Black Hat USA 2016

PinDemonium: a DBI-based generic unpacker for Windows executables

Sebastiano Mariani, Lorenzo Fontana, Fabio Gritti, Stefano D'Alessio

Contents

Chapter 1

Introduction

Nowadays malware authors, in order to hinder the analysis process, are increasingly using anti-analysis tools called *packers*. These tools can deeply change the structure of a program by obfuscating its code and resources, making automatic static analysis practically impossible and manual static analysis (i.e., reverse engineering) very hard. In this work we focus on how to automatically unpack a Windows executable and reconstruct a working version of it, exploiting the capabilities of a DBI framework (Intel PIN), in a way that is flexible with respect to the packer used by the malicious developer. Our goal is to leverage the fact that every packed executable has to unpack itself at run-time, in order to fulfill its goals. This run-time unpacking process differs from packer to packer, and its complexity can make it very hard to be predictable. Besides that, every unpacking routine shares a common behavior: it must write the new code in memory and then redirect the execution to it. Recognizing this process is the core of the modern analysis systems aimed to extract the original program code from a packed binary: these systems are commonly called *generic unpackers*. In order to be complete, the main problems that these tools must resolve are the correct detection of the Original Entry Point (OEP) of the program (i.e., where the real malicious code starts) and the reconstruction of a de-obfuscated version of it with a correct Import Directory (the list of the imported Windows libraries and functions), that implicitly hides the problem of detecting the Import Address Table (IAT) inside the process memory (i.e., finding the

structure that contains all the resolved addresses of the imported functions in the process memory). The former problem can be faced only using heuristics, since understanding algorithmically when the unpacking process is finished is a not decidable problem; different works proposed clever ideas that range from the use of the entropy to the detection of particular patterns of code. The latter problem is usually faced by recognizing the IAT pattern in memory (a series of addresses that points to nearby memory locations) and then dereferencing the addresses found in that position in order to discover the related APIs. However, this process is usually hindered by the so called IAT obfuscation techniques, that complicate the reconstruction of an Import Directory. Few of the analysis tools developed until now try to reconstruct a working version of the original binary, and none tries to combine together all the heuristics proposed in literature in order to collect useful information to increase the probability to detect the OEP of the binary. However, we made the consideration that combining together multiple heuristics can overcome the limitations of a single one and give a more precise indication of the correct dump that can include the OEP. Moreover, the majority of the current generic unpackers limit themselves to dump the program memory at the OEP, while others are only a support for Anti-Virus (AV) malware detection; for both of them the reconstruction of a de-obfuscated binary is out of scope either because this require an additional step that brings lots of new problems or because it is simply not needed for the purpose of the tool. Our approach in building PinDemonium exploits the general idea of a generic unpacker: tracking the written addresses in order to identify written and then executed memory regions and, after this condition is met, making a dump of the code at this moment and try to find all the imported functions in order to reconstruct a working program. We have overcome the limitations of existing generic unpackers by taking care also of dynamic unpacking of code on the heap and by defeating some of the IAT obfuscation techniques employed by malware. Moreover, since our strategy produces as final result different reconstructed binaries, we have introduced many OEP detection heuristics and unified them in a synergy aimed to identify the best candidate for being the correct working de-obfuscated program. Different approaches exist in order to build a generic unpacker: debuggers, kernel modules, hyper-

visor modules and Dynamic Binary Instrumentation (DBI) frameworks. In this work we choose to explore the possibilities offered by a Dynamic Binary Instrumentation (DBI) framework which allows a complete control over the analyzed program at a very fine granularity (i.e., every instruction can be inspected and modified). In particular, we choose PIN, since it is one of the most complete and well documented DBIs available. The core component of *PinDemonium* keeps track of instructions executed from previously written addresses, even inside external processes, in order to correctly identify new unpacked layers or eventually the layer that contains the OEP of the original packed program. When a written and then executed instruction is detected (and other conditions aimed to void too many dumps are met), we trigger a dump of the process by exploiting an external project that we have decide to include in our tool: *Scylla*. We have modified Scylla in both its core component: the *PE Reconstruction module* and the *IAT search and reconstruction Module*. The former has been enhanced in order to take care also of situations in which the new unpacked code is positioned on dynamic memory regions such as the heap, while the latter has been modified in order to allow the analyst to write its own deobfuscator and integrate it inside our tool. In order to save bit of performance and simplify the process of recognize written and executed code, we avoid tracking meaningless instructions usually not related to the unpacking process such as writes on the stack (since this is not a common behavior observed in packers) and writes on the Process Environment Block (PEB). Due to the deep differences among packers in their packing and run-time unpacking routines, a full generic unpacking algorithm that can fit all of them and can correctly reconstruct a working de-obfuscated program is really hard to develop, so we have decided to implement a series of ad-hoc techniques in order to take care also of particular run-time unpacking strategies that we have identified in some packers. These techniques can be enabled or not using a set of flags that can be passed to PinDemonium when starting the instrumentation process. In order to validate our work we have conducted two experiments, the first one is aimed to demonstrate the generality of our unpacking process against two known programs packed with 15 different known packers. With these tests we want to demonstrate that PinDemonium can extract the original program code from its packed

version independently of the packer employed to obfuscate the binary. The second experiment has the goal to demonstrate the effectiveness of PinDemonium against samples collected in the wild, packed with both known packers and not known ones. In these tests we have configured PinDemonium in the most generic way as possible and we have automatized the unpacking process over 1,066 malware collected from Virus Total. The final results of the latter experiment have been manually validated in order to discover if a working de-obfuscated program has been produced or not. The final results of this experiment show that PinDemonium manage to reconstruct a working de-obfuscated binary for 63% of the collected samples. These results can be further raised by resolving some of the PinDemonium limitations that we have identified and for which we have thought possible solutions proposed as future works. In conclusion, PinDemonium is a tool that can offer an unpacking service aimed to speed up the work of professional malware analysts in their daily battle against malicious packed programs.

Chapter 2

Approach

2.1 Dynamic Binary Instrumentation

Dynamic Binary Instrumentation is a technique for analysing the behaviour of a binary application through the injection of instrumentation code. The instrumentation code can be developed in an high level programming language and is executed in the context of the analysed binary with a granularity up to the single assembly instruction. The injection of instrumentation code is achieved by implementing a set of callbacks provided by the DBI framework. The most common and useful callbacks are:

- Instruction callback: invoked for each instruction.
- Image load callback: invoked each time an image (Dynamic Loaded library (DLL) or Main image) is loaded into memory.
- Thread start callback: invoked each time a thread is started.

Besides the callbacks the DBI framework allows to intercept and modify operative system Application Programming Interfaces (APIs) and system calls and this is very useful to track some behaviours of the binary, like the allocation of dynamic memory areas.

2.2 Approach overview

Our tool exploits the functionality provided by the Intel PIN DBI framework to track the memory addresses which are written and then executed with an instruction level granularity. More in details for each instruction the following steps are performed:

1. Written addresses tracking: keep track of each memory address which is written (even in a remote processes address space) in order to create a list of memory ranges of contiguous writes defined *Write Intervals* (Section 2.3.1).

2. *Write xor Execution (WxorX)* addresses notifier: check if the currently executed instruction belongs to a *Write Interval* (Section 2.3.2). This is a typical behaviour in a packer that is executing the unpacked layer and for this reason we trigger a detailed analysis which consists of:

   (a) Dump the memory range which triggered the WxorX rule violation: depending on its location this can be the main image of the Portable Executable (PE), a memory range on the heap (Section 2.3.3) or a memory range inside a different process address space.

   (b) Reconstructing the IAT and generating the correct Import Directory (Section 3.2.5).

   (c) Applying a set of heuristics to evaluate if the current instruction is the OEP (Section 2.3.5):

   - entropy: Check if the delta between the starting value of the entropy and the current one is above a certain threshold.
   - long jump: Check if the difference between the address of the current Extended Instruction Pointer (EIP) and the previous one is above a certain threshold.
   - jump outer section: Check if the current EIP is in a different section from the one of the previous EIP.
   - yara rules: Check if a set of yara rules is matched inside the dump which has broken the WxorX rule.

The result of our tool is a set of memory dumps or reconstructed PEs depending on the success of the IAT fixing phase and a report which includes the values of each heuristic for every dump. Based on these information we can choose the best dump, that is the one that has the greatest chance of work.

