Programmable Sandbox for Malware Analysis

(extended abstract of the MSc dissertation)

Diogo Vilela
Instituto Superior Técnico, Universidade de Lisboa

Advisor: Prof. Miguel Nuno Dias Alves Pupo Correia

***Abstract*—Ransomware has grown to be one of the largest cibersecurity threats in the past few years, so efforts in building better detection mechanisms have also increased. Ransomware's mainstream approach of encrypting 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 work 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.

I. Introduction

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. Ransomware, a specific type of malware, is considered to be one of the biggest cibersecurity threats currently, which due to 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. [1] 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.

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 solutions that allows researchers with access to the system to create and update Modules that then can be used by other researchers whilst maintaining the a previous version of those Modules that have possibly been used in experiments reported or worth reporting.

II. Dynamic Malware Analysis

To analyse a malware sample, one could "simply" execute it and monitor its actions in order to extract its behaviour. To this type of procedure is called Dynamic Malware Analysis.

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: The main idea behind monitoring the system call interface is similar to the verification process executed by the operating system (OS), where before a system call is pushed down to the kernel, the monitoring process logs the system call executed, its parameters and then its 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 system calls 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 OS. Additionally, due to updates to the OS, its internals may suffer modifications that can lead to malware written in kernel-mode to stop working as their are highly depended of the OSes internals.

b) Filesystem and Network Activity: 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 [2] and Barecloud [3] record the changes applied to the filesystem after sample execution and use them alongside with knowledge of Windows OSes internals to get the changes applied to the registry. Focusing on filesystem monitoring when analyzing ransomware samples, Kharraz et al. [4] propose monitoring the filesystem activity by monitoring the MFT on NTFS as it is expected that many changes are performed on its entries. To monitor the filesystem UNVEIL [5] used the Windows Filesystem Minifilter Driver, arguing it 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.

Regarding the communication to the outside, it can be an indicator of communication with Command and Control servers and/or attempts of propagation of the malware sample, so it is a common approach in literature to perform network logging, such as in [2], [3], [6], [7].

B. Execution Environments

Executing malware samples typically requires the existence of a dedicated environment which 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 [8]. The main difference between 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 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 [9]. The main four types of environments regarding transparency are the following:

a) Bare-Metal Machine: Bare-metal environments are the most transparent, as they do not execute the sample on top of virtualized/emulated hardware or OS. The lack of virtualization is what makes 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, this approach presents the challenges of restoring the initial system's state after performing each sample execution and extracting the behavior profile of a sample execution as the addition of an in-guest agent would compromise the transparency requirements.

Addressing the first challenge, Barebox [10] presents a new technique for system state restoration which is based on the idea of partitioning the physical memory of the system: one partition for the analysis environment while the other is used as a snapshot of the system to be restored. When the restoration of the system takes place, another OS – external to the memory partition belonging to the analysis environment – restores the system without requiring a reboot.

Addressing the second challenge, Barecloud [3] 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 1, Type I hypervisors have the hypervisor engine, also known as Virtual Machine Monitor (VMM), above the hardware lawyer. This engine is responsible for managing the VM running guest OSes and their access to the system state, i.e. hardware.

Due to the presence of a 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. Also, it is possible for an agent to perform monitoring activities to the memory and since the it 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 time taken by the processes that enable isolation and monitoring, 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 analysed and further stop execution.

Both Ether [11] and DRAKVUF [12] are 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

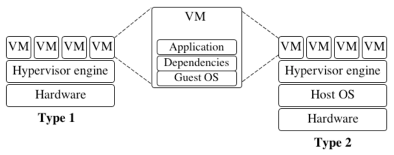


Fig. 1: Hypervisor Types [13]

`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, Type II Hypervisors, also known as Hosted Virtual Machines [14], have a hypervisor engine layer that stays under the VM layer. However, it is placed above the Host OS layer, as presented in Figure 1.

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 [2] 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 VMDK and PCAP files respectively.

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. 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 [15] using the QEMU.

Pandora's Bochs [16] 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 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, leading to more bugs possibly existing, making these systems the less transparent of them all.

C. Challenges

Dynamic malware analysis faces two main challenges, safety and environment detection.

a) Safety: When executing such samples the extent of their 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 [11] 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 [1] as this connection could be used to infected other systems or participate in bigger scale attacks, such as spam-serving or 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 [17] 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 II-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 [6] 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.

III. Proposed System

A. Requirements

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

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 Images (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.

2) Modular: As stated previously, 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 and the Data Processing System, 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 the its processing.

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

3) Scalable: 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.

4) Isolated: As seen in Section II-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 virtualized environments, thoroughly detailed in Section II-A, are often the chosen execution environment they 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 VMI imported to the system.

