ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

SCUOLA DI SCIENZE
Corso di Laurea Magistrale in Informatica

Dragonfly: next generation sandbox

Relatore:
Chiar.mo Prof.
Marco Prandini

Correlatori:
Dott. Mag. Davide Berardi,
Dott. Mag. Matteo Lodi,
Ph.D. Andrea Melis

Presentata da:
Simone Berni

Controrelatore:
Chiar.mo Prof.
Renzo Davoli

Sessione I
Anno Accademico 2019/2020

Introduction

An endless battle between malwares and malware analysts is fought every day. Many techniques of analysis are deployed, allowing the study of targets in a clean environment. Isolation is commonly provided by sandboxes, but it is not the only way: a new paradigm is emerging, emulation, that allows the study of targets without having to fear that its own infrastructure can be infected.
Malwares are detected and categorized using *rules*, simple regex queries that describe their behaviours and are matched against the static sample, but thanks to the emulation we can move this process a step further: Dragonfly allows deeper and more precise rules that are matched *during* the emulation of the target, allowing even the execution of custom user functions when a rule is matched to bring the analysis to its next step.

The thesis is divided into three chapters, and in the following their description.
Chapter1 provides a brief description of concepts that the reader must know, focusing on emulation, malware analysis methodologies, and sandboxes.
Chapter 2 describes the Qiling frameworks, with a focus on its goal, architecture, and functionalities. Moreover, its explained its usage, the issues that have been found, and the results when used with real world malwares.
The chapter 3 explains what is the personal contribution of the author in the malware analysis field: the main object of this dissertation is Dragonfly, its architecture, usage, rules, results and future development. A brief de-

scription of what has been upgraded inside Qiling to allow the creation of Dragonfly is provided.

Contents

List of Figures

Chapter 1

Overview

Malware is any software intentionally designed to cause damage to a computer, server, client, or computer network[1]. A wide variety of types of malware exist, including computer viruses, worms, Trojan horses, ransomware, spyware, adware, rogue software, and scareware.
Malware analysis is the study or process of determining the functionality, origin and potential impact of a given malware sample[2].
The method by which malware analysis is performed typically falls under one of two types:

- **Static analysis** is performed by dissecting the different resources of the binary file without executing it and studying each component.
- **Dynamic analysis** is performed by observing the behavior of the malware while it is actually running on a host system.

A **sandbox** is a security mechanism for separating running programs from the infrastructure, usually in an effort to mitigate system failures or software vulnerabilities from spreading: the ideal environment to perform malware analysis. A sandbox typically provides a tightly controlled set of resources

[1] `https://docs.microsoft.com/en-us/previous-versions/tn-archive/dd632948(v=technet.10)` (visited on 16/05/20)

[2] `https://web.archive.org/web/20160418151823/http://www.ijarcsse.com/docs/papers/Volume_3/4_April2013/V3I4-0371.pdf)` (visited on 16/05/20)

for guest programs to run in, such as storage and memory space. Network access, the ability to query the host system and read from input devices are usually disallowed or heavily restricted, not allowing the sample analysed to perform malicious activities on the host machine.

To solve the issues of configuring and maintaining a complex sandbox system, malware analysis is going toward a new paradigm, **emulation**. It is possible to emulate the malware, directly in the host machine, without building the infrastructure that a sandbox requires, and, it is possible to have a fine-grain control of the environment that the malware is using.

1.1 Emulation

Emulation commonly refers to the use of one hardware device to mimic the function of another hardware device. In that sense, emulation is the use of software to emulate hardware, and on top of it, it is possible to execute an higher level of software.
The core of the emulation is the **translation** of the target instruction set to the user instruction set. CPU instructions must be translated, but even the memory management and the GPU instructions have to be considered.
After having emulated the hardware, the foundations are done: now it is possible to build over this emulated hardware new software, add features, or let the user monitor and change the behaviour of the lower levels. The supervision of the emulation is done through *hooks*, a sort of APIs that allows the users to execute custom callback functions when a particular condition is found.

1.1.1 QEMU

QEMU[3] is a hosted virtual machine monitor: it emulates the machine's processor through dynamic binary translation and provides a set of different hardware and device models for the machine, enabling it to run a variety of guest operating systems. It can also be used with KVM[4] to run virtual machines at near-native speed (by taking advantage of hardware extensions such as Intel VT-x). QEMU can also do emulation for user-level processes, allowing applications compiled for one specific architecture to run on top of another one.
QEMU can be used in two different operating modes:

- **User-mode emulation**: it is able to run Unix programs that were compiled for a different instruction set.
- **System emulation**: it is able to run operating systems for any machine, on any supported architecture.

The core binary translation engine that allows QEMU to emulate foreign processors on any given supported host is called *The Tiny Code Generator*: the TCG works by translating each guest instruction into a sequence of host instructions. As a result there will be a level of inefficiency, which means TCG code will not be as fast as running native code.

1.1.2 Unicorn Engine

Unicorn Engine[5] is a lightweight, multi-platform, multi-architecture CPU emulator framework based on QEMU. Unicorn offers some unparalleled features:

- Multi-architecture: ARM, ARM64 (ARMv8), M68K, MIPS, SPARC, and X86 (16, 32, 64-bit)

[3]`https://github.com/qemu/qemu` (visited on 17/05/20)
[4]`https://www.linux-kvm.org/page/Main_Page` (visited on 17/05/20)
[5]`https://github.com/unicorn-engine/unicorn` (visited on 17/05/20)

- Clean/simple/lightweight/intuitive architecture-neutral API
- Implemented in pure C language, with bindings for Crystal, Clojure, Visual Basic, Perl, Rust, Ruby, Python, Java, .NET, Go, Delphi/Free Pascal, Haskell, Pharo, and Lua.
- Native support for Windows & *nix (with Mac OSX, Linux, *BSD & Solaris confirmed)
- High performance via Just-In-Time compilation
- Support for fine-grained instrumentation at various levels
- Thread-safety by design

Unicorn is designed to be used as a framework to create over it different tools: its showcase is endless, and some of the most famous ones that use or are built over Unicorn are the following:

- Radare2[6]
- Gef[7]
- Angr[8]
- Pwntools[9]
- Cuckoo[10]
- Unicorn-afl [11]

[6]`https://github.com/radareorg/radare2` (visited on 18/06/20)
[7]`https://github.com/hugsy/gefn` (visited on 18/06/20)
[8]`https://github.com/angr/angr` (visited on 18/06/20)
[9]`https://github.com/Gallopsled/pwntools` (visited on 18/06/20)
[10]`https://github.com/cuckoosandbox/cuckoo` (visited on 18/06/20)
[11]`https://github.com/Battelle/afl-unicorn` (visited on 18/06/20)

The power of Unicorn is that it is possible to insert *hooks* during the emulation: it is possible to read or write a specific address, execute a callback if a specific opcode is found, hook an entire block or a specific interrupt, making it useful to understand at a deeper level how the sample emulated really works.
Many bugs and zero-days are found using Unicorn, and the tools that are built over it: the most common practice these days is to use *fuzzing*[12].

1.1.3 WSL

Windows Subsystem for Linux (WSL) is a compatibility layer for running Linux binary executables (in ELF format) natively on Windows 10 and Windows Server 2019. The first release of WSL provides a Linux-compatible kernel interface developed by Microsoft, containing no Linux kernel code[13] which can then run a GNU user space on top of it, like Ubuntu or Debian. Such a user space might contain a bash shell and command language, with native GNU command-line tools, programming-language interpreters, and even graphical applications.
In May 2019, WSL 2 was announced, introducing important changes such as a real Linux kernel, through a subset of Hyper-V features[14].

1.2 Malware Analysis

1.2.1 Dynamic Analysis

Dynamic analysis is the technique to execute the malware and analyse its behavior during run time. A good practice is not to run the malware directly

[12]`https://docs.microsoft.com/en-us/previous-versions/software-testing/cc162782(v=msdn.10)` (visited on 18/05/20)
[13]`https://mikegerwitz.com/2016/04/gnu-kwindows` (visited on 19/06/20)
[14]`https://devblogs.microsoft.com/commandline/announcing-wsl-2/` (visited on 19/06/20)

to avoid any harm caused by it, so some steps to isolate the effects of the malware must be made.
Commonly a sample is analysed inside a Virtual Machine, allowing the creation of an isolated environment where the malware can be analysed in *almost* total security.

Many tools can be used to perform dynamic analysis: debuggers do the work for a manual analysis, but more complex tools must be used to automatize the process, like sandboxes. The description of what a sandbox is, how to use it, and what it can provide to its users, is provided in section 1.3.
The main issue that is found during dynamic analysis, is that a smart malware is able to recognize to be analysed: it will change its behaviour and signatures, making its study useless and misleading.

1.2.2 Static analysis

Static program analysis is the study of computer software that is performed without actually execute the sample, avoiding its malicious effects on the host machine.
There are an endless number of tools that can be used to perform static analysis on a sample, like Ghidra[15], IDA[16] or BinaryNinja[17].
It is possible to follow the execution flow of the sample, trying to understand its nature, or it is possible to check precisely elements, like strings, system-calls or Windows API, that can match a typical malware behaviour.
The main idea of static analysis is to understand the execution branches that the sample can follow, obtaining an overview of what it is going to do, and for which reason.

[15] `https://github.com/NationalSecurityAgency/ghidra` (visited on 21/06/20)
[16] `https://www.hex-rays.com/products/ida/` (visited on 21/06/20)
[17] `https://binary.ninja//` (visited on 21/06/20)

Signatures

As the two sections before tried to explain, many aspects of the sample must be analysed before it is possible to understand if the target is a malware or not. Moreover, this study requires time, a resource that sometimes analysts do not have. For these reasons, what it is commonly done, since it is not possible to manually analyse every sample that is found, is to use *signatures*. A signature encodes characteristics that distinguish a malware from a genuine executable, and must be created before being able to use them against a new target. For this reason it is still necessary to manually analyse samples, obtaining new features that distinguish malwares, or even characteristics that are present only in a precise malware family, recognizing them.

Obviously this method can produce false positive and false negative, but malware analysts are aware of that and precautions to lower these percentage have been taken.
In fact each rule has a weight, which embodies the probability to correctly indicate that the sample is a malware. The higher the weight, the higher the probability that the target is a malware. Every rule that an analyst has in its database can be matched against the same sample, increasing the probability that the binary analysed is a malware: if the total weight exceed a threshold, the target is considered malicious and a deeper study will be made.
Rules are commonly applied during static analysis against a sample, but nothing stops to use them against the output of a dynamic analysis tool. At the end, rules are matched using **regular expressions**, meaning that every type of files can be matched.
The standard to create and share signatures is YaraRules[18], and its made of four components:

- A unique **name** to distinguish rules between them.

[18]`https://github.com/Yara-Rules/rules`

- A **meta** section where the author, the weight and the references are provided.
- A **strings** section, where the strings, or bytes, that describe a malware behaviour are written.
- A **condition** section, where it is possible to specify constraints about the strings section.

