POLITECNICO DI TORINO

Corso di Laurea in Ingegneria Informatica

Tesi di Laurea Magistrale

Packer-Complexity Analysis in PANDA

Advisors
prof. Antonio Lioy
prof. Davide Balzarotti

Samuele AICARDI

January 2018

Summary

Run-time packers are very popular among malware authors, who employ them as a way to prevent static analysis and signature matching on malicious samples. While the concept of packing is simple, modern packers often adopt very complex mechanisms to protect the code of the target application by interleaving multiple layers of decryption and execution. To capture this complexity, in 2015 Ugarte-Pedrero proposed a six-value metric to measure the complexity of runtime packers [1]. In the paper, the authors performed a number of experiments using a set of dynamic binary analysis techniques implemented on top of TEMU [2]. Unfortunately, this choice limited the applicability of the solution to only Windows 32bit applications. The first goal of this project is to take advantage of the PANDA analysis framework [3] to implement a packer analysis plugin that can dynamically analyse a running application and compute the features required for the complexity classification. This can bring this packer analysis approach to the tens of thousands of malware samples that have been recorded with PANDA by other researchers. The goal of the second part of the project is instead to investigate how the classification tool can be applied to other architecture (e.g., ARM) and other operating systems (e.g, Linux) to allow for cross-system comparisons of the packer complexity

Contents

Chapter 1

Introduction

1.1 Context

Nowadays malicious programs represent a real threat against normal users. In the fight against malware authors, researchers need to come up with new techniques to instrument Anti-Viruses (AV) the proper way to detect new malware families. Malwares can be analysed using two main approaches. The first one is called *static analysis*. It consists in looking at the target binary without executing it, typically with the help of some disassembler tools such as Ida [4], Radare [5], Hopper [6] and BinaryNinja [7], disclosing the malware's disassembled code. This gives a broad view of what the target executable *can do*, regardless of all the environmental factors (time, location, machine, operating system and so on) that can influence the particular execution of the target executable. The second approach is called *dynamic analysis*. It consists of analysing the execution of the target binary without looking at its disassembled code. It gives a narrow view showing what that specific malware execution *actually does* with that precise set of environmental factors (at that time, on that machine and operating system, in that location).

These techniques can be combined to spot malicious behaviours among a set of executables. What a normal anti-virus does to check whether a target file is malicious or not is to compare it to some known malwares. The comparison is performed by checking the target executable's signature against a list of signatures, each of them corresponding to a well-known malware family. A signature is just a composition of some of the distinguishing traits of the target executable, typically something that can be derived from its source code, e.g. a list of used functions or the unkeyed hash of part of its code

1.2 History

The first runtime packers were designed when there still was the problem of running out of storage capacity. Runtime packers were in fact able to reduce (compress, or "pack") the space an executable required to be stored on disk without modifying the executable's behaviour at runtime. Packing an executable means to take its code and compress it as if it was a normal file that needs to be put in a zipped archive. Typically an executable can be compressed because its binary code contains on average several repeated instructions that can be represented in a more compact way. Packing an executable increases thus the entropy of the file itself. A packed executable is thus a smaller-in-size version of the original executable. Nevertheless, it doesn't show any modification to its original behaviour, so that it will execute almost the same instructions at runtime: the only thing that differs at runtime is the execution of a small routine that has the task of "unpacking" the code, i.e. to bring the original code back in memory, so that it can be executed. Typically the unpacking stub is one of the first routines to be executed after the program starts.

1.3 The arms race

Runtime packers are also very popular among malware authors because they can be used to trick AVs. The main idea is to use a runtime packer to hide the malicious code. By doing so an AV would not find anything suspicious after a normal scan, given that a typical scan performs only some static analysis. In such a way runtime packing can be considered a technique to "obfuscate" a part of the code (rather than to compress it), i.e. to shuffle the corresponding bytes such that the original code cannot be understood with static analysis.

Runtime packing has become quite a common technique in the last years, such that anti-viruses have developed good countermeasures against it, at least against the most common versions of packing. As it happened for many other arms races (e.g. the arms race between virtual machines and the anti-virtual machines techniques[1]), the better anti-viruses became in de-obfuscating (unpacking) malwares, the better malware authors became in finding new and more complicated ways to obfuscate malicious code. There are several techniques to increase the complexity of the unpacking routine such that unpacking cannot be done automatically by an anti-virus. Effort-wise, since a normal anti-virus scan tries to unpack the target executable with a standard unpacker (e.g. UPX [8]) one of the least expensive approaches consists in packing the original code multiple times. In this way the anti-virus would stop at the first unpacking round, flagging the executable as "not malicious". Another approach that malwares adopt to counter-attack automatic anti-virus scans is to interleave the unpacking routine with the execution of the already unpacked code, leaving no distinct separation between the unpacking routine and the original code. Some malwares even use multiple processes to disguise the malicious code: the unpacking routine can be split among different processes or there could be one process that accesses another process' memory to write the original code.

The former techniques are just some examples of how difficult to instruct an anti-virus in a proper way (i.e. to be able to identify malicious code in those different scenarios) can be. There are several packing tools that can be found on the internet. Some of them are even tunable and it is possible to choose different settings that will be applied while packing the executable. Some malware authors prefer instead developing their own packing program, in order to be able to fully customize it according to which malware they want to pack. This is the worst scenario for researchers because there would be no chance for them to apply some automated tools/analysis upon malwares packed by a custom packer. Either it is possible to get the source code of the packer or there is no way to statically understand it by looking at the source code.

A lot of malwares are being created every day, far too many to think about analysing them manually one by one. In this scenario it is more and more important to develop new dynamic analysis techniques because it seems to be the only efficient way of approaching a packed malware, if we want to detect it.

1.4 Need for a global picture

Nowadays it is more and more important to find a way to understand what are the most common packing techniques used by malware authors in order to be able to set up a proper instrumentation for anti-viruses. The need to have a global picture on the distribution of all the different techniques that can be used to obfuscate a malware started urging: what are the most used ones? Which is the most targeted operating system? For which architecture are they compiled?

In 2015 Ugarte-Pedrero et al. proposed an answer to those questions. They tried to verify what is the percentage of malwares that are packed with off-the-shelf packers with respect to the ones that use custom (and thus not so widespread) programs. They targeted malwares running for Windows XP 32bit (Windows PE executables). Subsections 1.4.1, 1.4.2 and 1.4.3 are taken from the paper by Ugarte-Pedrero et al. [1].

[1] https://www.cyberbit.com/anti-vm-and-anti-sandbox-explained/

1.4.1 A Longitudinal study of the complexity of run-time packers

The contribution of the paper can be summed up in:

- a taxonomy for run-time packers to measure their structural complexity
- a complete framework to analyse the complexity of run-time packers
- a study of the complexity of both off-the-shelf packers and custom packed malware submitted to the Anubis on-line sandbox covering a period of 7 years[2]

They designed a new way to classify the complexity of runtime packers according to a series of factors that were chosen as the most relevant:

Unpacking Layers A layer is, intuitively, a set of memory addresses that are executed after being written by code in another layer. When the binary starts its execution, the instructions loaded from its image file belong to the layer $\mathcal{L}_0$. Later on, if an address written by any of those instructions is executed, it will be marked as part of the next layer (in this case layer $\mathcal{L}_1$).

Parallel Unpacker Many packers employ several processes in order to unpack the original code. Some packers take the form of droppers and create a file that is afterwards executed, while others create a separate process and then inject the unpacked code into it.

Transition Model A transition between two layers occurs when an instruction at layer $\mathcal{L}_\mathrm{i}$ is followed by an instruction at layer $\mathcal{L}_\mathrm{j}$ with $\mathrm{i} \neq \mathrm{j}$. In particular, forward transitions ($\mathrm{j} > \mathrm{i}$) bring the execution to a higher layer, while backward transitions ($\mathrm{j} < \mathrm{i}$) jump back to a previously unpacked layer. In the simplest case, there is only one transition from each layer to the next one. This behaviour is called *linear* transition model. In case a packer does not satisfy this definition, and therefore contains backward transitions from a layer to one of its predecessors, the transition model is called *cyclic*.

Packer Isolation This feature measures the interaction between the unpacking code and the original program. Simple packers first execute all the packer code, and once the original application has been recovered, the execution is redirected to it. For these cases, a *tail transition* exists to separate the two independent executions.

Unpacking Frames One form of interaction between the protected code and the unpacking routine can lead to a situation in which part of the code (either the unpacking routine or the original binary) is written at different times. To model this behaviour, the concept of *Frame* needs to be introduced. Intuitively, an unpacking frame is a region of memory in which we observe a sequence of a memory write followed by a memory execution. Traditional run-time packers have one unpacking frame for each layer, because the code is fully unpacked in one layer before the next layers are unprotected. We call these packers *single-frame* packers. However, more complex cases exist in which the code of one layer is reconstructed and executed one piece at a time. These cases involve multiple frames per layer and are called *multi-frame* packers in our terminology.

Code Visibility In most of the cases the original code of the application is isolated from the unpacking routines, and no write to the original code occurs after the control flow reaches this code. However, more advanced multi-frame examples exist that selectively unpack only the portion of code that is actually executed. This approach is used as a mechanism to prevent analysts and tools from easily acquiring a memory dump of the entire content of the binary.

Unpacking Granularity In case the protected code is not completely unpacked before its execution, the protection can be implemented at different granularity levels. In particular, we distinguish three possible cases:

[2] http://anubis.iseclab.org/

1. Page level, in which the code is unpacked one memory page at a time.
2. Function level, in which each function is unpacked before it gets invoked.
3. Basic Block or Instruction level in which the unpacking is performed at a much lower level of granularity.

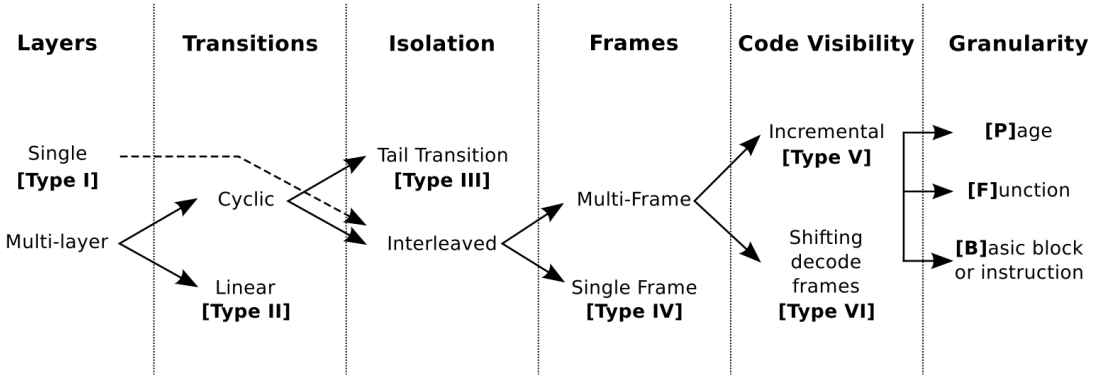


Figure 1.1: Six-value metric to rank the packer complexity.

If used together with Ugarte-Pedrero's classification, dynamic analysis can show how packers behave at runtime. The classification also divides the huge number of packer malwares into few bigger families, each one with a specific profile. The combination of all the former factors can be in fact used to distinguish six different packer families, from the most simple to the most complicated use of techniques. Figure 1.1 shows how to classify the complexity of a packer according to the features it presents at runtime. Each column represents a specific feature and it shows how it can be used to rank the packing complexity. From left to right we can distinguish the following families of complexity:

Type 1 Packers represent the simplest case, in which a single unpacking routine is executed before transferring the control to the unpacked program (which resides in the second layer).

Type 2 Packers contain multiple unpacking layers, each one executed sequentially to unpack the following routine. Once the original code has been reconstructed, the last transition transfers the control back to it.

Type 3 Packers are similar to the previous ones, with the only difference that now the unpacking routines are not executed in a straight line, but organized in a more complex topology that includes loops. An important consequence of this structure is the fact that in this case the original code may not necessarily be located in the last (deepest) layer. In these cases, the last layer often contains integrity checks, anti-debug routines, or just part of the obfuscated code of the packer. However, a tail transition still exists to separate the packer and the application code.

Type 4 Packers are either single- or multi-layer packers that have part of the packer code, but not the one responsible for unpacking, interleaved with the execution of the original program. For instance, the original application can be instrumented to trigger some packer functionality, typically to add some protection, obfuscation, or anti-debugging mechanisms. However, there still exists a precise moment in time when the entire original code is completely unpacked in memory, even though the tail jump can be harder to identify because the final execution may keep jumping back and forth between different layers.

Type 5 Packers are interleaved packers in which the unpacking code is mangled with the original program. In this case, the layer containing the original code has multiple frames, and the packer unpacks them one at a time. As a consequence, although Type-V packers have a tail jump, only one single frame of code is revealed at this point. However, if a snapshot of the

process memory is taken after the end of the program execution, all the executed code can be successfully extracted and analysed.

Type 6 Packers are the most complex in the taxonomy. This category describes packers in which only a single fragment of the original program (as little as a single instruction) is unpacked at any given moment in time.

In late 2015 a website [9] was created to host the service proposed in the paper for analysing the complexity of runtime packers [1]. It is hosted by the two universities where the authors of the paper worked, EURECOM [10] and DeustoTech [11].

1.4.2 Implementation

The main idea was to analyse the behaviour of a target executable at runtime, differentiate the unpacking stubs from the original code, understand which combination of techniques the executable uses for unpacking its code, then rank its complexity on a scale according to the six-value metric. In the paper, the authors performed a number of experiments to show the distribution of the most used packing techniques among malware authors for writing Windows 32bit executables. To compute the analysis, they developed a testing framework on top of TEMU [2] [12], a dynamic analysis platform built upon the emulator called QEMU [13]. They leveraged the binary tracing capabilities present in TEMU, extending them to fully support all the monitoring techniques that the analysis required. Since the framework needed to deal with complex runtime packers that could spawn multiple processes it was necessary to trace even the inter-process interactions. The framework is able to monitor inter-process interactions such as remote memory writes, shared memory sections, disk I/O, and memory-mapped files. It also tracks several Windows system calls that may interact with the program's memory. The analysis on a malware can be divided in the following steps:

1. *Trace the execution of the sample with TEMU.* The target executable will run inside the emulated environment, where its execution will be monitored by the code written for the analysis. The code is written as a TEMU plugin. Before starting the analysis it is necessary to add that specific plugin to the ones already running in the host part of TEMU. When the analysis is started and the program is being traced, the execution can proceed as normally. The target program will be dynamically monitored and the analysis will be performed on the go, at runtime (this will slow down the original execution of the target executable).

2. *Perform the analysis of the output of the first part.* When the live execution is finished the dynamic analysis has produced some output files that will be used in this phase of the analysis to produce the complexity graph. This graph will show a "general idea of how the unpacking routines have recovered the protected code" [17].

3. *Compute the complexity and the general statistics.* This third phase will take as inputs the outputs of the second phase and it will produce the ranking in the six-value metric according to the packer complexity. It will also retrieve some statistics about the packer family, the API calls done during the execution and other related information.

1.4.3 Results

In the paper the authors presented a packer taxonomy capable of measuring the structural complexity of run-time packers. They also developed an analysis framework that was evaluated on two different datasets: off-the-shelf packers and custom packed binaries. The lack of reference data-sets and the lack of tools for the analysis of the behaviour of packers suggests that the (un)packing problem has been put prematurely aside by the research community. Among the number of types of packers that have been found during the experiments, the authors described three packers that belong to three different categories: UPolyX 0.4 (Type 3), ACProtect 1.09 (Type 4), Armadillo 8.0 (type 6). Except for that, the authors also applied the analysis to two sets of malwares, the

first ones packed using off-the-shelf packers, the second ones packed with custom packers. The results of the experiments show that, while many runtime packers present simple structures, there is a significant number of samples that present more complex topologies. The first conclusion is that every unpacked code is not necessarily part of the original code. This goes against the way automatic unpackers work, showing that there is the need of further studying the implementation of automatic unpackers. The situation gets complicated even more if we consider that around 10% of the off-the-shelf packers and 14% of the custom packed malware did not have the original code in the last layer. The second conclusion is that the average packer belongs to the Type 3 category, i.e. cyclic unpacking routine working on multiple layers, with a visible *tail jump* and the original code in the deepest layer. Even though Type 5 and Type 6 packers existed in the off-the-shelf dataset, they are not very common in the wild. These packers require a significantly more complex development and they also impose a run-time overhead that may not be desired by malware writers.

This study can help security researchers to understand the complexity and structure of run-time protectors, to reverse engineer such malwares and to develop effective heuristics to generically unpack binaries. Unfortunately, the way the framework was developed limited the applicability of the six-value metric only to Windows XP 32bit executables. This is a pity because even though the framework is useful to classify a group of malwares, it is not able to give a proper view of the global situation regarding runtime packers usage. A considerable number of malware families simply cannot be classified by the 6-value scale of unpacking complexity because they have been compiled either for another architecture or for another operating system (or operating system version).

1.5 Goal

With this in mind, the overall goal of this thesis is to extend the range of such malware analysis. For the most part of this project I used a dynamic analysis platform called PANDA, which is able to 'record' the execution of a target application and to 'replay' it as many times the user wants. This offers a lot of freedom because one can replay an execution even without having access to the original executable. The first goal is to take advantage of such platform to implement a packer analysis plugin that can dynamically analyse a running executable and compute the features required for the complexity classification. This can bring the packer analysis approach to the tens of thousands of malware samples that have been recorded with PANDA by other researchers.

The second goal of the project is instead to investigate how the classification tool can be applied to other architectures (e.g., ARM) and other operating systems (e.g, Linux) to allow for cross-system comparisons of the packer complexity. In this way the six-value metric classification would cover a larger set of malware families, aiming to give an overall picture about the usage distribution of the different runtime packing techniques.

Chapter 2

Used Tools

2.1 Introduction

In 2015 Ugarte-Pedrero proposed a six-value metric to measure the complexity of runtime packers. In the paper, the authors performed a number of experiments using a set of dynamic binary analysis techniques implemented on top of TEMU [2].

TEMU is a whole-system emulator (built upon QEMU [13]) which is able to perform dynamic binary analysis. Among all the functionalities that it provides, Ugarte-Pedrero's analysis relies on the following:

- OS awareness. Information about OS-level abstractions, like processes and files, is important for many kinds of analysis. Using knowledge of the guest operating system (Windows XP or Linux), TEMU can determine what process and module is currently executing, what API calls have been invoked (with their arguments), and what disk locations belong to which files.
- In-depth behavioural analysis. TEMU is able to understand how an analysed binary interacts with the environment, such as what API calls are invoked, what files are read/written and what outstanding memory locations are accessed.

In 2016 the analysis has been converted to be run on DECAF [14]. It is the successor of the binary analysis techniques developed for TEMU and it is built upon it.

The first part of the current thesis was about porting the whole analysis from DECAF to PANDA [3] in order to exploit PANDA's capability of record and replay a malware execution.

2.2 QEMU

The three platforms included in the description (TEMU, DECAF and PANDA) are all based upon QEMU. QEMU is a processor emulator which can be run in two ways:

- user mode: it emulates the target architecture in order to execute a single application.
- full-system mode: it emulates a whole system running over the target architecture. This was the mode used for my experiments.

2.3 TEMU and DECAF

TEMU is the system which Ugarte-Pedrero's framework was initially based on.

It has the same basic behaviour of QEMU in full-system mode. In addition to that it offers specific functionalities to perform dynamic binary analysis.

DECAF (Dynamic Executable Code Analysis Framework) integrates TEMU's capabilities by extending them with:

- Precise Tainting
- Instruction Tracing
- Event-driven API (API tracing, key-logger detection)

As TEMU, it is a "platform-neutral full-system dynamic binary analysis platform" [14]

2.4 PANDA

PANDA (open-source Platform for Architecture-Neutral Dynamic Analysis) was the platform mainly used for the thesis. Based on QEMU, it is a full-system emulator that is able to perform Record & Replay of a target executable.

Since it is a whole-system emulator its analyses have access to all data and all code executing in the guest. It offers the ability to record and replay executions, enabling iterative, deep, whole-system analyses. Moreover, the replay log files are compact and shareable, allowing for repeatable experiments.

PANDA leverages QEMU's support of different CPU architectures to make analyses of those diverse instruction sets possible within the LLVM IR (Low-Level Virtual Machine Intermediate Representation). In this way, PANDA can have a single dynamic taint analysis, for example, that precisely supports many CPUs. PANDA analyses are written in a simple plugin architecture which includes a mechanism to share functionalities between plugins. They can interact with each other to increase code re-use and to simplify complex analysis development.

Ugarte-Pedrero's framework has been ported as a PANDA plugin having in mind the plugin-plugin interaction power. The most effective work-flow in PANDA is to record a piece of execution of interest (i.e. the malware's execution) and then analyse that recording over and over again.

2.4.1 Record & Replay

PANDA supports whole system deterministic record and replay in whole-system mode on the i386, x86_64, and arm targets. Deterministic record and replay is a technique for capturing the non-deterministic inputs to a system. This includes all the things that would cause a system to behave differently if it were re-started from the same point with the same inputs, from things like network packets to hard drive reads, mouse and keyboard input, etc.