2.3 Approach details

In this section we are going to describe in details the steps introduced in the previous section.

2.3.1 Written addresses tracking

All packers, in order to work correctly, present a common behaviour: they have to write the original program in memory and then execute it. For detecting this behaviour our tool tracks each write operation made by the program and builds a set of memory ranges which identify contiguously written memory addresses. A memory range is identified by a starting address and an end address and it is managed in order to take into account the memory operations that overlap it: the starting address and the end address are decreased or increased respectively when an overlapping writes is detected (see Figure 2.1).

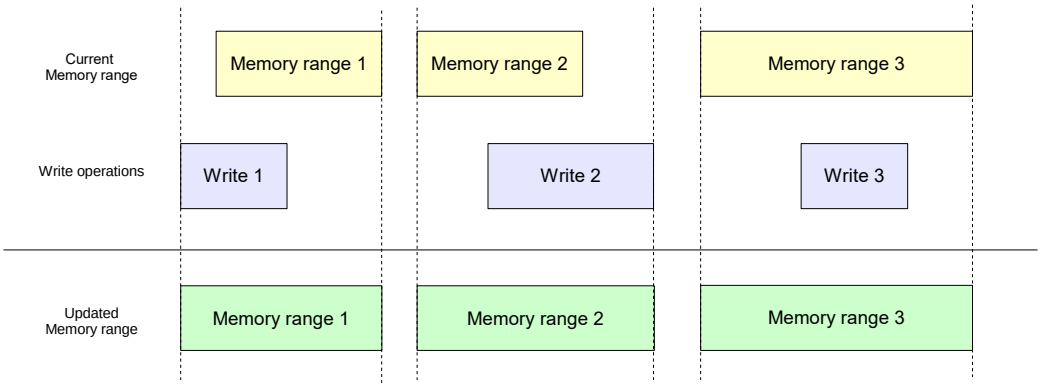

Figure 2.1: Memory range management

2.3.2 WxorX addresses notifier

As normal behaviour (except for Just in time (JIT) compilers) a particular memory address used by a program is either written or executed. For this reason we can say that usually the memory addresses handled by the binary satisfy the *WxorX rule*. On the other hand packers in order to unpack the original malware need to use addresses which breaks the *WxorX rule* since the original code needs to be written in memory and then executed.
Therefore, the first time we detect that the instruction pointer is inside one of the tracked memory ranges, all the analysis (Section 3.2.6) and dumping routines (Section 3.2.3) are triggered and the memory range is marked as *broken*. It is easy to deduct that the definition of a broken memory range is: a written memory range in which the execution has been redirected to.
The initial approach was to dump the code and trigger the analysis of it once the WxorX rule was broken in a *Write Interval* (set of contiguous memory address previously written) for the first time and then ignore all the subsequent executed instructions from the same *Write Interval*. However, when analysing a binary packed with *mpress* [46], we noticed the behaviour in Figure 2.2 that forced us to slightly modify the initial approach.

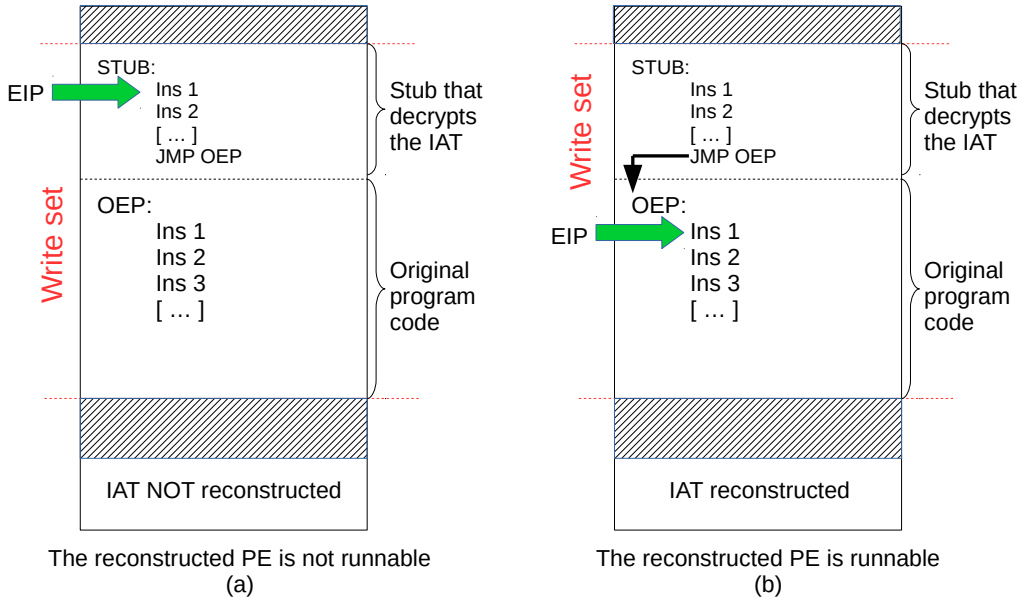

Figure 2.2: Mpress behaviour

When *mpress* [46] enters for the first time in the *Write Interval* that contains the original code (Figure 2.2(a)), it does not immediately execute the original program's code from the OEP but it starts the execution from an area of memory that contains a small stub, created by the packer itself, that reconstructs the IAT and eventually jumps to the original code (Figure 2.2(b)), that resides few bytes after in the same *Write Interval*. Using our original approach we would have dumped the code in the situation presented in Figure 2.2(a) since it is the first time that the WxorX rule is broken inside this Write Interval, but we would not have taken a dump after the jump showed in Figure 2.2(b), and so we would not been able to recreate a running PE because the IAT was not reconstructed yet.
Since this behavior can be shared among different packers, we have decided to generalize better our algorithm introducing the concept of *intra-write set analysis*. The *intra-write set analysis*, enabled optionally, specifies how many dumps should be taken inside the same Write Interval. An arbitrary number of dumps can be set, but, in this case, an *intra-write set analysis* of two jumps solves the problem raised during the analysis of mpress [46] since the

dump is taken both the first time the execution violates the WxorX rule (Figure 2.2(a)) and also in the situation displayed by Figure 2.2(b). In the second scenario the IAT is correctly fixed and it is possible to rebuild the *Import Directory* and reconstruct the PE.
However, in order to correctly implement this technique we need to take care also of another problem: we do not want to waste additional dumps by dumping the memory for each instruction inside the same *Write Interval*, since this would not increase the chance of reconstruction the PE and would generate a huge number of dumps. In order to understand better this problem we will use the next picture.

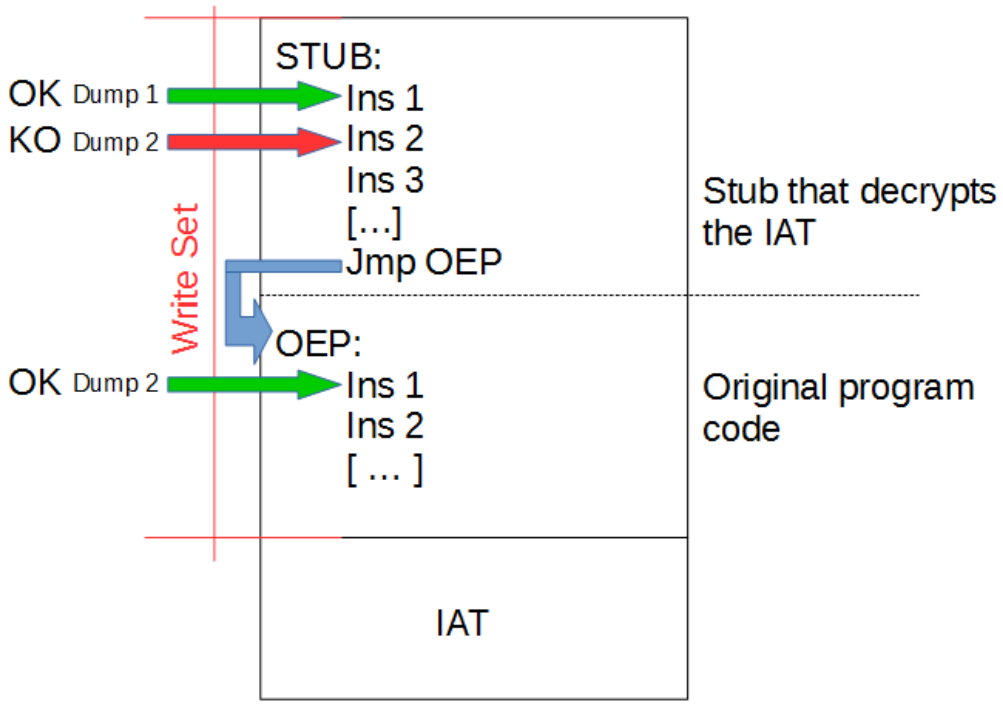

Figure 2.3: Inter write set analysis explanation

As we can see from the Figure 2.3 we are going to take the Dump 1 marked with the green arrows since it is the first time that the WxorX rule is broken in the write set, but after that we do not want to take the Dump