VirusTotal[19] created the Yara-Python[20] library, allowing the use of YARA from a Python program, increasing the ease of use. The following is an example of YaraRule that should match if the sample tries to disable the host antivirus.

```
 rule disable_antivirus {
 meta:
     author = "x0r"
     description = "Disable AntiVirus"
     version = "0.2"
 strings:
     $p1 = "Software\\Microsoft\\Windows\\CurrentVersion\\
         Policies\\Explorer\\DisallowRun" nocase
     $p2 = "Software\\Microsoft\\Windows\\CurrentVersion\\
         Uninstall\\" nocase
     $p3 = "SOFTWARE\\Policies\\Microsoft\\Windows Defender"
         nocase
     $c1 = "RegSetValue"
     $r1 = "AntiVirusDisableNotify"
     $r2 = "DontReportInfectionInformation"
     $r3 = "DisableAntiSpyware"
     $r4 = "RunInvalidSignatures"
     $r5 = "AntiVirusOverride"
     $r6 = "CheckExeSignatures"
     $f1 = "blackd.exe" nocase
     $f2 = "blackice.exe" nocase
     $f3 = "lockdown.exe" nocase
```

[19]https://www.virustotal.com/ (visited on 22/06/20)
[20]https://github.com/VirusTotal/yara-python (visited on 22/06/20)

```
        $f5 = "taskkill.exe" nocase
        $f6 = "tskill.exe" nocase
        $f7 = "sniffem.exe" nocase
        $f8 = "zlclient.exe" nocase
        $f9 = "zonealarm.exe" nocase
    condition:
        ($c1 and $p1 and 1 of ($f*)) or ($c1 and $p2) or 1 of (
           $r*) or $p3}
```

1.3 Sandbox

Sandbox is a traditional approach in which it is possible to execute files in a self-governing virtual computerized technology excluding any physical harm to the host machine. While running these files in sandbox, the sandbox system can highlight malicious activities, such as modification entry in registry, deleting and uploading files in a system[6]. Moreover, sandbox are able to retrieve artifacts from the analysed sample: temporary files that it creates, files deleted or modified, and even network traffic that it generates. The artifacts are then saved and it is possible to separate them independently, with different tools depending on the type of file.

A sandbox is the most common environment for analyse a malware, automating the analysis steps and producing a clear output about the predicted maliciousness of the sample, and the artifacts that it generates. The detection is commonly done through rules: this methodology was explained in section 1.2.2.
Many sandbox are available, some of them are open source, some are not. The following sections describe only few of the open source freely available, while closed source solutions, like VmWare[21] will not be discussed.

[21]https://www.vmware.com/

1.3.1 Cuckoo

Cuckoo[22] is the leading open source automated malware analysis system. It can be used to analyze:

- Generic Windows executables
- DLL files
- Microsoft Office documents
- PDF documents
- URLs and HTML files
- Almost every other possible attack vectors

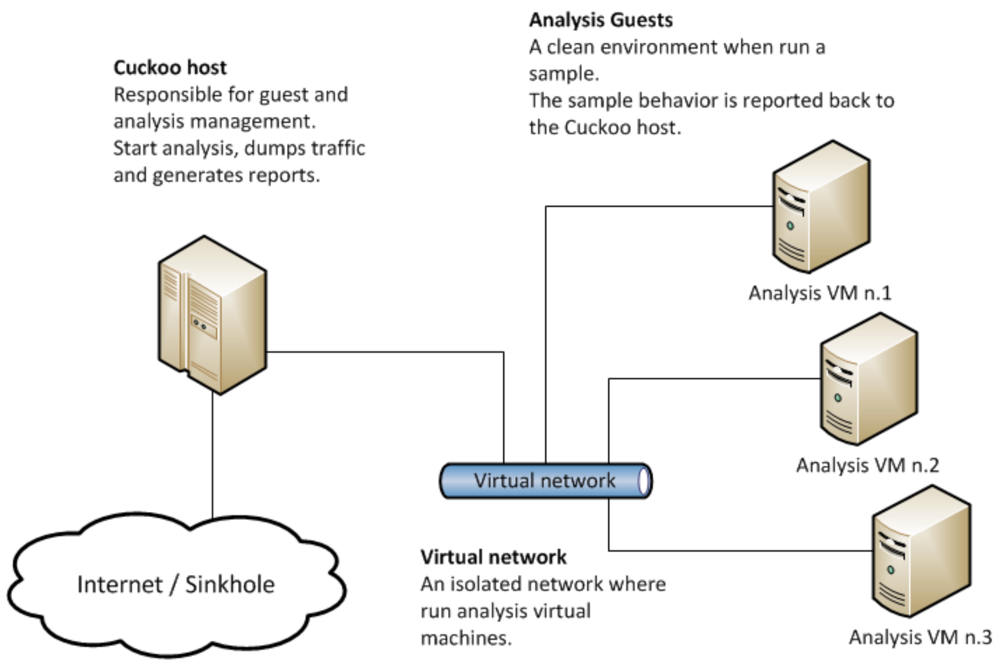

Analysis Guests
A clean environment when run a sample.
The sample behavior is reported back to the Cuckoo host.
Cuckoo host
Responsible for guest and analysis management.
Start analysis, dumps traffic and generates reports.
Analysis VM n.1
Analysis VM n.2
Virtual network
Internet / Sinkhole
Virtual network
An isolated network where run analysis virtual machines.
Analysis VM n.3

Figure 1.1: Cuckoo architecture

Its architecture is shown in figure 1.1, and it is made by two parts primary: the host that manages the guests virtual machines and is in charge of post

[22]https://github.com/cuckoosandbox/cuckoo

execution analysis, and many guest machines that are responsible to execute the samples that the host sends, storing every artifact that was possible to retrieve.
Cuckoo provides a huge number of rest API that can be used to submit and review the samples, for example:

- `/tasks/create/file`
- `/tasks/create/url`
- `/tasks/create/submit`
- `/tasks/view`
- `/tasks/summary`

Cuckoo analysis is split in two parts: the creation of the artifacts, and the analysis of them. The former is done by Cuckoo itself, thanks to the many different plugins that are installed:*procmemdump*, for example, is on them that, if enabled, dumps the entire memory of the process.
But Cuckoo does not manage the sample analysis itself: it uses external tools that must be installed in and executed into the host. Oletools[23], for example, is used to analyse *.doc* files, Wireshark[24] to find information inside *.pcap*, a memory dump is analysed using Volatility[25], a binary or text file with YARA, and so on.
This explains why Cuckoo setup and maintenance require an entire team:

- Every guest OS that may be used, must be installed and configured.
- Every guest software that will be used to execute the many different samples must be installed, remembering to install different versions for compatibility.

[23]`https://github.com/decalage2/oletools` (visited on 23/06/20)
[24]`https://www.wireshark.org/develop.html` (visited on 23/06/20)
[25]`https://github.com/volatilityfoundation/volatility` (visited on 23/06/20)

- Every host analysis tool must be installed, with probably many versions for each one, to keep compatibility.
- It is necessary to configure the network routing both between the host machine and the company infrastructure, and between the host and the many guests.
- Script everything!

For these reasons, Cuckoo requires more than a single person to run correctly, and many figures that are needed are sysadmin and programmers, not even malware analysts.
Moreover, if the OS environment that was created is not sufficient, meaning that the target analysed must be run against a different OS, with a different environment, it is necessary to rebuild and reconfigure from scratch the OS image. This operation requires time, probably a significant amount of time, that is not possible to have at all during a malware infection that is taking place.
Written in Python2, the idea to switch to Python3 started in 2015[26] but was never realized. For this reason, it is hard that new people and companies will invest their time, and money, in a tool that uses a language that is not longer supported. Of course there is an exception, and a Dutch company, called *hatching*[27] has participated for years in the development of Cuckoo: their main sell is to customize a Cuckoo instance, built over the client needs.

1.3.2 Drakvuf

Drakvuf[28]is a virtualization based agentless black-box binary analysis system, allowing in-depth execution tracing of arbitrary binaries (including operating systems), all without having to install any special software within the

[26]`https://github.com/cuckoosandbox/cuckoo/issues/594` (visited on 23/06/20)
[27]`https://hatching.io/`
[28]`https://drakvuf.com/` (visited on 24/06/20)

virtual machine used for analysis.
It currently supports the following guests OS:

- Windows 7-8, both 32-bit and 64-bit
- Windows 10 64-bit
- Linux 2.6.x - 5.x, both 32-bit and 64-bit

The main feature of Drakvuf is its almost undetectable footprint from the target point of view, allowing its use for malware analysis.
Its stealthiness is required because, as dynamic malware analysis systems have become widely deployed, malware has evolved to detect and evade such systems by either refusing to execute in a sandboxed environment, or by modifying its runtime behavior to lead the analysis system astray[9].

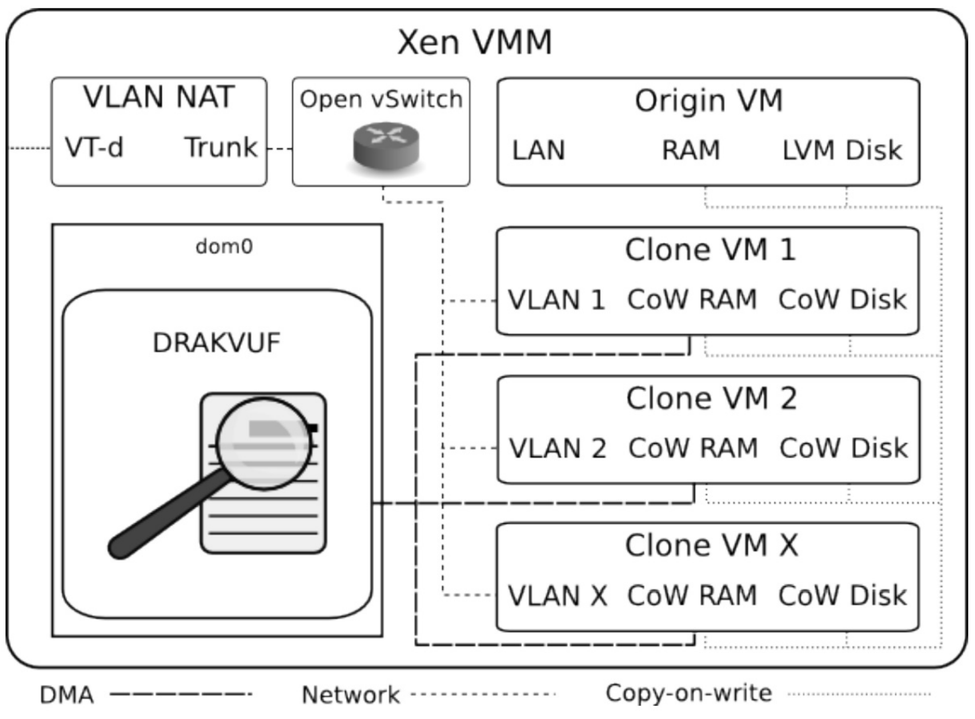


Figure 1.2: Drakvuf architecture

Drakvuf architecture is similar to Cuckoo, as shown in figure 1.2, the main difference is that Drakvuf is implemented with the use of virtualization technology, in particular it is built on the open-source Xen VM[29], as virtualization provides several benefits that dynamic malware analysis can take advantage of: this provide external access to the state of the virtualized hardware components, commonly referred to as virtual machine introspection (VMI), increasing the effectiveness of the analysis.
Drakvuf ability to hide itself from being detected by the target is done through the capacity to inject the target process inside another process running inside the guest, without the aid of any in-guest helper.
Moreover, this sandbox offers deep monitoring functionalities:

- **Execution tracing**: Drakvuf is able to trace not only windows API, but also kernel calls, by directly trapping internal kernel functions via breakpoints injection, allowing to monitor malicious drivers as well as rootkits. To establish a map of internal kernel functions, the Rekall[30] forensics tool has been used.

- **Monitoring file system accesses with memory events**: When a file is accessed, either by the OS or by a user-land process, a `_FILE_OBJECT` is allocated within the kernel heap with the accompanying tag, `Fil\xe5`. This allows Drakvuf to determine the full path of the file as well as the access privilege with which the file is accessed without the need to have any deeper understanding of the file system itself.

- **Carving deleted files from memory**: When files are created and destroyed rapidly, as it is often the case when malware is being dropped on a system, the files are never written to disk. In Drakvuf, the carving of deleted files is implemented by intercepting specific internal kernel calls that are responsible for file deletion, such as the `NtSetInformationFile` and `ZwSetInformationFile` routines.