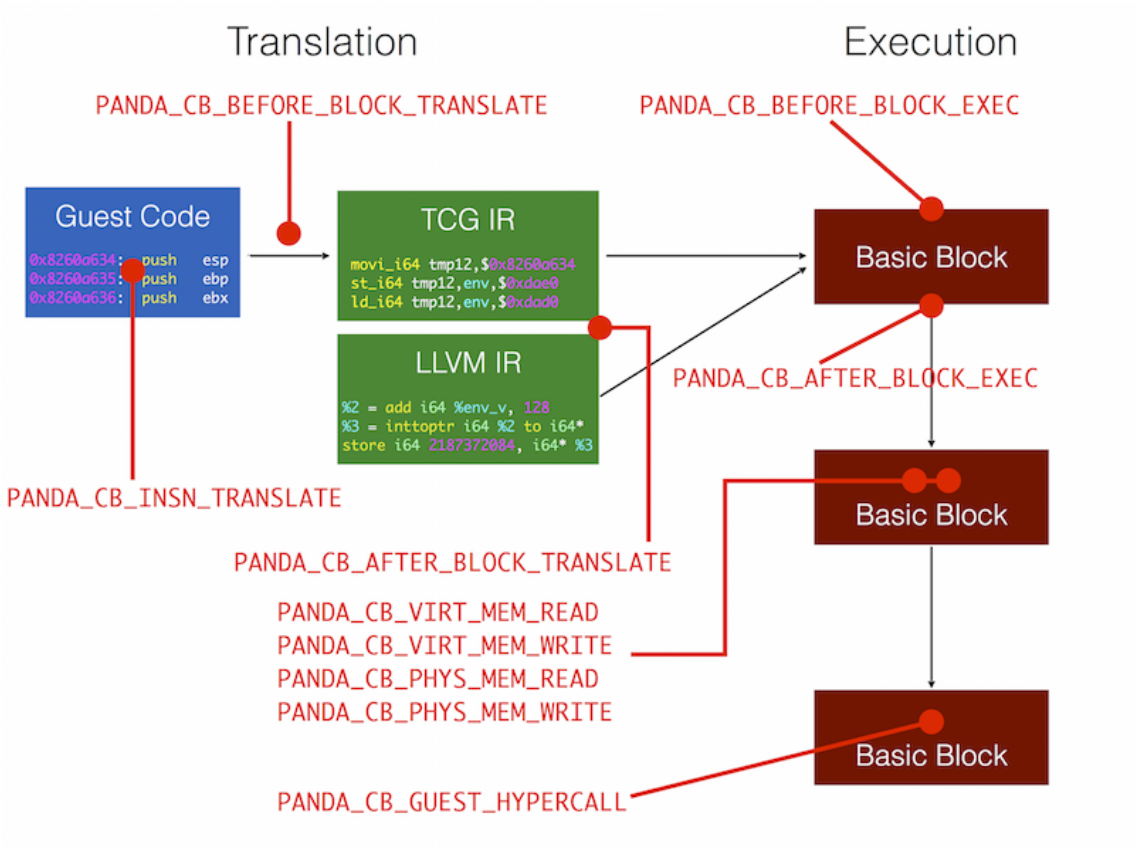
The implementation of record and replay focuses on reproducing code execution. That is, the non-deterministic inputs that PANDA records are changes made to the CPU state and memory such as DMA, interrupts, and so on. Because of some implementation simplifications PANDA does not record the inputs to devices. In order to get an idea of what is recorded one could imagine

drawing a line around the CPU and RAM: things going from the outside world to the CPU and RAM, crossing this line, must be recorded. Thanks to this feature it is possible to perform even the kind of analyses that would consume too much time with a common emulator. By creating a recording, which has fairly modest overhead, and performing analyses on the replayed execution, one can do analyses that simply aren't possible to do live.

Recording will create two files: <replay_name>-rr-snp, the VM snapshot at the beginning of the recording, and <replay_name>-rr-nondet.log, the log of all non-deterministic inputs. They are both necessary to replay the segment of execution.

Virtual machine instrumentation can be done at different granularity levels. In order to understand them it is necessary to describe how QEMU emulates guest code.

Let's consider a basic block of guest code that QEMU wants to emulate. It disassembles that code into guest instructions, one by one, simultaneously assembling a parallel basic block of instructions in an intermediate representation (IR). From this IR, QEMU generates a corresponding basic block of binary code that is directly executable on the host. This basic block of code is actually executed, on the host, in order to emulate guest behaviour. QEMU toggles between translating guest code and executing the translated binary versions. As a critical optimization, QEMU maintains a cache of already translated basic blocks.



Plugins allow you to register callback functions that will be executed at various points as QEMU executes. While it is possible to register and run callbacks during record, it is more usual for plugins to be used during replays. Here are some examples of the instrumentation granularity that PANDA offers:

- `PANDA_CB_BEFORE_BLOCK_TRANSLATE`, before the initial translation of guest code. The length of the block is not known at this point.
- `PANDA_CB_AFTER_BLOCK_TRANSLATE`, after the translation of guest code. In this case the length of the block is known.

- `PANDA_CB_BEFORE_BLOCK_EXEC`, after the block of guest code has been translated into code that can run on the host and immediately before QEMU runs it.
- `PANDA_CB_AFTER_BLOCK_EXEC`, immediately after the block of translated guest code has actually been run on the host.
- `PANDA_CB_BEFORE_BLOCK_EXEC_INVALIDATE_OPT`, right after the guest code has been translated into code that can run on the host, but before it runs.
- `PANDA_CB_INSN_TRANSLATE`, just before an instruction is translated, and allows inspection of the instruction to control how translation inserts other plugin callbacks such as the `INSN_EXEC` one.
- `PANDA_CB_INSN_EXEC` is just before host code emulating a guest instruction executes, but only exists if `INSN_TRANSLATE` callback returned true.

2.4.2 How-To

The procedure to record the execution of a binary can be summed up in the following steps:

1. Get a working `qcow2` image. This can be done by either:
   - downloading a snapshot of the target Operating System in the `qcow2` format
   - creating a `qcow2` image in this way:
     (a) `qemu-img create -f qcow2 <qcow2_image> <size_in_GB>G`
     [add the `-o compat=0.10` option for the real qcow3]
     where `<qcow2_image>` is the name we want to give to the new image and `<size_in_GB>` specifies how big we want to make the image.
     (b) `<PANDA_folder>/target-i386/qemu-system-i386 -boot d \ -cdrom <OS_iso> -hda <qcow2_image>`
     where `<PANDA_folder>` is the path where PANDA is installed and `<OS_iso>` is a `.iso` image of the operating system we want to install on the `qcow2` image.
2. Launch PANDA's version of QEMU:
   `<PANDA_folder>/qemu-system-i386 -hda <qcow2_image> -monitor stdio`
   The option `-monitor stdio` lets us interact with QEMU through its monitor. In this way we can write commands to interact with QEMU.
3. When PANDA has loaded the `qcow2` image, prepare the executable we want to record and start recording. In the QEMU monitor write:
   `begin_record <replay_name>`
   This will create the two files needed for the replay phase, `<replay_name>-rr-snp` and `<replay_name>-rr-nondet.log`. They will be filled as long as the executable is being recorded.
4. Once the executable has finished running, stop the recording. In the QEMU monitor write:
   `end_record`
   This will save the two files created in the previous step.

Now the the target executable can be replayed over and over as many times as we want. Thanks to the `<replay_name>-rr-nondet.log` file the execution will always be the same. In order to execute a replay it is sufficient to write:

`<PANDA_folder>/qemu-system-i386 -replay <replay_name>`

The next reasonable step is to execute some plugins along the basic replay. This will introduce the actual power of performing effective dynamic analysis:

```
<PANDA_folder>/qemu-system-i386 -replay <replay_name> -panda <plugin_name>
```

It is possible to load multiple plugins at the same time, always in the form `-panda <plugin_name>`. By doing so, several plugins can be combined together to perform even more powerful dynamic analysis. This is the basic idea behind the utilization of the plugin written for the topic of this thesis. It bases in fact part of the Operating System Introspection upon two plugins, namely `syscalls2` and `osi`. The former is in charge of placing hooks on specific system calls that the user wants to keep an eye on, the latter implements the Operating System Introspection, i.e. it monitors everything that is related to processes: process creation, process termination, retrieve the list of processes, get the current one, and so on and so forth. They will be explained more in details in the following chapters.

Chapter 3

The Framework

In this chapter I will explain how the starting framework is structured and how it is possible to get a complete report about the runtime packer complexity.

3.1 Deep Packer Inspection

The first implementation of the framework was developed to be a plugin for DECAF. Because of that Deep Packer Inspection (the name of the framework) contains some configuration files that truly depend on DECAF and only work for this specific implementation. The main idea is that, since DECAF is an emulator based on QEMU, it can emulate a whole operating system, Windows XP in this case. As QEMU, DECAF offers to the user an interface to give commands to the platform itself: the qemu monitor. Thanks to the qemu monitor it is possible to control the guest OS, see what is going on during a live execution and even modifying some structures. DECAF also offers the possibility to load what are called *plugins* on the live guest system: with those plugins the user can extend the control that she has over the running OS. To do so type in the qemu monitor:

```
load_plugin <plugin_name>
```

Deep Packer Inspector (DPI) implements thus a way to instruct the analysis through some configuration files, which specify e.g. the name of the sample that the user wants to analyse. The main information required to run such plugin are the following:

- The name of the process and its Entry Point
- The maximum time to spend waiting for the process to start. If the process has not started yet after this time, just close everything and exit
- The minimum analysis duration. The plugin will keep analysing the sample at least this number of seconds
- The maximum analysis duration. The plugin will keep analysing the sample at most this number of seconds
- The exception recovering time threshold. When an exception is produced, the plugin starts counting seconds. If the execution does not return to user code before the set threshold the analysis stops
- The granularity to monitor the process' activity. The plugin checks every N seconds if the process being monitored have a significant activity in the system (it checks the consumed CPU parameter from internal Windows kernel structures). If the process is idle and has not consumed any CPU in a period of 2 minutes, the analysis will be stopped

- The maximum size of the memory dumps to extract at the end of the execution

When the DPI plugin is loaded in DECAF it will :

1. Wait for a given process to start by monitoring the Program Counter (PC) of the guest OS
2. Start tracing the target process as soon as the PC reaches its entry point
3. Monitor several conditions in order to stop tracing and finalize everything:
   - wait up to N seconds for the sample to start
   - monitor it for a minimum number of seconds to a maximum number of seconds
   - between these 2 points, it will stop the analysis if all the analysed processes are killed, if none of the analysed process show a significant activity or if an exception is produced and the process does not recover.

The DPI plugin is composed of around 8000 lines of code, divided into several C/C++ files. The authors kept one file or a group of files (that interact with each other) for each of the main parts (and tasks) that compose the framework:

- Definitions of the main hooks, for windows functions that are somehow related to remote process memory injection. These functions include *WriteProcessMemory*, but also indirect ways of writing from process one process to another, e.g. the first process writes a certain file to disk, then the second process reads that same file from disk and executes the buffer
- Hooks for a number of API calls and then trigger all the necessary events. For instance, whenever there is a *WriteProcessMemory*, the framework delivers to the *execution model* (one of the main structures used for the analysis, in *ExecutionModel.cpp*) a memory write event. These events are generally logged to one of the log files (namely *events.log*) through the *EventLogWriter.cpp* proxy
- Virtual Machine Introspection (VMI) for Windows XP Service Pack 3. These functions are used to inspect the kernel data structures to retrieve information about the running system, such as getting the list of processes or extracting properties for each process (e.g. open handles)
- Definition of the *execution model* as a representation of the memory layers and regions. It is populated during the first execution of the sample with the different memory write and memory execution events that occur in the emulated guest OS.

The composition of those files can be seen as a black box that receives notifications about memory write and memory execution events, showing the address and size of each affected memory, along with source and destination process, as well as notifications about certain file write and file read operations. It will then take care of populating the *execution model*. Finally it will generate the output files that will be used in the next phase of the analysis.

3.2 Produce the Graph

At the end of the analysis described in Section 3.1 the DPI plugin will have produced some output files:

- *execution_model.log*. It lists the layers and the regions computed during the dynamic analysis with DECAF
- *trace.log*. It contains the extracted trace of the execution
- *events.log*. It contains the operating system specific events, such as which process created a process or which process wrote to another process' memory using which technique

- *auto_start.log*. It contains the details about the auto_start engine, and the analysis hit thresholds. It should answer questions like "Why did the analysis stop?"
- *functions.log*. For each monitored process it contains a list of addresses for each imported library and their names

At this point the dynamic analysis can be considered finished, while it is time for the post-processing analysis. This phase is in fact in charge of taking some of the output files of the dynamic analysis, elaborate them to understand the behaviour of the runtime packer and produce the output files that will be taken into account to produce the behavioural graph, showing how the packer unpacked the original code at runtime (i.e. how many layers of unpacking code did it use, how many processes were spawned during the execution and so on).

To run the second phase of the analysis:

```
./makegraph trace.log execution_model.log functions.log output_dir/
```

where the first 3 parameters are the result of running the first phase of the analysis and `output_dir/` is the directory where the tool will place:

- *general_statistics.log*. It contains several statistics about the graph, it is parsed by the following phase
- *graph.dot*. It contains the description of how to build the graph. It is in the *graphviz* format. To generate the actual graph:

  ```
  dot -Tpng -ograph.png graph.dot
  ```

- *regions_apis.log*. It is a list of API calls called by each region, in order to feed the database in the next phase
- *transitions.out*. It contains a serialized list of transitions. They will be parsed in the following phase too

3.3 Complexity Ranking

The final step of the analysis will produce the complexity ranking (on the six-metric scale) of the runtime packer, according to the results that can be gathered from the previous phases. It will also produce some information about the general behaviour of the packer based on what it did during the execution. The only output of this phase is a *txt* file similar to what is described in Appendix A.

Chapter 4

Implementation

4.1 Goal Description

Ugarte-Pedrero's work from 2015 mainly delivered two outputs. The first one is the six-value metric to measure the complexity of runtime packers, the second one is a plugin for DECAF that is able to bring out such analysis on Portable Executable (PE) files for Windows XP. Even though the analysis can be done on hundreds of malwares, there are a lot of other samples that cannot be analysed because either they are for other architectures (designed for x86_64 or ARM machines) or they have been compiled for other operating systems.

The main goal of the thesis is thus to extend the analysis to as many malware samples as possible. This implies adding the support for other architectures and operating systems. Once Ugarte-Pedrero's plugin had been ported on the PANDA framework it was divided into subsections, one subsection for each part of the analysis: process/thread monitoring, OS-specific introspection and memory read/write. The idea was to keep the part of the analysis that works for multiple systems 'as-is' and to replace only the pieces of the plugin that are architecture-dependent. In this way the plugin can be seen as a combination of pluggable components that can be combined together to analyse different classes of malwares.

At the end of the development of the plugin a long test session has been carried out to test several malware samples, with the final intent of generating an overview of statistics to see what is the packer usage scenario for a specific set of operating systems and architectures.

4.2 Why PANDA

Given the main goal of the thesis one can argue that it could have been carried out even by keeping the plugin implemented for DECAF, since DECAF supports multiple architectures and operating systems. The reason why it has been decided to go for PANDA is because of its capability to replay already recorded executions. Several malware families rely on the communication with a Command and Control (C&C) server to receive instructions or to download some infected payloads. It typically means that once the C&C server is not online any more the malware will just stay silent without doing anything suspicious. Situations like this one happen many times and they represent a huge problem for dynamic malware analysis: if someone wants to perform dynamic analysis on a piece of malware three years after it has been discovered, it may be simply not feasible to replicate the external environment inputs and outputs.

By using PANDA it is not necessary to have the original malware executable nor to be in the same external conditions of the original execution. It will always be the same no matter how different the environment will be, PANDA logs will keep the relevant not-deterministic inputs as they were at the moment of the recording. Citing the description on the PANDA man page, "To

get an idea of what is recorded, imagine drawing a line around the CPU and RAM; things going from the outside world to the CPU and RAM, crossing this line, must be recorded." [1]

Another reason for using PANDA is that it has been used by other researchers to record executions of several malwares. This gives us the chance to test the framework with a lot of already recorded samples. Dolan-Gavitt alone is hosting a database with more than 90000 records[2]. It is a database that receives malware samples on a daily basis. They are executed on PANDA and then their recordings are saved together with some statistics coming from `virustotal.com`.

4.2.1 Advantages and Limitations

In these days the panorama of dynamic analysis and reverse engineering frameworks include several academic projects, such as DECAF, PANDA, AVATAR [15] and PyReBox [16]. As one of them, PANDA has some advantages and limitations.

Advantages: Any kind of dynamic analysis that panda offers can be done during the replay phase, without interacting with the live system during the recording phase. This means that the execution of the target executable is not affected by any analysis (that will be done in the future). As a consequence, there is no need of tampering the original process execution, for example by attaching a debugger to the target process, to have additional information on the execution. This is a great advantage because a lot of malwares try to see if they are being traced at runtime. With PANDA, malwares will never see their execution tampered by some external agents.

Limitations: Since it is based on QEMU, which is an emulator, PANDA suffers all those problems that are typical of emulators. First of all, when there is the need of dealing with malwares there is always the (quite likely) chance that the target malware implements some anti-emulator techniques: the most common thing for a malware would be to look for emulator artefacts inside the emulated environment, because it may happen that the emulated environment presents several objects (such as specific system folders or files) whose name contains an emulator-specific word (e.g. "qemu-"), even though this scenario is more common in a virtualised environment. Another approach for a malware to detect n external analysis is to trigger some assembly instructions that produce different results in an emulated environment with respect to a bare-metal machine: an instruction useful to test this behaviour will be one that makes the physical machine crash on a bare-metal execution, while the crash will be handled by the emulator in the emulated environment. There is thus the chance that the target malware applies one of those emulator-detection mechanisms, triggering for example some particular instructions that would crush on a real system and not on an emulated one. It could also look for some "panda-" strings in the guest's filesystem.

4.3 The Starting Framework

The starting framework (by the authors of the paper) was firstly designed and written for TEMU, in 2015. When TEMU has been upgraded and the DECAF emulator was proposed out of it, the framework for analysing the complexity of runtime packers was ported to DECAF, to support and increase some of its functionalities. The decision to port the framework from one system to another was decided by the authors of the paper. As already said, both TEMU and DECAF are whole-system emulators based on QEMU. The packer complexity classification is currently available as a web service, where users can submit their executables. The service will analyse them and it will output the complexity analysis, ranking the sample in the six-value metric. The user can also specify if he wants to make the analysis results public or private. The framework originally supported only Windows XP Portable Executable (PE) files, which is the format adopted by all the recent Windows operating systems for executable programs and applications. PE executables can be compiled both for x86 and x86_64 machines but the support of this framework was specifically

[1] `https://github.com/panda-re/panda/blob/master/panda/docs/manual.md#background`

[2] `http://panda.gtisc.gatech.edu/malrec/`

only for 32bit programs. The choice did not represent a big issue because typically Windows malwares are designed to be run on as many machines as possible, thus most of them are 32bit executables. However, the limited support in terms of operating systems (and versions of the same OS) represents instead a considerable limitation in terms of applicability of the complexity classification (e.g. the OS support does not include the Unix family of operating system).

The internship can be divided into several temporal parts, each of which corresponds to a period in time where I worked for a specific goal of the thesis. There were two big phases: in the first one the objective was to have a working version the framework written as a plugin for PANDA. The first step was to port the whole plugin to PANDA, while the second step was to test if the new plugin was giving the same results as the old one. To do so I tested the plugin against some samples which I already had the results of (because I previously submitted them to packerinspector.com). By completing this first phase I had the same framework implemented as a PANDA plugin. In this way it was possible to extend the analysis to the thousands of samples that other researchers recorded using PANDA. I did this to achieve the first goal of the internship, that is to exploit the possibility of re-using samples that were recorded in the past, even for different purposes. In the second phase the main objective was to extend the plugin to support other architectures and other operating systems. We decided to extend the support of the analysis to Linux 32bit elf executables (running on x86 machines) and to the ARM architecture (again testing it with Linux elf executables). This led to the achievement of the second goal of the thesis, which was to extend the original support of the complexity analysis to other types of executable files.

4.4 Porting to PANDA

As previously stated, PANDA is a "Platform for Architecture-Neutral Dynamic Analysis", a full-system emulator based on QEMU. The main peculiarity that PANDA has with respect to other dynamic analysis platforms is the capability of performing record and replay of a live system. With PANDA one can record the execution of an executable and then replay it as many times as necessary to perform whichever kind of analysis, without affecting the 'real' execution in any way. PANDA also offers the possibility to perform some kinds of analyses that are not possible to be done with a live system, either because they require too much computation power or because they would drop down the performances for the runtime execution of the target sample. It is especially useful when dealing with malwares that implement logic bombs and anti-debugging techniques to detect when they are being traced and to avoid the analyst to spot the malicious behaviour. An example of such logic bombs could be a defence mechanism as simple as waiting some minutes (at runtime) before starting the actual malicious activity: in this way if an automatic dynamic analysis (which typically lasts few minutes) is performed on the malware it would be impossible to spot any malicious activity. Or the malware could also detect that it is under analysis by detecting any debugger attached to one of its processes (again, at runtime) by trying to call the *sys_ptrace* system call on itself: if the call does not succeed it is a symptom that something else is already tracing the same process. By replaying a recording the malware is not able to detect that it is under analysis: the analysis will be done on the execution itself, without attaching any debugger of just waiting the right amount of time. However, it can still detect that it is being run under an emulated system, but this is inevitable because PANDA is an emulator.

4.4.1 How PANDA works

PANDA implements several callbacks that can be registered to hook on a specific event, such as the creation/termination of a process, a memory write, the translation/execution of a basic block or a context switch. It offers several plugins to perform specific analyses, so there is no need of writing any code if the user needs to perform some already implemented analyses, such as tainting, tracing the system calls, monitoring the context switches, logging the open/closed file descriptors, logging the processes' start/termination time and duration. These are just some examples of what PANDA can offer in terms of analyses. If for some reasons the user needs to perform a different analysis, it is possible to create a new PANDA plugin that can both do something completely new in terms

of analysis and combine the already existing plugins, to perform even more powerful/complex analyses.