2 marked with the red arrow because in that case we would have the same problem that we would have dumped the process's memory when the IAT is not yet reconstructed. Rather we want to take the Dump 2 marked with the green arrow because in that moment the IAT has been reconstructed and we would be able to rebuild a working PE. In order to make the dump in the correct moment we have determined a threshold dependent on the write interval's size representing the minimum variation of the EIP needed to take the dump. This threshold has been determined through a small survey explained in Chapter 3 and its use allows us to avoid taking a dump for each instruction. Moreover a common behaviour in packer is to have a long jump right before reaching the OEP and therefore taking the dump only when the delta between the current and the previous EIP is above the threshold will not prevent us to dump the memory at the correct moment. Obviously the fixed threshold we have identified is only a possible value, but not absolutely general since, as said, packers' behaviors are really different and the layout of write intervals in memory can change deeply. In order to getting rid of this problem we have decide to propose our default value for the threshold to 5% of the write interval's size, but let finally the user to tune this value at his disposal.

2.3.3 Dumping process

Many memory dumping tools, programs that take care of copying the memory allocated by a process to a file, share a common behavior: they only dump the *main module* of the target program. The main module of a program is made up by different sections containing the code, the data and the resources used by the binary. An example of a main module of a Windows PE is shown in Figure 2.4.
The approach of dumping only this memory range correctly dumps all the sections of the binary, but fails to collect code that has been unpacked on dynamic memory regions such as the heap.

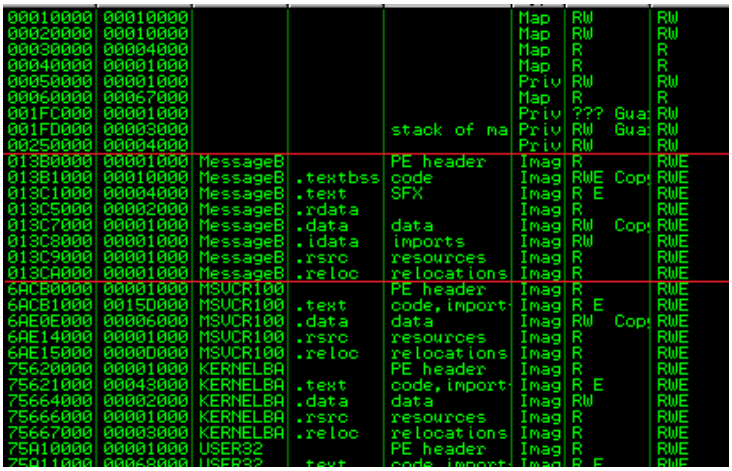

Figure 2.4: Main module of MessageBox in the red square

When packer's stub unpacks new code on the heap and then redirects the execution there, we correctly catch this using the *WxorX rule*, but using a naive dump strategy we would dump something meaningless since the *Write Interval* in which the *WxorX rule* is violated is not included in the dump, as we can see in Figure 2.5.

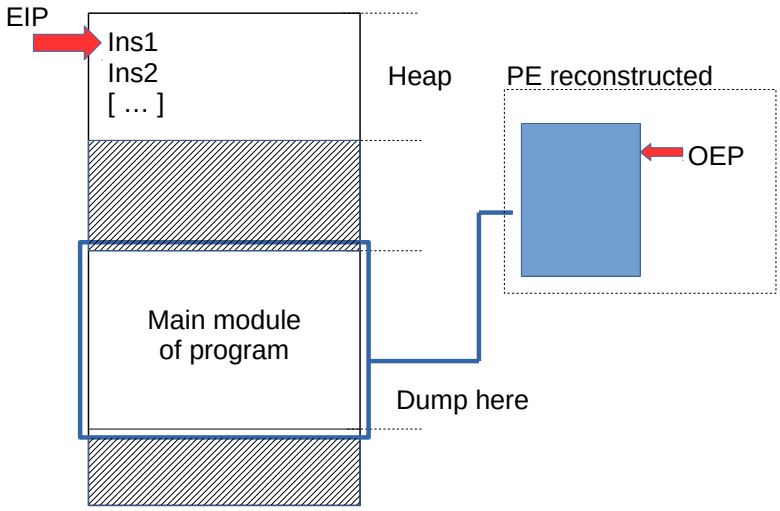

Figure 2.5: Heap not dumped using naive strategy

In fact, when the EIP reaches the heap we trigger the dump of the main module, but this will reconstruct a meaningless PE, since it will not contains the current write interval in which the EIP is.
A technique to solve this issue has been proposed by Jason Geffner in this post [23]. The proposed solution is to add a new section to the binary before the execution and then, following the code with a debugger, force the heap allocation functions such as *VirtualAlloc()*, *HeapAlloc()* to return an address inside the new added section. In this way, when a dump of the main module is taken it will be dumped correctly, since the heap is positioned inside a section. This technique has two main limits:

- It requires to add a new section to the binary before the unpacking and this could be a problem since many packers create a stub which verify, with an initial check-sum, the integrity of the PE and if this check fails the stub aborts the execution.
- It is fundamentally a manual approach because we need to know which is the right dynamic memory allocation used to contain the original payload.

In order to solve this problem we have decided to mark the write intervals on the heap with a flag that identifies them as *heap write intervals*. When the execution comes from an *heap write interval* we not only dump the program main module, but we also add to the final taken dump a new section containing the *heap write interval* data and finally set the Entry Point of the PE in this section (see Figure 2.6).

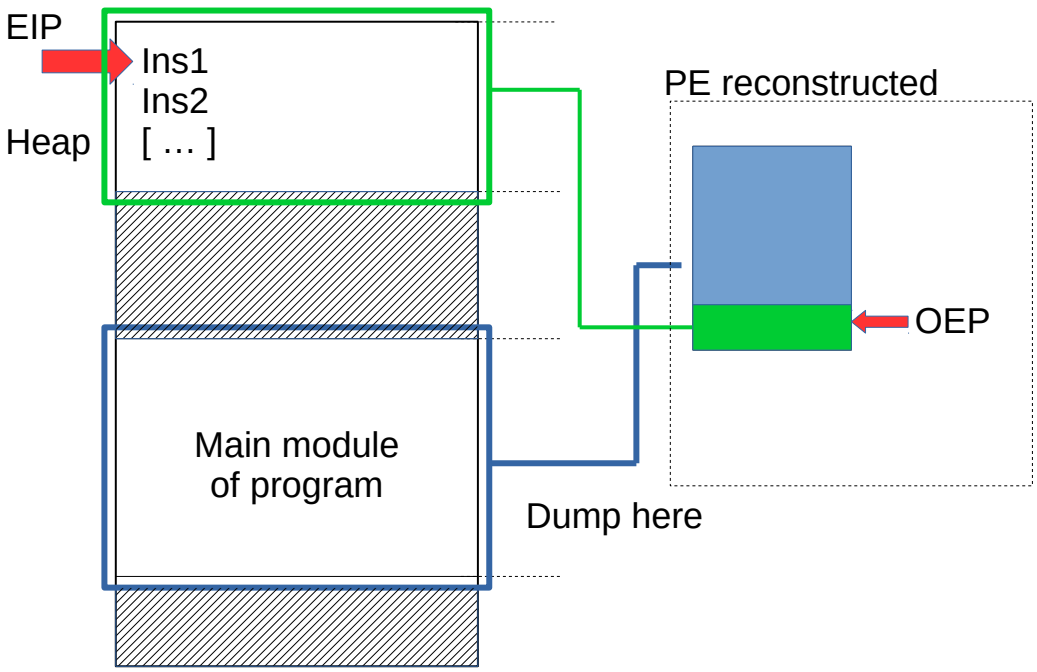

Figure 2.6: Heap dumped using our strategy

Besides dumping the *heap write interval* which triggers the WxorX rule we dumps all the other *heap write intervals* since there can be some references to data and code contained in them. In order to avoid dumping the same *heap write interval* multiple times we check through an MD5 hash that the same memory region hasn't been previously dumped. Finally through an IDA Python script we loads all the *heap write intervals* when the dump is opened in IDA and in this way the analyst is able to evaluate the behaviour of the unpacked stub through static analysis.