[29]`https://xenproject.org/` (visited on 24/06/20)
[30]`http://www.rekall-forensic.com/` (visited on 24/06/20)

Chapter 2

Qiling

Qiling[1] is an emulation framework that it is in an active state of development since October 2019, made by the same team that worked on the creation of Unicorn Engine.
Its main features are the following:

- Cross platform: Windows, MacOS, Linux, BSD
- Cross architecture: X86, X86_64, Arm, Arm64, Mips
- Multiple file formats: PE, MachO, ELF
- Emulate and sandbox machine code in a isolated environment
- Support cross architecture and platform debugging capabilities
- Provide high level API to setup and configure the sandbox
- Fine-grain instrumentation: allow hooks at various levels (instruction/basic-block/memory-access/exception/syscall/IO/etc)
- Allow dynamic hotpatch on-the-fly running code, including the loaded library

[1]`https://github.com/qilingframework/qiling` (visited on 10/06/20)

- True framework in Python, making it easy to build customized security analysis tools on top

Qiling is interesting because it tries to merge together the concepts of emulation and, partially, of a sandbox. Unicorn Engine is in charge of the emulation of the sample and, at the same time, is the core of Qiling. Like a sandbox needs an operating system environment before being able to execute the target binary, Qiling is able to build the environment without requiring any kind of software installation. Commonly, executables gain the knowledge of the external world thanks to systemcalls, if the sample is a Unix file, or via Windows Api, in the case of a Portable Executable file. Both the communication ways are emulated inside Qiling, allowing to control and monitor every aspect of the interaction between the target and its environment.
Qiling *is not* an analysis tool, but is a framework, designed to allow the construction of tools over it: storing every information that a possible tool may need, Qiling is the perfect foundation to build dynamic analysis tool, or to reproduce the concept of sandbox inside the emulation paradigm.

2.1 Comparisons

Qiling is the *next step* of the emulation paradigm, since it will emulate an entire operating system, something that was not possible before this project. It is necessary to consider the pros and cons of this framework, compared to other emulating projects or sandboxes, to understand if the Qiling idea is needed.

2.1.1 Cuckoo

As described in section 1.3.1, Cuckoo requires an enormous team work for its maintenance, that Qiling does not need. It is easy to change the configuration parameters of the emulation and the environment that sample must be run in, like OS variables and even hardware components.

But, while Cuckoo is able to test a multitude of attack vectors, like `.pdf`, `.doc` files, Qiling is able to emulate only executable files.
The installation of Qiling is literally done through a single command, `pip install --user qiling`, while Cuckoo requires a specific host configuration, making it a lot easier to install and deploy.
It is necessary to remember that Qiling is an emulation framework, not designed to be a malware analysis tool, nor a sandbox. For this reason, it is not able to produce an output of the goodness of the target, opposed to Cuckoo.

2.1.2 Unicorn Engine

While Unicorn Engine is able to emulate only CPU instructions, it is not able to understand high level concepts such as dynamic libraries, system calls, I/O handling or executable formats like PE, MachO or ELF. Qiling goal and design is to overcome this restrictions, allowing the emulation of an entire executable, but at the same time, keeping the ability to have fine-grain instrumentation.

2.1.3 QEMU

QEMU in user mode is able to emulate every Linux and BSD executable from a Unix host. For example it does not allow to emulate natively a Portable Executable file, like Qiling is able to.
Technically it is possible to use QEMU to emulate the entire Windows OS, then run the sample inside that emulation, and check the result, using the system mode that was described in section 2.1.3. There are many issues in that solution:

- It is not possible to retrieve information of what is happening inside the emulated OS, having zero visibility of the sample behaviour
- It is not possible to hook systemcall or modify the OS environment runtime

- It is necessary to install and configure entire OS

2.1.4 Usercorn

Usercorn[2] is very similar to Qiling as final goal: trying to emulate a complex environment but still allowing to have fine-grain instrumentation. Like Qiling, it is a framework, but the problem is that its scope is limited to emulate only Linux binaries, and since the vast majority of malwares are Windows PE, its use is too much limited for being used in a production environment.

2.1.5 Binee

Binee[3] is a tool written in Go by CarbonBlack. Its goal is similar to Qiling's one: the emulation of binaries, with a focus on introspection of all IO operations. Its primary feature is to provide a flexible environment for determining side effects on the system made by the sample. The issue is that it is able to emulate only Windows sample, limiting, like Usercorn, the numbers of sample that can be analysed.
Moreover, its Windows support is not fully implemented yet: the number of Windows API implemented are the same as the one implemented inside Qiling. For these reasons it is hard to find a real use case.

[2] `https://github.com/lunixbochs/usercorn` (visited on 26/06/20)
[3] `https://github.com/carbonblack/binee` (visited on 26/06/20)

2.2 Architecture

The architecture of Qiling is articulated, complex and undocumented: only the source code describes the real story of this project, and the documentation is still under development.

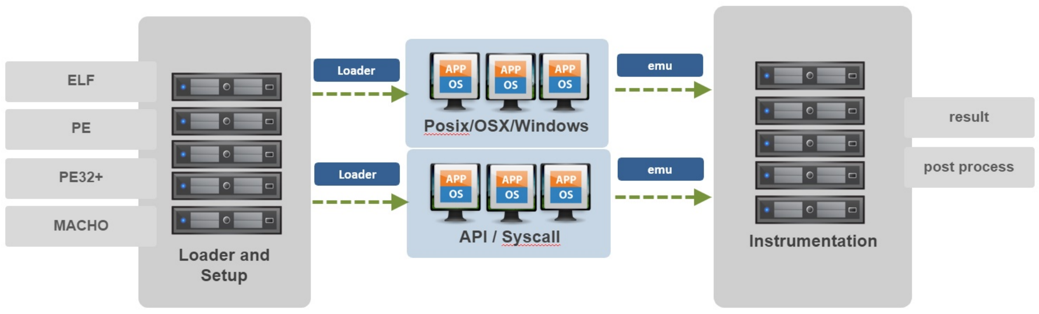


Figure 2.1: Qiling architecture [11]

Figure 2.1 shows how Qiling is structured from an high point of view. The concept is simple: the executable file, whatever kind it is, is loaded inside Qiling, and a general setup is started. Then the systemcalls or API are hooked, meaning that when the executable will call a function, a Qiling implementation will be called. At this point it is possible to connect instrumentation hooks, giving to its users the control. Finally a post process file is created, containing the information retrieved during the emulation.

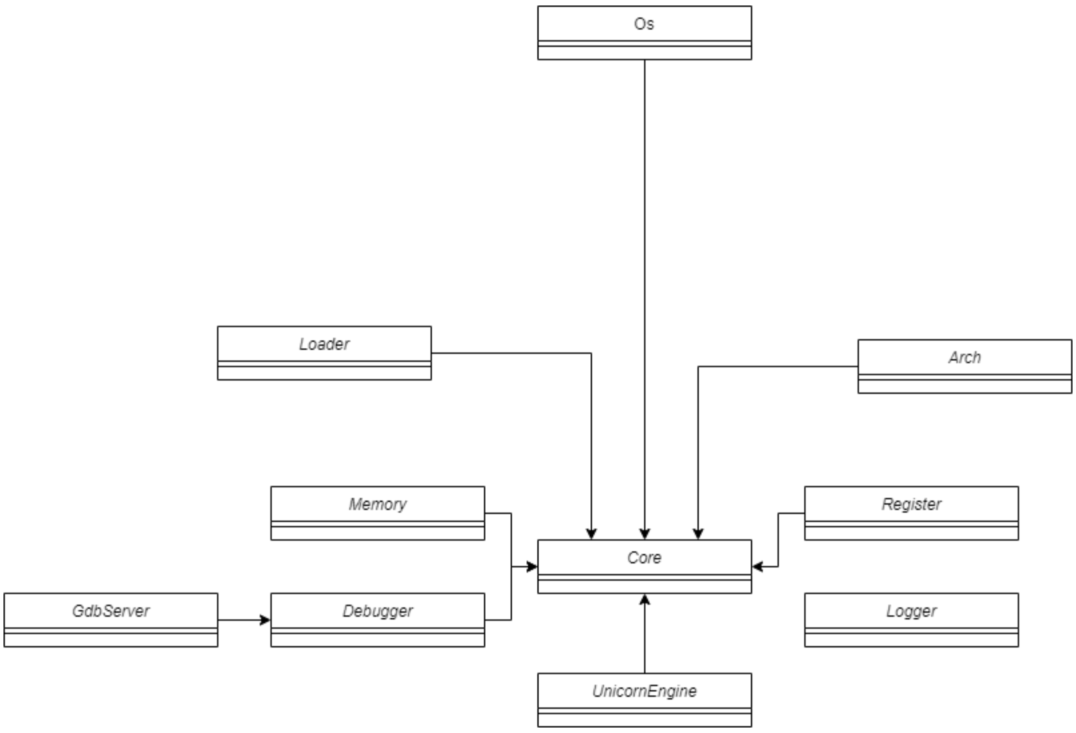


Figure 2.2: Qiling core class architecture

The class *Core* is the main class of the framework, and its task is to create the Qiling object and load inside its components, as figure 2.2 shows.
Each component is an abstract class that will have many, different implementations, depending on the OS that must be emulated, the hardware that should be used, and the type of executable that is given as input.

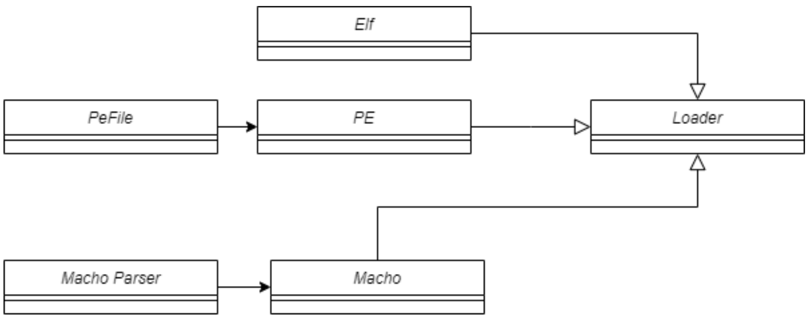


Figure 2.3: Qiling loader class architecture

The *Loader* task is to, as its name explains, load every information about the binary in memory. Figure 2.3 shows the classes that Qiling implements for doing that, each one tasked to load a different type of binary: the class *PE* will load a Portable Executable file, the *Macho* a macOs file, and so on. Moreover it is necessary to load libraries, that will change depending on the executable, basic structures that must be already in the memory, the arguments, and more.

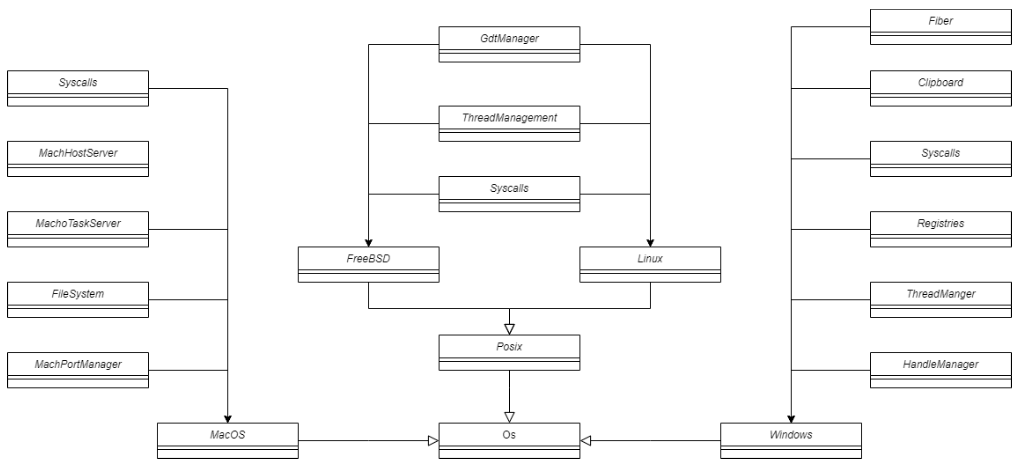


Figure 2.4: Qiling os class architecture

The second core component of Qiling, is the *Os* abstract class, implemented by many different objects, as shown in figure 2.4. Each class is a different operating system that Qiling supports, allowing its emulation to some degrees.
These classes are tasked to the setup and the hook of systemcalls, or Windows API, and right now Qiling implements 20% of the functions inside Kernel32.dll and 30% for Unix systemcalls. Another concept that every Os class must implement, is the multi threading, and, in the case of the Windows object, the registries and handle manager.