4.4.2 Steps

The first thing I needed to do in order to port the framework to PANDA was to study how the plugin was implemented in the original version, to understand the role of each part of the code and to see how different sections of code were combined together to perform the analysis of the complexity. Figure 4.1 shows the dependency graph of the initial implementation of the framework that I computed in order to have an idea of the interaction between the several files that compose the framework. Each node represents a file that was present in the original implementation as a DECAF plugin. All of them are C/C++ files, some of them are header files. Each arrow shows the dependency that occurs between two different files. A dependency from a generic file A to a file B is defined if there is a line of code file B that "includes" (in the C sense of the term) file A. Most of the files are distributed by following a hierarchical structure of dependencies. The names of the files are not visible because I wanted to give only a qualitative view of how things were organised at the beginning. Since the original plugin was written for DECAF there were

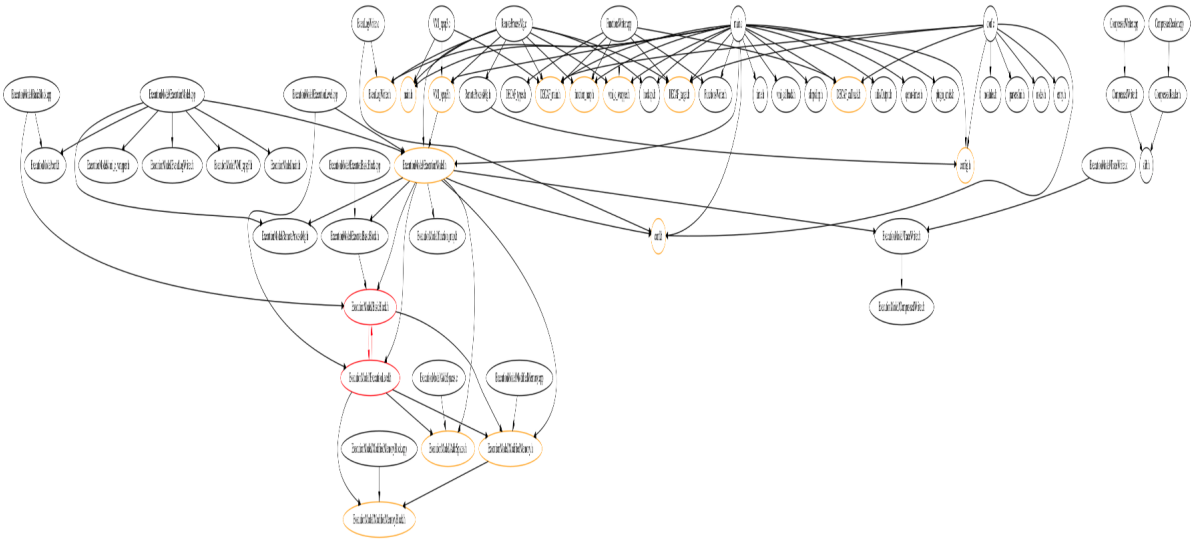


Figure 4.1: Dependency graph of the files composing the initial framework in DECAF

a lot of dependencies in the code that made the porting phase difficult at the beginning, in the sense that I needed to study how the dynamic analysis of DECAF interacted with the QEMU backbone, in order to be able to distinguish between the dependencies coming from DECAF and those coming from QEMU, because while I would have needed to modify the first ones, at the same time I could have re-used the second ones (because PANDA is built on top of QEMU too). Once I finished this preliminary part of understanding the general structure of the framework I started the porting phase by dividing the plugin into several components, each one with a specific purpose. I differentiated three main components of the plugin: one that deals with process tracing, a second one for all the OS-dependent parts, such as API call and system calls, a third one for the memory read/write tracing. I did so because I had in mind that this separation would have helped in the future to implement the plugin for other operating systems and architectures.

Once I finished this part I needed to test it in order to see if everything went the way it was supposed to go, so that there were no errors in the porting process. To test it I did the following: first of all I got some packed samples, I submitted them to the online service packerinspector.com and I received the results of those packed samples. Then I performed the complexity analysis on the same samples, this time using my version of the framework (i.e. the PANDA plugin). Eventually I compared the results of the two analyses to see if there were some discrepancies.

4.4.3 Plugin Components

To perform the complexity analysis I needed to combine my plugin with some of the plugins that PANDA offers, because I wanted to use some of the already implemented analyses. For the process tracing part the plugin that best suited for this role was the *osi* plugin. The name *osi* stands for "Operating System Introspection" and it is in charge of providing a set of callbacks that are useful to deal with process creation/termination (*on_new_process*, *on_finished_process*), retrieve a list of processes (*on_get_processes*) or retrieving just the running process (*on_get_current_process*). Figure 4.2 from the *osi* man page[3] shows how to use it. Because of its implementation, the *osi* plugin is just a "glue" layer that does not actually implement those callbacks, but it makes it easier to use the callbacks independently of the final operating system. This set of callbacks has

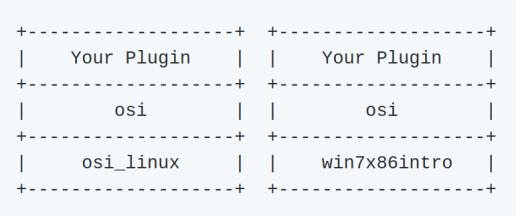


Figure 4.2: The *osi* plugin that acts as a "glue layer"

multiple implementations, one for each operating system that PANDA supports: the picture shows one implementation for Windows 7 (*win7x86intro*) and one implementation for Linux (*osi_linux*), then there is one for Windows XP as well, but it is also possible to create other plugins that work under *osi* if it is necessary. For the OS-dependent part (API calls, system calls) I used *syscalls2*. Previously there was another version of the same plugin, that only supported the previous version od PANDA (Panda1). This is the new version of the plugin that is able to hook system calls for different operating system versions also for the new version of PANDA, Panda2. For the memory write tracing there is *on_virt_mem_after_write*, which is a simple callback that the PANDA main infrastructure offers. There were then other methods that will be described later because they were different according to the operating system that I was analysing.

4.4.4 Problems

The trickiest part I had to go through during this phase was the beginning, because it took quite some time to figure out how the framework was organised in DECAF. Except for the time I had to spend in studying the DECAF dependencies there was also the issue of discerning which parts of the original code were useful to compute the complexity of the packer and which were not. Some parts of code had the only purpose of implementing some heuristics to try to guess the family of the packer, but this had nothing to do with the computation of the complexity. I took the general decision not to implement the part regarding the heuristics because I considered it as a side work that could be done after the completion of the relevant part. A final comment on this first part is that it was not possible to move from a stable version to another one. To see the first results it was necessary to port the whole plugin, otherwise nothing would have compiled. This ended up in a time consuming and error-prone approach, because at the end of the porting phase I had to solve all the dependency errors that occurred because of something that I could not test in advance.

4.5 Windows 7

The first implementation of the code supported Windows XP 32bit executables. To check that the porting has been completed successfully I started implementing the plugin with the intention of

[3] https://github.com/panda-re/panda/blob/master/panda/plugins/osi/USAGE.md

support the same kind of executables. I decided thus to write the PANDA plugin for Windows 7 recordings. This allowed me to test the plugin functionality with Windows XP executables, but at the same time I was able to extend the support for more executables, including the ones for Windows 7.

4.5.1 Plugin Components

For the process tracing part the *osi* plugin was needed as a "glue" plugin, but other two plugins were required for the Windows introspection. The first one was *win7x86intro*, the plugin that actually implemented the already-cited callbacks (*on_new_process*, *on_finished_process*, etc.). The second one is *wintrospection* and it was required by *win7x86intro* to work. For the OS-dependent part I used the plugin *syscalls2* provided hooks to system calls and I used those related to Windows 7 system calls. For the memory write tracing there were two different ways of tracing a memory write. The first one was to use *on_virt_mem_after_write*, the PANDA callback that is called every time there is a memory write. The second one was to log API calls to *WriteProcessMemory* by hooking every call to the *on_NtWriteVirtualMemory_enter* syscall. This hook was in charge of logging every remote memory write, stating that there was inter-process interaction during the execution of the target sample.

4.5.2 Heuristics

Part of the original plugin implemented some heuristics to guess the packer family. The heuristic is based on some well known patterns of the target executable's behaviour, such as the list of imported Dynamic-Link Libraries (DLLs) or the list of exported symbols, for each of the processes belonging to the target executable. The parts of the code that implemented the heuristics were not required to compute the packer complexity. As a consequence of it, I decided to implement the heuristics only for Windows 7 and not for other operating systems/architectures.

The basic behaviour of the heuristic can be summarized as follows: first of all, a list of the exported symbols from kernel modules is required. *on_get_modules*, a PANDA callback, works perfect for this purpose. Then for each element of the list it is necessary to get to two arrays that are stored in a kernel structure A similar process can be done for retrieving the loaded DLLs, using this time *on_get_libraries*

4.5.3 Results

In this section I will show some of the results that this particular implementation (for up to Windows 7 PE executables) produced, highlighting the peculiar aspects that appear in each graph.

Figure 4.3 shows the output graph of a simple *UnPackMe* executable. It comes from a collection of Windows PE executables that are packed with some runtime packers. These executables are not malwares nor useful programs. Their main purpose is to show how different runtime packers work. They are simple programs that output a "success" message when their code comes to an end. This means that the typical behaviour of such executables is to first be unpacked by the runtime packer and then to show that they finished their execution, either by prompting out a graphical pop-up message or by simply printing a line on the command prompt, accordingly to how they have been launched. The graph represented in Figure 4.3 shows how the executable behaved during the execution (recorded with PANDA) and how was the interaction with the runtime packer code (namely UnPackMe_WinKrypt). As we can see, the graph shows two layers of code, the upper one containing the initial code, i.e. the unpacking code from the runtime packer, while the lower layer contains the unpacked code (the original code of the program). We can see that the transition model is *linear* because there are only downwards arrows, meaning that there was no interleaving between the unpacking stub and the original code. The features described so far can be summed up by saying that the packer that packed (and unpacked) the code of this program was of complexity type 1. This is the simplest case of runtime packer. If we analyse the executable by using some reverse engineering technique we can see that the original code is preceded by what is called *tail*

jump, which is basically a assembly *JMP* instruction that separates the unpacking code (which comes first in the execution) and the original code.

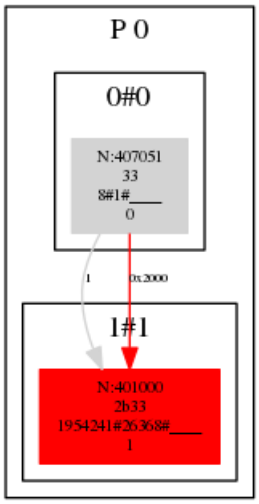


Figure 4.3: UnPackme_WinKrypt

Figure 4.4 shows the output graph of another *UnPackMe* Windows PE executable. This time the simple program is packed with another runtime packer called Zprotect. This graph looks more complicated than the previous one, in fact the packer is classified as a complexity 3 runtime packer. We can see that there are three layers of code, the top one is again the code where the execution started, while the other two layers represent the unpacked code. This case is more complex than the previous one because the layer in the middle is at the same time an unpacked layer (unpacked by the top layer) but also an unpacking layer since it unpacks the code contained in the bottom layer, where the original code is. The transition model is cyclic as we can see there are multiple interactions between the middle and the bottom layers (i.e. there are both downwards and upwards arrows). Even though it is more complex than the graph in Figure 4.3 we would find again a tail jump if we analysed the assembly code of this program.

4.5.4 Problems

Due to an improper implementation of the callback *win7x86intro* the callback *on_get_modules* always returned *NULL*. The reason for that was that the value for the KDBG offset was always zero. To cope with that, I modified the implementation of *win7x86intro* by computing the KDBG offset for each sample using the volatility plugin *imageinfo* on a memory dump of the running sample. There is still a open problem regarding the inter-process interaction between two processes of the same target executable. Some samples call the *WriteProcessMemory* function. The online service *packerinspector.com* correctly recognized and logged those calls, while my implementation of the plugin did not produce any traces of that in the output log files. Up to now there is still no solution for that.

Figure 4.5 shows an example of what a misinterpreted graph looks like. As we can see, there are two processes that were spawned by the program. The one on the left apparently does not present any unpacking behaviour, the one on the right instead has multiple layers of unpacking. What is missing here is the interaction between the two processes, in fact we can see there is no arrow connecting the first process with the second one, while it is known that Armadillo should present that kind of inter-process interactions[4].

[4] https://www.packerinspector.com/example/6

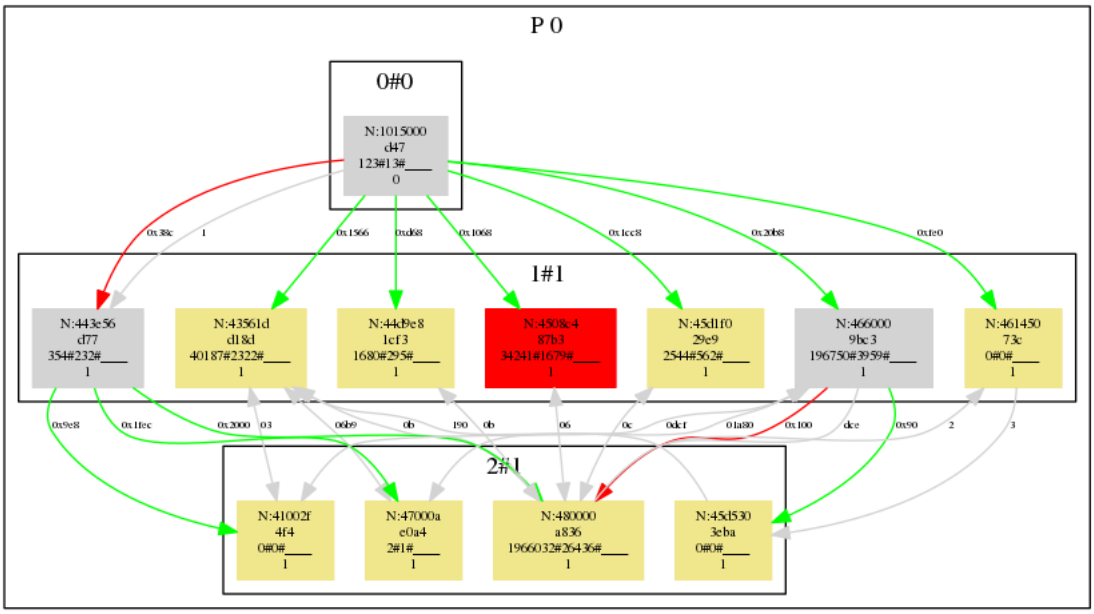


Figure 4.4: UnPackme_Zprotect

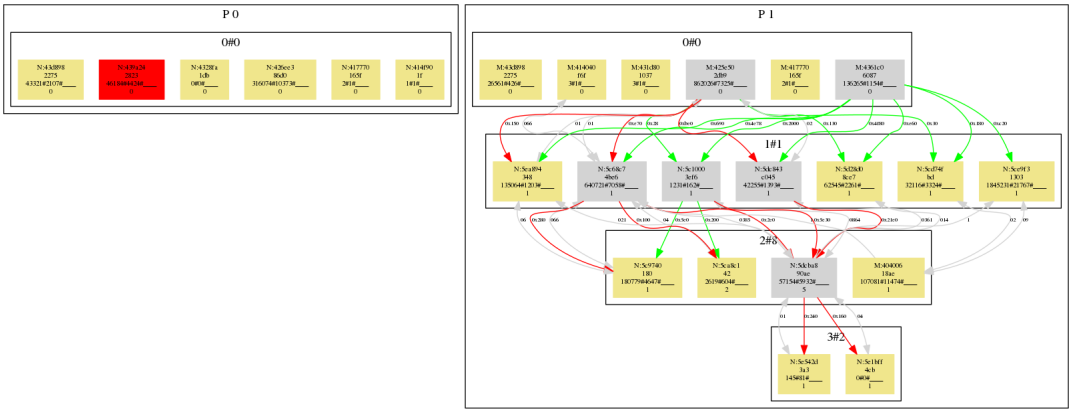


Figure 4.5: UnPackme_Armadillo v. 3.70

4.6 Linux

4.6.1 Plugin components

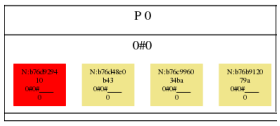
For the process tracing part the *osi* plugin was required as for the Windows 7 implementation. This time the fundamental callbacks were implemented in the *osi_linux* plugin, that was required as well. For the OS-dependent parts, again the *syscalls2* plugin was needed to be able to palce hooks on the Linux kernel system calls. This plugin implemented the system calls only for the 32bit version of the Linux kernel, and that is unfortunate because it means that complexity analysis cannot be extended to support 64bit versions of the Linux family of operating systems. For the memory writes several methods were possible with Linux to write to a process' memory. The first one is the classic *on_virt_mem_after_write* callback (that I used also for the other implementations), then there are a couple of ways to write a remote process' memory using some system calls. In this sense I hooked the following system calls:

- *on_sys_ptrace*, that is necessary to write on a remote process' memory. It is also possible to remotely write using only this system call, by specifying some specific flags

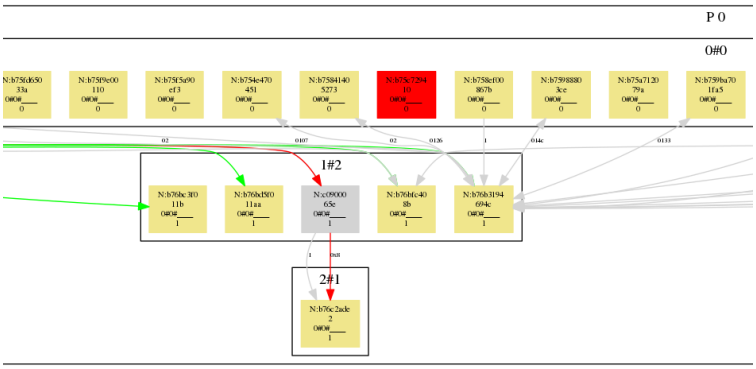
- *on_sys_pwritev* and *on_sys_pwrite64*, that are two system calls to write respectively arrays of integers or a single 64bit integer on a remote process' memory

4.6.2 Results

Here I will show some of the results I obtained after finishing the implementation for x86 Linux of the framework. The first thing I tried to do was to see the difference between the graph a normal program would have produced with respect to the graph produced by the same program, this time packed with a runtime packer. Figure 4.6 shows two instances of the same Linux program, *foremost*[5]. *foremost* is a command line program that takes as argument a file, typically a disk image, and tries to recover the files that are contained in that image according to their headers. I chose this program because its code is big and complex enough to be packed. To pack *foremost* I used UPX. Figure 4.6a represents the basic behaviour of a not packed program, with a single layer of code, containing the whole original code that has been executed at runtime. Figure 4.6b shows instead the behaviour at runtime of the same executable, this time packed with UPX. As we can see there are multiple layers of packed code, showing that the first executed code is the unpacking stub. Another thing that is worth to be mentioned is that here the original code is definitely not in the deepest layer (the lowest one), because it is too small to contain the original code of the program. This means that the original code is located in layer 1 (the one in the middle).



(a) *foremost*, not packed



(b) *foremost*, UPX packed

Figure 4.6: Different graphs for the same instance of *foremost*

Figure 4.7 shows instead a malware. It was packed as in the previous case with UPX. As a consequence, the resulting graph looks like the previous one. We can see that there are multiple layers, the top one containing the unpacking stub and the middle one containing the code of the application. The transition model is cyclic because there are both upwards and downwards arrows, showing that the execution went from the unpacking routine to the original code and vice versa. The graph overall classifies the malware as a complexity type 3 packer.

4.6.3 Problems

The process creation in a Linux shell was a bit problematic for the process monitoring part. All the samples were started from a shell, this means that they followed the classic process of process creation in Linux, i.e. at the beginning, the shell process forks itself, then the child calls the *execve* syscall. *on_new_process* from the *osi* plugin was able to spot only the newly created process (still named 'bash') and it was missing the process 'renaming' after the *execve*. The solution to this problem was to add a check for this scenario in the implementation of *on_new_process* in the *osi_linux* plugin.

[5] https://linux.die.net/man/1/foremost

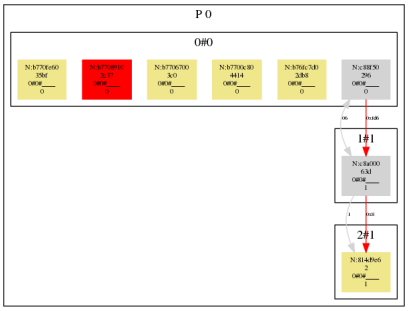


Figure 4.7: An x86 Linux malware, UPX packed

4.7 ARM

4.7.1 Plugin components

As for the other two implementations of the plugin, for the process tracing *osi* was required as a glue layer, under which there was again *osi_linux* because I developed the ARM plugin for a Linux 32bit operating system. For the OS-dependent parts, the *syscalls2* plugin offered the system calls for the Linux kernel, while for logging the memory writes the configuration were exactly the same that I used for the Linux x86 implementation, i.e. the *on_virt_mem_after_write* PANDA callback and the hooks on *on_sys_ptrace*, *on_sys_pwritev*, *on_sys_pwrite64*

4.7.2 Results

In this section I will present some example results of ARM recordings. To be able to make a comparison between two different implementations of my plugin I decided to test the same Linux program, *foremost*, which I beforehand packed with UPX for Linux ARM. Figure 4.8 shows the results of this first test. As we can see there are some differences between this graph and the one in Figure 4.6b. There is one additional layer of unpacked code here (four layers in total instead of three layers, as in the Linux x86 case). The overall classification of the complexity does not change though (still complexity type 3 packer), because the transition model is still cyclic and there would still be a tail transition if we had a look at the disassembled version of this elf executable. Figure 4.9 shows another complexity type 3 output graph. This time I packed with UPX another Linux command line program, *find*, whose behaviour is well known. As we can see, its graph is similar to the previous one. This is because the version of the packer (and so the family of the packer) was the same one I used for the previous test. Figure 4.10 shows instead a real malware that was already packed by its author. As for the other cases, there are multiple layers of unpacked code (two layers) and the double-arrow edges that connect different layers together mean that the transition model is cyclic.

4.7.3 Problems

The callback *on_asid_changed* was not defined for ARM targets. To cope with it, I added a check after every basic block execution (with the *on_after_block_exec* callback) to check for a possible context switch (i.e. every time there is a change of the page directory that identified each process' virtual memory mapping to the physical memory).