2.3.4 IAT Fixing and Import Directory Reconstruction

A PE in order to properly work must have a correct *Import Directory*. This structure is fundamental in the loading process because the loader needs to create the IAT starting from the Import Directory in order to allow the

program to use the imported functions. That step is mandatory because, for imported functions, the compiler does not know at compile time where these functions will reside in memory. The Import Directory is structured as a series of structs called *IMAGE_IMPORT_DESCRIPTOR* of 20 bytes (5 DWORD). For each external DLL imported by the program is present a IMAGE_IMPORT_DESCRIPTOR structure organized as follow:

- **1st DWORD**: Denoted as *OriginalFirtsThunk*, it points to a Relative Virtual Address (RVA) that, once followed, lends to a list of names of functions imported from the current DLL.
- **2nd DWORD - 3rd DWORD**: Not fundamental for the loading procedure.
- **4th DWORD**: Denoted as *ImportedDLLName*, it points to the name of the DLL.
- **5th DWORD**: Denoted as *FirstThunk*, it points to the IAT address that will be filled with the address of the imported functions at loading time.

Figure 2.7 shows the structure of the *Import Directory*.

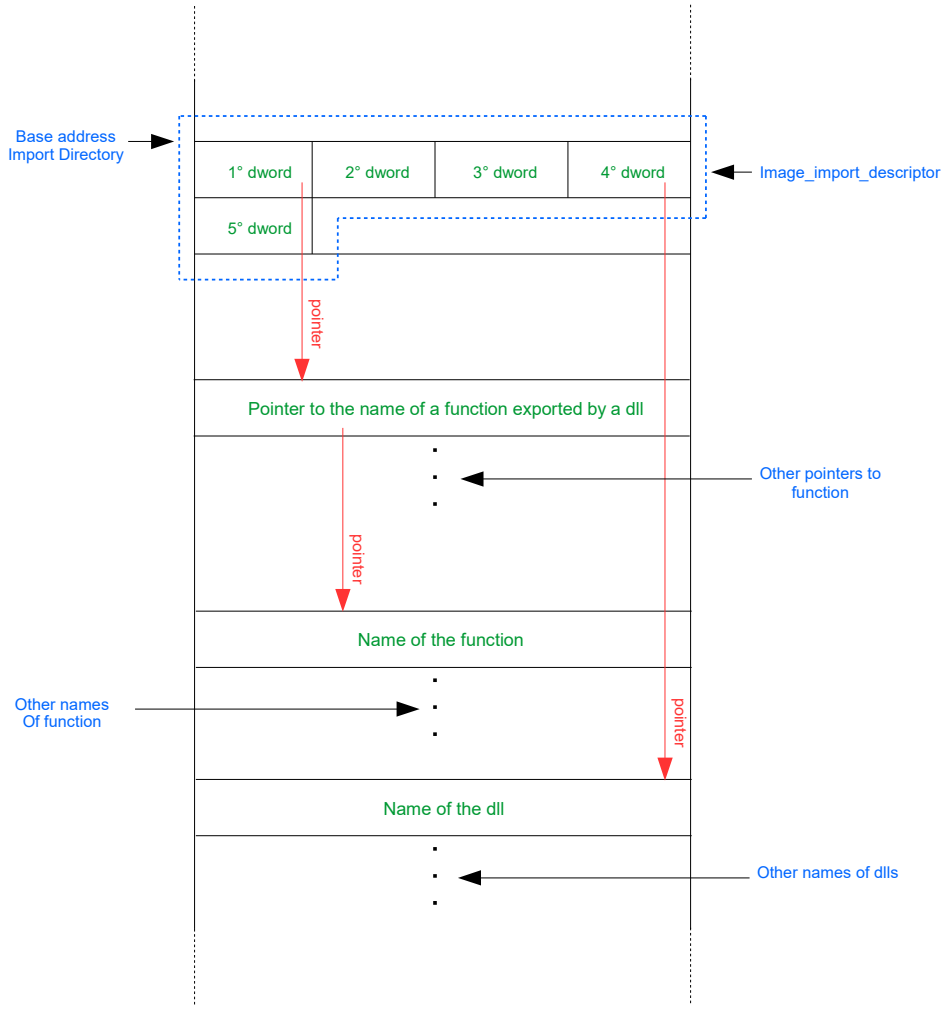

Figure 2.7: Import Directory structure

The loader parses the Import Directory and builds the IAT following these steps:

1. Once located the Import Directory, the loader scans all the *IMAGE_IMPORT_DESCRIPTORs* and for each imported DLL it loads the *entire* library in his memory space.

2. Following the *OriginalFirstThunk*, it retrieves the name of all the functions that the program needs in this DLL.

3. It retrieves their address and puts it at the RVA pointed by the *First-Thunk* of the *IMAGE_IMPORT_DESCRIPTOR*.

Packers, as standard behaviour, destroy the *Import Directory* of the original program substituting it with the one useful to them. The IAT building phase is performed directly by the packer itself while the loader, parsing the substituted *Import Directory*, loads only those functions necessary to the packer (usually *LoadLibrary* and *GetProcAddress*). More advanced packers do not just destroy the *Import Directory* but they obfuscate the IAT hindering the process of reverse engineering. It is possible to divide these techniques in two different categories:

- **Static API Obfuscation**: The obfuscation phase is done at **packing time** and the functions' instruction and addresses are the same for each execution.
- **Dynamic API Obfuscation**: The obfuscation phase is done at **run-time** while unpacking and the functions are read on the target machine and then copied on the heap during the execution.

Our challenge was to build a tool that it is able to correctly de-obfuscate the IAT at run-time, despite IAT obfuscation techniques, and consequently to reconstruct a meaningful *Import Directory* for our dump such as, when the dump is executed, the loader will be able to load the real functions imported by the original program and not those imported by the packer.

Since a generic algorithm that deobfuscate these kind of techniques in a generic way has not been developed yet, we decided to implement a system that can allow an analyst to integrate its own deobfuscator into PINdemonium without modifying it.

2.3.5 Heuristics description

The heuristics are a set of techniques that are used during the unpacking of a packed sample in order to understand if the OEP has already been reached. We need heuristics in order to accomplish this since, as demonstrated by Paul

Royal et al. [41], understanding if the unpacking process is finished or not is not a decidable problem. The heuristics we have collected come from different works and books [45], [32], [11], and we use them to tag a taken dump. These tags are useful at the end of the work of our tool in order to understand if a taken dump and the associated reconstructed PE represent the original program or not (since often the number of taken dumps is different than one).
The heuristics included in our tools are triggered run-time when we decide to dump the process and are the following:

Entropy

Entropy can be considered as a measure of the disorder of a program and can be used in order to detect if an executable has been compressed or encrypted and also when the process of decompression/decryption is nearly finished. The entropy is calculated following these step:

1. A probabilistic experiment is done on the entire binary (i.e., calculate the probability that a certain byte value appear in the binary).

2. Using the Shannon formula (Equation 2.1) with the probabilities retrieved in the previous point, the entropy value is calculated as:

$$H = -\sum_{i=0}^{255} P(i) log_2(P(i)) \tag{2.1}$$

According to the Equation 2.1 the higher is the entropy value the more random is the distribution of the byte values. We trace the entropy evolution of the main module of the target program and trigger the entropy flag when the difference of the current entropy compared with the original one is greater than a threshold. We adopt this heuristic because we expect that the entropy of a compressed/encrypted program is deeply different from the original program.

