analysis systems [20]. The main four types of environments regarding transparency are the following:

A – Bare-Metal Machine Of the four types, bare-metal environments are the most transparent ones, as they do not execute the sample on top of virtualized/emulated hardware or operating system. The lack of virtualization brings the advantage of making the environment much more transparent, as it is more difficult for a malware sample to detect that it is running on an analysis setup.

However, the bare-metal environment approach presents some challenges. One main challenge is restoring the initial system's state after performing each sample execution. Other common challenge is extracting the behavior profile of a sample execution as the addition of an in-guest agent would compromise the transparency requirements and make the system detectable.

Addressing the first challenge, Barebox [21] presents a new technique for system state restoration. The technique is based on the idea of partitioning the physical memory of the system: one of the partitions is for the analysis environment while the other is to be used as a snapshot of the system to be restored. When the restoration of the system takes place, another operating system – external to the memory partition belonging to the analysis environment – would then restore the system without requiring a reboot.

Addressing the second challenge, Barecloud [15] profiles the sample using network activity captured ”on the wire”. Regarding filesystem changes and registry-keys, as they are permanent changes they can be compared between the beginning and end of the sample's execution.This data can then be obtained without introducing the problematic agent into the system as the data is located or transmitted through the peripherals of the system.

B – Type I Hypervisor Adding a first layer of virtualization as shown in Figure 2.3, Type I hypervisors have the hypervisor engine, also commonly known as Virtual Machine Monitor (VMM), above the hardware lawyer. This engine is responsible for managing the VMs running guest operating systems and their access to the system state, i.e. hardware.

Due to the presence of the VMM in between the VM and the hardware it is now possible to isolate the VM's actions and have multiple VMs concurrently running in a single host. Furthermore, it is possible for

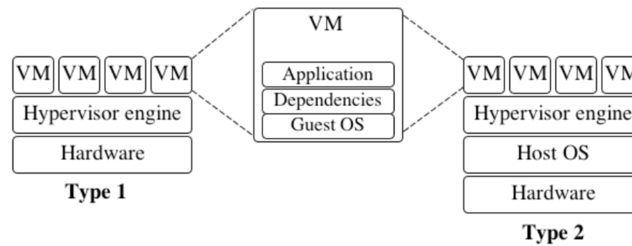

Figure 2.3: Hypervisor Types [3]

an agent to perform monitoring activities to the memory and since the agent is in a layer below the one executing the malware, the latter is unable to directly detect the agent's presence.

However, indirect detection can be achieved due to the fact that the processes that enable isolation and monitoring take more time to execute, possibly making the sample aware that it is being run at a slower rate than expected. Also, there can be bugs in the virtualization software that malware can exploit to detect they are being executed in an analysis environment and further stop execution.

Both Ether [22] and DRAKVUF [23] are systems based on the Xen hypervisor. Ether, using a modified version of Xen, makes use of the fact that page-faults can be configured to trigger `VMEXITs` events on chosen code locations and so Ether is able to trace the system calls occurred in the VM. DRAKVUF traces the system calls using direct memory access, with the use of LibVMI library, to inject breakpoint instructions into the VM's memory and then, similarly to Ether, it configures Xen to trigger a `VMEXIT` event when a breakpoint instruction is executed.

C – Type II Hypervisor Similarly to Type I, Type II Hypervisors, also known as Hosted Virtual Machines [24], have a hypervisor engine layer that stays under the VM layer. However, the former is placed above a new layer, the Host OS layer, as presented in Figure 2.3.

The addition of the Host OS layer further increases the problem of slow sample execution and adds the quirk that high-privilege instructions are executed from the Host OS, reducing the transparency of the system.

However, this approach brings the advantage of simpler usage and setup of hypervisor engine and VM layers.

AMAL [9] uses VMWare type II VM's to execute its samples. Upon execution completion, it proceeds to log the changes to file system and network activity by analyzing the Virtual Machine Disk (VMDK) and Packet Capture (PCAP) files respectively. Cuckoo sandbox [25], an open source project and probably one of the most popular dynamic malware analysis systems also uses type II virtualization.

D – Machine Emulator While in type I and II hypervisors low privileged instructions run directly in the host CPU, machine emulators execute every guest CPU's instructions using multiple host CPU's

instructions. This results in high portability as one CPU architecture can be emulated on another CPU architecture. Also, this means that emulators have control and monitor capabilities over every instruction executed by the guest system. Such capabilities allow new kinds of malware analysis such as information flow tracking on a binary level as demonstrated in Panorama [26] using the QEMU.

Pandora's Bochs [27] has the goal of recovering packed code so it extends the Bochs emulator to monitor the unpacking stubs that load the desired code into virtual memory and detect its execution.

However, due to the usual ratio of single guest instructions being translated into multiple host ones, the sample's execution is expected to be much slower than type I and II hypervisors. Additionally, emulating low-level CPU details is a difficult task to do accurately, meaning that more bugs can exist when compared against hypervisor solutions, making these systems the less transparent of them all.

2.2.2.C Challenges

As already stated above, dynamic malware analysis faces two main challenges, safety and environment detection. Below, it is presented more in-depth those challenges and proposed solutions.

A – Safety As previously established, malware has the intrinsic goal of performing malicious activities in the systems it infects. When executing such samples the extent of such malicious activities must be contained inside the analysis environment and only during the sample's execution.

In order to contain the impact of the sample's execution inside the analysis environment, virtualization and emulation are often used, as isolation is generally one of the main requirements on the design of such solutions. However, and even though it is commonly accepted that virtualization/emulation solutions are trustworthy, due to the fact that bugs also exist in the virtualization software, malware can target those bugs in order to escape the virtualized/emulated environment [22] enabling them to extend their impact to the whole infrastructure responsible for the analysis and further.

Additionally, it is usual for malware to require an Internet connection in order to begin its malicious activities. This presents a threat to the outside systems that needs to be addressed [8] as this connection could be used to infected other systems or participate in bigger scale attacks, such as spam-serving or Distributed Denial of Service (DDOS) attacks.

To deal with this trade-off between connectivity and safety the most common approach is to filter network traffic. By building rules based on traffic speed/number of connections, it is possible to mitigate participation in some less sophisticated DDOS attacks. It is also desired to build rules based on the port the samples tries to connect to, so that connections to common vulnerable services could be blocked, such as SMB that is commonly used as a propagation vector in self-propagating malware.

Another strategy is to emulate some common services and have the network traffic redirected to such them as done, for example, in [28] that builds a "mini-network" of such services, isolating the analysis

environment from the real Internet.

B – Environment Detection One way for malware authors to defeat dynamic malware analysis is to make samples able to detect that they are being executed in an analysis environment. Such detection is done by looking for unique traits in the execution environment that could indicate the malware is being executed in a system made for analysis. Upon positive detection, the malware usually behaves benignly or exits.

These unique traits, also known as fingerprints, can take the form of multiple environment variables, such as usernames, system settings, files in the user's folder, installed software and product keys, etc. A possible solution to this kind of fingerprinting is to pseudo-randomly generate such traits.

With the addition of virtualization/emulation, the environment becomes less transparent. In turn, this causes detection to be more easily achieved by a malware sample. One method, already mentioned, that malware can use to identify the virtualized system is slower execution speed in such systems. More advanced techniques also can identify timing discrepancies as time is hard to accurately simulate on virtualized hardware. Another method commonly used that was also already mentioned is to exploit bugs in the virtualization software, especially in the CPU virtualization part. These bugs, also known as "red pills" are a set of instructions that have different results when performed on virtualized CPU's.

As mentioned in Section 2.2.2.A binary rewriting to perform function hooking is a common technique to gain information on malware's behavior. This technique, however, can be detected if the malware checks its memory integrity. In order to mitigate such detection, CWSandbox [14] uses rootkit-like techniques to evade detection from the malware's point of view.

One last common technique to thwart malware analysis used by malware samples is to halt their activities for a significant amount of time or until user interaction is detected such as mouse movement or keyboard input.

2.3 Reproducibility

As recognized in recent years, rigorous empirical research plays an important role in computational science. However, in the security field, there's been a lack of adoption of sound scientific methods that are fundamental for the development and evolution of any scientific field.

At the core of the scientific method lies reproducibility. In computational science, good practices regarding reproducibility have practical advantages such as code reuse that enables more rapid progress as new approaches can be tested and built upon already developed and validated modules.

Testbeds have been proposed to deal with the mentioned problems as they offer great control over the whole experiment pipeline, enabling the recording of the whole experiment setup and detailed collection

of the results obtained. Examples of the proposed solutions are:

- DETER [29] which uses multiple machines and control software to provide a medium-scale infrastructure for safe and repeatable security-related experiments and simulations.
- ViSe [30] on the other hand uses virtualization techniques to set up the multiple virtual machines used in the testbed. The goal of this testbed is to have attacker and victim machines replaying and recording events, such as the attacker machine replay an attack and record the impact on a victim machine.