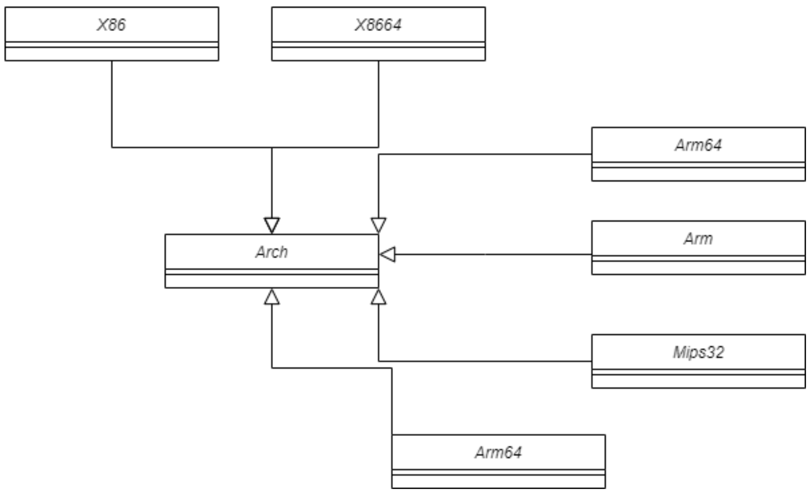


Figure 2.5: Qiling arch class architecture

The last core component is the *Arch* abstract class. Not only it is necessary to emulate different operating system, but even different family of instruction set architectures: figure 2.5 shows which architectures are available at the moment inside Qiling. Their task is to properly create, set and modify the registries that *Unicorn Engine* will use for the emulation, to manage the stack and the segments that the emulator needs.

2.3 Usage

Since Qiling is a Python framework, its basic usage is to import it as a package, and use the Qiling object with its parameters configurable during the object creation. The main function is *run()*, that, as the name denotes, starts the emulation. The sample is loaded in the memory, the shared libraries are loaded, and the emulation is run until the end of the execution. What was described is the basic usage of Qiling, but normally its users want the possibility to have instrumentation.
For doing that, Qiling offers hooks at different levels, giving the possibility to add a user defined callback that works with the entire Qiling object.

- **hook_address** when the program counter is at the defined address, the user defined function will be executed.

- **hook_mem_read** every time a read in memory is performed, the user defined function will be executed.
- **hook_mem_write** every time a write in memory is performed, the user defined function will be executed.
- **hook_code** before every instruction, the user defined function is executed.
- **set_syscall** a Qiling systemcall is replaced by the user defined function.
- **set_api** a Qiling Windows API is replaced by the user defined function.

When the emulation is completed, a log file is saved and it is possible to retrieve the log file created by Qiling, containing every library loaded, every Windows APIs (or systemcalls) called with its parameters. The following snippet shows what was described before, hooking a specific address to stop the emulation and printing the log file.

```
import qiling
def stop(ql, default_values):
        print("Ok for now")
        ql.emu_stop()
if __name__ == "__main__":
    ql = Qiling(["../examples/rootfs/x86_windows/bin/GandCrab502.bin"], "../
        examples/rootfs/x86_windows", output="debug", profile="profiles/
        windows_administrator.ql")
    ql.hook_address(stop, 0x40860f)
    ql.run()
    print(open("../examples/rootfs/x86_windows/GandCrab502.log").read())
```

2.4 Issues

Three main issues were found on the study of Qiling:

- A lack of operating system functions implemented : only 30% of Unix systemcalls have been implemented, and Windows APIs percentages are even lower.

- A configuration file is missing, meaning that an end user does not have the possibility to change the parameters of the emulation, or the OS environment.
- A way to save the results obtained through the sample emulation inside the Qiling object, as a real framework must do, removing the necessity to the final user to parse the log file.

2.5 Results

Qiling has been tested only in an academic environment, meaning that the binaries where not complicated, very linear and the majority were made by the same people that developed Qiling. A real malware was partially emulated, the famous ransomware WannaCry[4] , but only the first section, since it is not possible to use Qiling to make `post` and `get` requests.
When Qiling was tested with real word Windows malwares, the results were not satisfying enough: almost 99% of the tested sample crashed, and the number of APIs called is very low. The main cause of crashes is simply that not every Windows APIs have been implemented inside Qiling, making it impossible to continue the emulation of the binary target. There are some cases where Qiling does not correctly behave like it is supposed to, since most of APIs have many side effects that were not considered during the implementation.

[4]`https://blog.kartone.ninja/2019/05/23/malware-analysis-a-wannacry-sample-found-in-the-wild/` (visited on 26/06/20)

Chapter 3

Dragonfly

Dragonfly is a tool that merges together the concepts of sandbox and emulation. Built over Qiling, its goal is to detect if a particular sample is a malware or is safe to execute.
Inspired by *YARA*, the detection is done through rules that encode a malicious behaviour into a signature. Thanks to the emulation, it's possible, for example, to verify the goodness of a *Portable Executable* file from a Unix machine, a *Macho* executable from a Windows computer, and so on. With Qiling's power, the rules can be deep and precise as one wishes: it's possible to create signatures using strings, systemcalls or Windows API. It is possible to set watch points at any address, check if a Dll has been loaded or not, or if a Windows registry has been accessed.
Moreover, Dragonfly rules are matched **during** the emulation: signatures have the possibility to execute custom made functions, *actions*, when a match is found, empowering the rules to the next level.

Qiling, before this project, was designed to be only an emulation framework and the malware analysis was out of its scope. For this reason, a good part of the project was to design and implement the necessary changes inside Qiling to have enough expressive power, allowing the creation of Dragonfly.

3.1 Architecture

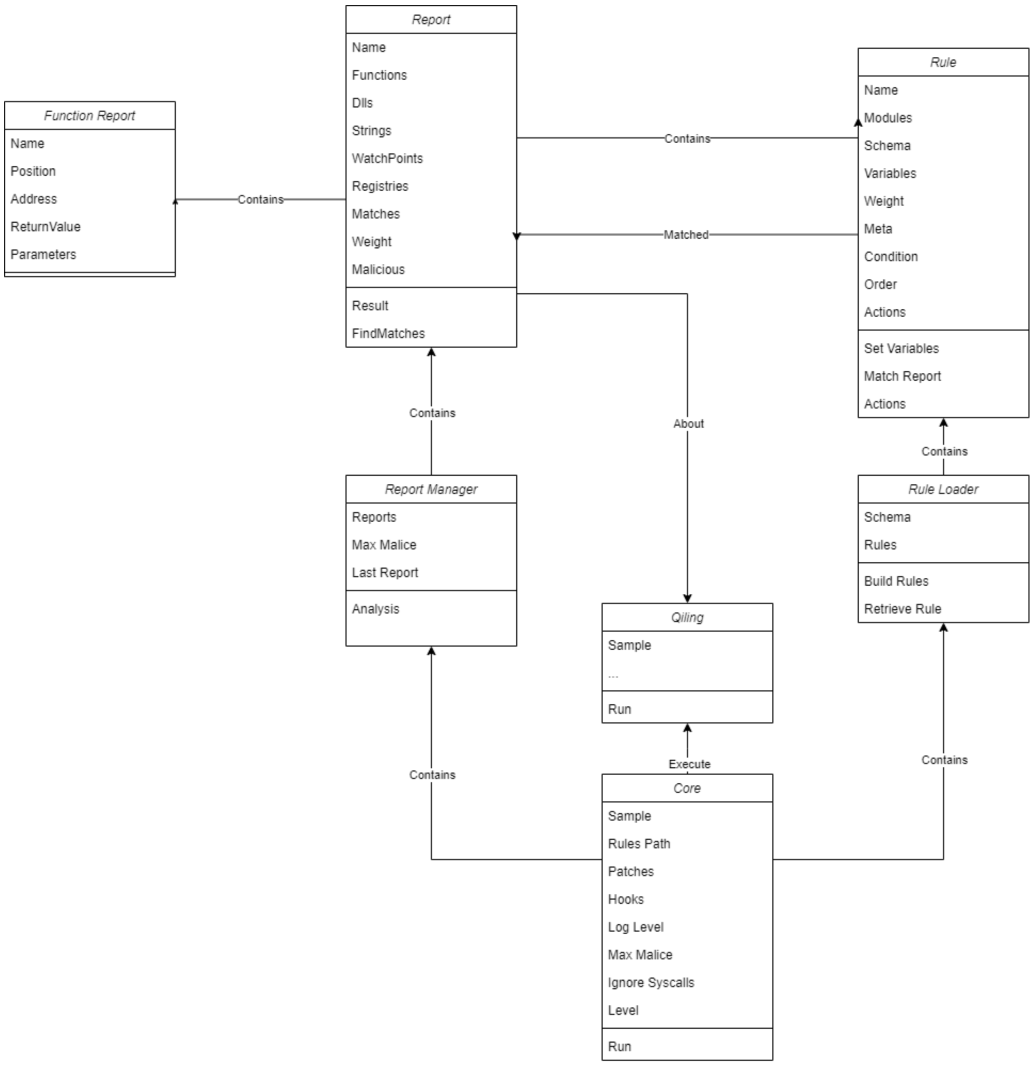


Figure 3.1: Dragonfly's class diagram

The architecture of Dragonfly is quite simple: everything was designed and built around Qiling. Figure 3.1 describes how each class is connected to the others. It's understandable that the fulcrum of Dragonfly is the `Core` class: its main task is to run the emulation of the sample, using Qiling as a library.

It's possible to split the diagram in two parts: the left side, composed of

`Report Manager`, `Report` and `Function Report` have the task to retrieve and store information about what happened during the emulation: which systemcall or Windows API has been called, which registry has been accessed, strings used and so on. In this way it is possible to standardize the information that Qiling has split inside many classes.