B. 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.

1) High-Level View: Section III-A 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 2.

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

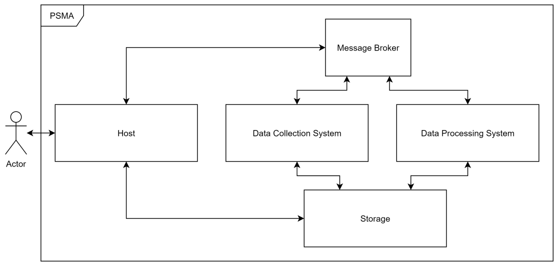


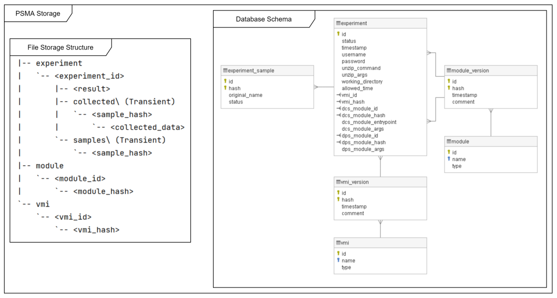
Fig. 2: PSMA's High-Level Architecture

will be run on the System, alongside the Modules and VMIs imported.

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) Message Broker: As seen in the Section III-A, 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.

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. 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.



```
|-- experiment
|   `-- <experiment_id>
|       |-- <result>
|       |-- collected\ (Transient)
|       |   `-- <sample_hash>
|       |       `-- <collected_data>
|       `-- samples\ (Transient)
|           `-- <sample_hash>
|-- module
|   `-- <module_id>
|       `-- <module_hash>
`-- vmi
    `-- <vmi_id>
        `-- <vmi_hash>
```

Fig. 3: PSMA's Storage

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 4.

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 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.

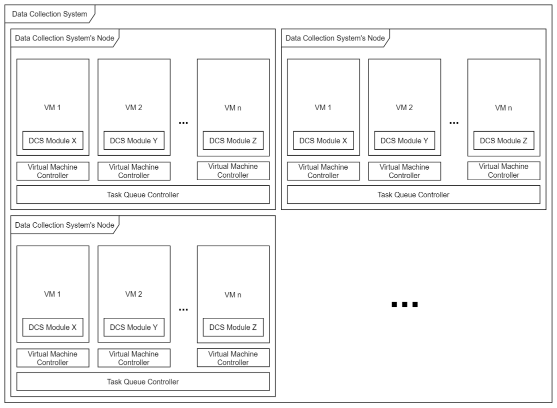


Fig. 4: Data Collection System Architecture

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 5. 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.

For the environment of execution of the data processing module, the system was design to it being 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.

C. 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

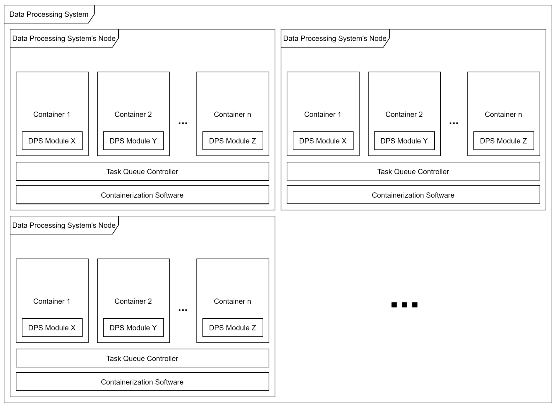


Fig. 5: Data Processing System Architecture

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 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 OSes, 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.

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 SFTP as it relies on the 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. 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 UUID randomly generated when the system receives a completely new object and the hash is the result of computing the the 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 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, 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

[2]https://www.python.org/
[3]https://docs.docker.com/compose/
[4]http://www.paramiko.org/
[5]https://hub.docker.com/r/atmoz/sftp/
[6]https://www.postgresql.org/
[7]https://flask-sqlalchemy.palletsprojects.com/en/2.x/
[8]https://hub.docker.com/_/postgres

itself upon the Alpine Linux project it has a much smaller storage footprint than the alternatives.

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 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, 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) 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 4.

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 OSes. 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 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 OSes. 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.

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 III-B6, 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

[9]https://docs.celeryproject.org/en/stable/
[10]https://redis.io/
[11]https://hub.docker.com/_/redis
[12]https://www.virtualbox.org/
[13]https://github.com/sethmlarson/virtualbox-python
[14]https://docker-py.readthedocs.io/en/stable/

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.

5) Host: As previously mentioned, the Host is the entry-point 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 I 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.