3

Proposed System

Contents

3.1 Requirements . . . 27
3.2 Architecture . . . 29
3.3 Implementation . . . 32
3.4 Usage . . . 38

This chapter presents the Programmable Sandbox for Malware Analysis (PSMA), a scalable dynamic malware analysis system that enables researchers to perform programmable and repeatable experiments. It starts by introducing the requirements considered while designing and implementing the solution in Section 3.1. Section 3.2 then presents the architecture that allowed to met the requirements imposed, followed by the implementation decisions and their rationale in Section 3.3. Finally, Section 3.4 presents the steps to setup the system and how to execute an experiment on it.

3.1 Requirements

In order to understand the proposed system's architecture, the following sections present the requirements had in consideration throughout its design and implementation.

3.1.1 Enable Reproducibility

Taking into consideration the goal of enabling reproducibility of experiments, the system must save as much information about the approaches used in each experiment as possible, enabling other researchers to confirm results or build new approaches based on previous work. To satisfy this requirement, while at the same time also enabling collaboration, the system was designed to have a storage component where the Modules and Virtual Machine Image (VMI) used are saved. Additionally, to allow for iterative work upon these Modules and VMIs, the storage component uses a versioning system to store them.

With the storage and versioning of the Modules and VMIs, an experiment can be now defined by the IDs and versions of these objects, the parameters required for their execution and the samples used. Moreover, the experiment is also saved inside the system with this information.

3.1.2 Modular

As stated in the previous chapters, malware analysis is commonly done in two phases: data collection and data processing. To comply with this norm, these two phases were translated into the creation of two subsystems independent from one another, the Data Collection System (DCS) and the Data Processing System (DPS), which are respectively responsible for the execution of the samples and analysing the data collected from their execution.

Considering that one of the main goals of the proposed system is to be useful in a range of malware research scenarios, the system must be flexible enough to adapt to the various research approaches. Although bound to the two-phase process detailed above, this flexibility can be achieved by having

the actions performed by the system being defined[1] by the user. These actions are passed to the system in the form of Modules that are uploaded to it, which are then referenced by the experiment making the system load them into the context of the experiment itself. Naturally, two module types were designed in order to align with the two-phase process mentioned: one to be executed inside the execution environment responsible for the collection of data and the other responsible for its processing.

3.1.3 Scalable

As motivated in Section 2.2, dynamic malware analysis systems must be highly performant as time is a valuable resource and the process of executing samples to collect their behavioural data can be very time consuming. However, leveraging the fact that sample execution is an isolated process, independent of other samples being executed in the system, the system can be designed to support horizontal scaling which is the process of increasing the number of resources in the resource pool of a system, contrasting with vertical scaling which is the process of increasing of the capacity of the resources in said resource pool.

Considering this new requirement and the independence between the Data Collection and the Data Processing systems, these were designed to be composed of multiple worker nodes, independently scalable on their own, allowing the parallel execution of samples in the Data Collection System, as mentioned previously, but also the parallel analysis of the collected data from different experiments in the Data Processing System.

3.1.4 Isolated

As seen in Section 2.2.2.C, safety poses as one of the main challenges for dynamic malware analysis, as infection and propagation of malware outside of the execution environment can affect the experiment's results, even hinder its completion, and raise ethical problems since systems outside the researcher's control can become infected. It was also explained that virtualization, thoroughly detailed in Section 2.2.2.A, is often the chosen execution environment as the virtualization software introduces an isolation layer between the execution environment and the outside systems and additionally allows for more control and monitoring of the environment's state itself.

For the reasons presented above, the execution environment inside the Data Collection System's worker nodes were designed to be Virtual Machines controlled by a Type II Hypervisor running inside each node. Additionally, to prevent the network propagation of malware, the virtual machines can be not connected to any network card. However, it's important to point that this is not a limitation of the system itself but a mere configuration of the Virtual Machine Image (VMI) imported to the system.

[1] Could be read as programmed, hence the system's name: **Programmable** Sandbox for Malware Analysis

3.2 Architecture

With the requirements for the system presented in the previous section, this section now presents the proposed architecture, that meets those requirements, by starting with the introduction of an high level view of the system followed by the detail of the architecture of the underlying systems/components.

3.2.1 High-Level View

Section 3.1 mentioned three components on the proposed system: the Data Collection, Data Processing and versioned Storage Systems. To complete the system, in addiction to those components, two others are required and were included in the architecture – the Host and Message Broker – as it can be seen in Figure 3.1.

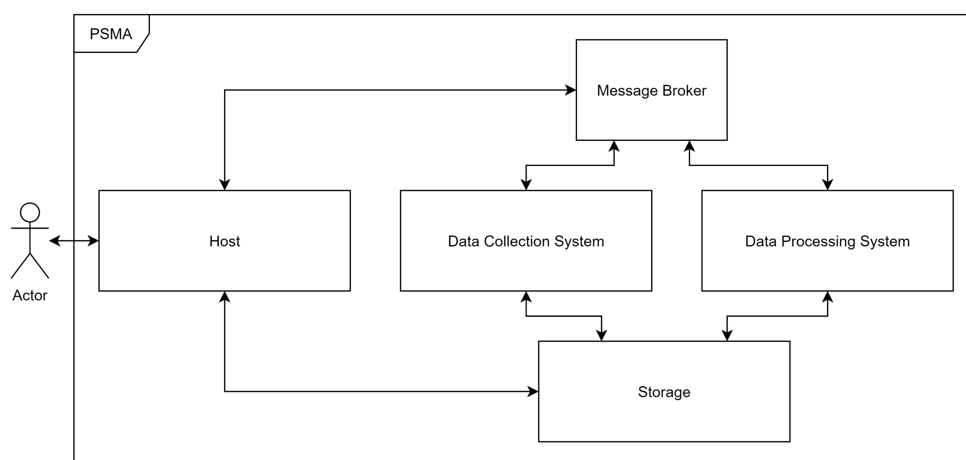

Figure 3.1: PSMA's High-Level Architecture

Regarding the actor in the figure, it corresponds to the researcher using the System, which will interact with it using the Host, as it will be explained in the next Section. The researcher is responsible for defining the experiments that will be run on the System, alongside the Modules and VMIs imported.

3.2.2 Host

As mentioned above, the Host is the entrypoint for the system. Each request made to the system is performed by calling the appropriate endpoint provided by the Host, which in turn communicates with the Storage system and/or Message Broker, depending on the task at hand. It is important to note that there is no direct communication with the Data Collection and Data Processing components, as they receive the tasks to perform from the Message Broker, hence the need for it. The Host can then be seen as being composed by a Web API and the necessary connectors for the Storage system and the Message Broker.

3.2.3 Message Broker

As seen in the Section 3.1, the Data Collection and Data Processing components have the requirement of being scalable and be composed of multiples workers. To coordinate the tasks run by these workers a communication system between them and the Host must exist. This is the role played by the Message Broker, which will receive the messages sent by the Host with the task definitions that will then be requested by the workers from the appropriate system for the task, making the Message Broker have a Task Queue like functionality.

3.2.4 Storage

A key component of the proposed system is the integration of the versioned storage solution. This storage is composed by two components as can be seen in Figure 3.2. The file storage is responsible to hold the actual files that compose the Modules, VMIs and experiments while the relational database is responsible to hold the metadata related to them. This allows for a fast lookup of information/state of these objects without having to transverse the much slower file storage.