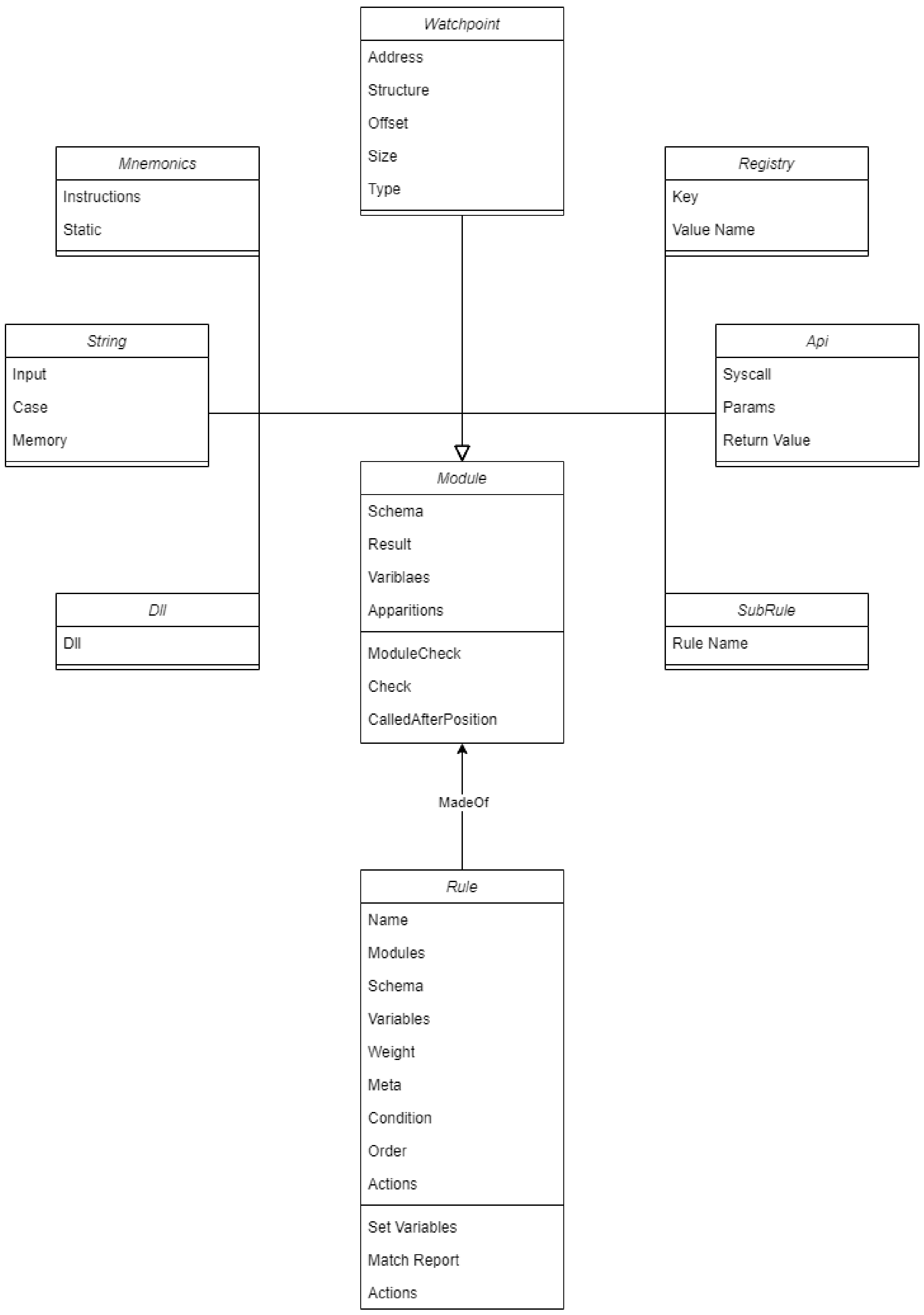


Figure 3.2: Dragonfly's rule implementation

The right part instead has, as its fulcrum, the `Rule` class. This object is made of many `Module`. The module is an abstract class that is implemented in seven different way, as figure 3.2 shows. Each module has to implement the function `check(Report report, int clock)`, where it is tasked to compute if the module matches with the report in input. All the other checks are done by the abstract class, removing complexity in each implementation, and allowing the project to scale in case that new modules are created.
A deeper description of how the rules work is presented in section 3.3.

3.2 Qiling for malware analysis

Qiling was not ready to emulate any kind of malwares when Dragonfly was designed, as already described in section 2.4. But Qiling was not alone in this issue: there was not any emulator that was able to manage a real malware and, at the same time, provide enough API and hooks to retrieve enough information allowing the creation of a tool on top of it.
For this reason, it was decided to help the Qiling community with the development of this framework, keeping in mind that the goal was to allow the emulation of complicated malwares. Since Dragonfly is designed to mainly work with Windows samples, since the majority of malwares are designed to work in a Windows environment, it has been decided to focus especially on the Windows section of Qiling. Three are the main concepts that Qiling did not have, as already described in 2.4, and each following section tries to describe how each issue was solved.

3.2.1 API

There was an enormous lack of Windows API implemented inside Qiling. This absence was justified by the small developers team: they had to focus their energies in others, more important, issues, like manage multithreading and make every different OS work, having the same foundations.

For that reason, at the beginning, the work done on Qiling was focused on implementing enough API to be able to execute malwares, and on learning how Qiling is designed and how it was possible to contribute in more meaningful ways. Moreover, since malwares are infamously known for using deprecated functions, with strong side effects, and even undocumented ones, this development took more time than anticipated.
At the time of writing, it is possible to claim that 40%-45% of kernel32.dll is implemented, and many other Dlls are partially reproduced inside Qiling.

3.2.2 Profiles

The first profile created was made for Windows in the early stage of the Qiling upgrade.
Now, profiles are one of the many features that the framework is proud to offer to its users. The concept is nothing more than a configuration file. It stores every information that the emulation may use, for example, the starting address of the stack, the address of where sample is loaded in memory or where each Dll should be loaded and so on.
But the profile can store much more than emulation variables: the configuration of each entire operating system is saved in this way. It is possible to describe which version of Windows should be emulated, its username and computer name, the machine IP address, and with which permission the sample should be executed. Every hardware specification that Qiling currently supports, like the number of processors or the drives and disks connected to the host, are set through the profile.
Every single of these variables can be easily replaced, added or removed, making possible to emulate the same sample in many, different, environments, as the situation requires.
From this configuration file, it is even possible to replace or add new Windows registry entries: the main goal was to give to the users an easy, quick and powerful way to manipulate the host environment as they please.

This upgrade solves the most important issue of the sandbox: it is now easy to change the environment and it is not needed to install and configure an entire OS.

The following snippet is an example of Windows profile that was made.

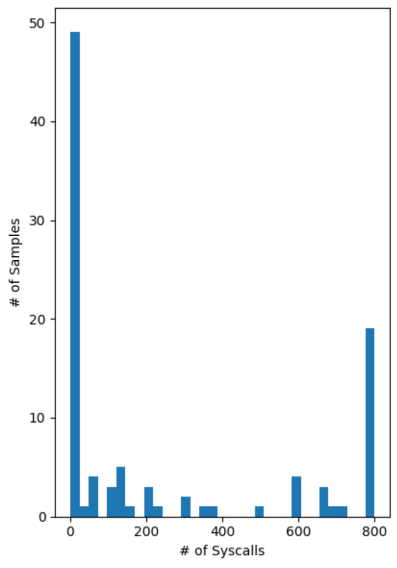
```
[OS64]
heap_address = 0x500000000
heap_size = 0x5000000
stack_address = 0x7ffffffde000
stack_size = 0x40000
image_address  = 0x400000
dll_address = 0x7ffff0000000
entry_point = 0x140000000
[OS32]
heap_address = 0x5000000
heap_size = 0x5000000
stack_address = 0xfffdd000
stack_size = 0x21000
image_address  = 0x400000
dll_address  = 0x10000000
entry_point = 0x40000
[SHELLCODER]
# ram_size 0xa00000 is 10MB
ram_size = 0xa00000
entry_point = 0x1000000
[KERNEL]
pid = 1996
parent_pid = 0
shell_pid = 10
[LOG]
# log directory output
# usage: dir = qlog
dir =
# split log file, use with multithread
split = False
[MISC]
# append string into different logs
# maily for multiple times Ql run with one file
# usage: append = test1
append =
automatize_input = False
[SYSTEM]
# Major Minor ProductType
majorVersion = 10
minorVersion = 0
productType = 1
language = 1093
VER_SERVICEPACKMAJOR = 0
computername = qilingpc
permission = root
[PROCESSES]
# process active in our env -> pid
csrss.exe = 1239
[USER]
username = Qiling
language = 1093
[PATH]
systemdrive = C:\
windir = Windows\
[REGISTRY]
registry_diff = registry_diff.json
[HARDWARE]
number_processors = 5
[VOLUME]
name = Volume1
serial_number = 3224010732
type = NTFS
sectors_per_cluster = 10
bytes_per_sector = 512
number_of_free_clusters = 12345
number_of_clusters = 65536
[NETWORK]
dns_response_ip = 10.20.30.40
```

3.2.3 Storing information

Qiling is a framework, but to learn about what happened during the emulation, API called, registries accessed, files created, it was necessary to parse its output file. This was a big issue: a framework *must* provide every information about its internal state to its users, using the object itself.
For this reason, it was developed a way to store every information that can be useful to an analyst inside the Qiling class: every string that has been used, read, or written in any part of the the memory; each systemcall called with its own parameters, address, and return value; the name of every Dll loaded and every registry that has been accessed.

3.2.4 Results

After these upgrades, it was possible to carry out a batch emulation with Qiling : 100 malwares have been emulated, unfortunately all Portable Executable files and all retrieved from a unclassified data set, demonstrating how Qiling behaviour improved from what was described in section 2.5.



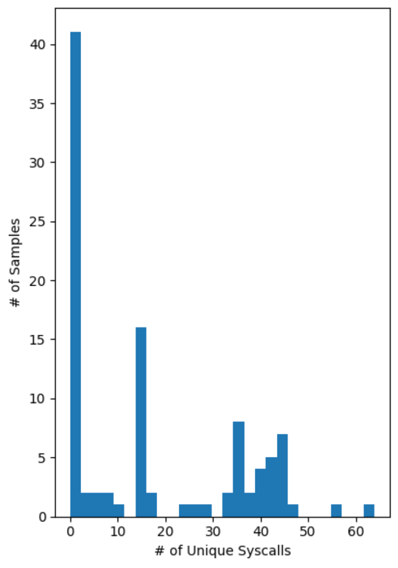


Figure 3.3: Number of Windows API called

Figure 3.4: Number of unique Windows API called

Figure 3.3 represents the total number of Windows API that a sample called during the emulation. The numbers sometimes are really high, even after having considered the low number of unique API that figure 3.4 shows. This happens because Qiling has to emulate the memory management too, done through API and normally transparent to the programmer. Every time the sample creates a variable, for Qiling and the operating system, is in reality just allocating memory. In reality is calling a kernel API that will manage the heap allocation, increasing exponentially the total numbers of calls

Figure 3.4 shows the numbers of unique Windows API that each sample calls: the values are statistically too low to had an entire execution of a malware, since a real binary normally ends its execution after an average of 100 unique systemcall, while the mean value of the samples analysed is 25. Moreover, a high number of samples does not even start their emulation, and this is caused mainly by the fact that Qiling does not support `.NET` executables yet.

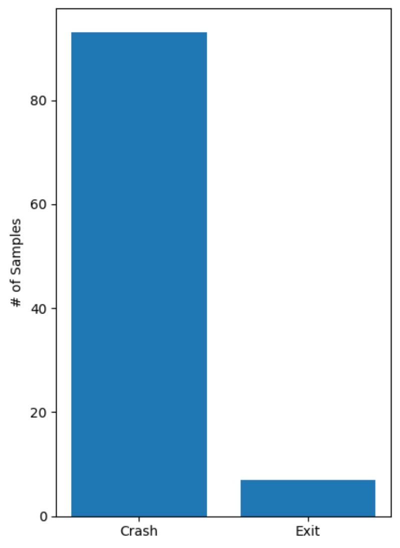


Figure 3.5: Number of samples that crashed during the emulation

For last, figure 3.5 describes how many samples crashed during the emulation. The numbers seems not very satisfying, but it is important to remember that *it is not necessary to emulate everything.* If the emulation can last enough to understand the sample behaviour, Qiling has correctly done its work, even if the emulation crashes.

.

3.3 Rules

The rules are the only way to use Dragonfly to identify if the sample analysed is a malware . This idea was used for the first time with the *YARA* project, that provides a rule-based approach to create descriptions of malware families, based on textual or binary patterns.

Why Dragonfly doesn't support YARA? Why was necessary to create another way to create rules? These are probably the questions that comes through the reader mind. The answer is, quite simple to grasp: YARA is not powerful enough.
Thanks to the emulation, people can create deeper and more precise rules: it is possible to check if a sample tries to access, in reading or writing mode, an interesting structure saved somewhere in memory, for example the `ProcessEnvironmentBlock` (PEB)[1] or the `ThreadEnvironmentBlock` (TEB)[2] inside a Windows environment. What YARA does, is just match the regex expressions that the user defines with the static sample, and checks if are present inside the binary. What Dragonfly tries to accomplish is to categorize every type of information that can be used to identify a malware. Moreover, Dragonfly can be used to call user defined functions, or *actions*, as they are called inside this project: to have an unique and consistent syntax, it was decided to create another way to write and use rules, distancing Dragonfly from YARA.

An *action* is an user defined function that is called when a rule matches. This concept is very powerful: it is possible to create an action that dumps the entire sample memory, an action that can retrieve values from the registries or memory structures, an action for anything that the user may be interested in. The user defined *action* has two parameters, the Qiling object, allowing the full control of the emulation from inside the function, and a dictionary, containing the variables and their respective values that were found during the emulation.
Inside a Dragonfly rule there are keywords that the user must use. Those that have the symbol `*` after the keywords means that are mandatory and must have an assigned value.