4.8 Porting to Panda1

The previous plugins were developed for Panda2, a newer version of PANDA. B. Dolan-Gavitt (one of the authors of PANDA) hosts a huge collection of malware recordings (`http://panda.gtisc.`

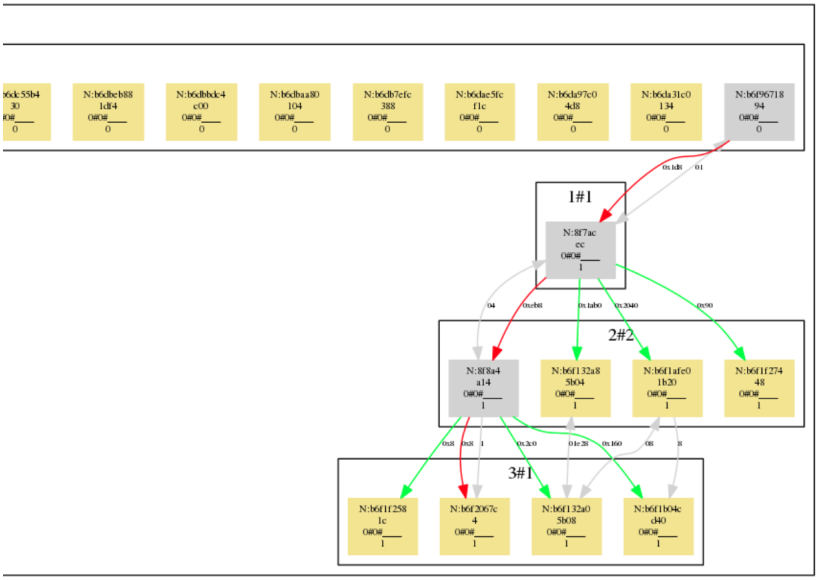


Figure 4.8: *foremost*, UPX packed

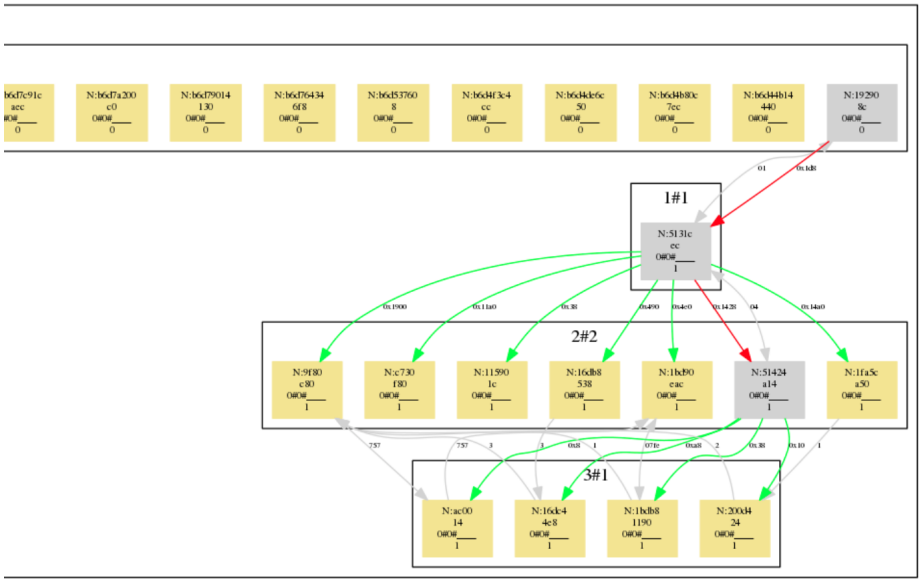


Figure 4.9: *find*, UPX packed

gatech.edu/malrec/). There are almost 90000 samples, all of them were recorded in a Windows 7 32bit environment with Panda1. Unfortunately, recordings cannot be converted from Panda1 to Panda2, so I needed to re-implement the plugin for Windows 7 on Panda1. I decided to do that because it was a huge dataset thanks to which I was able then to perform a lot of tests to see what is the distribution of packed samples in the wild.

4.8.1 Results

Here are described some of the results that I obtained from tests I ran in the last phase of the internship. Tests have been done on Windows 7 samples (from Dolan-Gavitt's database). They were all recordings of 32bit PE malwares. Around 10000 samples have been tested. Here are some statistics of the distribution of the complexity among the malwares that have been tested:

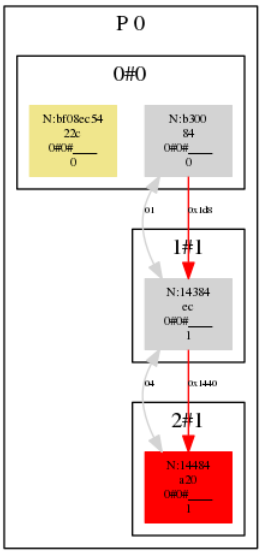


Figure 4.10: An ARM Linux malware, UPX packed

- Complexity type 0: 30.2 %
- Complexity type 1: 2.6 %
- Complexity type 2: 3.3 %
- Complexity type 3: 62.9 %
- Complexity type 6: 0.9 %

As we can see the most common class of complexity of runtime packers is Type 3. Almost every other sample was simply not packed, but there were some few cases where the packer belonged to a different complexity class. Here I will show one graph per category that has been found. The following graphs are representative for the majority of the samples that have been analysed by the framework.

Figure 4.11 shows how a graph of a non-packed executable looks like. As we can see, the graph is similar to the one in Figure 4.6a that showed the graph for a not packed binary for Linux. The features are the same: there is a single layer (layer 0) where the whole executed code lays.

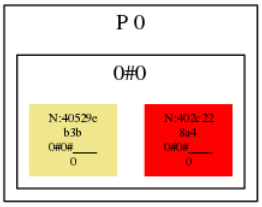


Figure 4.11: Complexity type 0, not packed.

Figure 4.12 shows a graph of a complexity type 1 runtime packer that has been found operating on a malware in one of the tested recordings. It has only two layers, one unpacking layer (on the top) and an unpacked one (on the bottom). The former contains the unpacking stub that de-obfuscates the latter. The transition model is linear because there are only downwards arrows.

Figure 4.13 shows a complexity type 2 graph. Here multiple layers start appearing. The highest one (on the top) contains the starting code, while the other three are unpacked layers. The

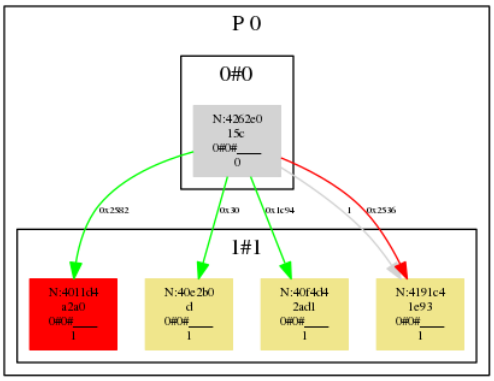


Figure 4.12: Complexity type 1, linear transitions, single layer, tail transition.

original code is contained in the deepest layer (the bottom one) since the transition model is linear (downwards arrows).

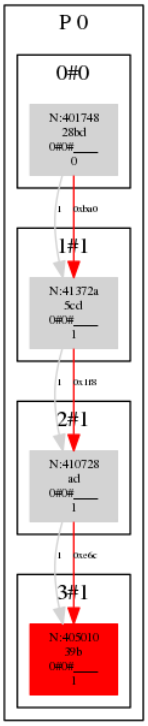


Figure 4.13: Complexity type 2, linear transitions, multiple layers, tail transition.

Figure 4.14 shows the most common graph so far. Type 3 is the complexity level that occurred the most while running the tests. Here we can see also upwards arrows that link together different layers, which means that the transition model is cyclic. As a consequence, the original code may not be found in the deepest layer.

Figure 4.15 shows a complexity type 6 output graph. It is clearly visible that this does not represent the actual graph of a Type 6 packer, but every other graph belonging to this category resulted in the same profile. This behaviour is due to the Windows issue with the *WriteProcessMemory* API call and the corresponding *NtWriteVirtualMemory* system call. Because of that the inter-process interaction between the two processes that appear in the figure is missing, resulting in a not reliable source of information.

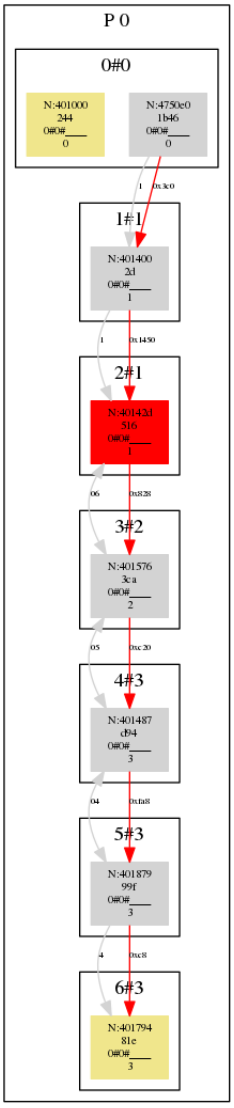


Figure 4.14: Complexity type 3, cyclic transitions, multiple layers, tail transition.

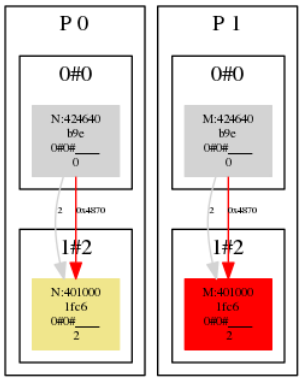


Figure 4.15: Complexity type 6, two processes, graph may depend on the Panda1 issue of *NtWriteVirtualMemory* in the process interaction.

4.8.2 Problems

In this phase of the thesis I did not encounter many problems. Being another version of a plugin for analysing Windows PE files there were the aforementioned issues involving the monitoring of Windows executables, one for all the failing implementation of the *NtWriteVirtualMemory* system call. Even in this case there were situations in which that system call correctly traced API call to *WriteProcessMemory* while other times it did not register any system call, bringing erroneous results, as I explained Figure 4.15. Apart from that, the only other problem was the amount of samples that I needed to analyse in order to build up the statistics. Many samples did not produce a valuable result that could be compared and classified in the six-value scale for the unpacking complexity. For this reason I actually had to analyse way more samples than the thousand that produced a valid output to compute the statistics.

4.9 Non-implemented features

The original implementation of the framework, developed as a plugin for TEMU/DECAF, included some additional features that were not necessary to compute the complexity of the runtime packer. Such features were, for example, all the heuristic parts necessary to determine the packer family. As I have already said in Section 4.5.2 I decided not to implement those features because they were not relevant in computing the complexity of the runtime packer. In my implementation of the framework there is also another part missing, which is the one in charge of computing how many and what type of API calls the program does during its execution. This is also related to try to guess the family of the packer.

Chapter 5

Automated Testing Framework

In the final phase of the internship I wanted to assess the actual working status of the framework. To do so, I needed to perform a series of tests that required an automation process. I decided to build what I called "automated testing framework", that is a combination of Python and Bash scripts (totalling approximately 1100 lines of code) that make the analysis of one or more samples automatic.

5.1 The architecture

When I wanted to create the Automated Testing Framework (ATF) there were several situations where having it would have made sense. Among all the possible scenarios where the ATF would have helped I decided to focus on two main situations. The first one required an easy way to perform all the analysis by running a single command, without the need of running every single part of the analysis to come to the final result. To do so I combined together the components of the framework as if they were in a pipeline, connecting the outputs of one phase to the inputs of the next one. The second one dealt with the multiple samples I wanted to test from Dolan-Gavitt's collection of recordings. In this case I extended the automation done for the first scenario to test a long list of recordings. In the following sections I will go through the important aspects of this part of the internship that contributed to the final results of the thesis.

5.2 Simplify the execution

The original plugin was divided into three main parts, each one producing some outputs to be combined together to obtain the final results. The beginning part of the analysis simply took the sample recording and performed the preliminary analysis (dynamic analysis) thanks to PANDA. The second part organised the results of the dynamic analysis to produce the behavioural graph that shows what the packer did at runtime. The final part computed the complexity of the runtime packer according to the results of the first two phases. In the last phase of the internship I wanted to simplify the computation of the complexity by making everything more user-friendly and easy to be executed. I decided then to write a script that is in charge of doing everything, starting from the sample executable file to the final results. To do so there were three main steps that needed to be done. For each file that I wanted to analyse I needed at first to create a PANDA recording from the sample, then to dynamically analyse the recording again with PANDA and in the end to produce the final results for the runtime packer complexity analysis. Appendix B shows the code of the Python script (*run_target_mal.py*) I wrote to accomplish this task.

5.2.1 Requirements

Before running the script it is necessary to understand how it will combine things together and how the required files will be used to generate the final report. The first step of the automation is the generation of a recording of the target sample's execution in PANDA. The PANDA commands described in Chapter 2 are sufficient to complete the first step. A requirement to be able to record the execution of a sample in PANDA is having the actual sample executable in the emulated operating system. To do so there are several methods that depend on the emulated OS. If it is a Windows OS one option is to transfer the target file using a USB image:

1. Create a USB image:
   - On the host:
     `qemu-img create usb_image.img 50M`
   - Start the guest OS with the `-usb` option as an additional command line argument:
     `<path/to/PANDA/qemu-system-i386 -hda <image> [other_options] -usb`
   - Add a USB device using the PANDA-qemu monitor:
     `usb_add disk:<path/to/usb_image.img>`
   - Wait until the guest installs the drivers
   - Open the disk administrator, initialize and create a partition (with FAT32 format)
   - Eject the USB image
   - (Optional) It is also possible to take a snapshot at this point
2. Copy the sample executable into the USB image
   - Mount the USB image on the host:
     `mount -o loop,offset=<offset> <path/to/usb_image.img> </mnt/point>`
   - Copy the sample executable into the USB image:
     `cp <path/to/target_file> </mnt/point>`
   - Un-mount the USB image:
     `umount </mnt/point>`
3. Attach the USB drive with data to the guest OS
   - Start the guest OS with the `-usb` option as an additional command line argument (it is possible to start from a snapshot in the same way):
     `<path/to/PANDA/qemu-system-i386 -hda <image> [other_options] -usb`
   - Add a USB device using the PANDA-qemu monitor:
     `usb_add disk:<path/to/usb_image.img>`
4. Copy the sample executable to the target directory according to the guest OS

Another option for a Windows guest OS is to download the target executable from the internet, e.g. from a private repository on a cloud service. It is thus necessary to provide the guest with an internet connection. A simple PANDA-qemu command line argument to get connectivity is the `-net nic -net user` option:

```
<path/to/PANDA/qemu-system-i386 -hda <image> [other_options] -net nic -net user
```

If the guest operating system is instead Linux-based (e.g. Ubuntu or Debian) a possible choice to transfer the sample executable from the host to the guest is to use a connection via *ssh*:

1. Install `ssh-server` on the guest OS:
   `apt-get install openssh-server`

2. Copy the target file from the host to the guest:

   `scp -P <port> <target_file> <username>@localhost:<path/to/file>`

   where:

   - `<port>` is the port on which the guest's openssh-server is listening for connections
   - `<target_file>` is the target executable that we want to copy over the guest OS
   - `<path/to/file>` is the path to the destination folder on the guest OS

PANDA recordings will produce two outputs, a log file for all the non-deterministic inputs of the execution and a memory snapshot that is taken at the beginning of the recording. Both of them can be very big, reaching few GB of size, so it is necessary to have at least 10 GB of free space on the host machine.

Another requirement is to have a configuration file that needs to be set up in order have a structure like the following:

```
[Main]
basedir = /home/samaicardi/replays
panda = /home/samaicardi/my_panda2
images = /home/samaicardi/.panda
makegraph_dir = /home/samaicardi/deep_packer_inspector_makegraph
dpi_home = /home/samaicardi/my_panda2/panda/plugins/packer_inspector

[VM]
mem = 1G
exec_time = 4
replay_time = 200
repeat_replay = 5
```

where:

- `basedir` is the directory where your target file is, on the host
- `panda` is the path to the folder where PANDA is installed, on the host
- `images` is the directory where the qcow2 images are stored, on the host
- `makegraph_dir` is the path to the folder where the code for second part of the analysis is stored, on the host
- `dpi_home` is the directory where framework for the complexity analysis as a PANDA plugin is stored, inside the PANDA root directory, on the host
- `mem` is the amount of RAM that we want to give to the guest OS, as a command line argument
- `exec_time` is the maximum amount of time (in seconds) that the execution of the target executable can last, on the guest OS. After `exec_time` seconds have passed, if the program is still running, it is terminated.
- `replay_time` is the maximum amount of time (in seconds) that the replay of the recording can last, on the host OS. It corresponds to the first part of the analysis, i.e. the dynamic analysis with the PANDA plugin.
- `repeat_replay` is the maximum number of times that the ATF can replay the recording. There are situations in which the first replay crashes because the execution reached a non-stable point in the recording (due to several external factors depending both on PANDA-qemu and the used plugins). In these cases the recording is started again with a timer that will trigger a signal handler after (0.8*`replay_time`) seconds: by doing so it is less likely that the program will crash in the next execution. If the second run still crashes this process will be repeated until `repeat_replay` times, each time decreasing the time of previous run by 0.8

5.2.2 How-to

The Python script I wrote to simplify the execution of a complete test is called *run_target_mal.py* and it is typically called in this way:

```
./run_target_mal <path/to/sample_name> <output_dir> <config_file> <arch> \
<vm_folder> [--make]
```

where:

- `<path/to/sample_name>` is the path to the executable we want to analyse
- `<output_dir>` is the directory that will contain the output files, both the intermediate and the final ones
- `<config_file>` is a file that contains some global variables required by the program. They have to be in a separate file because of scalability reasons. It is the one described in the previous section.
- `<arch>` is the architecture which the executable is compiled for. It can be *arm*, *i386* or *x86_64*
- `<vm_folder>` is the directory that contains the instance of the sample executable on the emulated operating system
- `[--make]` is an optional argument thanks to which it is possible to specify if the *packer_inspector* plugin needs to be recompiled. One situation where the plugin needs to be recompiled may be, for example, when we want to test a Windows 7 PE executable after having tested a Linux elf file. There would be some internal configuration files that depend on the type of the executable we want to analyse

5.2.3 Create the recording

The script is divided into three main parts, each of them is called by the main function that manages the command line arguments and sets up the required signal handlers and the file loggers. The basic idea of the main function (better described in Appendix B is to execute each part of the script one after the other, stopping the execution as soon as it comes across an error or an exception. The first part of the script (corresponding to the code inside the function called `run_sample`) is in charge of record the execution of the target malware, which needs to be already in the guest OS's filesystem. There is an important if condition that divides the execution in three branches according to the value of `{arch}` (either *i386*, *arm* or *x86_64*). If the architecture is *i386* or *x86_64* then the execution is quite similar: first of all the script sends to the qemu monitor the command to start the recording (`begin_record <record_name>`), then it types in the guest OS the commands to start the target executable. If the architecture is *arm* then the recording of the target executable will be taken twice. The reason behind it is that it seems that the recording of an *arm* executable does not work the first time that it is taken, while the second time it works normally. When the executable has finished running (either because it terminated or it was stopped) the script sends to the qemu monitor the command to stop the recording (`end_record`).

The interaction between the script and the qemu monitor is handled by the function `mon_cmd` that sends the commands character by character via a *telnet* connection. This method is also used for the interaction between the script and the guest OS because the qemu monitor offers a command (`sendkey`) that serves for this specific purpose.

5.2.4 Perform the analysis

Once the first part is finished we have the recording of the execution of the target sample, i.e. the log file of all the non-deterministic inputs and the memory snapshot taken at the beginning of the execution. Now it is time to perform the first round of analysis on the recording. It will be a dynamic analysis using PANDA record&replay. The second function that is called by the main function is `perform_analysis`. In this phase the script calls PANDA-qemu and it specifies *packer_inspector* as a command line argument. It is time to use my PANDA plugin version of the framework to dynamically analyse the recording of the target executable. First of all, the script sets up a time alarm that will notify when the execution of the replay needs to be stopped because the time exceeded a fixed threshold. When the dynamic analysis has finished (and PANDA stops replaying the execution) the script will save the intermediate results that will be required by the last phase of the analysis. If the replay did not end in the correct way the analysis will be performed again, this time by decreasing the time threshold for the test, hoping there will not be errors due to a shorter execution time.

5.2.5 Produce the final results

The final part of the script (function `third_part`) takes care of the intermediate outputs of the previous part (*events.log*, *execution_model.log*, *functions.log* and *trace.log*) by moving them in the right directory. After that it calls a bash script, *automated_graph_production.sh*. This script is in charge of producing the unpacking behaviour graph and the complexity analysis. Additional information on the structure of the script can be found in the Appendix B. At the end of the execution of *run_target_mal.py* we can find all the results (both the intermediate and the final ones) in a single folder inside *deep_packer_inspector_makegraph/*.

5.3 Automate large scale tests

The second scenario where I decided to implement the ATF was to test a lot of samples from Dolan-Gavitt's collection of malware recordings. As already said, all the recordings from that database were taken with Panda1, the previous version of PANDA, on a 32bit version of a Windows 7 guest OS. The basic idea for this part of the work was to create a series of pluggable scripts that could be used to apply the runtime packer complexity analysis to a large set of recordings, without the need of executing manually the analysis for each sample. With the current status of the work, a potential user would just need to execute the script called *start_analysis.sh* by specifying some parameters that will be described later in this section. This time the procedure is a little different with respect to the previous one, in fact there is no need of recording the target executable because it has already been recorded by Dolan-Gavitt's framework [1]. Appendix C shows the main scripts that are required by this part of the internship.

5.3.1 Requirements

There are few things that need to be done to be able to start the large scale analysis. First of all we need to get the recording of target executable. Since a single recording (snapshot + non-deterministic log) can be very heavy in terms of space on disk, Dolan-Gavitt's database does not store every memory snapshot. They can instead be derived from a starting memory snapshot that is taken as a reference by computing a sort of binary `diff` between the two snapshots. What typically happens on Dolan-Gavitt's framework is the following:

1. The execution of the target malware is recorded, producing a memory snapshot and a log of all the non-deterministic inputs of the CPU

[1] https://github.com/moyix/panda-malrec

2. A binary `diff` between this memory snapshot and the reference snapshot is computed and saved in a text file (<*malware_hash*>*.patch*)

3. The non-deterministic log and the patch are compressed in a *txz* archive and then stored in the database

4. Some additional files are stored in the database too, especially a *json* file containing the information coming from a VirusTotal report (we will need it later)

There is a Python script that can be used to generate the snapshot for a specific sample, starting from the original reference snapshot and the non-deterministic log. It is called *bpatch.py* and it is well described in a post by Irfan Ul Haq[2].

Since we want to apply the analysis to a large set of malwares we need to create a list of all the samples we want to analyse. The script is designed to download the *txz* archive containing the necessary files directly from one of the two websites that host Dolan-Gavitt's collection, `http://panda.gtisc.gatech.edu/malrec/` and `http://giantpanda.gtisc.gatech.edu/malrec/`. As we can see if we visit any of the two websites there are several textual columns, each one displaying information about the malware hosted on a particular row. Specifically, to download the *txz* archive we need the information contained in the first column, the *UUID* of the sample, because the archive is called <*UUID*>*.txz*. Once we have decided which samples we want to analyse we will need to write a list of their UUID in a *txt* file that has to be called *panda_malware_list.txt*.