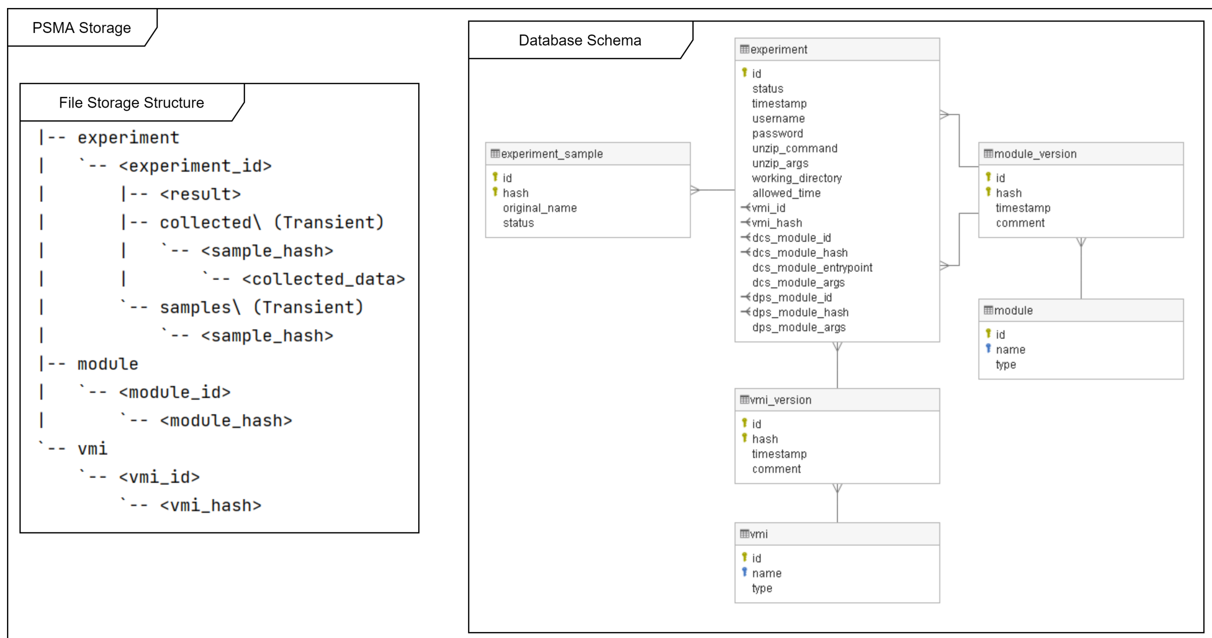

Figure 3.2: PSMA's Storage

3.2.5 Data Collection System

As mentioned before, the collection of data from the execution of the tested samples is done inside the Data Collection System which is composed by a variable number of nodes as showed in Figure 3.3.

To control the tasks executed at each node on the Data Collection System, each node has a Task Queue Controller which connects itself to the Message Broker. Each time a message arrives at the Message Broker with the identification of a Data Collection task, the Task Queue Controller (if the node has capacity) creates a Virtual Machine Controller which interfaces with the Hypervisor running on the node. This Virtual Machine Controller then connects to the Storage System to download the Virtual Image, the Data Collection System module and sample, with which it will launch a virtual machine and upload the module and sample into it. After, the Virtual Machine Controller instructs the virtual machine to execute the module, which is responsible to run the sample itself and collect data about its behaviour. Finally, after the module collects the data from the execution of the sample, the Virtual Machine Controller extracts the data from inside the virtual machine and uploads it to the Storage System.

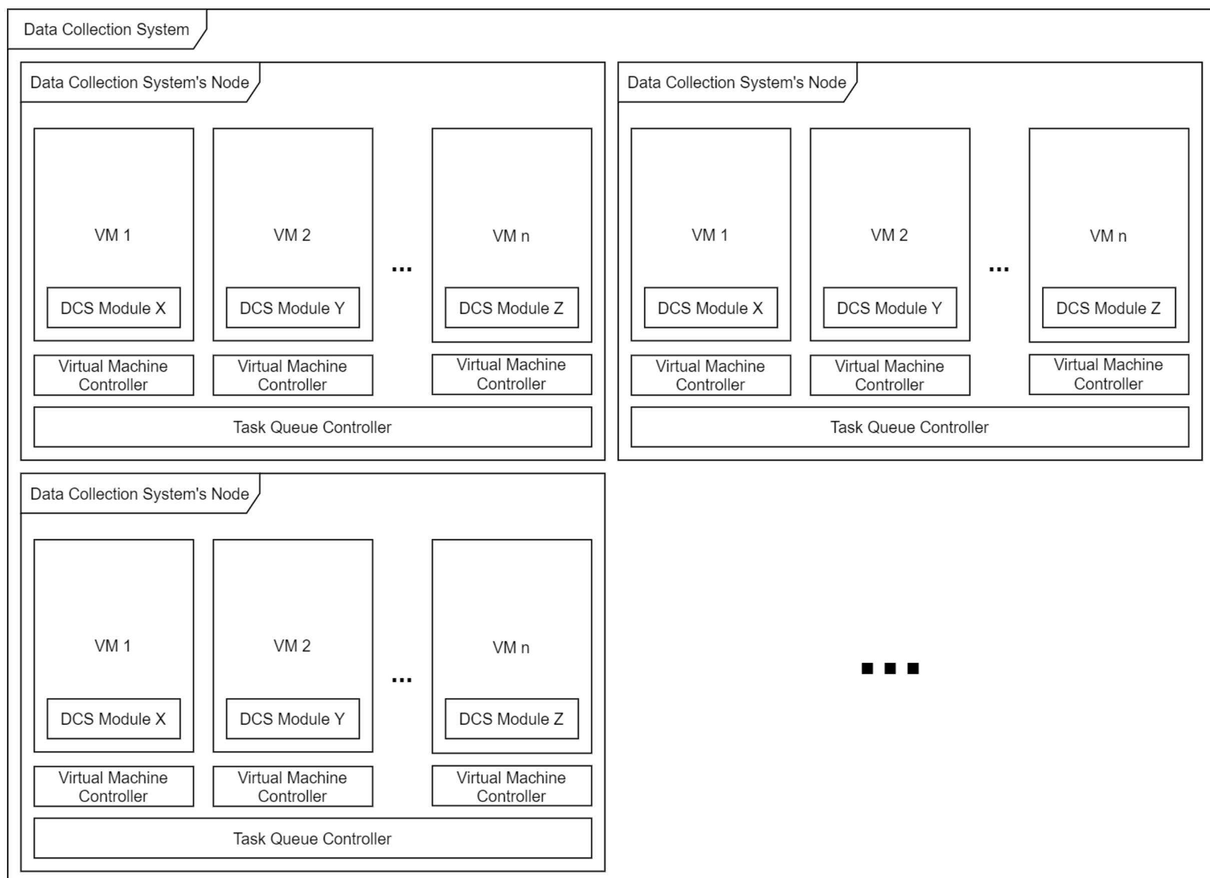

Figure 3.3: Data Collection System Architecture

3.2.6 Data Processing System

Similar to the Data Collection System, the Data Processing System is composed of a variable number of nodes as it is scalable, as can be seen in Figure 3.4. Again, much like the DCS nodes, the DPS

nodes connect to the Message Broker using a Task Queue Controller which will receive the tasks to be performed.

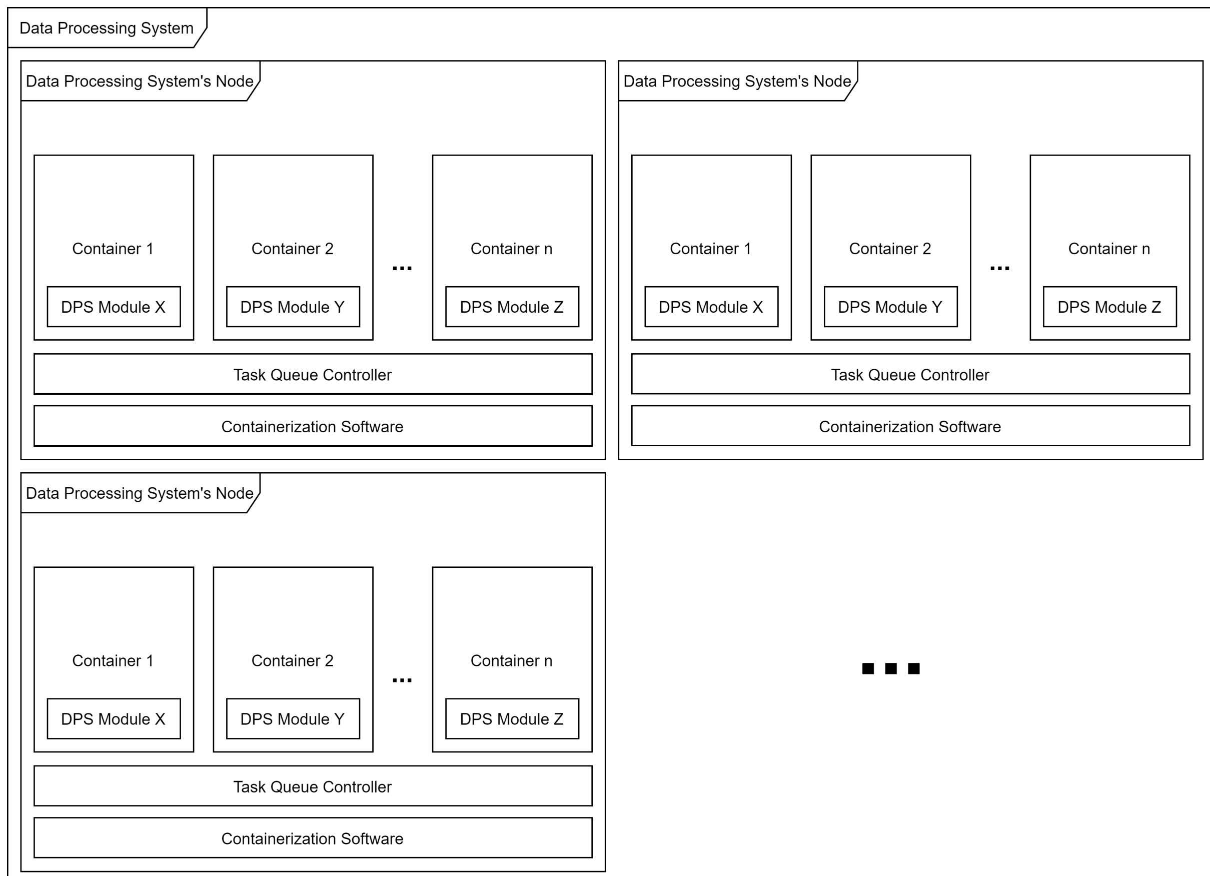

Figure 3.4: Data Processing System Architecture

For the environment of execution of the data processing module, the system was designed to be a containerized environment, as it provides much flexibility to the user, allowing them to import all of the dependencies needed inside the container whilst not impacting the installed software in the node environment. The Task Queue Controller manages the containers orchestration as it directly interfaces with the containerization software present in the node.