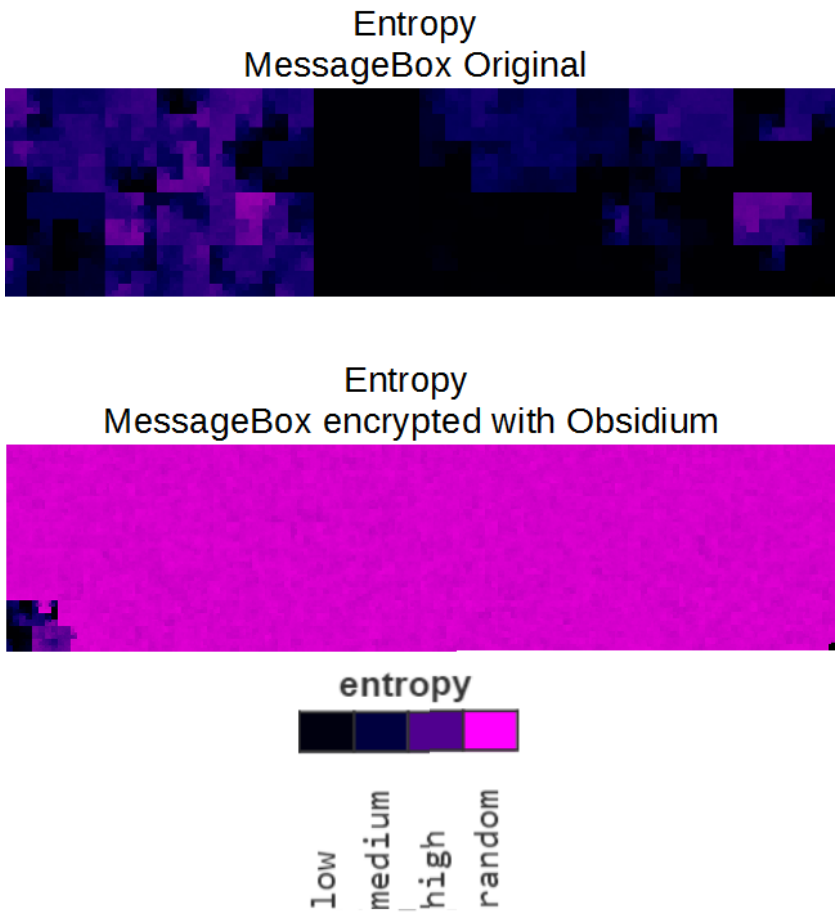

Figure 2.8: Entropy difference between the PE of a test program not encrypted and encrypted with Obsidium. This figures shows a representation of each byte of the binary, organized for convenience in rows in order to have a rectangle representation, along with its probability value expressed with a color scheme. The lower is the probability that the byte value appears in the binary the more its color tend to magenta

As we can notice from the Figure 2.8 the encrypted program, due to the fact that an encryption function must respect the principles of diffusion and confusion returning as output a stream of bytes as random as possible, tend to have an higher entropy value (i.e., low probability for each byte value to appear in the binary) respect to the non-encrypted one as expected.

Long jump

A *long jump* is defined as a jump in which the deviation from the previous EIP address and the current one is greater than a fixed threshold.
Based on the observation that it is very uncommon to have in memory an unpacking stub really near to the code where we can find the OEP (Figure 2.9, cases (a) and (b)), we can say that usually the control transfer from the packer's stubs to the original program's code is performed via a long jump as presented in Figure 2.9, case (c). This behavior can be identified as the final *tail jump* if it is from a stub to the original program's code or simply the control transfer from a stub to another stub.

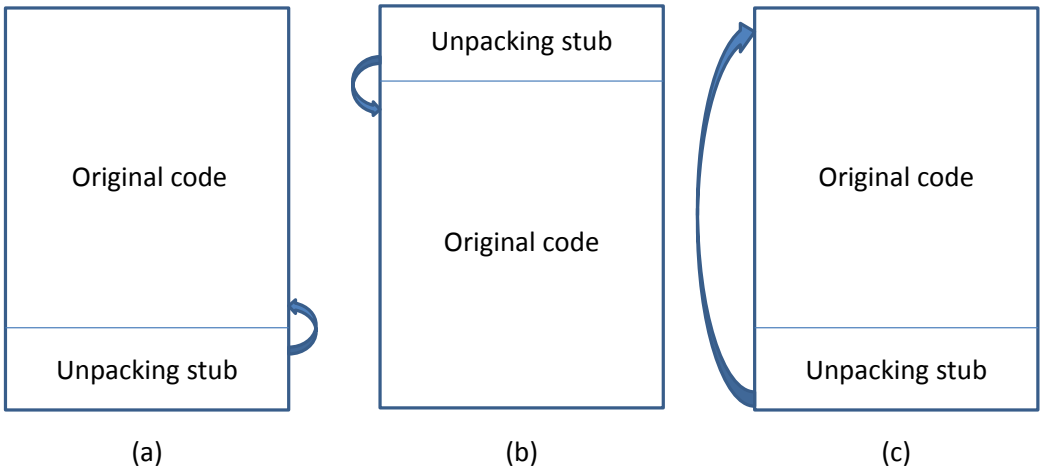

Figure 2.9: Different jumps from the unpacking stub to the original code

A previous work [45] has noticed that the *tail jump* is characterized by a long jump from a section to another different section: this is used also as an additional heuristic in order to detect the final control transfer from packer's stub to the original program's code.

Yara rules

Yara rules are a very powerful tool for identifying and categorizing known malware samples. However since they are based on patterns they can't work if

the malicious payload is encrypted. For this reason we try to match the yara rules inside the dumped and unpacked memory regions every time the WxorX rule is broken. The currently matched rules have two main functionalities:

- *Detect end of unpacking*: if a yara rule, executed on one unpacked layer, matches a know malware family it may mean that the unpacking stub is ended the dump contains the final payload. Moreover we have the information about the malware family which the current sample belongs to.

- *Identify evasion techniques detection*: if a yara rule matches a know evasion technique and the unpacking process seems to be finished prematurely it may mean that the malware has identified the analysis environment and the payload hasn't been executed. Moreover knowing the matched evasion technique the analyst can be able to handle the evasion attempt and keep on analysing the malware.

Finally the user can define his own rules which will be matched on every layer of the unpacking stub and reported inside the Pindemonium report.

Chapter 3

Implementation details

We have divided this chapter in two parts: in Section 3.1 we describe all the external tools used by PinDemonium (PIN, Scylla and IDA); in Section 3.2 we show the implementation of all the modules described in Section 2.3 and how they interact with each other.

3.1 System architecture

The high level overview of our generic unpacker can be summarized in Figure 3.1.

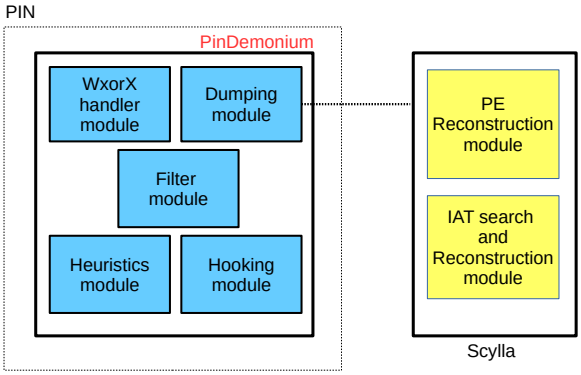

Figure 3.1: Overview of our generic unpacker

After different discussions about which solutions would be better in order to build a generic unpacker we finally decided, given our goals and final purpose, to employ a DBI framework. We have taken this decision for these reasons:

- It gives us full control over the code executed by a program and so the possibility to analyze deeply what the binary is doing.
- It is intrinsically immune to anti-debugging and anti-disassembly techniques. We are going to clarify the reason of this in the next section.
- It has a rich and documented API set really useful to extract information from the running program and the possibility to modify its behavior run-time.

There are different DBIs available online, some of the most discussed and used are: PIN [31], Valgrind [36] and DynamoRIO [2]. We have finally decided to employ PIN [31], a DBI framework developed by Intel, to build our generic unpacker. In the next section we are going to explain briefly how this DBI works and its pro and cons.