5.3.2 How-to

To start the large scale analysis it is sufficient to run the script called *start_analysis.sh*:

```
./start_analysis.sh
```

The script will launch *automated_win7_script.sh*:

```
./automated_win7_script.sh
```

which will take care of all the required input to the Python script that actually performs the analysis, *execute_win7_malware_replay.py*:

```
./execute_win7_malware_replay.py <sample_dir> <config_file> <arch> \
<sample_name> <exe_name>
```

where

- `<sample_dir>` is the directory where the recording is stored
- `<config_file>` is the configuration file that contains all the global variables, in the same format of the one explained in Sub-Section 5.2.1
- `<arch>` is the architecture used for the emulated guest OS (*x86_64* in this case)
- `<sample_name>` is the UUID of the sample, taken from one of the two websites
- `<exe_name>` is the actual name of the executable, used in the guest OS

[2] `https://irfanulhaq.info/2015/12/09/replay-panda-malware-recordings/`

5.3.3 Scripts execution

If we satisfied all the requirements we are ready to run *start_analysis.sh*. The script just cleans up some directories and then calls another script, *automated_win7_script.sh*. This one reads the UUIDs from the list and for each of them it performs the following actions:

1. It downloads the *tgz* archive from one of the two websites
2. It extracts the non-deterministic log file and the patch for the memory snapshot
3. It restores the actual snapshot starting from the reference snapshot and the patch
4. It starts *execute_win7_malware_replay.py*
5. It does some clean-ups for the next iteration

execute_win7_malware_replay.py is the final script that is executed for this type of analysis. It is quite similar to *run_target_mal.py*: it calls the same functions except for `run_sample` that is not necessary any more.

5.4 Performance

Few words need to be spent regarding the performances of all these analyses. For what concerns the computational power there are no big issues. The framework itself is quite complex in terms of memory accesses and control flow instructions, but the way PANDA manages the memory usage is very good. There are a lot of structures that need to be created (i.e. allocated in the heap) during the execution of a replay, e.g. all the *OsiProc* structures that are returned when calling *get_current_process* or *get_processes*. Nevertheless PANDA implements also the corresponding functions to free the memory allocated for those structures, so if a proper use of memory is adopted when writing a plugin the amount of memory required to run the whole recording can be as small as 0.23 GB. If we talk about time performances, instead, we need two separate two main families of analyses: the Linux family and the Windows family. While the former does not have big duration issues (we are talking about *seconds* of total time per analysis), the latter typically needs several *minutes*. This is due to the fact that Windows replays tend to last way more time than Linux replays, both because the emulation of a Windows OS (with a graphical interface and so on) is heavier and the CPU cannot go as fast as the one in a Debian Linux Server (no graphics, only the essential shell), and also because Linux executables typically last less seconds than the Windows PE files.

Chapter 6

Comparison with the State of the Art

6.1 Comparison with the State of the Art

In this section I will compare some of the results I obtained from my implementation with the State of the Art (SoA) for this particular problem. As I already explained before, there is a web service that is available at `www.packerinspector.com`. This service is what can be considered the SoA for evaluating and ranking the complexity of runtime packers. In order to be able to compare my results with the SoA, I submitted several samples of packed executables to the online service and I retrieved the results of those analyses, especially by taking into account the output graph showing how the packer behaved at runtime.

As we can see from Figure 6.1 the website is very simple: it is possible to select a sample from your local machine, upload it and then start the analysis by clicking on the *Analyze* button. The service will analyse the sample according to its behaviour at runtime and it will show the results of the analysis in a format similar to what is represented in Figure 6.2. I will focus on the output graph because it is the best and most immediate way to understand the behaviour of the tested runtime packer.

Once I had enough results from the web service I started analysing the same group of samples with my implementation of the framework. Then I compared the corresponding results for each sample by paying particular attention to shape and the content of the two graphs (mine and the one resulting from packerinspector.com). Unfortunately, the web service can only analyse PE executables, i.e. programs that can run only on a Windows operating system. For this reason I could not compare all of my samples against the SoA, but only the ones that were designed for Windows. Besides, it was anyway a good mean to test whether my initial implementation of the complexity analysis was OK and if it was producing the correct outputs. In the next sections I will show some of the results I got from my experiments and how I compared them with the SoA. I will not show all the comparisons I have done during the testing phase, neither half of them because they simply are too many. Furthermore, they are not all relevant for the purpose of this thesis, they are required only to show that everything worked fine and that my implementation produced the same results of the web service. With that in mind, I will thus show only a couple of examples of the comparisons I did in order to test my system, pointing out for each of them something that is peculiar of that particular comparison.

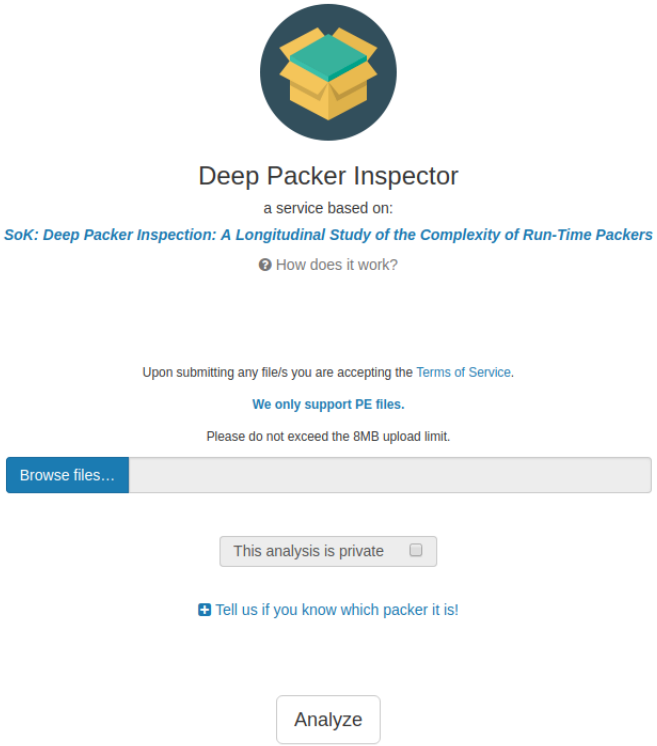


Figure 6.1: Main page at www.packerinspector.com

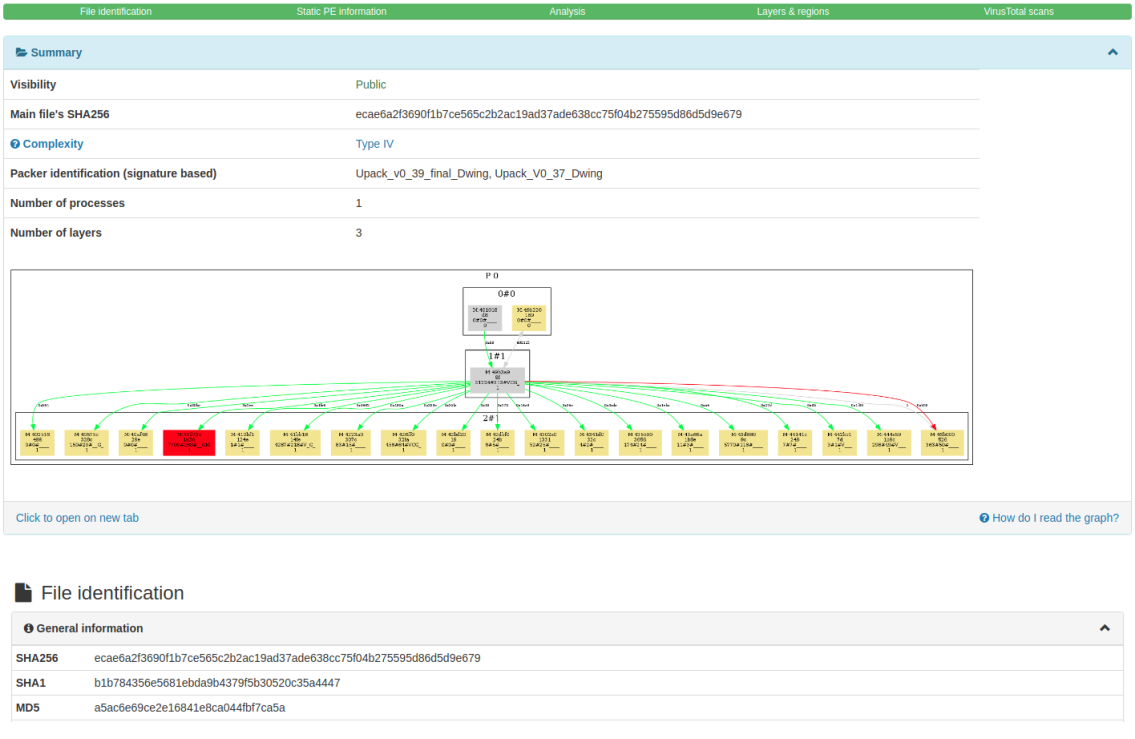


Figure 6.2: A sample result

6.2 Interpret the graphs

The results of my implementation present the information about the complexity analysis in the same way packerinspector.com does. It is thus reasonable enough to explain how to interpret the web service's output graph, that represents the behaviour of the analysed sample. This section is taken from packerinspector.com [17]

6.2.1 Colour Blindness

The unpacking graphs are colour-coded and the explanations below use colours to refer to certain parts of the graphs. Figure 6.3 shows the colours and colour names that we will refer to. Note that there are two types of greens, Arrow Green and Box Green, that appear on arrows (connectors) and boxes respectively.

Figure 6.3: Colour Blindness

6.2.2 Processes

The web service monitors all the processes created during execution, as well as those the sample interacts with. A graph may show one or several processes, if the system detects, for example, that the binary has injected code to another process. Each process is designated by a process number (e.g. P0, P1). The unpacking layers and memory regions of each process are contained in a separate box for each process. The graph in Figure 6.4 shows 2 processes.

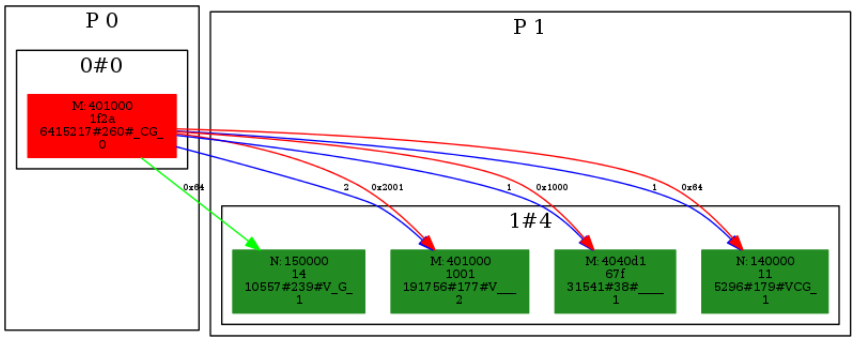


Figure 6.4

6.2.3 Layers

Each process will have at least 1 layer of code (if some code was executed), and up to any number of layers. Each layer is represented as a blank box containing horizontally aligned coloured boxes (that represent memory regions). Each layer has a header that follows the format *[LayerNumber]#[NumberOfFrames]*. The first one indicates the layer number: 0 for the executed code that was present in the binary (typically, the code of the packer), and greater than 0 for every unpacked layer. The second number represents the number of frames of code that the layer contains. Figure 6.5 represents a packer with 4 layers (3 of them containing unpacked code, because they are grey, see Subsection 6.2.4). All the layers contain a single frame. Nevertheless, if we look at the previous example, the layer 1 in process P1 contains 4 different frames, given that it is an incremental packer

that unpacks memory pages on-demand, just before they are executed (i.e., there is one frame per memory page executed).

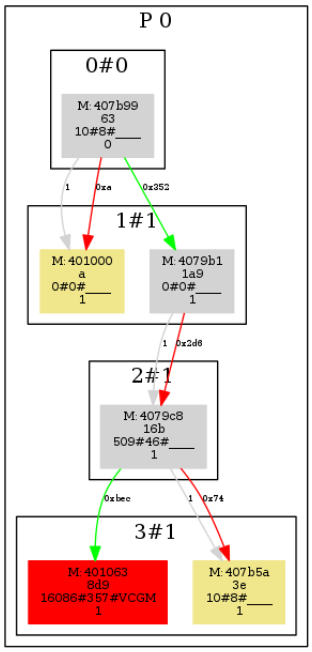


Figure 6.5

6.2.4 Memory regions (boxes)

The coloured boxes inside each layer are memory regions. Each memory region represents a set of contiguous memory addresses that were executed. Furthermore, we group into the same region all executed instructions located at a distance lower than one memory page (4096 bytes). This does not mean that the binary executed ALL the possible instructions in that region, but we group them together to facilitate visual representation. These regions follow a simple colour scheme:

Yellow regions. Represent regions in which there is not a single instruction that wrote the memory of another region. In other words, it represents a piece of code with no unpacking behaviour.

Gray regions. These regions, on the contrary, contain at least one instruction that wrote the memory of another region: it shows some unpacking behaviour.

Green regions. There regions represent memory that has been written remotely from another process (either via *WriteProcessMemory*, shared memory regions, or by loading a file that was written by another process).

Red regions. You will only find one red region in each graph, and it contains the last instruction that was executed during analysis.

Also, the regions contain 4 lines of text with different types of information:

Line 1: Type of memory and base address. The online service distinguishes between 3 types of memory. "M" for module address space, "H" for heap, "S" for stack, and finally, we will use "N" whenever our system does not properly retrieve the memory type. In my implementation there is no such distinction because it was not needed for computing the packer complexity, it was only needed by some heuristics to try to guess the packer family.

Line 2: Size. The size of the region in bytes (in hexadecimal).

Line 3: APIs executed. It shows 3 attributes separated by #:

[NumAPICalls]#[NumDiffAPICalls]#[APICallsByFamily].

The first one represents the total number of API calls made from the region. The second, the number of different API calls executed, and finally, there are 4 spaces for 4 letters: "VCGM". Each letter represents the presence of a given family of API calls (an underscore "_" represents the absence of such API call). "V" corresponds to the *GetVersion* function family, "C" corresponds to the *GetCommandLine* function family, "G" corresponds to the *GetModuleHandle* function family, and finally "M" corresponds to the *MessageBox* related group of functions. The first 3 groups of APIs are related to typical C runtime API calls, and are sometimes used as a way to locate the original entry point of an application. *MessageBox* related functions were also monitored for testing purposes, and are left because many unpack-me challenges show a message box as a payload. See API call families for more information. Also this information is not present in my implementation because of the same reason, it was not needed to compute the packer complexity.

Line 4: Frames. Finally, the last line represents the number of frames that the region contains.

6.2.5 Memory write operations (green and red connectors)

Memory write operations between regions are represented as green and red connectors. The green colour is used whenever a region writes the memory of another region. In contrast, the red colour is used whenever there is a memory write and a execution transition between the same pair of regions. If a region writes another region and then the execution jumps to this code, the connector will be represented in red. Each connector has an hexadecimal number next to it, showing the number of bytes written. For clarity, only the connectors between contiguous layers are shown. Showing all the connections would produce and unreadable graph in certain cases.

6.2.6 Execution transitions (grey and blue connectors)

Execution transitions are depicted as grey connectors, and show the execution jumps from one region to another region in the following layer. Like for memory write operations, execution transitions that occur inside the same layer are omitted, as well as transitions between non-contiguous layers. The number shown next to each connector represents the number of transitions observed. If the connector is blue instead of grey, it means the transition occurred between two different processes. An inter-process transition does not imply process synchronization and might just be a consequence of process scheduling.

6.2.7 Frames

As described before, the number of frames is represented at two different points: at layer level, and at region level. These numbers may not coincide, but why? A frame represents a set of memory regions written and executed at one time. For instance, imagine a packer that first unpacks and executes a given routine, then goes back to the packer code, unpacks another one, and then executes it. This packer would present two frames, one for each routine. The explanation for counting the number of frames with two different granularities (layer and region) is simple: these two frames may be located in the same layer (and therefore, the layer header would show "2" next to the layer number), but the code for each frame might be located in different regions, and thus each region would contain only one frame of code. Now, look at Figure 6.4. Layer 1 in process 1 has 4 frames. Nevertheless, only the region starting at 0x401000 contains 2 frames. This packer protects each memory page separately, so, whenever the execution jumps to a protected memory page, the packer comes in and decrypts its contents (resulting in a new frame). The only region with a size greater than one page is the one at 0x401000 (with 0x1001 bytes), and as a consequence, it presents 2 frames.

6.3 Comparisons

6.3.1 Example 1

Figure 6.6 shows the comparison between my results and the SoA. In this example I submitted the PE executable to the online service and I got back the graph on the right and the information on the upper part of the figure. The results of my implementation can be seen on the leftmost graph (the one in the window called *ImageMagic: graph.png*). The executable I chose for this first test was a simple program that was packed with EXECryptor [18], a Type 3 packer. As we can see the two graphs are very similar, in fact they have the same number of regions, layers and arrows, and each arrow connects the same pair of memory regions. This means that in both graphs (and thus in both analyses) the behaviour of the analysed runtime packer was the same: there was a single layer of unpacking code (the upper one, *0#0*) that unpacked the original code (the lower one, *1#1*). By looking at the numbers on the arrows and on the second line of the boxes we can also see that the memory writes were of the same size. The red box, which indicates the last executed memory area, is different in the two graphs, but this can be related to some differences in the final behaviour of the application in the two executions, also because this particular executable had a graphical interface to show the end of the unpacking stub. The grey arrow on the right of both pictures links together two blocks on different layers. This is the result of the cyclic transition model that the packer has. Finally my graph does not present information on the type of API calls because, as previously said, it was not relevant for the final purpose of my thesis.

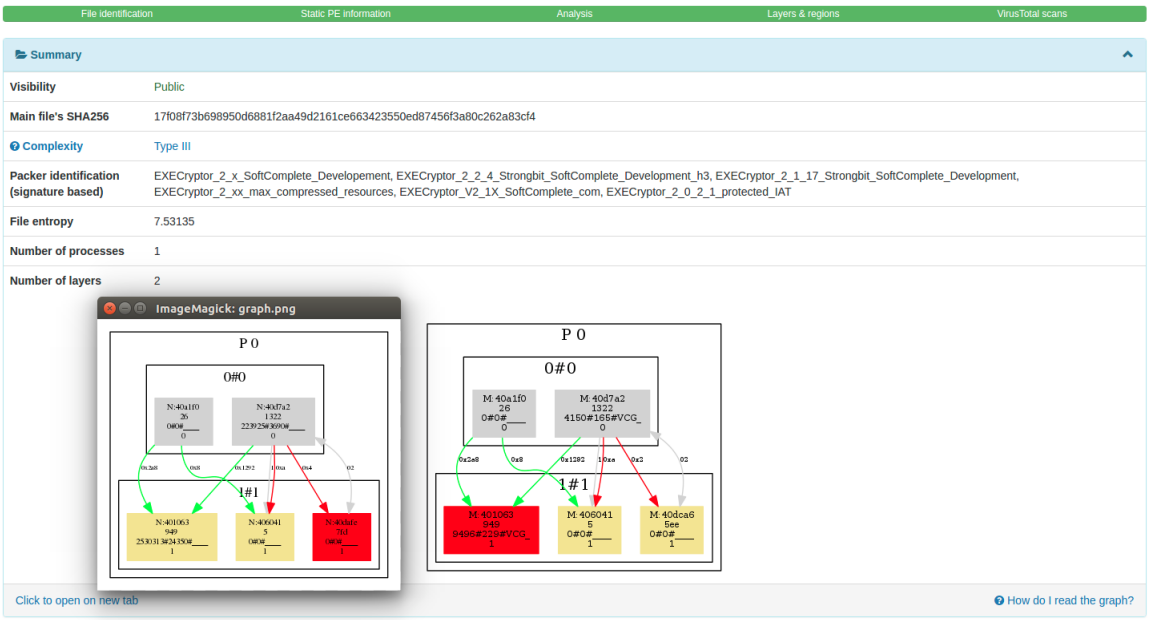


Figure 6.6: Example 1. Comparison between my implementation (left) and the SoA (right)

6.3.2 Example 2

Figure 6.7 shows a similar result in terms of accuracy. This time I used an actual malware which was packed with an unknown packer (md5: cecf3c6e7139985101e181a235e90aea, it can be found on virustotal.com [19]). In this case we can see that there are two layers of unpacked code (the middle and the bottom layer) and two layers of unpacking code (the top and the middle layer). This means that the executable started with the first unpacking stub, which unpacked the first layer (*1#1*), which in a second time wrote itself into memory where the execution finally jumped, generating the second (and final) unpacked layer (where the execution finished, given the red box). As in the previous example the packer has a cyclic transition model. This time we can see from the information on top of the graphs something unusual for a packed executable, i.e. that the

entropy of this particular malware was below 7.0. This is something unusual, in fact the entropy of a packed executable typically ranges between 7.0 and 8.0, since the original code is obfuscated as much as possible (generating thus a lot of randomness with few redundancies, which means high entropy). By looking at the graphs it is possible to deduce that the two executions that the malware performed (one in my framework and the other on the web service) were pretty much the same. One thing that is different between the two graphs is that layer 1 was mapped on different memory areas, as we can see from the first line of information in the grey box (N:<address>). This could be the consequence of a normal memory allocation for that region of code during the execution. It is in fact common that normal *malloc*s are done without specifying the destination address of the memory allocation in the heap. As in the first example there is no information (in the graph resulting from my implementation) about neither the API calls nor the types of the memory regions (Module, Stack, Heap, None) and in this case the web service was not able to determine the packer family based on a signature.