3.3 Implementation

With the architecture of the system detailed above, this section presents the implementation decisions and technologies adopted for the creation of the first prototype of the Programmable Sandbox for Malware Analysis (PSMA) which was mainly developed using the Python 3[2] programming language due to its simplicity for prototyping and large community support. Regarding the deployment of the system's

[2] https://www.python.org/

components, most of the components were deployed using Docker Compose[3], which is a tool for defining and running multi-container Docker applications. This way the infrastructure needed can be run on a single machine while communicating over a network, hence removing the need for a multi-machine setup for the system to be tested and used. Additionally, the use of a containerized approach to the system's deployment allows for support of multiple operating systems, as long they support the containerization technologies used, and isolation from the host which means less changes are required to be made to the software installed on it.

Similarly to the previous sections, this section is further composed by subsections detailing the multiple parts of the system, in this case regarding its implementation.

3.3.1 Storage System

As previously mentioned, the Storage System is composed by a file storage component and a relational database. As the system is design to be scalable, network access to the file storage system is mandatory and therefore a communication protocol must be followed by all the parties involved in the file transfer (i.e. clients and server). After analysing several file transfer protocols, the chosen protocol was the SSH File Transfer Protocol (SFTP) as it relies on the Secure Shell (SSH) protocol providing an encrypted connection for both authentication and file transfer. Additionally, with its popularity, many libraries for connecting to SFTP servers using Python were available, from which Paramiko[4] was picked to be at the core of a wrapper developed called `SFTPController` that manages the connection to the server on each file/folder operation. As for the SFTP server itself and following the strategy of using Docker to containerize the infrastructure of the system, the atmoz/sftp[5] Docker image was chosen.

Regarding the file system structure on the SFTP server, three folders were created as depicted in Figure 3.2. The first two, the `vmi` and `module` folders, have similar file structure as both VMIs and Modules are versioned in the same manner. From a file storage point of view, both objects are composed by two identifiers: the id, which identifies the group of version of an object, and the hash which identifies each version of the object itself. The id is a Universally Unique Identifier (UUID) randomly generated when the system receives a completely new object and the hash is the result of computing the the Secure Hash Algorithm (SHA) 256-bit version over the file stored. Given these two values the file is stored in the file system as follows: `/[vmi|module]/<id>/<hash>`.

Lastly, the other folder created was the `experiment` folder which stores every file regarding the experiments (i.e. samples, data collected and result). As expected, to identify each experiment, an UUID is randomly generated which will be the root folder for the subsequent folders created for the experiment. For the samples being used in the experiment, the `samples` folder was created, where each sample

[3] https://docs.docker.com/compose/
[4] http://www.paramiko.org/
[5] https://hub.docker.com/r/atmoz/sftp/

stored is identified by the SHA-256 of the file. As for the data collected from the analysis of the sample inside the DCS, it is stored on the folder: /experiment/<experiment_id>/collected/<sample_hash>/. The files that result from the conclusion of the experiment are stored inside the root folder for the experiment. At the end of the experiment the samples and collected data folders are deleted as to efficiently use the available storage.

As for the relational database, the system is using a PostgreSQL[6] database to hold the metadata for the objects related to the VMIs, Modules and Experiments. As can be seen in Figure 3.2, the versioned objects – VMIs and Modules – have a similar data structure where each object is defined by an id (the same id referenced before), a name and a type. As for the version of the objects, these have a reference to the id of the object, the hash of the current version of the object file, the timestamp of when the version was uploaded to the system and finally an optional string for comments on the specific version. The type of the module refers to whether the module is to be used on either DCS or DCS nodes, while the type of the VMI refers to what kind of virtualization software is to be used with the VMI file stored, even with the current prototype only supporting one virtualization software. As for the experiment, it is defined by an id, status and by the id and hashes for the VMI, DCS and DPS modules alongside the needed parameters/arguments. For each sample in the experiment is then added an entry to sample table with the id of the experiment, the hash of the sample file, the sample's original file name and its status in regard to the data collection phase.

For the connection to the database, the system uses the Flask-SQLAlchemy[7] library as the host, as will be detailed later, is running a Flask application. The deployment of the PostgreSQL server is made using the postgres:alpine[8] version Docker image as it is sufficient for the use case and by building itself upon the Alpine Linux project it has a much smaller storage footprint than the alternatives.

3.3.2 Message Broker

The Message Broker plays a vital part on the system, being the component that allows for the scalability in terms of nodes in both DCS and DCS as it will act almost as a task queue accessible by all the nodes. While this Section is related to the Message Broker implementation/design choices, it is first required to explain how the nodes and host connect to it as the chosen technology impacted the choice of the Message Broker. After analysing the alternatives for implementing a distributed task queue in Python, it was decided to use the Celery framework[9] as it is one of the most popular frameworks for asynchronous task distribution in Python. To create the two-phase approach workflow mentioned previously, the system uses a Celery feature called Chord. Chords allow to create a group of tasks that will be executed in

[6] https://www.postgresql.org/
[7] https://flask-sqlalchemy.palletsprojects.com/en/2.x/
[8] https://hub.docker.com/_/postgres
[9] https://docs.celeryproject.org/en/stable/

parallel followed by a task that will execute when all of the tasks in the group are finished. To support this feature, Celery requires a Results Backend to be configured, for which Redis[10] is usually used. As Celery can also use Redis to play the role of a Message Broker, Redis was the message broker selected to be used in the system. For the deployment of the Redis server the system is using the redis:6-alpine[11] version Docker image.

3.3.3 Data Collection System

As seen in the previous Section, the DCS nodes will use Celery to connect to the Message Broker. This is achieved by having each worker node being deployed as a Celery Worker which listens to the task queue correspondent to the DCS tasks. For the management and virtualization of the execution environments, the system currently only supports Oracle VM VirtualBox[12] hypervisor which is controlled using the developed `VBoxController` that corresponds to the Virtual Machine Controller in Figure 3.3.

To interface with VirtualBox two main options were available: using the command line interfacing binaries that come with the standard installation of VirtualBox into a system or using the API developed by VirtualBox. Although the usage of binaries had the advantage diminishing the requirements in terms of libraries used in the code of the worker's system, it had the big disadvantage of requiring OS awareness due to the different ways command line interfaces can be used in different OSs. For that reason and to have all the state of VirtualBox system inside the worker context the `VBoxController` was developed around the virtualbox-python library[13] which itself is a wrapper around the API developed by VirtualBox.

When the worker receives a task, it will start by commanding the `VBoxController` to clone the VMI reference on the experiment which makes the `VBoxController` start by querying VirtualBox to check if there's a virtual machine in the system created based off the requested VMI. If VirtualBox can't find the virtual machine then the `VBoxController` will load the VMI from the Storage System and then create a virtual machine using it. With the base virtual machine present in the system, then the `VBoxController` creates a clone of it to be used as the execution environment where the sample is going to be analysed. To increase efficiency in storage used and in the process itself of cloning, the clone created uses a new differencing disk image based on the base virtual machine disk hence reducing the storage and copy of duplicated information.

With the virtual machine required to analyse the sample created, the `VBoxController` is then instructed by the worker to power it up and upload the files required into it. These files which correspond to DCS module referenced by the experiment and the sample to be analysed and are passed to the virtual machine in a compressed folder which then will need to be extracted using the command passed

[10] https://redis.io/
[11] https://hub.docker.com/_/redis
[12] https://www.virtualbox.org/
[13] https://github.com/sethmlarson/virtualbox-python

to the experiment by the user. The decision of requiring the user to give the extraction command was made due to the different tools available in different OSs. With all the files in the virtual machine, the `VBoxController` then executes the entrypoint of the DCS module, which is then responsible to analyse and run the sample. The module is then expected to save its results in a directory that is going to be fetched by the `VBoxController` at the end of the allocated analysis time. With the sample analysed and the results extracted from the virtual machine, the `VBoxController` uploads the results to the Storage Systems as mentioned in previous subsections and powers down and deletes the virtual machine from the system.