3.1.1 PIN

PIN is one of the most complete and well documented DBI framework, it is easy to use and provides efficient instrumentation by using a *JIT compiler*. PIN works with the so called *pintools*, user developed DLLs which implement the routines to perform during the instrumentation of a target binary. Once the program is launched with PIN it first spawns a new process for the target binary and then injects inside that process the user developed *pintool* and the pinvm.dll, a lightweight virtual machine used to control the execution of the instrumented program. Moreover PIN allocates a memory region on the heap called *code cache* which is used to copy instrumented traces that will be afterward executed.

The granularity wherewith PIN works are essentially three (see Figure 3.2):

- **Instruction**: A single instruction inside a collected trace.

- **Basic blocks**: A sequence of instructions terminated by a conditional jump.
- **Trace**: A sequence of basic blocks terminated by an unconditional jump.

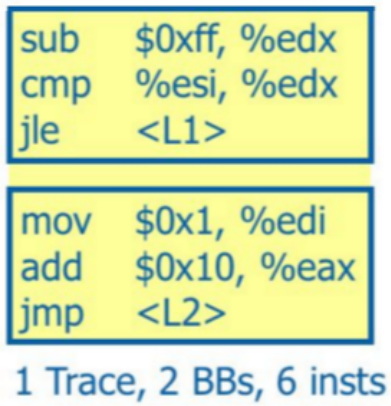

Figure 3.2: Granularity overview

A pintool is essentially composed by two important components:

- **Instrumentation routines**: Callback executed when a trace / instruction is collected statically. They are useful in order to analyze the properties of the code and insert the analysis routines in the appropriate position.
- **Analysis routines**: Functions executed when the instructions placed in the code cache are ran. Accordingly to the position specified in the instrumentation routine, the analysis routine will be executed before or after the current instruction.

After the injection PIN starts to statically build a *trace* from the first instruction of the program to the first instruction representing an unconditional jump. Once this trace is copied on the heap the JIT recompile the code and after that it is ready to be instrumented (see Figure 3.3).

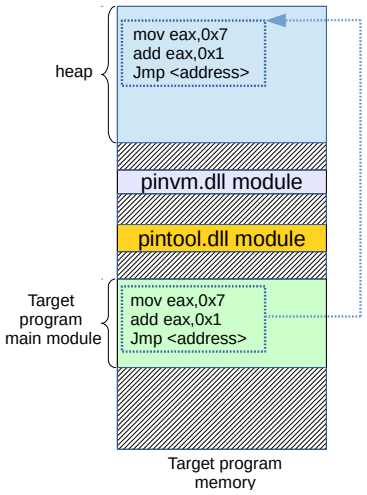

Figure 3.3: PIN collects a trace from the original program code

Now the trace collected is analyzed and modified using the functions that we have specified inside our pintool. First of all PIN passes the control to the *instrumentation routines* which will do all the tasks specified by the user and will patch the trace inserting in the appropriate places the call to the *analysis routines*, if specified. After these operations the code is placed in the *code cache* (see Figure 3.4).

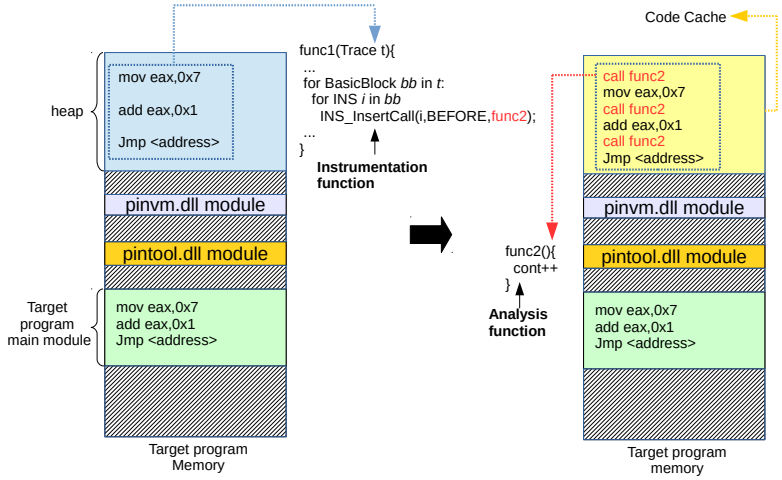

Figure 3.4: Before and after the insertion of the analysis routine

Now the execution will start from the first instruction of the new modified trace until reaching the end of it. When this happens an exception is thrown, PIN catches it, resolves the target address of the execution, recreates a trace and starts again the process explained before.

Note that, during the whole process, the original code (the one belonging to the main image) is never modified or executed by PIN, but it is useful only as a reference to collect traces.

In the Figure 3.5 we can see a diagram that schematically represents the PIN flow just explained.

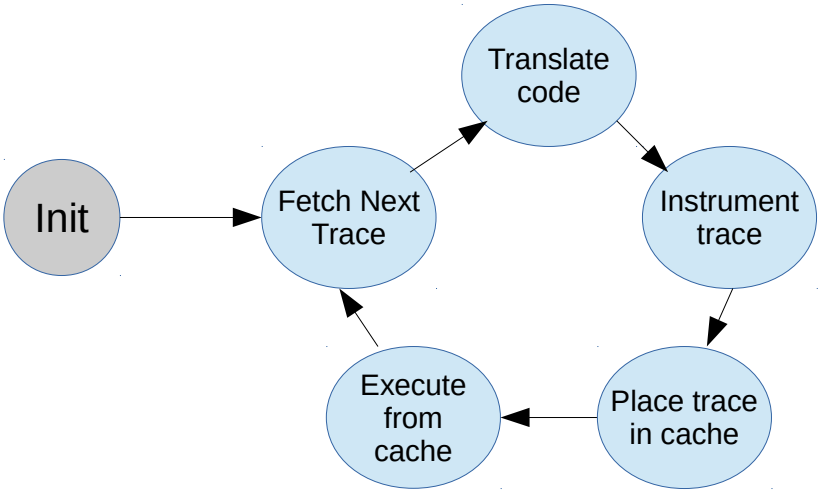

Figure 3.5: Overview of the pin flow

A DBI is a quite complex tool, it not only has to take care of a tricky process as it is a dynamic instrumentation, but also performances are really important in order to avoid to add too much overhead during the execution of the original program. Lots of details have been omitted in the previous explanation that take care of different problems such as the performance. For a complete and better explanation of PIN internal we refer to the official documentation and user's manual [28].

3.1.2 Scylla

There are lots of different tools available to dump a process and reconstruct a correct Import Directory Table such as *ImpREC* [5], *CHimpREC* [1] and *Imports Fixer* [4]. Among these choices we have decided to employ Scylla [38] since it has been recently developed and is currently supported. Moreover this project is open source allowing us to fix some bugs and extend it in order to deal also with *IAT Redirection* and *Stolen API* techniques.
The main features of Scylla are the IAT Search and the Import Directory reconstruction. The IAT Search functionality tries to locate the starting address and size of the IAT in memory employing two different techniques: a basic one and a more advanced one. Since the structure of the IAT is characterized by a set of contiguous addresses of the imported functions, the basic idea behind the IAT Search is to look for *call* instructions whose target addresses point to a set of close memory addresses.

Basic IAT search

The basic IAT search can be summarized as follows:

1. Scylla receives as input a *start address* from which try to search the IAT.

2. The executable page which contains the *start address* is scanned to identify *calls* or *jumps*. Each target address of these instructions is considered as a candidate for a IAT entry pointer.

3. The value contained in the target address of the *call* or the *jump* is check against a set of possible API addresses obtained by enumerating the export functions of the loaded DLLs. The possible IAT pointers are filtered considering only the addresses which pass the check before.

4. The memory is scanned starting from the IAT entry pointers found before until four zero bytes are reached and this is the end address of the IAT. The same approach is used to find the start address by scanning the memory in the reverse way.

Advanced IAT search

The only difference between the basic and the advanced search is that the advanced one searches for IAT pointers through all the executable pages of the program and not only through the executable page of the *start address* given as input. Then, it filters all these pointers in order to eventually remove the invalid ones.

3.2 System details

In this section we are going to explain in detail the implementation of the most important components of our tool.