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 I: 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 file in the request alongside the object file. 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 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 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.

[15]https://flask.palletsprojects.com/en/2.0.x/

IV. Evaluation

A. 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[16], it was considered that 10 megabytes roughly corresponded to the amount of captured data in a minute, therefore it was the chosen allotted time 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 II. 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 II: Evaluation System's Specifications

Regarding the metrics over which the evaluation was done, from the logs it was collected the total time of the experiment 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[17], cAdvisor[18] and Celery Exporter[19] – were responsible for the monitoring of the system's resources, docker container's resources and celery related metrics, respectively.

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

[17]https://hub.docker.com/r/prom/node-exporter

[18]https://hub.docker.com/r/google/cadvisor/

[19]https://hub.docker.com/r/danihodovic/celery-exporter

These metrics were then scrapped by a Prometheus[20] service every 15 seconds and then presented in a Grafana[21] service in the form of dashboards, which will be presented in the next Section. Finally, to assess the impact of the monitoring services mentioned, it will be compared the execution times of the experiments both with and without them.

B. 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 III, Figure 6 and Figure 7.

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 III: Experiment Tasks' Statistics

Table III 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.

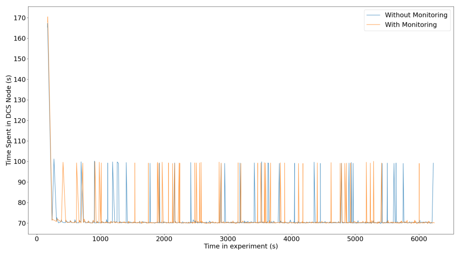


Fig. 6: DCS Task Execution Time Time-series

In Figure 6 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.

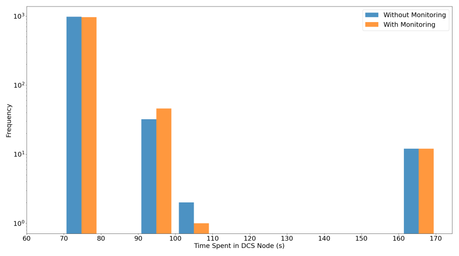
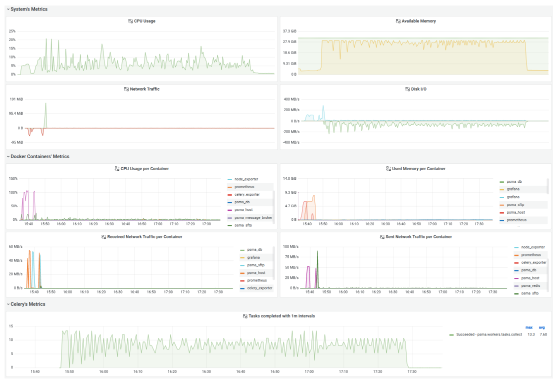


Fig. 7: DCS Task Execution Time Histogram

Figure 7, 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.



System's Metrics
CPU Usage
Available Memory
Network Traffic
Disk I/O
Docker Containers' Metrics
CPU Usage per Container
Used Memory per Container
Received Network Traffic per Container
Sent Network Traffic per Container
Celery's Metrics
Tasks completed with 1m intervals

Fig. 8: Monitoring Dashboard

Figure 8 presents the monitoring dashboard designed for the

[20]https://hub.docker.com/r/prom/prometheus
[21]https://hub.docker.com/r/grafana/grafana

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.

V. Conclusion

A. Summarized Contribution Analysis

This work presented the Programmable Sandbox for Malware Analysis, 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, it was developed two modules that simulate 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.

B. 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 at a small scale 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 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.

References

[1] 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 *2012 IEEE Symposium on Security and Privacy*. IEEE, 2012, pp. 65–79.

[2] 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.

[3] D. Kirat, G. Vigna, and C. Kruegel, "Barecloud: bare-metal analysis-based evasive malware detection," in *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, 2014, pp. 287–301.

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

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

[6] 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.

[7] M. Lindorfer, C. Kolbitsch, and P. M. Comparetti, "Detecting environment-sensitive malware," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2011, pp. 338–357.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] R. Morabito, J. Kjällman, and M. Komu, "Hypervisors vs. lightweight virtualization: a performance comparison," in *2015 IEEE International Conference on Cloud Engineering*. IEEE, 2015, pp. 386–393.

[14] 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.

[15] 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.

[16] L. Böhne, "Pandora's bochs: Automatic unpacking of malware," *University of Mannheim*, vol. 6, 2008.

[17] 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 *2008 IEEE International Conference on Communications*. IEEE, 2008, pp. 1715–1721.