3.3.4 Data Processing System

Similarly to the DCS nodes, the Data Processing System nodes also are deployed as Celery workers that connect to the Message Broker to receive the tasks to perform. As previously stated in Section 3.2.6, the module execution is done inside a containerized environment and, as to no surprise, the chosen environment was Docker Containers which are managed by the worker using the Docker SDK for Python[14]. This requires the user of the system to define the DCS modules as a valid Docker image with the Dockerfile in the root of the module. Upon receiving a task, the worker starts by loading the collected data from the Storage System to a directory which will then be mounted as a volume in the Docker container to make it available from inside the container. After the collected data is loaded into the directory, the worker then builds the Docker image defined by the Dockerfile and creates a Docker container based on the image just created. The container then runs the command defined in the Dockerfile `ENTRYPOINT` section. It is then expected for the execution of the command in the container produces a directory inside the mounted volume with the results of the processing of the collected data. Upon completion of the execution of the Docker container, the worker uploads the results directory to the Storage System and deletes the local volume and container created as they are no longer needed. At this point the experiment is concluded and the worker updates its status and cleans the Storage System from the intermediary files created.

3.3.5 Host

As previously mentioned, the Host is the entrypoint for the system, connecting the user to the Storage System and Message Broker (and indirectly to the DCS and DCS nodes). The user interacts with the host using a Web API developed using the Flask Framework[15] which is composed of the endpoints presented in Table 3.1 that allow for CRUD (create, read, update, delete) operations over VMIs and Modules and for the execution of experiments, querying their status and deleting them from the system.

[14] https://docker-py.readthedocs.io/en/stable/
[15] https://flask.palletsprojects.com/en/2.0.x/

Method	Route	Explanation
GET	/v1/[vmi|module|experiment]/	Gets all objects
POST	/v1/[vmi|module|experiment]/	Creates a new object
GET	/v1/[vmi|module|experiment]/<id>	Gets the information of the object
DELETE	/v1/[vmi|module|experiment]/<id>	Deletes the object
PUT	/v1/[vmi|module]/<id>	Creates a new version of the object
GET	/v1/[vmi|module]/<id>/download	Downloads the latest version of the object
GET	/v1/[vmi|module]/<id>/<hash>	Gets the information of an object's version
DELETE	/v1/[vmi|module]/<id>/<hash>	Deletes an object's version
GET	/v1/[vmi|module]/<id>/<hash>/download	Downloads an object's version
GET	/v1/experiment/<id>/status	Get experiment's and samples' statuses
GET	/v1/experiment/<id>/result	Downloads the experiment's result

Table 3.1: PSMA's endpoints

When using the endpoints that create a new object (i.e. Module, VMI), the related metadata should be included in a YAML Ain't Markup Language (YAML) file in the request alongside the object file, as it will be presented in Section 3.4. The result of the GET methods also displays the information of the object in the YAML format as to have consistency in the input and output of the system. The choice of using the YAML format against JavaScript Object Notation (JSON), which is also a very popular format for information passing in Web APIs, was based on the readability of YAML files by humans and its massive adoption by the Docker community.

Regarding the object files, in the case of the VMI endpoint the passed object file must correspond to the Open Virtualization Format (OVA) file that contains the image of the base virtual machine to be used; for the Module endpoint, the object file must be a ZIP file containing the required file/structure for the type of module it represents; and finally the object file of the experiment is a ZIP file with all the samples to be analysed in the experiment.

Upon called in one of the endpoints, the Host verifies if the information passed is valid such as validating if object to delete is present or that no duplicates are introduced in the system. If required, then the Host executes the changes in the SFTP server and if all is performed successfully then it applies the change to the PostgreSQL database. In the case of the endpoint that initiates an experiment (i.e. POST `/v1/experiment/`), the Host first uploads the samples to the SFTP server and creates the correspondent metadata for them and for the experiment in the PostgreSQL database. After committing the addition of the metadata, it then proceeds to launch the experiment by calling the Chord feature from Celery which was discussed earlier. This will produce the tasks in the Message Broker which will then be received by available nodes in the system.

3.4 Usage

Finally, this Section presents how a researcher could use PSMA to perform his experiments, starting by describing the setup procedures so PSMA can properly run followed by the interactions needed with PSMA to perform an experiment.

3.4.1 System Setup

As stated previously, most of the system can be run using the Docker Compose utility. Additionally, for the execution of the DPS modules, Docker is used as the containerization environment. Hence, installing Docker[16] is mandatory and Docker Compose[17] is highly recommended.

However, as both DCS and DPS nodes need to interface with virtualization and containerization software installed on the host, the nodes will not run inside the containers. For them to work in the researcher's system, Python 3[18] must be installed and PSMA's library dependencies must be met. To not interfere with the libraries already present in the researcher's system, it is advised to make use of Virtual Environments[19].

So, to create the Virtual Environment and install the dependencies required, in the root directory of the PSMA project, type the following commands:

```
$ python3 -m venv ./.venv
$ source ./.venv/bin/activate
$ pip install -r requirements.txt
```

Finally, as explained in Section 3.3.3, for the current version of PSMA, only VirtualBox is supported as an Hypervisor, so to run a DCS's node VirtualBox[20] must be installed alongside its SDK, which must be installed in the Virtual Environment where the nodes will run.

3.4.2 Experiment Execution

To simplify the process of starting PSMA's components, multiple scripts were developed which can be found in the `scripts/` directory. To bootstrap all of the components, the `full_system_run.sh` should be used as it will start the Storage component, the Message Broker and the Host in the Docker environment followed by the execution of one node for each DCS and DPS.

With the PSMA up and running, the Host is then available at the address `localhost:8080`, which the researcher can reach to interact with the system.

[16] https://docs.docker.com/engine/install/
[17] https://docs.docker.com/compose/install/
[18] https://www.python.org/downloads/
[19] https://docs.python.org/3/tutorial/venv.html
[20] https://www.virtualbox.org/wiki/Downloads

For the execution of an experiment, three objects must be created inside the system – the VMI, the DCS Module and the DPS Module.

3.4.2.A VMI Creation

As explained previously, the VMI corresponds to the OVA file defining the Virtual Machine to be used which has two requirements. First, as explained in the previous section, the Virtual Machine has to have an utility installed that can be called in order to perform the extraction of the files inside the ZIP uploaded into it. Secondly, in order to have the Virtual Machine being operated from the outside, the VirtualBox Guest Additions package must be installed on it.

With the OVA file created, the YAML file should be created following the format presented in Figure 3.5. Additionally, it should have the `name` field, be exactly same as the name of the Virtual Machine from which the OVA file was extracted, due to a limitation on VirtualBox's stored metadata.

```yaml
vmi:
  name: Windows7
  type: virtualbox
  comment: Windows 7 base image
```

Figure 3.5: VMI Input YAML File

Now with both files created and saved as `vmi.ova` and `vmi.yaml`, make a request to PSMA on the `/v1/vmi/` endpoint as follows:

```
$ curl --location --request POST 'http://localhost:8080/v1/vmi/' \
--form 'vmi=@"vmi.ova"' \
--form 'vmi_definition=@"vmi.yaml"'
```

To which PSMA should respond with a file similar to the one presented in Figure 3.6.

```yaml
vmi:
  id: e0c60eb1-302b-4f2a-9597-b823bcc2ab65
  name: Windows7
  type: virtualbox
  versions:
  - comment: Windows 7 base image
    hash: f6821e23bb1e10c52438cc02242dd4f5b7469d3a6abb93780edde14a1598c80b
    timestamp: 2021-07-26 13:17:59.030769
```

Figure 3.6: VMI Output YAML File

3.4.2.B DCS Module Creation

As mentioned previously, the DCS module is responsible for the execution of the sample and its monitoring. To comply with the time allocated for the experiment, it is recommended that the module's

`entrypoint` (i.e. the file that will be executed to run the module) is asynchronous and returns immediately. The module also should contain all the files necessary for it to able to run as connection to the Internet is not guaranteed. With the entrypoint file created and its dependencies met inside the same folder, a ZIP file of such folder should be created as it is what PSMA expects to receive as a Module.