3.2.1 WxorX handler Module

Within this module we have implemented the core functionality of the entire tool: it detects when an executed instruction has been previously written by the program itself and it triggers all the other modules' features. In order to detect this behaviour two important functionality have been implemented:

- Written addresses tracking: keep track of the memory addresses written by the program
- WxorX addresses notifier: notify when a previously written address is executed

Written addresses tracking

The building block of the written address tracking phase is the *WriteInterval*, a structure that contains the characteristics of a set of contiguous writes: the start address of the write interval, the end address representing the last contiguous written address, a Boolean value that indicates if the write interval has been analyzed yet and all the results of the heuristics applied to the write interval (for more information about heuristic see Section 3.2.6). The set of *WriteIntervals* created is stored inside a C++ vector called *WritesSet*. Using these structures we check each instruction, including writes, to see if it

executes from one of the *WriteIntervals*. If this is the case, then we proceed with our analysis; in the other case we execute the instruction and go to the next one. In both cases the *Write Intervals* are preserved, the reason will be clear in Section 3.2.3.
The whole process can be summarized in Algorithm 1.

Algorithm 1: WxorX handler Module

```
   Input: The instruction currently analyzed ins
1  if isWriteInstruction(ins) then
2  |  updateWritesSet(ins);
3  end
4  if isInsideWriteInterval(ins) then
5  |  TriggerHeuritics();
6  |  DumpAndFix();
7  |  MarkWriteIntervalAsAnalyzed();
8  end
```

The following steps explain how *WriteIntervals* are created and updated:

1. For each instruction we check if it is a write.
2. If so, we insert an analysis routine before it that will retrieve the address where the operation will write and the size of the memory that will be written. With these information we compute the start and end addresses of the write.
3. Now we proceed to the construction or the update of the *WriteInterval*. We have five cases:
   (a) The memory written by the instruction neither is contained nor overlaps with another. *WriteInterval*. In this case we create a new one and add it to the *WritesSet* vector.
   (b) The start address of the write is before the start of a *WriteInterval*, but the end address is inside it. In this case we update the *WriteInterval* setting as start address the start of the write, but leaving unaltered the end address.

(c) The same as case (b), but this time regarding the end address. Consequently, we only update the end of the *WriteInterval*.

(d) The memory written by the instruction completely contains a *WriteInterval*. In this case we update both the start and the end of the *WriteInterval*.

(e) The memory written by the instruction is completely contained by an existing *WriteInterval*. In this case we do nothing.

WxorX addresses notifier

As described in details in Section 2.3.2, the *WxorX addresses notifier* identifies when a written address is executed and it is useful to trigger the dumping process. This happens when the WxorX rule is broken for the first time inside a *Write Interval* or when a jump bigger than a *threshold* is taken inside the same *Write Interval* (*Intra Write Intervals* analysis). In order to establish an acceptable value for the threshold, we did a survey analyzing how we can distinguish the tail jump to the OEP from the regular jumps inside the *Write Interval*, whose results are schematized in Section 4.1.1.

Process Injection Handling

A slightly different approach for executing previously written memory is based on the process injection techniques. A common behaviour of packers consist of hiding the malicious payload by injecting it in a legitimate process. In order to achieve this result the packer need to perform a write operation inside the memory space of the target process and trigger its execution by using thread related functions. PinDemonium is able to identify this behaviour by hooking the system calls used to write inside the remote process memory:

- *NtWriteVirtualMemory*: system call eventually called when executing memory writing functions.
- *NtMapViewOfSection*: system call which maps a view of a section into the virtual address space of a process.

When one of these system calls is invoked Pindemonium keep tracks of the addresses written in the remote process by using an hashmap which maps the pid of the injected process with a list of *Write Intervals*. Then our tools monitor the execution of these addresses by hooking a set of thread related functions commonly used to execute the injected payload:

- *NtCreateThreadEx*: Creates a thread that runs in the virtual address space of another process.
- *NtResumeThread*: Resume the execution of a previously suspended thread
- *NtQueueApcThread*: Adds user defined routine to thread's APC queue. This routine will be executed when thread will be signaled.

When one of the previous system calls is invoked Pindemonium checks if the pid of the process involved is contained in the hashmap which keeps track of remote write operations. If this happens our tool dumps all the memory ranges written in the remote process and executes a set of heuristics on them.

3.2.2 Hooking Module

In the hooking module we have implemented all the hooks of the Windows APIs and system calls we need in order to track the activity of the instrumented program. This is possible by exploiting the PIN's APIs, which permits to insert *callback* functions before or after an API call or a system call.

Windows API Hooks

Windows APIs hooks implemented in our tool are very few since these hooks can be bypassed by malware using direct syscalls. For example, instead of using the *IsDebuggerPresent* API a malware can directly check if the BeingDebugged byte-flag (located at offset 2 in the Process Environment Block (PEB) structure) is True or not.
In order to give the possibility to easily add a new function hook we have

implemented a simple *hook function dispatcher* that consists of 3 main components:

- **FunctionsMap**: This is a configurable list of names of Windows APIs that an user wants to hook.
- **ImageLoadCallback**: This callback is provided by PIN and executed every time a new image of a DLL is loaded inside the process memory.
- **HookDispatcher**: This function is called inside the ImageLoadCallback and receives as parameter the image just loaded. This function has the job to iterate over all the routine (RTN) objects (an abstraction of the functions inside a DLL), in order to check their names against the FunctionsMap: if a match is found, the PIN's APIs are exploited in order to insert a call to a predefined hook for that function whenever it will be called during the execution (see Figure 3.6).

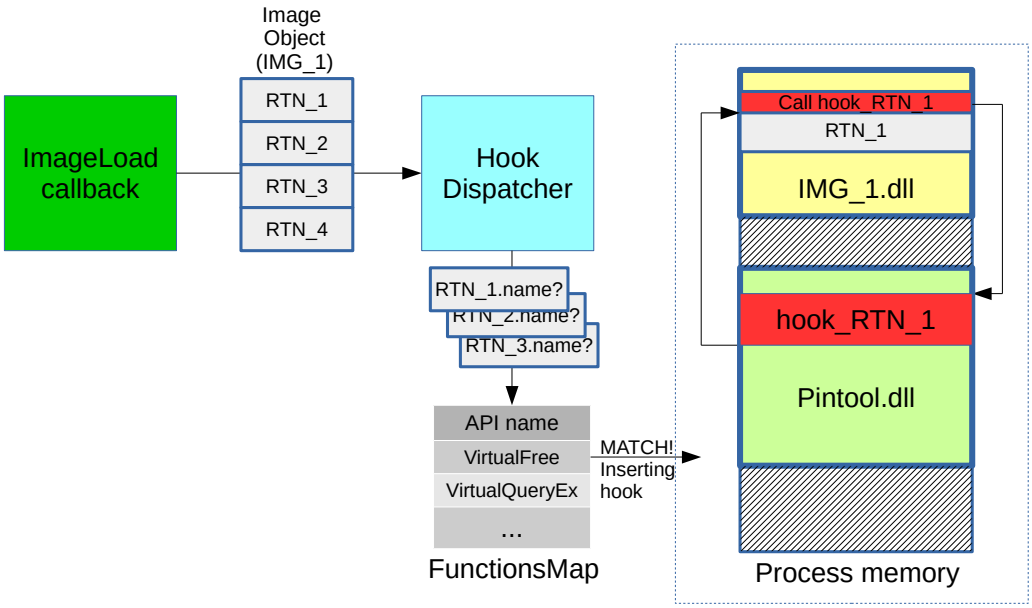

Figure 3.6: The hook function dispatcher places a hook before a routine

We use this hooking functions system principally in order to keep track of code unpacked on the heap for those functions that do not finally use a call to the syscall *NtVirtualAlloc* (hooked as a syscall and explained in the next section).

System calls Hooks

We use syscalls hooking in order to track the binary activity at the deepest possible level. As before, we have implemented an *hook syscall dispatcher* that based on the current called syscall, redirects the execution to the proper hook before or after the system call. This *hook syscall dispatcher* consists of 3 components:

- **SyscallHooksList**: This is a configurable list of names of syscalls that an user wants to hook with the corresponding hook to call. In this list we also embed the information about the moment in which we want to call the hook (before the syscall execution or after that).
- **SyscallEntryGlobalHook**: This is a global hook provided by PIN called at the entry point of every syscall: this function has the job to recognize the syscall called and, using the SyscallHookList, understand if a call to an hook must be triggered or not.
- **SyscallExitGlobalHook**: This is a global hook provided by PIN called at the exit of every syscall: this function has the job to recognize the syscall called and, using the syscallHookList, understand if a call to an hook must be triggered or not.