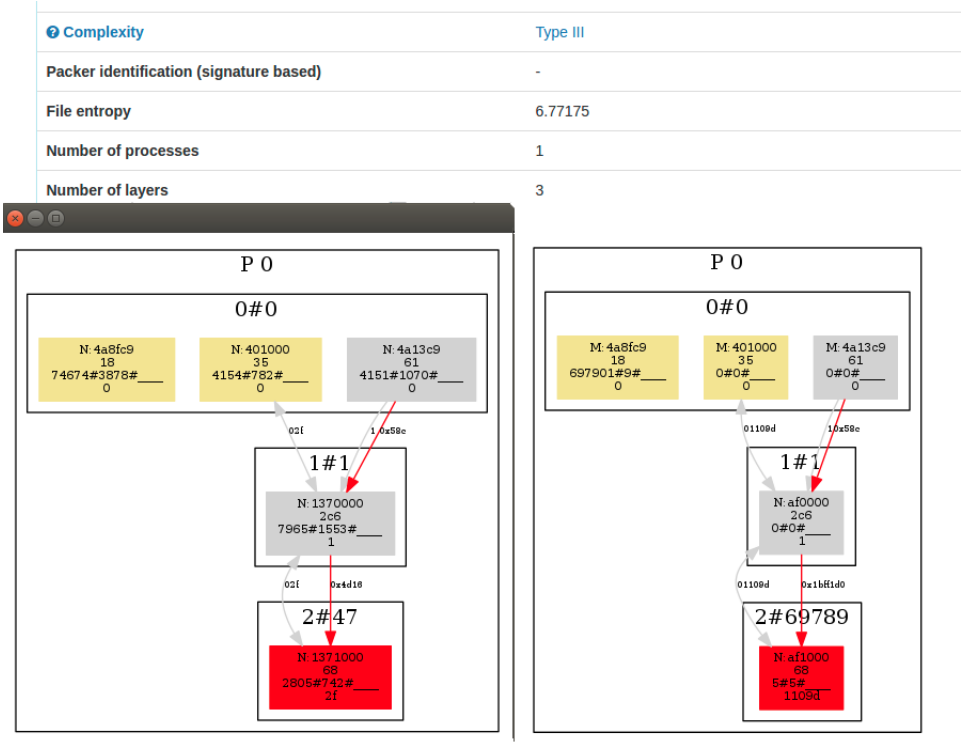
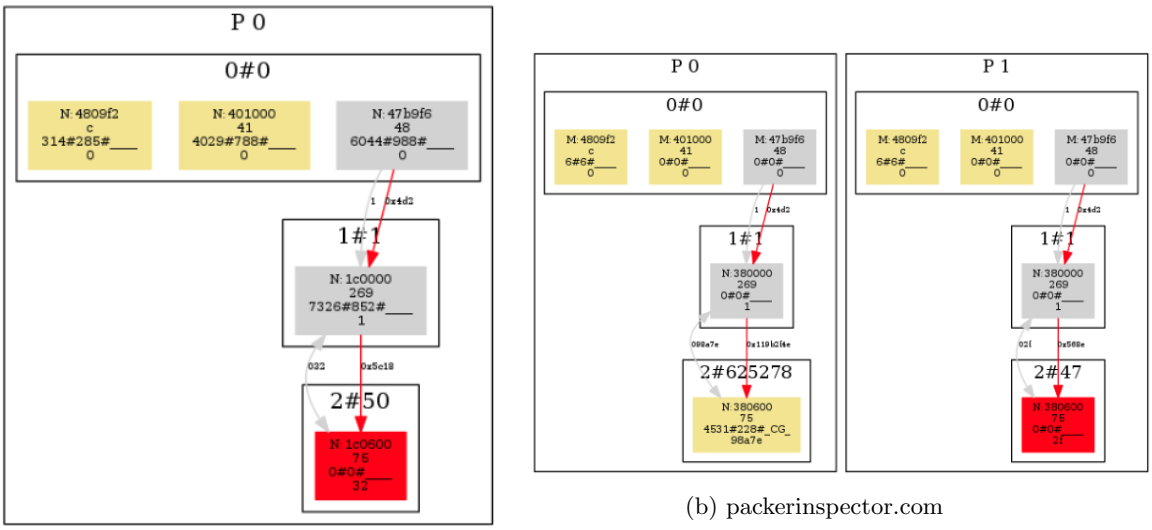


Figure 6.7: Example 2. Comparison between my implementation (left) and the SoA (right)

6.3.3 Example 3

Figure 6.8 shows instead a scenario in which there were some differences between the two executions of the sample. Like in the previous example I used a packed malware (md5: c63bb9913158e8afc4cc6-80e02a027de on virustotal.com [20]) with entropy equals to 7.78. As we can see from the figure, the web service produced the graph in 6.8b with two distinct processes that had the same behaviour: they have the same structure and number of layers (the top one containing the original code, the other two containing the unpacked code), both processes wrote the same memory regions. They look like two copies of the same process. On the left side of the figure (6.8a) is the result of my implementation of the framework. The main difference with 6.8b is that there is only a single process. This is due to a different behaviour of the execution of the sample when the recording was taken. It could depend on some logic bombs inside the malware that were triggered by PANDA, or it could simply mean that one process did not start before the end of the recording. I wanted to stress out that situations like this one are quite common, because they fall into the big category of those problems related to the dynamic analysis of an hostile executable, which will do anything it can to prevent us from understanding its behaviour.



(a) My implementation

(b) packerinspector.com

Figure 6.8: Example 3. Differences between the two executions

Chapter 7

How-to

In this chapter I will explain in few words how to install my implementation of the plugin and how to execute the PANDA replays. PANDA replays can be recorded in the way explained in chapter 2.4.1. First of all we need to get access to the GitHub repository on which I worked during the six months of internship, so:

```
git clone https://github.com/SamAicardi/My-Panda.git panda
```

This command will download the whole PANDA repository with all the modifications that I did in order to make everything work. The downloaded directory will have three different branches:

- *master*. This is the main branch, where we can find the version of the plugin designed to analyse Windows executables.
- *linux*. In this branch we can find the implementation of the plugin that supports the analysis of Linux executables for the x86 architecture.
- *arm*. In this branch we can find the implementation of the plugin that supports the analysis of Linux executables for the ARM architecture.

To change from one implementation to the other:

```
cd panda/
git checkout <branch>
```

where `<branch>` can assume one value among *master*, *linux* and *arm*. To see which branch is the current one:

```
git branch
```

It will show the list of all the branches that we visited so far, highlighting the current one as in figure 7.1.

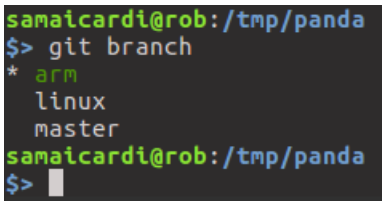


Figure 7.1: There are three different branches

The implementation of the framework as a PANDA plugin can be found under `./panda/plugins/packer_insp`
At this point it is possible to configure the plugin itself to be able then to compile the whole PANDA repository:

```
cd <panda_folder>/panda/plugins/packer_inspector
./configure --panda-path=<panda_folder>
```

where <panda_folder> is the absolute path to where PANDA has been downloaded. Then we need to open the Makefile and change DPI_HOMEDIR in the first line:

```
DPI_HOMEDIR=<panda_folder>/panda/plugins/packer_inspector
```

Finally we just have to build the whole PANDA project as any other project:

```
cd <panda_folder>
./build.sh
```

Once we have built the PANDA repository we can just use the newly-compiled plugin to analyse a normal replay. There are three different ways to execute the packer complexity analysis, one for each version of the plugin. Appendix D explain in details what the three different methods do in order to accomplish to the results. To summarise the options in few words, we can see the three different commands that will be called by each script in the different scenarios. To run Windows replays we can use the following command:

```
$> <path_to_panda>/i386-softmmu/qemu-system-i386 -replay <replay_name> -panda \
   syscalls2:profile=windows7_x86 -panda packer_inspector:name=<exe_name>,os=win \
   -os windows-32-7 -m <size>
```

To run Linux x86 replays we can use the following command:

```
$> <path_to_panda>/i386-softmmu/qemu-system-i386 -replay <replay_name> -panda \
   syscalls2:profile=linux_x86 -panda osi -panda osi_linux:kconf_file=\
   <path_to_panda>/panda/plugins/osi_linux/kernelinfo.conf,kconf_group=\
   my_debian_i386 -panda packer_inspector:name=<exe_name>,os=linux \
   -os linux-32-* -m <size>
```

To run Linux ARM replays we can use the following command:

```
$> <path_to_panda>/arm-softmmu/qemu-system-arm -M versatilepb -kernel \
   <path_to_images>/vmlinuz-3.2.0-4-versatile -initrd \
   <path_to_images>/.panda/initrd.img-3.2.0-4-versatile -hda \
   <path_to_images>/.panda/debian_wheezy_armel_standard.qcow2 -append \
   'root=/dev/sda1' -panda syscalls2:profile=linux_arm -panda osi -panda \
   osi_linux:kconf_file=<path_to_panda>/panda/plugins/osi_linux/kernelinfo.conf\
   ,kconf_group=my_debian_arm -panda packer_inspector:name=<exe_name>,os=linux \
   -os linux-32-* -replay <replay_name> -m <size>
```

For each of the three commands listed above:

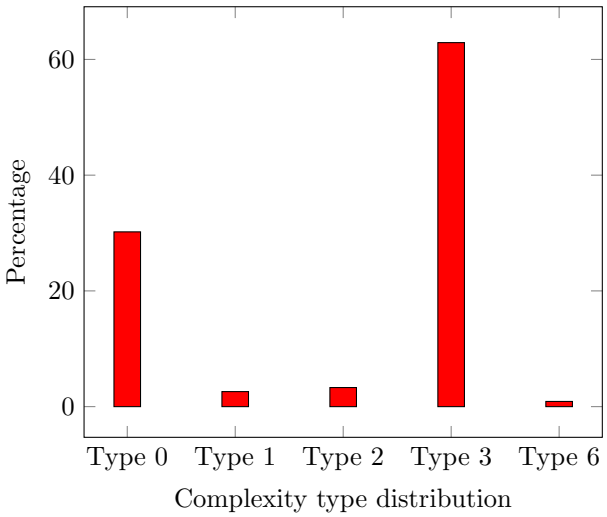
- <path_to_panda> is the root directory for the PANDA repository. It is where we decided to clone my version of the plugin
- <replay_name> is the prefix name of the files that are necessary to run a replay, i.e. the memory snapshot and the non-deterministic inputs log file
- <exe_name> is the name of the executable, either for Linux or Windows, that was started in the emulated guest operating system
- <size> corresponds to how much space we want to assign to the RAM memory of the emulated OS. It must be expressed either in MB or in GB
- <path_to_images> is the absolute path to the folder where we stored the *qcow2* images that are required to emulate the guest operating system

Chapter 8

Conclusions

During the six months I spent working on this thesis I had two main tasks: as a first step I needed port the complexity classification of runtime packers from DECAF to PANDA in order to get the access to the thousands of malware executions that other researchers already recorded with PANDA. The second main task was to extend the applicability of the runtime packer complexity classification to other operating systems and other architectures (with respect to Windows XP PE executables).

As a result of the six months of work I was able to port the framework as a working PANDA plugin, being able to test over ten thousands of malware recordings taken from Dolan-Gavitt's collection. I also extended the applicability of the analysis to ARM machines and to Linux x86 elf executables. With Dolan-Gavitt's collection of malware samples it was possible to observe how runtime packers are used in the wild and what is the usage distribution in terms of packing complexity. Since it is typically easier to find malware designed for Windows with respect to Linux malwares (and also there are way more packer families for Windows rather than for Linux) this was the fastest way to draw some statistics on a huge test set.



By looking at the results of these tests we can see that the most used runtime packer was a Type 3 packer (multiple unpacking layers, cyclic transition model, typically single process, tail transition, for more details see Section 1.4). It is a confirmation that Type 3 runtime packers are the optimal trade-off between complexity and performance. The more complex the packer is (i.e. the more unpacking layers and/or processes it has) the better is the achieved obfuscation and the harder is for an automatic scanner to detect potentially malicious hidden code, but it is also true that the more complex the packer is the more onerous the unpacking routine will be. As a consequence, the execution of a malware packed with a very complex runtime packer (e.g. a Type 6 packer) may not be the best thing for the author of the malware, because the malware

may become so slow in unpacking itself that an anti-virus can detect it even before the unpacking routine is finished. Type 3 packers seem to be a good compromise between unpacking effectiveness and unpacking efficiency.

For what concerns the analysis of Linux runtime packers, there was no such a big collection of samples, so I did some preliminary tests by analysing a series of sample programs (packed with UPX) which did not have malicious code. After this initial phase I tested a small set of 32bit elf real malwares. There were 788 Linux x86 executables that I tested using the Automated Testing Framework. The samples were already labelled as either *packed* or *not_packed* so that I could then check the accuracy of the ATF. It produced 868 results (some of the samples have been executed twice because the first run just crashed for some reasons), out of which I got the following statistics:

- 589 finished the analysis, producing a final report as explained in Chapter 3
- 85 aborted because of some errors during the first phase (the dynamic analysis)
- 83 produced an error message in the second phase of the analysis
- 111 did not finish for other reasons

The total number of samples that ended up being classified as *packed* was 58, with respect to 63, which is the total number of malwares that were present inside the 788 tested samples. The five samples that are missing from the packed list were not recognised because it was not possible to dynamically analyse them, as the executable files ended up being corrupted (they prompted a "bus error" message). The only way to spot their maliciousness was to manually analyse each one of them statically, by looking at their disassembled code.

I used then a similar process for the analysis of Linux 32bit ARM executables. I tested 738 samples, where 45 of them were already classified as *packed* malwares. Here is the need of pointing out that the record&replay platform in PANDA for ARM images is not a hundred percent reliable. There were in fact a lot of samples whose recordings simply did not work for some reasons. This is why I needed to repeat, for many of them, the execution in the guest OS multiple times just to be sure that eventually I would have got at least one working recording. The ATF produced in this scenario 1960 results, out of which I got the following statistics:

- 430 samples finished and produced a proper analysis
- All the other recordings just did not finish for the reasons I explained before

In this case there were almost half of the packed samples (20 out of 45) that could not be executed because of some problems at runtime: either they produced a segmentation fault, or they had illegal instructions, or again they reported a bus error. In the small group of packed malwares that the framework was able to execute 21 out of 25 were correctly detected as *packed*.

Further Work

As I have already explained in the previous sections, my implementation of the framework is not as complete as the one available at packerinspector.com. I decided to skip all those parts of the code that were implementing the heuristics to determine the packer family: the collection of API calls, the separation between different types of memory (i.e. stack, heap or modules), imported and exported symbols, because I preferred to focus on implementing what was actually necessary to compute the complexity of runtime packers. A further step towards the complete realization of the framework in PANDA can be achieved by implementing those features.

As the plugin is implemented up to now, it would be quite easy to extend the support the 64bit versions of both Linux and Windows executables. I could not extend the support to the x86_64 architecture because there were some missing PANDA plugins, specifically the *syscalls2* plugin did not support neither Linux nor Windows 64bit replays. The *syscalls2* plugin is fundamental because it implements all the hooks that the user can register on system calls (and thus on API

calls), and up to now it does not offer the support for 64bit system calls. It would be quite easy to extend this support by looking at the implementation of the 32bit version of the same system calls that can be found in the PANDA man page[1].

Another improvement could be trying to extend the support of the analysis to another architecture. We have already taken into account the three main architectures, x86, x86_64 and ARM. By looking at the PANDA man page: "PANDA support is reasonably strong only for x86, arm, and ppc" [2]. So the next target in terms of architectures may be Power PC, but further investigations would be required in order to be sure that the main PANDA plugins used for the complexity classification can extend the support to this new architecture.

[1] https://github.com/panda-re/panda/blob/master/panda/docs/syscalls2.md

[2] https://github.com/panda-re/panda/blob/master/panda/docs/manual.md#emulation-details

Appendix A

Results of the analysis

In this Appendix I will show how the file *results.txt* looks like. Just to remind it, this is the output file that is produced at the end of the analysis, when the runtime packer has already been monitored and when the behavioural graph has already been produced.

Figure A.1 shows the output graph of a runtime packer called *UnPackMe_Obsidium*. As we can see there are multiple layers of unpacking code, starting from the top layer, which contains the first piece of code that has been executed at runtime.

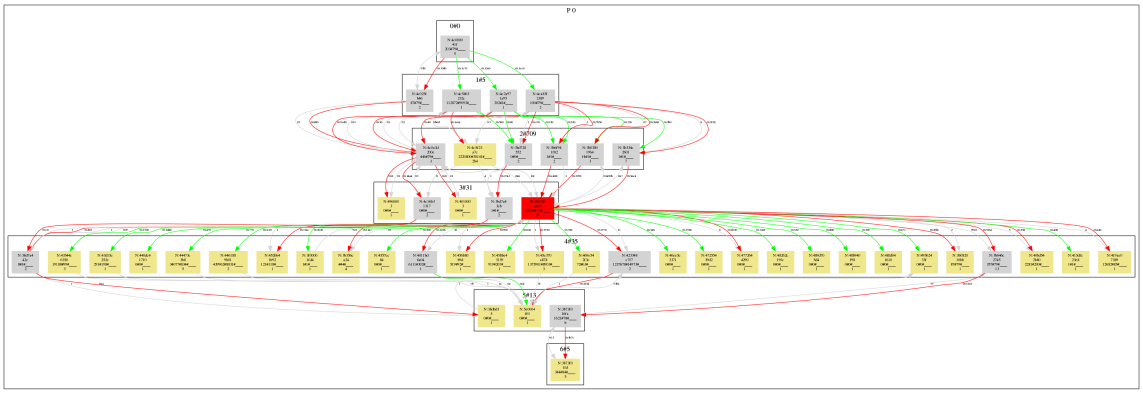


Figure A.1: Behavioural graph of the unpacking routine for UnPackMe_Obsidium v. 1.4.0.9

results.txt

```
GENERAL FEATURES
================

Transition model: cyclic
Footprint:
Original code located in last level: True
Number of regions with special API calls: 0
Granularity:
Layer with original code: None
COMPLETITY TYPE: 3
Total analysis (execution) time: 0.0
Number of instructions decoded: 0
Number of exceptions: 0
Number of processes: 1
Number of upward transitions: 94916
Number of downward transitions: 92479
Number of processes with interprocess communication: 0
Termination reason: Process exit


LEVEL STATISTICS
```

```
================


Level number 0
---------------
Number of regions: 1
Number of frames: 0
Min. BB size: 2
Avg. BB size: 6
Max. BB size: 85

    Region number 0
    -----------------
    Region address 4c0000
    Region size 41f
    Region Type N
    Region nb frames 0
    Region has been modified remotely: 0
    Region total API calls: 203
    Region different API calls: 79
    Region sp_v: False
    Region sp_c: False
    Region sp_g: False

Level number 1
---------------
Number of regions: 4
Number of frames: 6
Min. BB size: 1
Avg. BB size: 13
Max. BB size: 165

    Region number 0
    -----------------
    Region address 4c025f
    Region size b46
    Region Type N
    Region nb frames 2
    Region has been modified remotely: 0
    Region total API calls: 87
    Region different API calls: 79
    Region sp_v: False
    Region sp_c: False
    Region sp_g: False

    Region number 1
    -----------------
    Region address 4c2a97
    Region size 1a95
    Region Type N
    Region nb frames 1
    Region has been modified remotely: 0
    Region total API calls: 202
    Region different API calls: 1
    Region sp_v: False
    Region sp_c: False
    Region sp_g: False

    Region number 2
    -----------------
    Region address 4c5863
    Region size 232c
    Region Type N
    Region nb frames 1
    Region has been modified remotely: 0
    Region total API calls: 112872
    Region different API calls: 9092
    Region sp_v: False
    Region sp_c: False
    Region sp_g: False

    Region number 3
```

```
    -----------------
    Region address 4ca83f
    Region size 2389
    Region Type N
    Region nb frames 2
    Region has been modified remotely: 0
    Region total API calls: 100
    Region different API calls: 79
    Region sp_v: False
    Region sp_c: False
    Region sp_g: False
```

...

results.txt

```
Level number 5
---------------
Number of regions: 3
Number of frames: 13
Min. BB size: 2
Avg. BB size: 6
Max. BB size: 53

    Region number 0
    -----------------
    Region address 3b73f0
    Region size 16fa
    Region Type N
    Region nb frames 11
    Region has been modified remotely: 0
    Region total API calls: 1621
    Region different API calls: 78
    Region sp_v: False
    Region sp_c: False
    Region sp_g: False

    Region number 1
    -----------------
    Region address 3bdbd3
    Region size 5
    Region Type N
    Region nb frames 1
    Region has been modified remotely: 0
    Region total API calls: 0
    Region different API calls: 0
    Region sp_v: False
    Region sp_c: False
    Region sp_g: False

    Region number 2
    -----------------
    Region address 5e0004
    Region size ff0
    Region Type N
    Region nb frames 1
    Region has been modified remotely: 0
    Region total API calls: 0
    Region different API calls: 0
    Region sp_v: False
    Region sp_c: False
    Region sp_g: False

Level number 6
---------------
Number of regions: 1
Number of frames: 5
Min. BB size: 1
Avg. BB size: 4
Max. BB size: 20
```

```
    Region number 0
    -----------------
    Region address 3b73f0
    Region size 1fd
    Region Type N
    Region nb frames 5
    Region has been modified remotely: 0
    Region total API calls: 344
    Region different API calls: 84
    Region sp_v: False
    Region sp_c: False
    Region sp_g: False


POTENTIALLY ORIGINAL REGIONS
============================



MORE GENERAL STATISTICS
=======================

Number of layers with multiple frames: 6
...out of 7 layers
...that have 49 regions together
...of which 0 have calls to special APIs
Last executed region: Proc: 0 Level: 3 Number: 0 Address 3b0380 Size: a465


INFO ABOUT MONITORED PROCS
==========================

PID: 430
Position: 0
Loaded modules:
---------------
    No_name - 6d0000(281000)


REMOTE MEMORY WRITE INFORMATION
===============================
```

At the end of the file we can find other information that summarize the content of the analysis (e.g. the number of layers with multiple frames, the total number of layers), the Process Identifier (PID), the list of loaded modules (if any) and also the remote memory write statistics (in this case there were no remote memory writes as there was only one process). There are also some missing information, such as the potentially original regions or the runtime packer footprint (at the very beginning of the file) because they are part of the heuristics to understand and try to guess the packer family, which did not succeed in this case.

Appendix B

Simplify the Execution

Listing B.1 shows some utilities that will be called by the three main functions, plus few data structures (Python dictionaries) to store global values. `keymap` is a dictionary that contains associations between ASCII characters and their representation in the qemu-monitor. In this way it will be possible for the script to send keystrokes to the guest operating system, using the qemu-monitor command `sendkey <character>`. `arch_set` and `src_port_base` are other two dictionaries that just keep track of the guest operating system that has been selected by the corresponding command line argument. Function `mon_cmd` is used to send any command to the qemu-moniror, while `guest_type` is used for the communication between the qemu-monitor and the guest operating system. `handler`, `SnoozeAlarm` and `snoozealarm` represent the implementation of a timer, that will be used by the function `perform_analysis` in the second phase of the execution. The timer is implemented through a signal handler and a daemon that will send the *SIGALRM* signal to the signal handler. `download_log` is a function that can be called if the user wants to download a specific file from the guest to the OS. It is basically a wrapper of the bash command `scp`, handling all the possible exceptions that it may throw. `md5_for_file` is a function that computes the md5 hash of a target file.