Similarly to the VMI, a YAML file should be created to accompany the ZIP file in the request made to the system. In Figure 3.7 is shown an example with the expected YAML file structure.

```yaml
module:
  name: SimpleDCSModule
  type: data_collection
  comment: Simple DCS module
```

Figure 3.7: DCS Module Input YAML File

Now with both files created and saved as `dcs_module.zip` and `dcs_module.yaml`, make a request to PSMA on the `/v1/module/` endpoint as follows:

```
$ curl --location --request POST 'http://localhost:8080/v1/module/' \
--form 'module=@"dcs_module.zip"' \
--form 'module_definition=@"dcs_module.yaml"'
```

To which PSMA should respond with a file similar to the one presented in Figure 3.8.

```yaml
module:
  id: 46faa557-da7f-4f72-8222-d204819c9427
  name: SimpleDCSModule
  type: data_collection
  versions:
  - comment: Simple DCS module
    hash: 81425fe3e614bd5485939365d908c9a03ecf6031f4386001f3dda0d399b9d227
    timestamp: 2021-07-26 13:18:41.575810
```

Figure 3.8: DCS Module Output YAML File

3.4.2.C DPS Module Creation

As for the DPS Module, it should correspond to a valid definition for a Docker container (i.e. it should contain a valid `Dockerfile` file at the root of the module). As it is common practice in Docker container creation, the additional files necessary for the container should be in the folder of the Module and should be copied using the `COPY` instruction in the `Dockerfile` file. Similarly to the DCS Module, from this folder should be created a ZIP file with its contents.

As expected, a YAML file should also be created to accompany the ZIP file in the request made to the system. In Figure 3.9 is shown an example with the expected YAML file structure.

```yaml
module:
  name: SimpleDPSModule
  type: data_processing
  comment: Simple DPS module
```

Figure 3.9: DPS Module Input YAML File

Now with both files created and saved as `dps_module.zip` and `dps_module.yaml`, make a request to PSMA on the `/v1/module/` endpoint as follows:

```
$ curl --location --request POST 'http://localhost:8080/v1/module/' \
--form 'module=@"dps_module.zip"' \
--form 'module_definition=@"dps_module.yaml"'
```

To which PSMA should respond with a file similar to the one presented in Figure 3.10.

```yaml
module:
  id: d8660458-6799-48b1-be12-cdd69bd578b0
  name: SimpleDPSModule
  type: data_processing
  versions:
  - comment: Simple DPS module
    hash: c5fbd55c54a976ea1b79ce56877a07eefcb8d9a3e8dfbcd7d52ef625e96e339e
    timestamp: 2021-07-26 13:19:13.491374
```

Figure 3.10: DPS Module Output YAML File

3.4.2.D Experiment Launch

With the three required objects in the system, the only thing left to do to launch an experiment is to create the YAML file defining it and create the ZIP file with the samples to be analysed. An example of a YAML file referencing the previously generated objects is presented in Figure 3.11.

For the file's creation it is mandatory to reference the IDs generated for the objects, so that the PSMA knows what to load into the experiment's context. However, due to the presence of only a single version of each object in the system, the `hash` fields can be left out, as the system will default those values to the last version present. Additionally, the `args` and `allowed_time` fields are also optional as they default to empty arrays and 60 (i.e. 60 seconds), respectively.

Performing the request as follows, starts the experiment defined in the `experiment.yaml` file against the samples inside the `samples.zip`.

```
$ curl --location --request POST 'http://localhost:8080/v1/experiment/' \
--form 'samples=@"samples.zip"' \
--form 'experiment=@"experiment.yaml"'
```

```yaml
experiment:
  allowed_time: 60
  credentials:
    username: psma
    password: password
  unzip:
    command: C:\Program Files\7-Zip\7z.exe
    args:
    - -o$WORK_DIR\files
    - e
    - $ZIP_FILE
  working_directory: C:\Users\psma\Desktop
  vmi:
    id: e0c60eb1-302b-4f2a-9597-b823bcc2ab65
    hash: f6821e23bb1e10c52438cc02242dd4f5b7469d3a6abb93780edde14a1598c80b
  dcs_module:
    id: 46faa557-da7f-4f72-8222-d204819c9427
    hash: 81425fe3e614bd5485939365d908c9a03ecf6031f4386001f3dda0d399b9d227
    entrypoint: entrypoint.bat
    args: []
  dps_module:
    id: d8660458-6799-48b1-be12-cdd69bd578b0
    hash: c5fbd55c54a976ea1b79ce56877a07eefcb8d9a3e8dfbcd7d52ef625e96e339e
    args: []
```


Figure 3.11: Experiment Input YAML File

The request results in the YAML file presented in Figure 3.12, which fills the defaults values, adds the samples' metadata and adds both the status for the experiment and the samples. The status for the experiment transactions as follows:

`NOT_STARTED -> COLLECTING_DATA -> PROCESSING_DATA -> FINISHED`

Whereas, the status for the samples transitions according to:

`NOT_STARTED -> COLLECTING_DATA -> DATA_COLLECTED`

In order to query the status of the experiment, the `/v1/experiment/<experiment_id>/status` endpoint should be used as demonstrated below, resulting in a output similar to the one depicted in Figure 3.13.

```
$ curl --location --request GET \
'http://localhost:8080/v1/experiment/ff1b4ec4-4c47-4487-91d5-af3883eeb5bf/status'
```

Finally, after the experiment status reaches `FINISHED`, the result of the experiment can be obtained using the following request:

```
$ curl --location --request GET \
'http://localhost:8080/v1/experiment/ff1b4ec4-4c47-4487-91d5-af3883eeb5bf/result'
```

```yaml
experiment:
  allowed_time: 60
  credentials:
    password: password
    username: psma
  unzip:
    args:
    - -o$WORK_DIR\files
    - e
    - $ZIP_FILE
    command: C:\Program Files\7-Zip\7z.exe
  working_directory: C:\Users\psma\Desktop
  vmi:
    hash: f6821e23bb1e10c52438cc02242dd4f5b7469d3a6abb93780edde14a1598c80b
    id: e0c60eb1-302b-4f2a-9597-b823bcc2ab65
  dcs_module:
    args: []
    entrypoint: entrypoint.bat
    hash: 81425fe3e614bd5485939365d908c9a03ecf6031f4386001f3dda0d399b9d227
    id: 46faa557-da7f-4f72-8222-d204819c9427
  dps_module:
    args: []
    hash: c5fbd55c54a976ea1b79ce56877a07eefcb8d9a3e8dfbcd7d52ef625e96e339e
    id: d8660458-6799-48b1-be12-cdd69bd578b0
  id: ff1b4ec4-4c47-4487-91d5-af3883eeb5bf
  samples:
  - hash: 2734cc2af4d39fa9e9895c4411041e7d0b29802260623138fef9badb88b3c138
    original_name: sample1.exe
    status: NOT_STARTED
  - hash: 76c1bc3ad8a48c6f8fd45e7050d00f77886666215b8903e463177e3e5fda61d8
    original_name: sample2.exe
    status: NOT_STARTED
  status: NOT_STARTED
  timestamp: 2021-07-26 14:33:58.921368
```


Figure 3.12: Experiment Output YAML File

The output of request above should be the ZIP file generated by the DPS module, after the processing of the collected data.

```yaml
experiment:
  id: ff1b4ec4-4c47-4487-91d5-af3883eeb5bf
  overall_status: NOT_STARTED
  samples_status:
  - hash: 2734cc2af4d39fa9e9895c4411041e7d0b29802260623138fef9badb88b3c138
    status: NOT_STARTED
  - hash: 76c1bc3ad8a48c6f8fd45e7050d00f77886666215b8903e463177e3e5fda61d8
    status: NOT_STARTED
```

Figure 3.13: Experiment Status YAML File

4

Evaluation

Contents

4.1 Evaluation Design 47
4.2 Evaluation Results 48

This chapter describes the experimental evaluation done of implemented system. As it will be further discussed in Section 4.1, the system was evaluated in terms of performance. In Section 4.2, it is presented the results from the such evaluation.

4.1 Evaluation Design

To test the performance of the system, a module for each DCS and DPS was developed. Since the quality of the experiment-resulting model is out-of-scope, these modules could simply simulate the data collection and data processing phases.

Furthermore, since the data collection and processing were simulated, the samples used did not need to be real malware and could instead be blobs of randomly generated data. However, the samples' size was required to be close to the average size of samples caught in the wild, to closely model the performance of a real experiment. Thankfully, VirusTotal provided a dump of approximate 300 gigabytes of submitted samples, from which it was deduced an upper bound average size of two megabytes per sample. For the experiment to be run, 1024 samples were generated, corresponding to two gigabytes of samples that were analysed by the system.

As for the size of the generated collected data, after some experimentation with system calls monitors, and specifically SpyStudio API Monitor[1], it was considered that 10 megabytes roughly corresponded to the amount of captured data in a minute, therefore this was the time span chosen to execute each sample.

The VMI selected to be loaded into the experiment corresponds to a Windows 7 Ultimate Service Pack 1 virtual machine with two gigabytes of memory and one virtual CPU, which has six gigabytes of size.