[1]`https://en.wikipedia.org/wiki/Process_Environment_Block` (visited on 27/05/20)

[2]`https://en.wikipedia.org/wiki/Win32_Thread_Information_Block` (visited on 27/06/20)

- **Name*** unique identifier of the rule.
- **Meta** a dictionary containing information about the author, the description of the rule, and every information that may be interesting.
- **Modules*** a list of modules, each one represented as a dictionary. More information about every module available inside Dragonfly in section 3.3.1.
- **Weight*** every rule has a weight indicating how much the signature reflects a malware behaviour.
- **Condition*** it can be `Any` or `All`, indicating if the rule should trigger when just one of the module matches, or if every module matches.
- **Order** a Boolean value that describes if the modules should match in order or not.
- **Actions** a list of functions names that will be called when the rule is matched.
- **Variables** a list of variables that are possible to use inside the modules.

Inside appendix A.0.1 it is possible to retrieve the rules that were made as examples, to better understand these concepts.

3.3.1 Modules

Every rule is made by at least one module, each one is independent from the other and the functionalities that it exports are provided via keywords. When writing a rule, the module is nothing more than a dictionary, where the value of the key **module** describes which type of module made inside Dragonfly the user wants to use.
The following sections describe the functionality of the modules present in Dragonfly, their keywords and their use.

Module - WatchPoint

A watch point is a specific address that needs to be monitored. A use case is malwares that try to retrieve the value of the parameter `DebuggingFlags` inside the `PEB` to understand if they are debugged or not.
Since the address of these types of structures can not be known before starting the emulation, the *WatchPoint* module provides the feature of directly inserting the name of the structure that should be monitored and, if necessary, an offset from its base address. Another example of structure that can be interesting to monitor is the `TEB`.
The following keys, and consequently, the following features are provided:

- **Address*** the address that, when accessed, triggers the module.
- **Structure*** the structure that should be checked. It is used to retrieve dynamically the address, since it is not possible to know its address statically.
- **Offset** an integer indicating the distance from the address of the selected structure.
- **Size** an integer indicating the range of address to monitor from the base address.
- **Type*** it can be `Write` or `Read`, indicating if the module should match when the address (or range of addresses) is accessed in write or read mode.

The key *offset* can be used only if the key *structure* has been used too, while it is not possible to have both the keys *structure* and *address*.

Module - Registry

It is possible to use this module only if the binary analysed is a Portable Executable for an obvious reason: only Windows implements the concept of

a system registry. A Macho or Unix file will not check these values because their operating system does not support it.
The *Registry* module matches when a specific registry key has been accessed and, if the user desires more precision, when the value specified has been queried.
A use case is malwares that try to use registries to identify if they are executed in a controlled environment, like a sandbox: for example the key `SOFTWARE\VMware, Inc.\VMware Tools` is present only if Vmware is installed.
Another example are malwares that will add themselves in the key `HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\RunOnce` to execute their malicious payload at every boot of the host.
The following keys, and consequently, the following features are provided:

- **Key*** the registry key that, if accessed, matches with the module.
- **Value_Name** a string describes which of the many values contained inside a key must be queried to match with the module.

Module - Api

The common way to understand the behaviour of a sample, is through its APIs calls: they can give powerful insights to the analyst that is able to correctly read them and understand what is going on.
A simple use case: if the sample is calling the Windows API `CreateToolhelp32Snapshot` with parameter `dwFlags` the value `TH32CS_SNAPPROCESS`, it means that it is trying to enumerate every process in the entire system. There are very few legit motivation for doing so.
For this reason the module *Api* was created, leaving to its users the power of specifying with high granularity and precision when the module should match, allowing even to check the parameters and the return value of the API called.

- **Syscall*** which API should be matched against.
- **Return_Value** an array containing the possible return values that make the module match.
- **Params** parameters of the API as a dictionary. Each key is the name of the parameter and each value is an array containing one or more possible values that will make the module match.

Module - Dll

Another module that was designed to work against a Portable Executable file, since only Windows use the concept of Dll.
A use case are malwares that tries to learn more about the environment, checking if a specific Dll was loaded or not: this happens because some debugger, virtual machine and sandboxes will require the installation of particular Dlls. If the sample will try to load these, it means that is trying to understand if some debug tools are present in the host machine.

- **Dll*** name of the Dll that must be loaded to match the module

Module - SubRule

This module has, as its goal, the increase of expressive power of rules. Creating a rule that, has as one of its module, the *SubRule* module, allows to concatenate rules together and rule reusability.

- **Rule*** name of rule that must have matched.

Module - String

Another technique that can be used to identify and categorize malwares is through hard-coded and unique strings that are present in the file or, during the execution, in their memory.
The module *String* has the goal to match when a particular string, or even raw bytes, are found. It is possible to search the entire memory, or the static

file, or to use the strings that are used as parameters during the calls of APIs or systemcalls.

- **Input*** the string, or an array of bytes, that must be present in the sample.
- **Case** Boolean value, if true the case is relevant, otherwise it is not.
- **Memory** Boolean value, if true the entire memory is relevant, otherwise it is not.

Module - Mnemonic

This module is a surplus, meaning that its scope is covered by the *String* module: a mnemonic in fact is the human way to represent opcodes, that are nothing more than simply bytes.
The module *Mnemonic* was made to find specific opcodes inside the sample, with a focus on the possibility that malware will use shellcodes to obfuscate their behaviour.
The mnemonics can be given in whatever instruction set the maker prefers: thanks to **keystone-engine**[3] they will be translated to the instruction set that the target was made for.

- **Instructions*** an array of mnemonic instructions that if executed, will make the module match.
- **Static** Boolean value, if true the module matches if the instructions are present, if false only if are executed.

3.3.2 Variables

How to use variables inside the rules, and why this feature even exists can not be very immediate: the usage is made of two parts, firstly it is necessary to declare the variables that are going to be used inside the rule, then it is

[3]`https://www.keystone-engine.org/` (visited on 29/06/20)

possible to use them inside every module that the rule is made of.
When a module has a variable as one of its parameter, the module matches if all other constraints are respected. Once a match is found, the value of the parameter is assigned to the variable, and the emulation continues. The rule in its entirety matches if, every module that it is made of, that shares the same variables, has at least one value in common.

3.3.3 Json-schema

The reader can understand that it will be easy to insert a wrong type in one of the many parameters that the modules support. A wrong type will possibly make Dragonfly crash or, even worse, have an unexpected behaviour. To limit this issue it has been decided to use json-schema[4], allowing to specify the type of each parameter in each module. The typing must be done by the module maker, defining a *schema* if new modules are added to Dragonfly, and two are the main benefits of this library:

- The module must behave consistently only with the types defined in its schema.
- Dragonfly will not have an unexpected behaviour in case of a wrong type.
- A Dragonfly user will have a clear exception, where the motivations of the typing error is reported.

Each schema that has been made for each module is provided in appendix B.

3.4 Usage

Dragonfly was born to be used as a Python package. This means that its users can simply import Dragonfly inside its own project, instantiate the

[4] `https://python-jsonschema.readthedocs.io/en/stable/` (visited on 29/06/20)

object, and run the analysis with the method `run()`. It is necessary to discuss how it is possible to configure the Dragonfly object and how many types of analysis exists and how they work.
Three are the possible *levels* that can be set when the `run()` function is called:

- **Level 0** will emulate the sample and a single analysis is done at the end of the emulation. It is the faster analysis and it does **not support** the use of *actions*.
- **Level 1** will create a report at every *step* and each module is matched with the new section. When an entire rule matches, its *actions* are called sequentially. A *step* is made every time a Windows API, or systemcalls, is called. It is possible to not increase the step, black listing specific functions.
- **Level 2** will make a *level 1* analysis, plus it will try to defeat anti-debug techniques that a sample can adopt. This level is still in development, and more information are provided in section 4.2.3.

The `run()` method will return a JSON object, if the parameter `json` is set to `True`. The user can even retrieve every bit of information about the rules that matched, setting the parameter `verbose` to `True`, otherwise a Boolean value, describing if Dragonfly thinks that the sample is a malware or not, is returned.
The Dragonfly object is customizable in many ways:

- Every parameter that Qiling requires during the object creation, is set at the creation of Dragonfly
- `Hooks` and `Patches`, to modify directly the sample behaviour
- `ignore_syscalls` a list of systemcalls, or Windows API that, when called, will not produce a partial report to analyse.

- `max_malice` an integer describing the threshold after which the sample is considered a malware. Each rule matched will increase the malice of the sample.

- `rules_path` the path of the rules directory.