B.1 run_target_mal.py

Listing B.1: "Helper functions"

```
#!/usr/bin/env python

import ConfigParser
import logging
import os
import shutil
import socket
import subprocess
import sys
import telnetlib
import time
import hashlib
import samples
import pexpect
import signal
import threading
import logging
import time
import string

# key mapping for the guest-host interaction
keymap = {
    '-': 'minus',
    '=': 'equal',
    '[': 'bracket_left',
```

```
    ']': 'bracket_right',
    ';': 'semicolon',
    '\'': 'apostrophe',
    '\\': 'backslash',
    ',': 'comma',
    '.': 'dot',
    '/': 'slash',
    '*': 'asterisk',
    ' ': 'spc',
    '_': 'shift-minus',
    '+': 'shift-equal',
    '{': 'shift-bracket_left',
    '}': 'shift-bracket_right',
    ':': 'shift-semicolon',
    '"': 'shift-apostrophe',
    '|': 'shift-backslash',
    '<': 'shift-comma',
    '>': 'shift-dot',
    '?': 'shift-slash',
    '\n': 'ret',
}

arch_set = ['i386', 'x86_64', 'arm']

scp_port_base = {
    'i386' : 11022,
    'x86_64' : 12022,
    'arm' : 10022
}

# some global variables for debugging
CLEAN_ALREADY_ANALYZED_SAMPLES = 0 # not used for the moment
PARTIAL_ANALYSIS = 1
# if PARTIAL_ANALYSIS == 0 but we still want to log everything
LOG_ANYWAYS = 1
DELETE_SNAPHOTS = 1

# to enable writing to a logfile
LOG_DEBUG = 0

global_start_time = 0
global_var_child = 0

########################################################################
#               Interaction with the guest OS                          #
########################################################################
def mon_cmd(s, mon):
    mon.write(s)
    logging.info(mon.read_until("(qemu)"))

def guest_type(s, mon):
    for c in s:
        if c in string.ascii_uppercase:
            key = 'shift-' + c.lower()
        else:
            key = keymap.get(c, c)
        mon_cmd('sendkey {0}\n'.format(key), mon)
        time.sleep(.1)

def handler(signo, frame):
    print "Timeout exceeded, sending SIGTERM (elapsed time: {0})".format(time.time() -
     global_start_time)
    global global_var_child
    global_var_child.kill(signal.SIGTERM)
    sys.stdout.flush()


########################################################################
#                              Timer                                   #
########################################################################
class SnoozeAlarm(threading.Thread):
    def __init__(self, zzz):
```

```
        threading.Thread.__init__(self)
        self.setDaemon(True)
        self.zzz = zzz

    def run(self):
        time.sleep(self.zzz)
        os.kill(os.getpid(), signal.SIGALRM)

def snoozealarm(i):
      SnoozeAlarm(i).start()


######################################################################
#                              Utils                                 #
######################################################################
# download some files from the guest OS
def download_log(srcpath, dstpath, port):
    try:
        var_password  = "user"
        var_command = "scp -P {2} user@localhost:{0} {1}".format(srcpath, dstpath, port)

        var_child = pexpect.spawn(var_command)
        i = var_child.expect(["password:", '(yes/no)? ', pexpect.EOF])

        if i==0: # send password
            var_child.sendline(var_password)
            var_child.expect(pexpect.EOF)
        elif i==1: # send yes
            var_child.sendline('yes')
            i = var_child.expect(["password:", pexpect.EOF])
            if i==0: # send password
                var_child.sendline(var_password)
                var_child.expect(pexpect.EOF)
            elif i==1:
                print "Got the key or connection timeout"
                pass
        elif i==2:
            print "Got the key or connection timeout"
            pass
    except Exception as e:
        print "Oops Something went wrong buddy"
        print e


def md5_for_file(fname, block_size=2**20):
    f = open(fname, 'rb')
    md5 = hashlib.md5()
    while True:
        data = f.read(block_size)
        if not data:
            break
        md5.update(data)
    digest = md5.hexdigest()
    f.close()
    return digest
```

Listing B.1 shows the first function that is called by `main`. It is in charge of recording the execution of the target sample inside the guest OS. It supports three different architectures that can be emulated with panda-qemu: *x86*, *x86_64* and *arm*. According to the command line arguments that the script receives the corresponding combination of architecture and operating system will be emulated. The command line arguments that need to be passed when launching panda-qemu also differ from version to version. To cope with that, some global and local structures (Python dictionaries) are used to keep track of the selected OS/architecture, so that it is easy to choose the right parameters. `run_sample` will produce the PANDA recording files (i.e. the initial memory snapshot and the non-deterministic input log file), that will be used by the second function, `perform_analysis`.

Listing B.2: "Run sample"

```
######################################################################
#                              Steps                                 #
######################################################################
def run_sample(filename, sample_dir, conf, arch, vm_folder):

    # Setup from config
    instance = 0
    monitor_port = 1234 + instance
    basedir = conf.get('Main', 'basedir')
    qcow_dir = conf.get('Main', 'images')
    exec_time = int(conf.get('VM', 'exec_time'))
    panda_exe = os.path.join(conf.get('Main', 'panda'), '{0}-softmmu'.format(arch), 'qemu-system
    -{0}'.format(arch))

    basedir = conf.get('Main', 'basedir')
    sample_name = filename + '_' + time.strftime('%Y%m%d.%H.%M.%S')
    logfile = sample_name

    if LOG_DEBUG:
        logger = logging.getLogger()
        logger.handlers[0].stream.close()
        logger.removeHandler(logger.handlers[0])

        file_handler = logging.FileHandler("{0}.log".format(logfile))
        file_handler.setLevel(logging.DEBUG)
        formatter = logging.Formatter('%(asctime)s %(levelname)s %(message)s')
        file_handler.setFormatter(formatter)
        logger.addHandler(file_handler)

        # Startup msgs
        logging.info("Config file: {0}".format(sys.argv[2]))
        #logging.info("UUID: {0}".format(run_id))
        logging.info("Sample: {0}".format(filename))


    arch_params = {
        'i386': ['-hda', '{0}/debian_wheezy_i386_standard.qcow2'.format(qcow_dir), '-m', '1G',
                 '-loadvm', '4', '-net', 'user,hostfwd=tcp::{0}-:22'.format(scp_port_base[arch]+
    instance),
                 '-net', 'nic'
                 #,'-nographic' # if there is no need to see the guest OS
                 ],
        'x86_64': ['-hda', '{0}/debian_wheezy_x86_64_standard.qcow2'.format(qcow_dir), '-m', '1G'
    ,
                 '-net', 'user,hostfwd=tcp::{0}-:22'.format(scp_port_base[arch]+instance), '-net'
    , 'nic', '-loadvm', '3'
                 #,'-nographic' # if there is no need to see the guest OS
                 ],
        'arm': ['-M', 'versatilepb', '-kernel', '{0}/vmlinuz-3.2.0-4-versatile'.format(qcow_dir),
                '-initrd', '{0}/initrd.img-3.2.0-4-versatile'.format(qcow_dir), '-hda',
                '{0}/debian_wheezy_armel_standard.qcow2'.format(qcow_dir), '-append',
                '"root=/dev/sda1"', '-loadvm', '7',
                '-net', 'user,hostfwd=tcp::{0}-:22'.format(scp_port_base[arch]+instance), '-net',
     'nic'
                 #,'-nographic' # if there is no need to see the guest OS
                 ]
    }

    if LOG_DEBUG:
        # compute sample md5
        sample_md5 = md5_for_file(filename)
        logging.info("MD5: {0}".format(sample_md5))


    panda_args = []
    panda_args.append(panda_exe)
    for el in arch_params[arch]:
        panda_args.append(el)
    panda_args.append('-monitor')
    panda_args.append('telnet:localhost:{0},server,nowait'.format(monitor_port))
```

```
    # Start the QEMU process
    panda_stdout_file = '{0}.stdout'.format(sample_name)
    panda_stdout = open(panda_stdout_file, 'w')
    panda_stderr_file = '{0}.stderr'.format(sample_name)
    panda_stderr = open(panda_stderr_file, 'w')
    panda = subprocess.Popen(panda_args, stdin=subprocess.PIPE, stdout=panda_stdout, stderr=
     panda_stderr)

    # Connect to the monitor
    # Give it time to come up...
    tries = 10
    mon = None
    for i in range(tries):
        try:
            if LOG_DEBUG:
                logging.info('Connecting to monitor, try {0}/{1}'.format(i, tries))
            mon = telnetlib.Telnet('localhost', monitor_port)
            break
        except socket.error:
            time.sleep(1)

    if not mon:
        if LOG_DEBUG:
            logging.error("Couldn't connect to monitor on port {0}".format(monitor_port))
        sys.exit(1)
    else:
        if LOG_DEBUG:
            logging.info("Successfully connected to monitor on port {0}".format(monitor_port))

    # Wait for prompt
    mon.read_until("(qemu)")

    ##########################################################################
    #                                    arm                                 #
    ##########################################################################
    # NOTE: for arm it seems necessary to record the same execution twice
    # with the same qemu session. the first attempt rarely works.
    # it needs to be mutually exclusive with i386
    if arch == 'arm':
        # Write the sample name
        if LOG_DEBUG:
            logging.info("Writing sample name.")
        guest_type(os.path.join(vm_folder, filename), mon)
        #guest_type("./{0}".format(filename), mon)
        #guest_type("strace ./{0} 2>{0}_strace_stdout.trace".format(filename), mon)

        # Begin the record
        if LOG_DEBUG:
            logging.info("Beginning record.")
        #record_name = os.path.join('./', 'recs', sample_dir, '{0}_debian_{1}'.format(arch,
     sample_name))
        mon_cmd("begin_record {0}\n".format(logfile), mon)

        # Run the sample
        if LOG_DEBUG:
            logging.info("Starting sample.")
        guest_type("\n", mon);

        # Wait
        if LOG_DEBUG:
            logging.info("Sleeping for {0} seconds...".format(exec_time))
        time.sleep(exec_time)

        # End the record
        if LOG_DEBUG:
            logging.info("Ending record.")
        mon_cmd("end_record\n", mon)

        #-------------------

        # now do it again
        # Recall sample name
```

```
        if LOG_DEBUG:
            logging.info("Recalling sample name.")
        send_ctrl_c(mon)
        send_ctrl_c(mon)
        send_ctrl_c(mon)
        send_up(mon)

        # Begin the record
        if LOG_DEBUG:
            logging.info("Beginning record.")
        mon_cmd("begin_record {0}\n".format(logfile), mon)

        # Run the sample
        if LOG_DEBUG:
            logging.info("Starting sample.")
        guest_type("\n", mon);

        # Wait
        if LOG_DEBUG:
            logging.info("Sleeping for {0} seconds...".format(exec_time))
        time.sleep(exec_time)

        # End the record
        if LOG_DEBUG:
            logging.info("Ending record.")
        mon_cmd("end_record\n", mon)


    ########################################################################
    #                                  i386                                #
    ########################################################################
    if arch == 'i386':
        # Write the sample name
        if LOG_DEBUG:
            logging.info("Writing sample name.")
        guest_type(os.path.join(vm_folder, filename), mon)
        #guest_type("strace ./{0} 2>{0}_strace_stdout.trace".format(filename), mon)

        # Begin the record
        if LOG_DEBUG:
            logging.info("Beginning record.")
        #record_name = os.path.join('./', 'recs', sample_dir, '{0}_debian_{1}'.format(arch,
     sample_name))
        mon_cmd("begin_record {0}\n".format(logfile), mon)

        # Run the sample
        if LOG_DEBUG:
            logging.info("Starting sample.")
        guest_type("\n", mon);

        # Wait
        if LOG_DEBUG:
            logging.info("Sleeping for {0} seconds...".format(exec_time))
        time.sleep(exec_time)

        # just to be sure that the program stops
        send_ctrl_c(mon)
        send_ctrl_c(mon)
        send_ctrl_c(mon)

        # End the record
        if LOG_DEBUG:
            logging.info("Ending record.")
        mon_cmd("end_record\n", mon)

    ########################################################################
    #                                x86_64                                #
    ########################################################################
    # it needs to be mutually exclusive with arm
    if arch == 'x86_64':
        # chmod u+x <sample_name>
        if LOG_DEBUG:
```

```
            logging.info("chmod u+x <sample_name>")
        guest_type("chmod u+x {0}\n".format(os.path.join(vm_folder, filename)), mon)

        # Write the sample name
        if LOG_DEBUG:
            logging.info("Writing sample name.")
        guest_type(os.path.join(vm_folder, filename), mon)
        #guest_type("strace ./{0} 2>{0}_strace_stdout.trace".format(filename), mon)

        # Begin the record
        if LOG_DEBUG:
            logging.info("Beginning record.")
        #record_name = os.path.join('./', 'recs', sample_dir, '{0}_debian_{1}'.format(arch,
    sample_name))
        mon_cmd("begin_record {0}\n".format(logfile), mon)

        # Run the sample
        if LOG_DEBUG:
            logging.info("Starting sample.")
        guest_type("\n", mon);

        # Wait
        if LOG_DEBUG:
            logging.info("Sleeping for {0} seconds...".format(exec_time))
        time.sleep(exec_time)

        # just to be sure that the program stops
        send_ctrl_c(mon)
        send_ctrl_c(mon)
        send_ctrl_c(mon)

        # End the record
        if LOG_DEBUG:
            logging.info("Ending record.")
        mon_cmd("end_record\n", mon)

    if LOG_DEBUG:
        logging.info("Quitting PANDA.")
    mon.write("q\n")

    #print "closing files.."
    panda_stderr.close()
    panda_stdout.close()

    os.remove(panda_stderr_file)
    os.remove(panda_stdout_file)
    #os.remove(logfile)

    return logfile
```

Listing B.1 shows the content of the second function that `main` calls during the main loop, `perform_analysis`. It is in charge of analysing the output files that have been produced in the first phase to produce the intermediate results for the analysis, i.e. *execution_model.log*, *events.log*, *functions.log* and *trace.log*. First of all `perform_analysis` moves those files in the right directory, checking also if those files exist or not. In case any of them is not present, the execution is terminated with an exception. Then it will start the dynamic analysis by replaying the recording with panda-qemu. This function is also in charge of setting a timer (which is handled by a signal handler) to notify when the execution reaches the maximum time threshold. If this happens, the execution is stopped and the function returns.

Listing B.3: "Perform analysis"

```
def perform_analysis(arch, record_name, filename, output_dir, elapsed_time, do_make, vm_folder):
    script_os = {
        'arm'   : 'arm',
        'i386'  : 'linux_32',
        'x86_64': 'linux_64'
    }
    arch_options = {
        'arm'   : '--null',
```

```
    'i386'  : "'-m 1G'",
    'x86_64': "'-m 1G'"
}
arch_setup = {
    'arm'   : '--null',
    'i386'  : '--null',
    'x86_64': '--null'
}


try:
    run_replay = os.path.join(conf.get('Main', 'panda'), 'panda/plugins/packer_inspector', '
 run_replay_{0}.sh'.format(script_os[arch]))
    exe_name = filename[:15] # in the replays program names are truncated to the first 15
 chars
    options = arch_options[arch]
    setup = arch_setup[arch]
    folder = 'results_malware/test_{0}'.format(arch)
    # this is to stop execution_model.log even though all the processes didn't finish
    # i.e. stopping the program with ctrl-C
    exit_method = '--dirty'

    replay_output_log = os.path.join(sample_dir, '{0}.replay_log'.format(filename))

    var_command = run_replay +' '+ record_name +' '+ exe_name +' '+ options +' '+ setup +' '+
  folder +' '+ exit_method +' '+ do_make
    print '\t' + var_command
    sys.stdout.flush()

    replay_time = int(conf.get('VM', 'replay_time'))

    global global_var_child
    global_var_child = pexpect.spawn(var_command, timeout = replay_time)

    # ./run_replay_<arch>.sh > outfile
    outfile = file(replay_output_log, 'w')
    global_var_child.logfile = outfile

    # set an alarm equal to the 80% of the previous elapsed time
    if elapsed_time == 0:
        snoozealarm(replay_time)
    else:
        snoozealarm(elapsed_time)

    global global_start_time
    global_start_time = time.time()

    # start the script
    i = global_var_child.expect(["End of Plugin", pexpect.EOF, pexpect.TIMEOUT])

    if i==0: # send password
        # disable the timer
        signal.alarm(0)
        print "plugin finished"
        sys.stdout.flush()
        return 1
    elif i==1: # send yes
        # disable the timer
        signal.alarm(0)
        print "plugin stopped before finishing"
        sys.stdout.flush()
        return 0
    elif i==2:
        # disable the timer
        signal.alarm(0)
        print "\tReplay didn't finish after {0} seconds, sending SIGTERM".format(replay_time)
        #global_var_child.kill(signal.SIGTERM)
        #sys.exit(1)
        return 1
except Exception as e:
    print "Oops Something went wrong buddy"
    print e
```

```
        return 0

    return 0
```

Listing B.1 shows the last function that is called by `main`, `third_part`. It is in charge of parsing the intermediate output files to produce the behavioural graph, that shows how the packer unpacked the original code at runtime, and the unpacking complexity ranking on the six-value scale. When the output files are produced, it cleans up the environment and sets up all the files produced in the current run to be all in a single directory, in order to be as clean and easy as possible.

Listing B.4: "Third part"

```
def third_part(sample_name, arch, output_dir):
    makegraph_dir = conf.get('Main', 'makegraph_dir')
    results_dir = os.path.join(conf.get('Main', 'dpi_home'), 'results_malware/test_{0}'.format(
     arch))

    try:
        os.rename(os.path.join(results_dir, 'functions.log'), os.path.join(makegraph_dir, 'TRACE'
     , 'functions.log'))
    except Exception:
        print 'creating file {0}'.format(os.path.join(results_dir, 'functions.log'))
        with open(os.path.join(results_dir, 'functions.log'), 'w') as f:
            f.write('Functions for PID 000')
        os.rename(os.path.join(results_dir, 'functions.log'), os.path.join(makegraph_dir, 'TRACE'
     , 'functions.log'))

    try:
        os.rename(os.path.join(results_dir, 'events.log'), os.path.join(makegraph_dir, 'TRACE', '
     events.log'))
        os.rename(os.path.join(results_dir, 'execution_model.log'), os.path.join(makegraph_dir, '
     TRACE', 'execution_model.log'))
        os.rename(os.path.join(results_dir, 'trace.log'), os.path.join(makegraph_dir, 'TRACE', '
     trace.log'))
    except Exception as e:
        print e
        print "\t{0} didn't produce the correct results. Exiting.".format(sample_name)
        return 0

    script_name = os.path.join(conf.get('Main', 'makegraph_dir'), 'automated_graph_production.sh'
     )
    var_command = script_name +' '+ sample_name +' '+ arch

    var_child = pexpect.spawn(var_command, timeout = 30)
    var_child.logfile = sys.stdout

    # ./automated_graph_production.sh
    i = var_child.expect(["Done.", "Wrong.", pexpect.EOF])

    if i==0:
        print "\tScript finished"
        try:
            shutil.move(os.path.join(makegraph_dir, 'all{0}'.format(arch)), os.path.join(
     makegraph_dir, 'test_{0}'.format(arch)))
        except Exception:
            print 'ERROR moving files directory {0} to directory {1}'.format(os.path.join(
     makegraph_dir, 'all{0}'.format(arch)), os.path.join(makegraph_dir, 'test_{0}'.format(arch)))
        return 1
    elif i==1:
        print "\tWrong arguments"
        return 0
    elif i==2:
        print "\t[{0}] - Error".format(sample_name)
        return 0
```

Listing B.1 shows the starting point of the script. It is is charge of initialising the data structures which will be required in the three main functions that will be called during the execution. First of

all, it checks that each of the input parameters has been passed correctly to the script, then it opens some log files to keep track of the execution and it retrieves the global variables from *malrec.config*, which will be assigned to `config_file`. Then it comes the main loop, which is iterating over two variables: until everything went OK (`ok` is initialised as false) or we overcame the maximum number of iterations (set to 5 for this specific analysis). Inside the main loop there are three important function calls. The first one calls `run_sample`, which will start panda-qemu and will record the execution of the target sample. The second one calls `perform_analysis`, which will perform the actual dynamic analysis with PANDA, producing the intermediate outputs. The third one will call `third_part`, which will start the final part of the analysis, producing the behavioural graph and the complexity ranking, and setting up the output files in the right directory. The main loop is repeated exactly the same every iteration, except for the maximum time threshold for the second function call: it decreases from time to time according to how much time it took in the previous round (`elapsed_time * 0.8`). This is done in order to avoid some samples to crash unexpectedly because the recording lasted too much.

Listing B.5: "Main function"

```
if __name__ == '__main__':
    if len(sys.argv) < 6:
        print 'Usage: {0} <path/to/sample_name> <output_dir> <config_file> <arch> <vm_folder> [--
     make]'.format(sys.argv[0])
        sys.exit(1)

      if sys.argv[4] not in arch_set:
        print 'Error: <arch> not in arch_set'
        sys.exit(2)

    # rename all the arguments
    sample_name = os.path.basename(sys.argv[1])
    sample_dir = os.path.dirname(sys.argv[1])
    output_dir = sys.argv[2]
    config_file = sys.argv[3]
    arch = sys.argv[4]
    vm_folder = sys.argv[5]
    do_make = '--nomake'
    if len(sys.argv) == 7:
        if sys.argv[6] == "--make":
            do_make = sys.argv[6]

    # instantiate the event logger
    conf = ConfigParser.ConfigParser()
    conf.read(config_file)

    if LOG_DEBUG:
        # Init the logger
        logging.basicConfig(filename='/tmp/tmp', level=logging.DEBUG, format='%(asctime)s %(
     levelname)s %(message)s', filemode='w')

    # instantiate a signal handler for perform_analysis()
    signal.signal(signal.SIGALRM, handler)

    print "Executing sample: " + sample_name
    sys.stdout.flush()
    ok = 0
    count = 0
    max_count = int(conf.get('VM', 'repeat_replay'))
    elapsed_time = 0
    while not ok and count<max_count:
        # 1. record the execution
        record_name = run_sample(sample_name, sample_dir, conf, arch, vm_folder)
        time.sleep(2)

        # 2. try to perform the analysis with the plugin
        start_time = time.time()
        ok = perform_analysis(arch, record_name, sample_name, output_dir, (elapsed_time * 0.8),
     do_make, vm_folder)
        end_time = time.time()

        # 3. compute elapsed time
```

```
        elapsed_time = end_time - start_time
        print '\tElapsed time: '+str(elapsed_time)

        sys.stdout.flush()
        count += 1
        if DELETE_SNAPHOTS:
            print "\tDeleting snapshot and non-det log file of file {0} ..".format(record_name)
            os.remove(record_name+'-rr-snp')
            os.remove(record_name+'-rr-nondet.log')

    if count < max_count:
        print "\tReplay saved."

        # 4. go on with the third part only if the analysis was successful
        third_part(os.path.basename(record_name), arch, output_dir)
```

B.2 automated_graph_production.sh

Listing B.2 shows the bash script to automate the generation of the behavioural graph and the complexity ranking log file. They were initially generated during different steps, now they can be generated with a single script execution. *automated_graph_production.sh* receives the output files of the dynamic analysis as an input and it produces the graph and the complexity classification as an output. In the meantime it manages all the required directories that are touched during the execution, moving and deleting the appropriate files. The output graph is generated by calling `make test`, while the file containing the complexity ranking and other statistics can be obtained by the final call to `parseLogs.py`.