The system in which the experiment was executed had the specifications defined in Table 4.1. Taking into account the memory and processor in the system and the required resources for each virtual machine, it was decided to execute 12 DCS Celery workers and one DPS Celery worker.

Resource	Value
Processor	`AMD Ryzen 9 5900X 12-Core Processor 3.70 GHz`
Memory	`DDR4-3600MHz CL16 16GBx2 + 18 GB Swap`
Disk	`500 GB Gen.4 PCIe NVMe M.2`
OS	`Ubuntu 18.04.5 LTS`
Docker version	`20.10.7, build f0df350`
VirtualBox version	`5.2.42_Ubuntu`

Table 4.1: Evaluation System's Specifications

Regarding the metrics over which the evaluation was done, the total time of the experiment was

[1] https://www.nektra.com/products/spystudio-api-monitor/index.html

collected from the logs alongside the individual time for each sample spent inside a DCS node. Additionally, in order to monitor the resources consumption by the system, three extra monitoring services were defined in the `docker-compose.yaml` file. These services – Node Exporter[2], cAdvisor[3] and Celery Exporter[4] – were responsible for the monitoring of the system's resources, docker container's resources and celery related metrics, respectively.

These metrics were then scrapped by a Prometheus[5] service every 15 seconds from which were then presented in a Grafana[6] service in the form of dashboards, which will be presented in the next Section. Finally, to assess the impact of the monitoring services mentioned, execution times of the experiments (both with and without them) will be compared.

4.2 Evaluation Results

As previously mentioned, DCS task execution time related metrics were extracted from the logs of the multiples DCS Celery workers. These metrics were then processed to generate Table 4.2, Figure 4.1 and Figure 4.2.

Monitoring	Min (s)	P50 (s)	P75 (s)	P90 (s)	P99 (s)	Max (s)	Avg (s)	Std Dev
False	69.78	70.25	70.45	70.91	165.54	167.29	72.42	11.48
True	69.71	70.25	70.48	71.16	168.24	170.45	72.82	12.14

Table 4.2: Experiment Tasks' Statistics

Table 4.2 presents the result of various statistical measures from which it can be concluded that most of the samples spend similar time inside a DCS worker and a few outliers take significantly more time, specially when considering the samples are executed inside the Virtual Machines always for 60 seconds. Discarding the fixed execution time, what is left is the time the DCS worker needs to create a new instance of a `VMController`, clone the base Virtual Machine, download the sample and DCS module, boot the Virtual Machine up, power it off, delete it and upload the collected data to the SFTP server. From further analysis of the logs, most of this time is spent in booting up, shutting down and delete the Virtual Machine, as expected.

Additionally, from the logs it was possible to discover that part of the outliers, were due to the Virtual Machine being marked as locked after shutdown, making the deletion of it wait for exactly 29 seconds until reported as unlocked from VirtualBox, which could be the manifestation of an internal scheduled update on the lock status of the virtual machines.

[2] https://hub.docker.com/r/prom/node-exporter
[3] https://hub.docker.com/r/google/cadvisor/
[4] https://hub.docker.com/r/danihodovic/celery-exporter
[5] https://hub.docker.com/r/prom/prometheus
[6] https://hub.docker.com/r/grafana/grafana

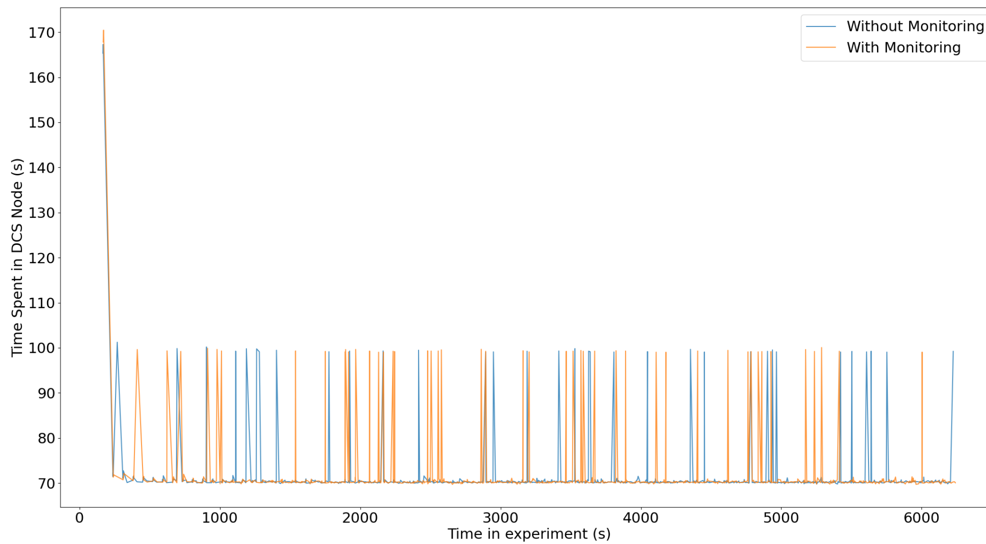

Figure 4.1: DCS Task Execution Time Time-series

In Figure 4.1 it is possible to observe the outliers mentioned previously, alongside the other group of outliers which are the first samples analysed by the workers. This group of samples have higher execution times due to the import of the not-present base VMI which is used in the experiment. This import time, which correspond to time taken to download the VMI and actually import it into VirtualBox, will affect all workers running in a node, as while one worker is performing these actions the others will stay locked before the clone operation, so that multiples instances of the same VMI would be imported into the node.

Figure 4.2 presents the distribution of sample execution times in the DCS node and, re-affirming the previous conclusions, it is possible to observe the three groups mentioned, i.e. the vast majority of samples executing around the 70 seconds mark, the samples which were locked by VirtualBox which run for about 100 seconds and the first batch of samples executed by the workers. For more extended experiments, it can be predicted that the ratio between the first two groups would remain similar to the presented here, while the number of samples locked due to import is only proportional to the number of workers in a node and would be less statistical important with the increase in the length of the experiment.

Furthermore, comparing the experiments with and without monitoring, it can be seen, as expected, a slight increase in task execution time when the monitoring containers are deployed. However, from further analysis of the collected data it is possible to conclude that the statistical increase in DCS task execution times can be attributed mostly to the increase in the number of locked Virtual Machines, as the experiment without monitoring had 34 locked Virtual Machines while the one with monitoring had 47.

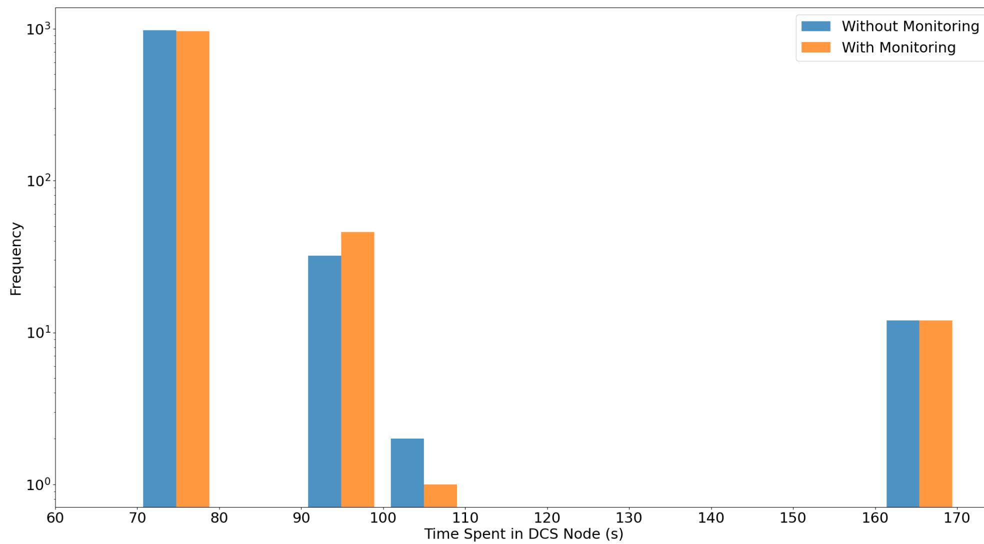

Figure 4.2: DCS Task Execution Time Histogram

Figure 4.3 presents the monitoring dashboard designed for the system during Module and VMI uploads and the execution of the experiment. From it, it is possible to observe the memory exhaustion due to the Virtual Machines running in the system during the experiment and conclude that it is main bottleneck of this system for this configuration, as the CPU usage stayed always below 25%.

Additionally, it is possible to notice the impact on the containers from to the VMI upload and download from the containers due to its size, corresponding to the first burst in CPU and memory usage in the `psma_host` container. The second burst is related to the samples' upload, which had also a considerate size.

Lastly, apart from the initial pressure for the creation of the VMI, and posterior download from the `psma_sftp` container to the DCS node, the containers remained at low resource consumption as expected.