3.4.1 Flow

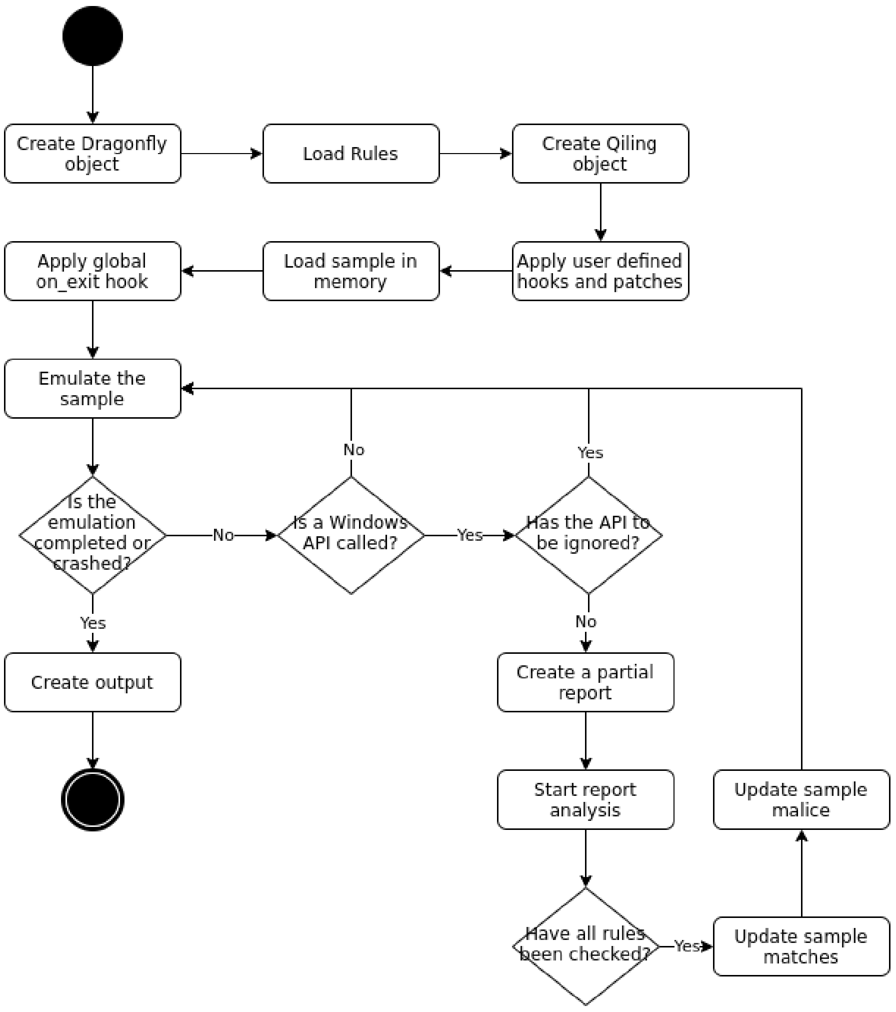


Figure 3.6: Dragonfly's analysis flow

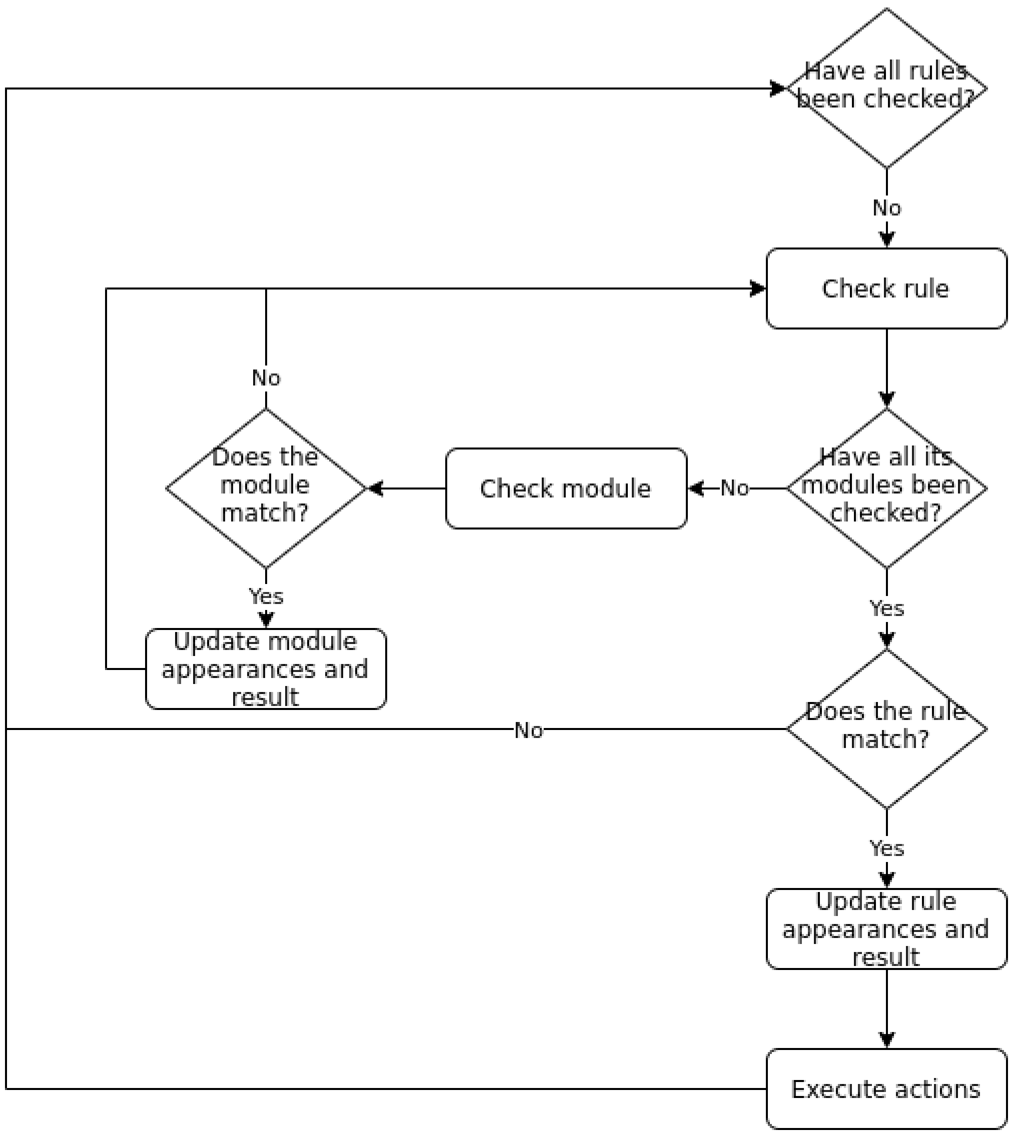

Have all rules been checked?
No
Check rule
No
Does the module match?
Check module
No
Have all its modules been checked?
Yes
Update module appearances and result
Yes
No
Does the rule match?
Yes
Update rule appearances and result
Execute actions

Figure 3.7: Dragonfly's analysis flow

Figure 3.6 and figure 3.7 show how the analysis of a Windows sample inside Dragonfly is really done, which components are made and how they communicate with each other. The figures describe an analysis of level 1: the difference with a level 0 analysis is that the lower level does not support *actions* and rules are matched at the end of the entire emulation, not at every step.

Before starting the emulation, the rules are loaded inside Dragonfly and the Qiling setup is started, applying hooks and patches that the user could have set and loading the libraries that the sample and the Dlls that it requires, into in its own memory.
After that Dragonfly will hook every Windows API *on exit*, meaning that a custom function will be called after the execution of a Windows API. The custom function will create the *report* and find rules that matches. It is possible to ignore some API using the parameter `ignore_syscalls`. The partial report, contains every bit of information that Dragonfly will use to find modules that match. When every module of a rule is matched, the rule itself matches, and their actions are executed, handing over the execution to the user.
At the end end, an output is generated, containing the analysis results.

Chapter 4

Results and Future Work

4.1 Testing

Dragonfly percentage to be able to correctly detect if the sample analysed is a malware, requires the maximum number of information at its disposal. An issue appears when its core, Qiling, is not able to emulate the sample. The problems present in the framework were explained in section 2.4.

Having said that, two samples have been deeply analysed, using before a manual approach, and then with Dragonfly: Gandcrab[1] and Al-Khaser[2]. The former is a famous ransomware, the latter is a proof of concept malware that aims to stress an anti-malware system, checking if the sandbox is fortified and stealthy enough to bypass the checks that it makes.
These two samples are interesting because, Al-Khaser can be used to translate its checks in Dragonfly rules, while Gandcrab can be used to create new rules, to understand if the rules implemented are enough to detect a malware, and which artifacts it is possible to be extracted.

[1]`https://www.vmray.com/cyber-security-blog/gandcrab-ransomware-evolution-analysis` (visited on 29/06/20)

[2]`https://github.com/LordNoteworthy/al-khaser` (visited on 29/06/20)

4.1.1 Al-Khaser

Al-Khaser is a proof of concept malware application with good intentions that aims to stress an anti-malware system. It performs many common malware tricks with the goal of seeing if the system stays under the radar. It offers many different features:

- Anti-debugging attacks
- Anti-Dumping
- Timing Attacks
- Human Interaction
- Anti-VM
- Anti-Disassembly

Only the anti-debugging attacks have been encoded as rules, of the many possible tricks that Al-Khaser implements: since these checks can be very precise, utilizing not common Windows structures and API with uncommon side effects, Qiling still has issues to emulate the entirety of Al-Khaser. Among these checks, half are simply done through Windows APIs, and thanks to the module *Api*, the encoding in rules is trivial.

The `PEB` tricks are more interesting: these can be hard to be detected with a naive sandbox, or with a manual analysis. The *WatchPoint* module comes to help in this scenario, allowing to watch the entire structure and learn which of the many information contained in the PEB, the sample tries to query.
Inside the appendix A.0.3, some of the rules, that were made thanks to the study of Al-Khaser, are described.

4.1.2 Gandcrab

Gandcrab has been studied a lot for this project and many rules have been made thanks to the knowledge obtained through this ransomware: appendix A.0.2 contains rules, that were made to recognize if a sample belongs to the Gandcrab family, and also more general ones, that can be used to detect malware behaviours.
Three are the main paths of the Gandcrab execution:

- Encrypt the file system if it was run with administrator privileges.
- Launch itself again with more privilege if it was run with user privileges.
- Stop the execution and cancel itself from the system if some system constraints are not satisfied.

In any case, the first check that Gandcrab does is using the Windows API `CreateToolhelp32Snapshot` to enumerate every process inside the host and `strcmpiW` to compare the name of the process with an hard-coded list. If a match is found, it will try to kill the process. This technique can be easily detected thanks to Dragonfly, using, as a rule, two *API* modules, checking first if `CreateToolhelp32Snapshot` is called, then `strcmpiW` adding, as possible values of the comparisons, common processes that are present inside an host. Moreover, it is necessary to set the keywords *condition* to `All` and *order* to `True`.

Gandcrab at this point will try to gain more knowledge about the system, using `VerifyVersionInfoW` and `GetTokenInformation` to understand which Windows version is the user using, and with which privilege Gandcrab has been executed. The first Windows API has, as one of its parameter, the address of a `OsVersionInfoW`[3] structure. This object contains the information about the operating system asked by the sample, and that must be compared

[3]`https://docs.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-osversioninfoexa` (visited on 30/06/20)

with what the host has installed. Thanks to a simple *action*, as shown in the third rule in the appendix, it is possible to retrieve the entire structure from the memory.

If the permissions are not sufficient to execute its real payload, meaning it does not have administrator privileges, Gandcrab will call the Windows API `ShellExecute`, with the parameter `verb` having value `runas`, and `file` to run `"C:\Windows\System32\wmic.exe"`, to which it passes the parameter `"process call create 'cmd /c start <GandcrabPath>'"`. This trick is used to execute itself again, earning the administrator privilege. Obviously the API `ShellExecute` should be monitored very closely.

Another operation that Gandcrab does is checking if the keyboard layout or the user language in the host, is Russian. If this is the case, the ransomware will deactivate itself using the API `ShellExecute`. It is possible to create a rule that matches this behaviour, as showed in the second rule of the appendix A.0.2, using the module *Registry* with key `HKEY_CURRENT_USER\Keyboard Layout\Preload`. It is possible to try different keyboard layout modifying the input profile, as section 3.2.2 described.

4.2 Future Developments

Both Dragonfly, and its core, Qiling, should keep improving their behaviour, the first to detect malwares, the latter to emulate them. Three are the main points that have been identified that need to be polished, and the following sections will try to explain and give a solution to each issue.

4.2.1 Rules

An aspect that in this work that was not considered enough, is the creation of new rules. Gandcrab and Al-Khaser have been used, yes, but it is not enough to have a product that will correctly identify every malware that

analyse.
The bare minimum is the translation of YARA rules to Dragonfly rules, adding, where it is necessary, *actions* to use Dragonfly as the best of its capabilities.
A reminder that make rules is a business, and companies sell and distribute them, making it a job that needs experience, a deep knowledge of malware behaviours and reverse engineering abilities.

4.2.2 Modules

Dragonfly architecture was designed to support the creation of new modules, without having to learn the entire project. If it is found that a new module is necessary to better describe a malware behaviour, its implementation should be easy.
The modules that Dragonfly has, are enough to describe the majority of behaviours, but they still need to be polished, or modified, as the knowledge of how malware works increase.
An example of a module that is not implemented, because Qiling can't support it yet, is the *Connectivity* module, that will identify the connections with the command center.

4.2.3 Anti anti-evasion

The main concept that Dragonfly was not able implement is the analysis level 2, or the *anti anti-evasion* level. This was though as an advanced feature that should be used to try to defeat evasion techniques implemented by malwares. When a malware detects that is in a controlled environment, it will change its behaviour, or simply will stop its execution, making its analysis quite useless. To detect when this happens, the idea was to use *Symbolic execution*[4], understanding the check that the sample implements, and allowing to bypassing it changing the correct variable in the operating

[4]`https://en.wikipedia.org/wiki/Symbolic_execution` (visited on 30/06/20)

system environment.

Conclusion

Dragonfly is a tool that merges together the concepts of dynamic analysis, emulation and sandbox, increasing the knowledge of the sample behaviour. Thanks to the emulation it is possible to have control of the emulation and the entire operating system environment, making easy to customize and modify it.

Dragonfly is able to recognize and distinguish malwares through *rules*. Rules, in turn, are made by *modules*. Differently from YARA, that simply matches the user regex with the file selected, Dragonfly modules are designed to have a limited scope and are matched against a precise type of information. The module *Registry* is used to verify the accesses of the system registry, *Api* with systemcalls and Windows API, *Strings* with the strings that are present inside or used by the sample.

The emulation core is Qiling, and upgrades were necessary to allow the emulation of malwares: the first *profile* was made in this work, allowing to easily customize the operating system environment and even variables used during the emulation itself. More Windows API have been implemented to a considerable extent, and moreover it is not necessary to parse the log file anymore to retrieve information about the emulation.

Two samples in particular have been used to build and test Dragonfly rules: Gandcrab, and Al-Khaser. Thanks to them, it was possible to see how

Dragonfly's core, Qiling, behaves against real world examples, and it was possible to create rules that synthesize the behaviour of the ransomware and the checks of Al-Khaser.