Listing B.6: "Automated graph production"

```
#!/bin/bash

set -e
set -o xtrace

if [[ -z $1 ]] ; then
    echo "usage: $0 <sample_name> <arch>"
    echo "Wrong."
    exit 1
fi

basedir=/home/samaicardi/deep_packer_inspector_makegraph
sample_name=$1
arch=$2

cd $basedir

make test > ./make_graph.out

cd $basedir/TRACE-output-cpp
dot -Tpng -ograph.png graph.dot
cd ..

mv TRACE/* TRACE-output-cpp/
mv ./make_graph.out TRACE-output-cpp/
mv TRACE-output-cpp $sample_name
rm -f TRACE/*
mv $sample_name ./all$arch/
cd ./all$arch/

# generate auto_start.log file since the analysis doesn't produce it anymore
echo -e '[FINISHED]\nReason: process exited' > ./$sample_name/auto_start.log

python ../post_processing/parseLogs.py ./$sample_name/ > $sample_name/results.txt

echo "Done."
```

Appendix C

Large Scale Analysis

Listing C shows the bash script that is in charge to start the large scale analysis. It basically calls `automated_win7_script.sh` redirecting its output both to standard output and to a *txt* file. It is wrapped in a *for loop* in case something crashes during execution.

Listing C.1: "Start analysis"

```
#!/bin/bash

for i in {0..200} ; do
    ls ./logs/rr/ | grep -v references | while read line; do rm -f ./logs/rr/$line ; done
    rm -f *.txz
    rm -f *.json
    ./automated_win7_script.sh | tee -a output_analysis.txt
done
```

Listing C shows the script that is called by `start_analysis.sh`. The first thing that it does is to prepare the list of the samples that it will need to analyse. To do so, the script compares the list of samples that are written in *panda_malware_list.txt* and it removes all the already executed samples that are stored in *already_executed_samples_<arch>.txt* (which starts empty). Then it downloads the *txz* sample from the database, unpacks it and creates the PANDA recording with *bpatch.py*

Listing C.2: "Initialization"

```
#!/bin/bash

set -e
#set -o xtrace

# save current working directory
current_wd=‘pwd‘


#cat panda_malware_list.txt | while read malware_name; do
cat already_analysed_samples_x86_64.txt already_analysed_samples_x86_64.txt panda_malware_list.
     txt | sort | uniq -u | while read malware_name; do

    echo "================================================================================"
    echo "             $malware_name"
    echo "================================================================================"

    echo $malware_name >> already_analysed_samples_x86_64.txt
#============================================
#             download sample
#============================================
    #echo http://panda.gtisc.gatech.edu/malrec/rr/$malware_name.txz
    wget http://panda.gtisc.gatech.edu/malrec/rr/$malware_name.txz


#============================================
```

```
#             generate snapshot
#============================================
    tar xvf $malware_name.txz
    bpatch.py logs/rr/$malware_name.patch


#============================================
#      launch replay and store results
#============================================
    wget http://panda.gtisc.gatech.edu/malrec/vt/$malware_name.json
    exe_name=$(cat `echo -n $malware_name.json` | egrep -o 'md5[^,]*,' | awk 'BEGIN{FS="\"}{print
      $3}')

    cd $current_wd
    ./execute_win7_malware_replay.py ~/results ./malrec_win7.config x86_64 $malware_name
     $exe_name

#============================================
#           delete not needed files
#============================================
    cd $current_wd
    rm $malware_name.json
    rm $malware_name.txz
    rm logs/rr/$malware_name.patch
    rm logs/rr/$(echo -n $malware_name)-rr-nondet.log
    rm logs/rr/$(echo -n $malware_name)-rr-snp

done
```

The script calls then `execute_win7_malware_replay.py`, which is in charge of performing the actual analysis of the target sample. *execute_win7_malware_replay.py* is a script which is derived from *run_target_mal.py*, shown in Appendix B, where basically function `run_sample` (Listing B.1) is stripped out, as there is no need to generating another PANDA recording.

Appendix D

How to run the replays

Listing D shows the content of the script that is used to run replays for Windows 7 executables. It contains a lot of comments, in this way it is easy to understand the script because it is rather self explanatory.

Listing D.1: "Run replay Windows 7"

```
#!/bin/bash

DPI_HOME="/home/samaicardi/my_panda2/panda/plugins/packer_inspector"
#REPLAY_DIR="/home/samaicardi/replays"

# stop on any error
set -e

if [[ -z $1 ]] ; then
    echo "Usage: $0 <replay_name> <exe_name> <options> <setup> <folder>"
    echo "<replay_name> has to be in the usual PANDA format"
    echo "<exe_name> is the name of the executable we want to trace"
    echo "<options> for other PANDA options (--null if not used)"
    echo "<setup> is --setup (to recompute the address of psActiveProcessHead and the KDBG) or --
     null"
    echo "<folder> is where to store the log files"
    exit 1
fi

output_dir="$DPI_HOME/$5"

if [[ ! -d $output_dir ]] ; then
    echo "$output_dir: no such file or directory"
    exit 2
fi

echo "#define PLUGIN_PATH \"$output_dir\"" > $DPI_HOME/output_dir.h
touch $DPI_HOME/packer_inspector.cpp

#echo '#define OS_WINDOWS 1' > $DPI_HOME/os_define.h

#==================================
#            OPTIONS
#==================================
options=$3
if [[ $3 == "--null" ]] ; then
    options=""
fi


if [[ $4 == "--setup" ]] ; then
#==================================
#       PsActiveProcessHead
#==================================
    echo -ne "\e[1m[$(basename $0)] Retrieving PsActiveProcessHead from ./$1-rr-snp ... \e[0m"
```

```
    psActiveProcessHead=$(vol.py -f $1-rr-snp --profile=Win7SP1x86 kdbgscan 2>&1 | grep
     PsActiveProcessHead | uniq | awk 'BEGIN{FS=": "}{print $2}')
    #psActiveProcessHead=$(vol.py -f $1-rr-snp --profile=Win7SP1x86 kdbgscan | grep
     PsActiveProcessHead | uniq | awk 'BEGIN{FS=": "}{print $2}')

    re='^0x[0-9a-f]+$'
    if ! [[ $psActiveProcessHead =~ $re ]] ; then
        echo -e "\e[1merror: unable to find psActiveProcessHead\e[0m" >&2; exit 1
    fi
    if [[ -z $psActiveProcessHead ]] ; then
        echo -e "\e[1merror: unable to find psActiveProcessHead\e[0m" >&2; exit 1
    fi

    echo -e "\e[1m$psActiveProcessHead\e[0m"
    echo "#define psActiveProcessHead $psActiveProcessHead" > $DPI_HOME/psActiveProcessHead.h   #
      already included by VMI_win7.cpp

#==================================
#              KDBG
#==================================
# IMPORTANT: I need to hardcode the value of the kdbg in win7x86intro.cpp every time I run
     another recording
    if [[ ! -e ./$1.dd ]] ; then
        echo -e "\e[1m[$(basename $0)] Generating memory dump $1.dd ... \e[0m"
        #echo "~/my_panda2/i386-softmmu/qemu-system-i386 -replay $1 $options -panda memsavep:
     percent=1,file=$1.dd 1>/dev/null 2>&1"
        #~/my_panda2/i386-softmmu/qemu-system-i386 -replay $1 $options -panda memsavep:percent=1,
     file=$1.dd 1>/dev/null 2>&1

        command="/home/samaicardi/my_panda2/i386-softmmu/qemu-system-i386 -replay $1 $options -
     panda memsavep:percent=1,file=$1.dd"
        echo -e "\e[1m[$(basename $0)] $command \e[0m"
        #$command 1>/dev/null 2>&1
        $command
        echo -e "\e[1mdone.\e[0m"
    fi

    echo -ne "\e[1m[$(basename $0)] Retrieving KDBG from ./$1.dd ... \e[0m"
    imported_kdbg=$(vol.py -f $1.dd imageinfo 2>&1 | grep 'KDBG :' | awk 'BEGIN{FS=": "}{print$2
     }' | awk 'BEGIN{FS="L"}{print $1}')
    #imported_kdbg=$(vol.py -f $1.dd imageinfo  | grep 'KDBG :' | awk 'BEGIN{FS=": "}{print$2}' |
      awk 'BEGIN{FS="L"}{print $1}')

    re='^0x[0-9a-f]+$'
    if ! [[ $imported_kdbg =~ $re ]] ; then
        echo -e "\e[1merror: unable to find KDBG (res = $imported_kdbg)\e[0m" >&2; exit 1
    fi
    if [[ -z $imported_kdbg ]] ; then
        echo -e "\e[1merror: unable to find KDBG (zero)\e[0m" >&2; exit 1
    fi

    echo -e "\e[1m$imported_kdbg\e[1m"
    echo -n "#define imported_kdbg $imported_kdbg" > $DPI_HOME/../win7x86intro/import_kdbg.h
    touch $DPI_HOME/../win7x86intro/win7x86intro.cpp

elif [[ $4 != "--null" ]] ; then
    echo "error: wrong <setup>" >&2; exit 1
fi


#==================================
#      OS HEADER GENERATION
#==================================
#echo '#include "win7/VMI_win7.h"' > $DPI_HOME/os_include.h
#echo '#include "win7/win7_api_call.h"' >> $DPI_HOME/os_include.h
#echo '#include "win7/win7_functions.h"' >> $DPI_HOME/os_include.h


#==================================
#       MAKEFILE GENERATION
#==================================
#echo -n "DPI_OS_DEPENDENT_FILES=" > $DPI_HOME/os_makefile.mak
```

```
#echo "$DPI_HOME/win7/win7_api_call.o $DPI_HOME/win7/win7_functions.o $DPI_HOME/win7/VMI_win7.o"
     >> $DPI_HOME/os_makefile.mak


#==================================
#        CLEAN-UP and MAKE
#==================================
echo -e "\e[1m[$(basename $0)] Clean-up and make ...\e[0m"
bash -c "cd $DPI_HOME && make clean"
bash -c "cd $DPI_HOME/../../../ && make"
echo -e "\e[1m[$(basename $0)] Removing old log files in $output_dir ...\e[0m"
rm -f $output_dir/functions.log $output_dir/events.log $output_dir/execution_model.log
     $output_dir/trace.log


#==================================
#            LAUNCH
#==================================
echo -e "\e[1m[$(basename $0)] Launching PANDA... [command_injection?]\e[0m"
panda_command="/home/samaicardi/my_panda2/i386-softmmu/qemu-system-i386 -replay $1 -panda
     syscalls2:profile=windows7_x86 -panda packer_inspector:name=$2,os=win -os windows-32-7
     $options"
echo -e "\e[32m$panda_command\e[0m"

$panda_command
```

Listing D shows the content of the script that is used to run replays for Linux x86 elf executables. As we can see there are differences from the previous Listing.

Listing D.2: "Run replay Linux x86"

```
#!/bin/bash

DPI_HOME="/home/samaicardi/my_panda2/panda/plugins/packer_inspector"
#REPLAY_DIR="/home/samaicardi/replays"

# stop on any error
set -e

if [[ -z $1 ]] ; then
    echo "Usage: $0 <replay_name> <exe_name> <options> <setup> <folder>"
    echo "<replay_name> has to be in the usual PANDA format"
    echo "<exe_name> is the name of the executable we want to trace"
    echo "<options> for other PANDA options (--null if not used)"
    echo "<setup> is --setup (to recompute the address of psActiveProcessHead) or --null"
    echo "<folder> is where to store the log files"
    exit 1
fi

output_dir="$DPI_HOME/$5"

if [[ ! -d $output_dir ]] ; then
    echo "$output_dir: no such file or directory"
    exit 2
fi

echo "#define PLUGIN_PATH \"$output_dir\"" > $DPI_HOME/output_dir.h
touch $DPI_HOME/packer_inspector.cpp

if [[ $4 == "--setup" ]] ; then
    # do nothing for the moment
    echo -n ""
elif [[ $4 != "--null" ]] ; then
    echo "error: wrong <setup>" >&2; exit 1
fi


#==================================
#      OS HEADER GENERATION
#==================================
#echo '#include "linux_32/VMI_linux_32.h"' > $DPI_HOME/os_include.h
```

```
#echo '#include "linux_32/linux_32_functions.h"' >> $DPI_HOME/os_include.h
#echo '#include "linux_32/linux_32_api_call.h"' >> $DPI_HOME/os_include.h
#echo '#include "VMI_linux_32.h"' > $DPI_HOME/os_include.h
#echo '#include "linux_32_functions.h"' >> $DPI_HOME/os_include.h
#echo '#include "linux_32_api_call.h"' >> $DPI_HOME/os_include.h
#echo '#define OS_LINUX 1' > $DPI_HOME/os_define.h


#==================================
#       MAKEFILE GENERATION
#==================================
#echo -n "DPI_OS_DEPENDENT_FILES=" > $DPI_HOME/os_makefile.mak
#echo "$DPI_HOME/linux_32_api_call.o $DPI_HOME/linux_32_functions.o $DPI_HOME/VMI_linux_32.o" >>
     $DPI_HOME/os_makefile.mak


#==================================
#            OPTIONS
#==================================
options=$3
if [[ $3 == "--null" ]] ; then
    options=""
fi


if [[ $6 == "--dirty" ]] ; then
    echo '#define DPI_DIRTY_EXIT 1' > $DPI_HOME/exit_strategy.h
else
    echo '#define DPI_CLEAN_EXIT 1' > $DPI_HOME/exit_strategy.h
fi
touch $DPI_HOME/packer_inspector.cpp

do_make=1
if [[ $7 == "--nomake" ]] ; then
    # do nothing for the moment
    do_make=0
elif [[ $7 != "--make" ]] ; then
    echo "error: wrong <make> option" >&2; exit 1
fi

#==================================
#        CLEAN-UP and MAKE
#==================================
echo -e "\e[1m[$(basename $0)] Clean-up and make ...\e[0m"
if [[ $4 == "--setup" ]] ; then
    bash -c "cd $DPI_HOME && make clean"
fi
if [[ $do_make == 1 ]] ; then
    bash -c "cd $DPI_HOME/../../../ && make"
fi
echo -e "\e[1m[$(basename $0)] Removing old log files in $output_dir ...\e[0m"
rm -f $output_dir/functions.log $output_dir/events.log $output_dir/execution_model.log
     $output_dir/trace.log


#==================================
#            LAUNCH
#==================================
echo -e "\e[1m[$(basename $0)] Launching PANDA... [command_injection?]\e[0m"
panda_command="/home/samaicardi/my_panda2/i386-softmmu/qemu-system-i386 -replay $1 -panda
     syscalls2:profile=linux_x86 -panda osi -panda osi_linux:kconf_file=/home/samaicardi/my_panda2
     /panda/plugins/osi_linux/kernelinfo.conf,kconf_group=my_debian_i386 -panda packer_inspector:
     name=$2,os=linux -os linux-32-* $options"
echo -e "\e[32m$panda_command\e[0m"

$panda_command
```

Listing D shows the content of the script that is used to run replays for Linux ARM elf executables. It is quite similar to what is shown in Listing D because the executions in the two architectures (x86 and arm) have many points in common, as already said previously.

Listing D.3: "Run replay Linux ARM"

```
#!/bin/bash

DPI_HOME="/home/samaicardi/my_panda2/panda/plugins/packer_inspector"
#REPLAY_DIR="/home/samaicardi/replays"

# stop on any error
set -e

if [[ -z $1 ]] ; then
    echo "Usage: $0 <replay_name> <exe_name> <options> <setup> <folder>"
    echo "<replay_name> has to be in the usual PANDA format"
    echo "<exe_name> is the name of the executable we want to trace"
    echo "<options> for other PANDA options (--null if not used)"
    echo "<setup> is --setup (to recompute the address of psActiveProcessHead) or --null"
    echo "<folder> is where to store the log files"
    exit 1
fi

output_dir="$DPI_HOME/$5"

if [[ ! -d $output_dir ]] ; then
    echo "$output_dir: no such file or directory"
    exit 2
fi

echo "#define PLUGIN_PATH \"$output_dir\"" > $DPI_HOME/output_dir.h
touch $DPI_HOME/packer_inspector.cpp

if [[ $4 == "--setup" ]] ; then
    # do nothing for the moment
    echo -n ""
elif [[ $4 != "--null" ]] ; then
    echo "error: wrong <setup>" >&2; exit 1
fi


#==================================
#      OS HEADER GENERATION
#==================================
#echo '#include "arm/VMI_arm.h"' > $DPI_HOME/os_include.h
#echo '#include "arm/arm_functions.h"' >> $DPI_HOME/os_include.h
#echo '#include "arm/arm_api_call.h"' >> $DPI_HOME/os_include.h
#echo '#include "VMI_arm.h"' > $DPI_HOME/os_include.h
#echo '#include "arm_functions.h"' >> $DPI_HOME/os_include.h
#echo '#include "arm_api_call.h"' >> $DPI_HOME/os_include.h


#==================================
#       MAKEFILE GENERATION
#==================================
#echo -n "DPI_OS_DEPENDENT_FILES=" > $DPI_HOME/os_makefile.mak
#echo "$DPI_HOME/arm_api_call.o $DPI_HOME/arm_functions.o $DPI_HOME/VMI_arm.o" >> $DPI_HOME/
    os_makefile.mak


#==================================
#            OPTIONS
#==================================
options=$3
if [[ $3 == "--null" ]] ; then
    options=""
fi


if [[ $6 == "--dirty" ]] ; then
    echo '#define DPI_DIRTY_EXIT 1' > $DPI_HOME/exit_strategy.h
else
    echo '#define DPI_CLEAN_EXIT 1' > $DPI_HOME/exit_strategy.h
fi
touch $DPI_HOME/packer_inspector.cpp
```

```
do_make=1
if [[ $7 == "--nomake" ]] ; then
    # do nothing for the moment
    do_make=0
elif [[ $7 != "--make" ]] ; then
    echo "error: wrong <make> option (you typed: $7)" >&2; exit 1
fi

#==================================
#        CLEAN-UP and MAKE
#==================================
echo -e "\e[1m[$(basename $0)] Clean-up and make ...\e[0m"
if [[ $4 == "--setup" ]] ; then
    bash -c "cd $DPI_HOME && make clean"
fi
if [[ $do_make == 1 ]] ; then
    bash -c "cd $DPI_HOME/../../../ && make"
fi
echo -e "\e[1m[$(basename $0)] Removing old log files in $output_dir ...\e[0m"
rm -f $output_dir/functions.log $output_dir/events.log $output_dir/execution_model.log
     $output_dir/trace.log


#==================================
#            LAUNCH
#==================================
HOMEDIR=/home/samaicardi
echo -e "\e[1m[$(basename $0)] Launching PANDA... [command_injection?]\e[0m"
panda_command="$HOMEDIR/my_panda2/arm-softmmu/qemu-system-arm -M versatilepb -kernel $HOMEDIR/.
     panda/vmlinuz-3.2.0-4-versatile -initrd $HOMEDIR/.panda/initrd.img-3.2.0-4-versatile -hda
     $HOMEDIR/.panda/debian_wheezy_armel_standard.qcow2 -append 'root=/dev/sda1' -panda syscalls2:
     profile=linux_arm -panda osi -panda osi_linux:kconf_file=/home/samaicardi/my_panda2/panda/
     plugins/osi_linux/kernelinfo.conf,kconf_group=my_debian_arm -panda packer_inspector:name=$2,
     os=linux -os linux-32-* -replay $1 $options"
echo -e "\e[32m$panda_command\e[0m"

$panda_command
```

Bibliography

[1] X. Ugarte-Pedrero, D. Balzarotti, I. Santos, P. G. Bringas, "SoK: Deep Packer Inspection: A Longitudinal Study of the Complexity of Run-Time Packers", Proceedings of the IEEE Symposium on Security and Privacy, San Jose, 2015 DOI 10.1109/SP.2015.46

[2] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, P. Saxena, "BitBlaze: A New Approach to Computer Security via Binary Analysis", Proceedings of the 4th International Conference on Information Systems Security, Keynote invited paper, Hyderabad, India, Dec 2010, DOI 10.1007/978-3-540-89862-7_1

[3] PANDA, https://github.com/panda-re/panda#panda

[4] IDA, https://www.hex-rays.com/products/ida/

[5] Radare, http://rada.re/r/

[6] Hopper, https://www.hopperapp.com/

[7] BinaryNinja, https://binary.ninja/

[8] UPX, https://upx.github.io/

[9] packerinspector.com, https://www.packerinspector.com/

[10] EURECOM, http://www.eurecom.fr

[11] DeustoTech, http://deustotech.deusto.es

[12] BitBlaze: Binary Analysis for Computer Security, http://bitblaze.cs.berkeley.edu/

[13] QEMU, https://qemu.org/

[14] A. Henderson, A. Prakash, L. K. Yan, X. Hu, X. Wang, R. Zhou, H. Yin, "Make It Work, Make It Right, Make It Fast: Building a Platform-neutral Whole-system Dynamic Binary Analysis Platform", Proceedings of the 2014 International Symposium on Software Testing and Analysis, San Jose, CA, USA, 2014, pp. 248-258, DOI 10.1145/2610384.2610407,

[15] AVATAR, http://www.s3.eurecom.fr/tools/avatar

[16] PyReBox, https://github.com/Cisco-Talos/pyrebox

[17] How to read graphs on packerinspector.com, https://www.packerinspector.com/reference#ex-graphs

[18] EXECryptor, http://www.strongbit.com/execryptor_inside.asp

[19] Virustotal results of a malware with md5: cecf3c6e7139985101e181a235e90aea, https://www.virustotal.com/#/file/0aa794ec929696e378e25835119e7ab86e1c0ee9e4f41f09d70235fe5822ad9f/details

[20] Virustotal results of a malware with md5: c63bb9913158e8afc4cc680e02a027de, https://www.virustotal.com/#/file/186ea13ea561ba34cba9382db452ab16f2b369e4e2e5f65432c2ec35e4211b4a/details