Figure 4.3: Monitoring Dashboard

5

Conclusions

Contents

5.1 Summarized Contribution Analysis 55
5.2 Future Directions 55

5.1 Summarized Contribution Analysis

This work presented the Programmable Sandbox for Malware Analysis (PSMA), a scalable dynamic malware analysis system that enables researchers to perform programmable and repeatable experiments, focusing also on the proper reporting and storage of experiment parameters. The approach used was based on the two-phase norm, data collection and data processing, commonly performed in the dynamic malware analysis research area and used VirtualBox Virtual Machines as the execution environment for the samples to be analysed.

To test the performance of the implemented solution, two modules were developed for simulating the data collection and data processing mechanisms. Additionally, monitoring containers were deployed which confirmed the expectations for the system's performance by running an experiment with 1024 samples of two megabytes.

5.2 Future Directions

The system presented is a proof of concept on what could be a full fledged Platform As A Service (PAAS). As such, some additions and modifications could be made to the system:

A – Change into more performant hypervisors In order to serve more researchers and be capable of performing more experiments at the same time, the change to Type I Hypervisors seem obvious as it removes the Host OS layer, which is resource consuming.

B – Addition of authentication mechanisms For the system to be transformed into a PAAS, its access must be restricted to authorized entities, i.e users. Additionally, if access control to the objects is implemented, it would be possible for researchers to test their approaches privately on the system while they were not ready for publishing.

C – Addition of administrative interface In the tested scenarios, the system was used in a limited environment, which allowed for a rather easy management of the its nodes. However for large scale deployments, this tasks rapidly could become cumbersome. One possible solution for this problem would be extending the Host to include administrative endpoints, from which would be possible to both monitor and manage the various components of the system.

D – Addition of Graphical User Interface Finally, one of the biggest improvements for the usability of the system would have to be the introduction of a Graphical User Interface, be it in the form of a web app or native application.

Bibliography

[1] B. A. S. Al-rimy, M. A. Maarof, and S. Z. M. Shaid, “Ransomware threat success factors, taxonomy, and countermeasures: A survey and research directions,” *Computers & Security*, vol. 74, pp. 144–166, 2018. [ix, 9, 11]

[2] A. Resh, M. Kiperberg, R. Leon, and N. Zaidenberg, “System for executing encrypted native programs,” *International Journal of Digital Content Technology and its Applications*, vol. 11, 2017. [ix, 15]

[3] R. Morabito, J. Kjällman, and M. Komu, “Hypervisors vs. lightweight virtualization: a performance comparison,” in *Proceedings of the 2015 IEEE International Conference on Cloud Engineering*. IEEE, 2015, pp. 386–393. [ix, 20]

[4] VirusTotal. File submission statistics. Accessed 12th December 2020. [Online]. Available: https://www.virustotal.com/en/statistics/ [3]

[5] A. Gazet, “Comparative analysis of various ransomware virii,” *Journal in computer virology*, vol. 6, no. 1, pp. 77–90, 2010. [3]

[6] M. Spagnuolo, F. Maggi, and S. Zanero, “Bitiodine: Extracting intelligence from the bitcoin network,” in *Proceedings of the International Conference on Financial Cryptography and Data Security*. Springer, 2014, pp. 457–468. [3]

[7] Q. Chen and R. A. Bridges, “Automated behavioral analysis of malware: A case study of wannacry ransomware,” in *Proceedings of the 2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA)*. IEEE, 2017, pp. 454–460. [3]

[8] C. Rossow, C. J. Dietrich, C. Grier, C. Kreibich, V. Paxson, N. Pohlmann, H. Bos, and M. Van Steen, “Prudent practices for designing malware experiments: Status quo and outlook,” in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*. IEEE, 2012, pp. 65–79. [4, 21]

[9] A. Mohaisen, O. Alrawi, and M. Mohaisen, “Amal: High-fidelity, behavior-based automated malware analysis and classification,” *Computers & Security*, vol. 52, pp. 251–266, 2015. [14, 18, 20]

[10] W. Lee, S. J. Stolfo, and K. W. Mok, “A data mining framework for building intrusion detection models,” in *Proceedings of the 1999 IEEE Symposium on Security and Privacy (Cat. No. 99CB36344)*. IEEE, 1999, pp. 120–132. [14]

[11] G. Wicherski, “pehash: A novel approach to fast malware clustering.” *LEET*, vol. 9, p. 8, 2009. [15]

[12] Mandiant. Tracking Malware with Import Hashing. Accessed 12th December 2020. [Online]. Available: https://www.fireeye.com/blog/threat-research/2014/01/tracking-malware-import-hashing.html [15]

[13] A. M. Turing, “On computable numbers, with an application to the entscheidungsproblem,” *Proceedings of the London mathematical society*, vol. 2, no. 1, pp. 230–265, 1937. [16]

[14] C. Willems, T. Holz, and F. Freiling, “Toward automated dynamic malware analysis using cwsandbox,” *IEEE Security & Privacy*, vol. 5, no. 2, pp. 32–39, 2007. [17, 18, 22]

[15] D. Kirat, G. Vigna, and C. Kruegel, “Barecloud: bare-metal analysis-based evasive malware detection,” in *Proceedings of the 23rd {USENIX} Security Symposium ({USENIX} Security 14)*, 2014, pp. 287–301. [18, 19]

[16] A. Kharraz, W. Robertson, D. Balzarotti, L. Bilge, and E. Kirda, “Cutting the gordian knot: A look under the hood of ransomware attacks,” in *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2015, pp. 3–24. [18]

[17] A. Kharaz, S. Arshad, C. Mulliner, W. Robertson, and E. Kirda, “{UNVEIL}: A large-scale, automated approach to detecting ransomware,” in *Proceedings of the 25th {USENIX} Security Symposium ({USENIX} Security 16)*, 2016, pp. 757–772. [18]

[18] M. Lindorfer, C. Kolbitsch, and P. M. Comparetti, “Detecting environment-sensitive malware,” in *Proceedings of the International Workshop on Recent Advances in Intrusion Detection*. Springer, 2011, pp. 338–357. [18]

[19] I. Goldberg, D. Wagner, R. Thomas, E. A. Brewer *et al.*, “A secure environment for untrusted helper applications: Confining the wily hacker,” in *Proceedings of the 6th conference on USENIX Security Symposium, Focusing on Applications of Cryptography*, vol. 6, 1996, pp. 1–1. [18]

[20] A. Bulazel and B. Yener, “A survey on automated dynamic malware analysis evasion and counter-evasion: Pc, mobile, and web,” in *Proceedings of the 1st Reversing and Offensive-oriented Trends Symposium*. ACM, 2017, p. 2. [19]

[21] D. Kirat, G. Vigna, and C. Kruegel, “Barebox: efficient malware analysis on bare-metal,” in *Proceedings of the 27th Annual Computer Security Applications Conference*. ACM, 2011, pp. 403–412. [19]

[22] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, “Ether: malware analysis via hardware virtualization extensions,” in *Proceedings of the 15th ACM conference on Computer and communications security*. ACM, 2008, pp. 51–62. [20, 21]

[23] T. K. Lengyel, S. Maresca, B. D. Payne, G. D. Webster, S. Vogl, and A. Kiayias, “Scalability, fidelity and stealth in the drakvuf dynamic malware analysis system,” in *Proceedings of the 30th Annual Computer Security Applications Conference*. ACM, 2014, pp. 386–395. [20]

[24] G. Pék, B. Bencsáth, and L. Buttyán, “nether: In-guest detection of out-of-the-guest malware analyzers,” in *Proceedings of the Fourth European Workshop on System Security*. ACM, 2011, p. 3. [20]

[25] C. Guarnieri, A. Tanasi, J. Bremer, and M. Schloesser, “The cuckoo sandbox,” *Accessed: Dec*, vol. 16, p. 2018, 2012. [20]

[26] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, “Panorama: capturing system-wide information flow for malware detection and analysis,” in *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007, pp. 116–127. [21]

[27] L. Böhne, “Pandora’s bochs: Automatic unpacking of malware,” *University of Mannheim*, vol. 6, 2008. [21]

[28] D. Inoue, K. Yoshioka, M. Eto, Y. Hoshizawa, and K. Nakao, “Malware behavior analysis in isolated miniature network for revealing malware’s network activity,” in *Proceedings of the 2008 IEEE International Conference on Communications*. IEEE, 2008, pp. 1715–1721. [21]

[29] T. Benzel, “The science of cyber security experimentation: the deter project,” in *Proceedings of the 27th Annual Computer Security Applications Conference*. ACM, 2011, pp. 137–148. [23]

[30] A. Årnes, P. Haas, G. Vigna, and R. A. Kemmerer, “Digital forensic reconstruction and the virtual security testbed vise,” in *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2006, pp. 144–163. [23]