Having said that, Dragonfly still requires some polish, many more signatures must be made, and more modules should be made. Its core, Qiling, has to improve its performance against complicated samples, allowing Dragonfly to learn more about the target analysed.

Appendix A

Rules

A.0.1 Examples

```
{"name": "Example - Use of modules",
      "meta": {
       "author": "Ossigeno",
       "description": "Different modules can be used together"
     }
     "modules": [
       {"syscall":"VirtualAlloc",
         "params:{
             "lpAddress":["address]
         }
       },
       {"structure": "PEB",
         "offset": 2
       },
       {"registry": "HKEY_CURRENT_USER\\Keyboard Layout\\Preload"
           ,
         "value_name": "1"
       }
     ],
     "weight": 5}
```

```
{"name": "Example − Order can be important",
     "meta": {
      "author": "Ossigeno",
      "description": "Sometimes the order of when the modules is
           matched is important"
    }
    "modules": [
      {"syscall":"VirtualAlloc",
        "params:{
             "lpAddress":["address]
        }
      },
      {"structure": "PEB",
        "offset": 2
      },
      {"registry": "HKEY_CURRENT_USER\\Keyboard Layout\\Preload"
        ,
        "value_name": "1"
      }
    ],
    "weight": 5,
    "condition" : "All",
    "order":"True"
    }
```

```
 {"name": "Example − Variables",
     "meta": {
      "author": "Ossigeno",
      "description": "Here you can understand how to use
          variables for your own rules"
    },
    "variables: ["address"]
    "modules": [
      {"syscall":"VirtualAlloc",
        "params:{
             "lpAddress":["address]
        }
        },
        {"syscall":"memcpy",
          "params:{
              "dest":["address]
          }
        },
        {"syscall":"VirtualProtect",
          "params:{
              "lpAddress":["address]
          }
        }
      ],
      "weight": 10}
```

A.0.2 Gandcrab

Keyboard Layout and User Language

```
{"name": "Gandgrab - keyboard layout and user language",
    "meta": {
      "author": "Ossigeno",
      "description": "The sample tried to check the keyboard
          layout"
    },,
    "modules": [
      {
        "module": "Registry",
        "key": "HKEY_CURRENT_USER\\Keyboard Layout\\Preload",
        "value_name": "1"
      },
      {
        "module": "Api",
        "syscall": "GetUserDefaultUILanguage"
      },
      {
        "module": "Api",
        "syscall": "GetSystemDefaultUILanguage"
      }
    ],
    "weight": 3,
    "condition": "Any"}
```

Antivirus Process

```
{
  "name": "Gandgrab - processes",
   "meta": {
    "author": "Ossigeno",
    "description": "The malware tried to gain knowledge about
        other processes, in particular is checking if antivirus
         are present"
  },
  "modules": [
    {
      "module": "Api",
      "syscall": "CreateToolhelp32Snapshot",
      "params": {
        "dwFlags": [2]
      }
    },
    {
      "module": "Api",
      "syscall": "lstrcmpiW",
      "params": {
        "lpString1": ["sqlbrowser.exe"]
      }
    }
  ],
  "weight": 8,
  "condition": "All",
  "order": true
}
```

Dump Os Version

```
{"name": "Gandgrab - OS",
    "meta": {
      "author": "Ossigeno",
      "description": "The malware tried to use VerifyVersion to
          check which Windows version is running"
    },
    "variables": ["addr"],
    "modules": [
      {
        "module": "Api",
        "syscall": "VerifyVersionInfoW",
        "params": {
          "lpVersionInformation": ["addr"]
        }
      },
      {
        "module": "Api",
        "syscall": "GetTokenInformation",
        "params": {
          "TokenInformationClass": [25]
        }
      }
    ],
    "weight": 5,
    "condition": "Any",
    "actions": ["dumpOs"]}
```

```python
def old_dumpOs(ql, variables):
    syscall = ql.os.syscalls["VerifyVersionInfoW"][-1]
    addr = syscall["params"]["lpVersionInformation"]
    print(hex(addr))
    osVersion = OsVersionInfoExA(ql)
    osVersion.read(addr)
    maj = osVersion.major[0]
    minv = osVersion.minor[0]
    prod = osVersion.product[0]
    print(SYSTEMS_VERSION.get(str(maj) + str(minv) + str(prod)))
```

59

A.0.3 Al-khaser

Peb Checks

```
{
  "name": "Peb checks",
  "meta": {
    "author": "Ossigeno",
    "description": "Inside the PEB there are some specific
        addresses that contains information about the debugging
        environment"
  }
  "modules": [
    {
      "module": "WatchPoint",
      "structure": "PEB",
      "offset": 2,
      "type": "Read"
    },
    {
      "module": "WatchPoint",
      "structure": "PEB",
      "offset": 24,
      "type": "Read"
    }
  ],
  "weight": 6,
  "condition": "Any"
}
```

Debugging API

```
{   "name": "Debugging API",
    "meta": {
      "author": "Ossigeno",
      "description": "The sample tried to gain information the
          environment, querying one debugger API"
    },
    "modules": [
      {"syscall": "IsDebuggerPresent"},
      {"syscall": "CheckRemoteDebuggerPresent"},
      {"syscall": "WudfIsUserDebuggerPresent"},
      {"syscall": "WudfIsAnyDebuggerPresent"},
      {"syscall": "WudfIsKernelDebuggerPresent"},
      {"syscall": "DebugActiveProcess"}
    ],
    "weight": 4}
```

Appendix B

Json-Schema

Rule

```
{   "type": "object",
    "properties": {
        "name": {"type": "string"},
        "meta": {"type": "object"},
        "condition": {"type": "string", "enum": ["Any", "All"]},
        "order": {"type": "boolean"},
        "weight": {"type": "integer", "exclusiveMinimum": 0},
        "variables": {"type": "array"},
        "modules": {
            "type": "array",
            "items": {
                "type": "object",
                "properties": {"module": {"type": "string", "
                    enum": modules_names}},
                "required": ["module"],
            },
            "uniqueItems": True,
            "minItems": 1,
        },
        "actions": {
            "type": "array",
            "items": {"type": "string"},
            "uniqueItems": True,
            },
        },
        "required": ["name", "condition", "weight", "modules"],
        "additionalProperties": False,
}
```

Api

```
{"type": "object",
          "properties": {
              "module": {"type": "string"},
              "params": {"type": "object",
                          "patternProprieties": {
                              "*": {
                                  "type": "array",
                                  "items": {
                                      "type": ["integer", "string"]
                                      },
                                  "uniqueItems": True
                              }
                          }
                          },
              "return_value": {
                  "type": "array",
                  "items": {
                      "type": "integer"
                  },
                  "uniqueItems": True
              },
              "syscall": {"type": "string"}
          },
"required": ["syscall"],
"additionalProperties": False}
```

Dll

```
{"type": "object",
         "properties": {
             "module": {"type": "string"},
             "dll": {"type": "string"}
         },
"required": ["dll"],
"additionalProperties": False}
```

String

```
{"type": "object",
         "properties": {
             "module": {"type": "string"},
             "string": {"type": "string"},
             "case": {"type": "boolean"}
         },
"required": ["string"],
"additionalProperties": False}
```

WatchPoint

```
{"type": "object",
         "properties": {
             "module": {"type": "string"},
             "structure": {"type": "string",
                           "enum": ["PEB"]},
             "offset": {"type": "integer",
                        "exclusiveMinimum": 0},
             "address": {"type": "integer"}
         },
"additionalProperties": False,
"oneOf": [
     {"required": ["structure"]},
     {"required": ["address"]}]}
```

Registry

```
{"type": "object",
         "properties": {
             "module": {"type": "string"},
             "key": {"type": "string"},
             "value_name": {"type": "string"},
             "set": {"type": "string"}
         },
"required": ["key"],
"additionalProperties": False}
```

SubRule

```
{"type": "object",
         "properties": {
             "module": {"type": "string"},
             "sub_rule": {"type": "string"}
         },
"required": ["sub_rule"],
"additionalProperties": False}
```

Bibliography

[1] O. P. Samantray, S. N. Tripathy and S. K. Das, ”A study to Understand Malware Behavior through Malware Analysis,” *2019 IEEE International Conference on System, Computation, Automation and Networking (ICSCAN)*, Pondicherry, India, 2019, pp. 1-5, doi: 10.1109/ICSCAN.2019.8878680.

[2] D. A. Mundie and D. M. Mcintire, ”An Ontology for Malware Analysis,” *2013 International Conference on Availability, Reliability and Security*, Regensburg, 2013, pp. 556-558, doi: 10.1109/ARES.2013.73.

[3] P. Black and J. Opacki, ”Anti-analysis trends in banking malware,” *2016 11th International Conference on Malicious and Unwanted Software (MALWARE)*, Fajardo, 2016, pp. 1-7, doi: 10.1109/MALWARE.2016.7888738.

[4] C. A. B. d. Andrade, C. G. d. Mello and J. C. Duarte, ”Malware Automatic Analysis,” *2013 BRICS Congress on Computational Intelligence and 11th Brazilian Congress on Computational Intelligence*, Ipojuca, 2013, pp. 681-686, doi: 10.1109/BRICS-CCI-CBIC.2013.119.

[5] A. Case, M. Jalalzai, M. Firoz-Ul-Amin, R. Maggio, A. Ali-Gombe, M. Sun, G. Richard, ”HookTracer: A System for Automated and Accessible API Hooks Analysis”, *Digital Investigation 2019*, vol: 29 pp: S104-S112

[6] S. Jamalpur, Y. S. Navya, P. Raja, G. Tagore and G. R. K. Rao, ”Dynamic Malware Analysis Using Cuckoo Sandbox,” *2018 Second*

International Conference on Inventive Communication and Computational Technologies (ICICCT), Coimbatore, 2018, pp. 1056-1060, doi: 10.1109/ICICCT.2018.8473346.

[7] C. Greamo and A. Ghosh, "Sandboxing and Virtualization: Modern Tools for Combating Malware," in *IEEE Security & Privacy*, vol. 9, no. 2, pp. 79-82, March-April 2011, doi: 10.1109/MSP.2011.36.

[8] F. Al Ameiri and K. Salah, "Evaluation of popular application sandboxing," *2011 International Conference for Internet Technology and Secured Transactions*, Abu Dhabi, 2011, pp. 358-362.

[9] Lengyel, Tamas & Maresca, Steve & Payne, Bryan & Webster, George & Vogl, Sebastian & Kiayias, Aggelos, (2014), "Scalability, Fidelity and Stealth in the DRAKVUF Dynamic Malware Analysis System", 10.1145/2664243.2664252.

[10] Cohen, Michael, (2017), "Scanning memory with Yara". *Digital Investigation*, 10.1016/j.diin.2017.02.005.

[11] Lau Kai Jern & Simone Berni, Building Next-Gen Security Analysis Tools With Qiling Framework, HITBSecConf, 25 April 2020

Acknowledgements

The first person that I want to thank from the bottom of my heart, is my girlfriend and thesis reviewer, Teresa. This work would not be possible if she didn't waste her time to review and correct every section. Moreover, she put up with me for three, long, years, supporting my ideas and projects in this entire time. I can't really thank her enough.

I want to thank the entire Ulisse team, the Unibo cybersecurity group, to have welcomed me, two years ago, when I had less than zero knowledge about this field, and taught me everything that they knew. They made me passionate about this new world, and I will have a debt forever. A special thank to Dave and Melis, for having reviewed my thesis with zeal and having engaged me in interesting discussions on the choices that I made for creating Dragonfly, pointing to flaws and problems that I would have never found.

A special thank to the Certego team, especially to Matteo e Pietro: the creation of Dragonfly was done during my internship at Certego, and they are the minds behind the Dragonfly's idea in the first place. I have to thank them to have make me passionate about malware analysis, and I can't wait to join officially their team.

At last, thanks to Massimo and Cristina, my parents, to have supported me in these five years. I would not have been the person that I am now, and I probably should have appreciate more the work that you have done for me.

Thank you all.