The *hook syscall dispatcher* is schematized in Figure 3.7.

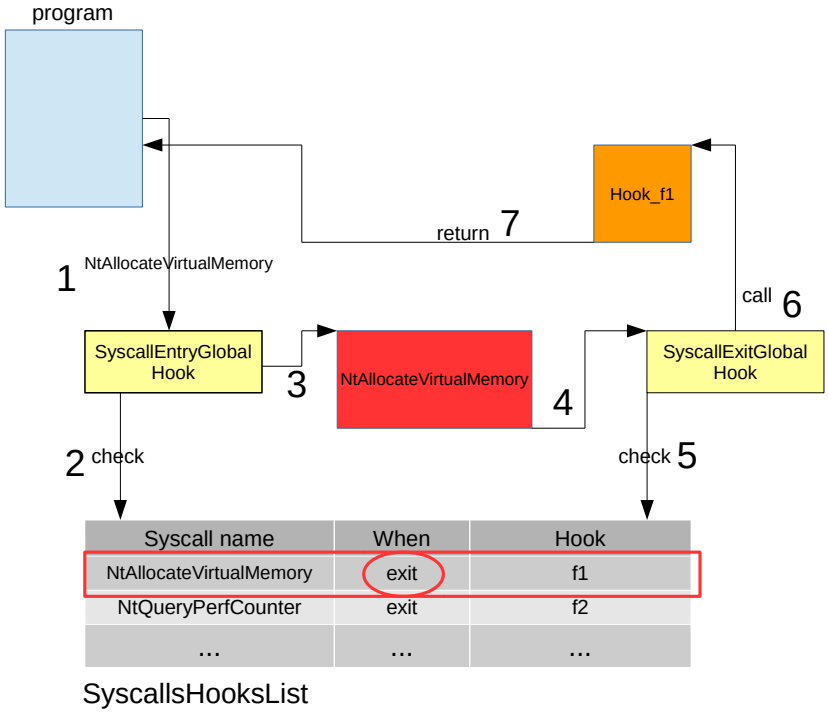

Figure 3.7: The hook syscall dispatcher places a hook before the NtVirtualAlloc

We use this hooking system principally in order to keep track of code unpacked on the heap. Since the Windows APIs that can allocate dynamic memory are numerous, an ad-hoc hook for everyone of them would be a bad solution, rather we have noticed that the majority of them eventually call the *NtAllocateVirtualMemory* syscall and so we have decided to hook this syscall.

The Figure 3.7 shows an example of the functioning of the hooking module, in particular it shows the hook of the *NtAllocateVirtualMemory*.

1. Before the execution of the syscall the *SyscallEntryGlobalHook* is called.
2. It checks if the name of the syscall invoked has an hook bound to it at the *entry* position, if so the hook is executed.
3. The original syscall is executed.

4. After the end of the execution of the syscall the *SyscallExitGlobalHook* is invoked

5–6 It checks if the name of the syscall invoked has an hook bound to it at the *exit* position, if so the hook is executed.

7 The normal execution is restored.

3.2.3 Dumping module

The dumping module, as explained in Chapter 2, relies on the implementation made in Scylla. This takes care of creating the dumps and trying to reconstruct the *Import directory* and the final PE.
As explained in Section 2.3.3, we do not use a naive approach that dumps only the main module of the target program, but rather if the execution comes from a dynamically allocated region we insert into the reconstructed PE a new section containing the data in the write interval in which the EIP was positioned and we finally set the OEP in this new section. In order to do this, we need to track all the heap allocations, and as explained in the previous section, this is the work of the hooking module.

When we find out that an instruction executes from one of the *Write Intervals* the analysis goes through the steps illustrated in Figure 3.8:

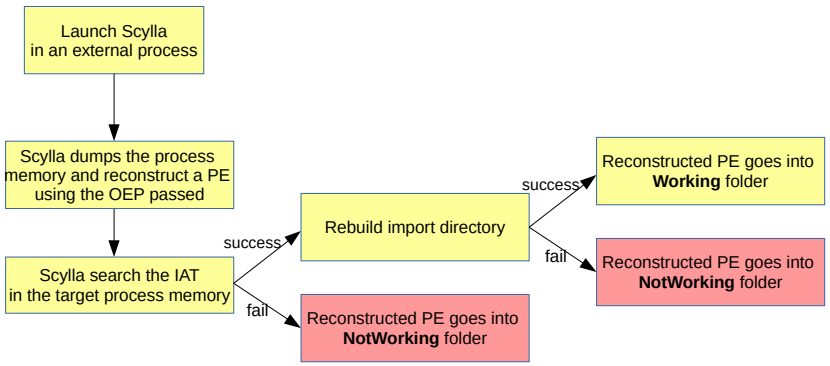

Figure 3.8: Overview of the dumping process

After these steps, if the dump has been triggered by a broken WxorX rule inside an *heap write interval*, the corresponding data in that write interval are inserted as a new section inside the reconstructed PE and the OEP is moved inside that section.

3.2.4 IAT search and reconstruction Module

Our tool uses Scylla, an open source project, that tries to automatically detect and fix the IAT and reconstruct the Import Directory of the dump made (the process of how Scylla works has been explained in the Section 3.1.2). Scylla can detect the IAT in memory using two different techniques: the *Basic search* or the *Advanced search* (explained in details in Section 3.1.2). During our tests we noticed that using only one of these was not sufficient in order to obtain a generic result despite the packer analyzed, so we decided to combine the two techniques together in order to obtain the best outcome possible. The dump taken is marked as 'working' or 'not working' depending on the results of the two search operation and the fix operation accordingly with the Finite State Automata (FSA) of Figure 3.9.

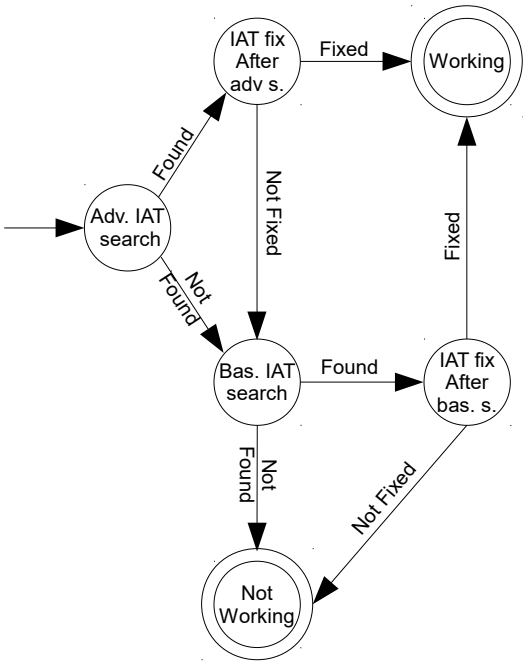

Figure 3.9: FSA IAT search and fix

The original code of Scylla implemented in the IAT fix operation does not take into account any technique of IAT obfuscation such as *API Redirection*, *Stolen API* or its generalization.

3.2.5 IAT Fixing and Import Directory Reconstruction

Due to the lack of an algorithm that can deal with IAT obfuscation techniques in a generic way we decided to implement a system where an analyst can integrate its own deobfuscation routine inside PINdemonium without modifying it. In order to do so we extended the plugin system already present inside Scylla to work even when the tool is used as Dll.

This system gives the analyst two important structure:

- An handle to the process that is beign analyzed. This allow to directly modify the instrumented process from the plugin.

- A list containing all the unresolved imports found by Scylla along with their IAT addresses and the pointed targets.

and two fundamental helper functions :

- *readMemoryfromProcess* : this function gives the possibility to read the memory of the instrumented process at a given address.
- *writeMemoryToProcess* : this function allow to write a local data buffer to the memory of the instrumented process.

To test the proper functioning of this system we decided to implement a custom plugin to defeat the IAT obfuscation technique employed by the packer *PESpin*. In this techniques multiple part of the original API are copied starting from the address pointed by the IAT entry and they are connected together by absolute jumps until the last one reaches the original API code. The process can be better understood following the example illustrated in Figure 3.10.

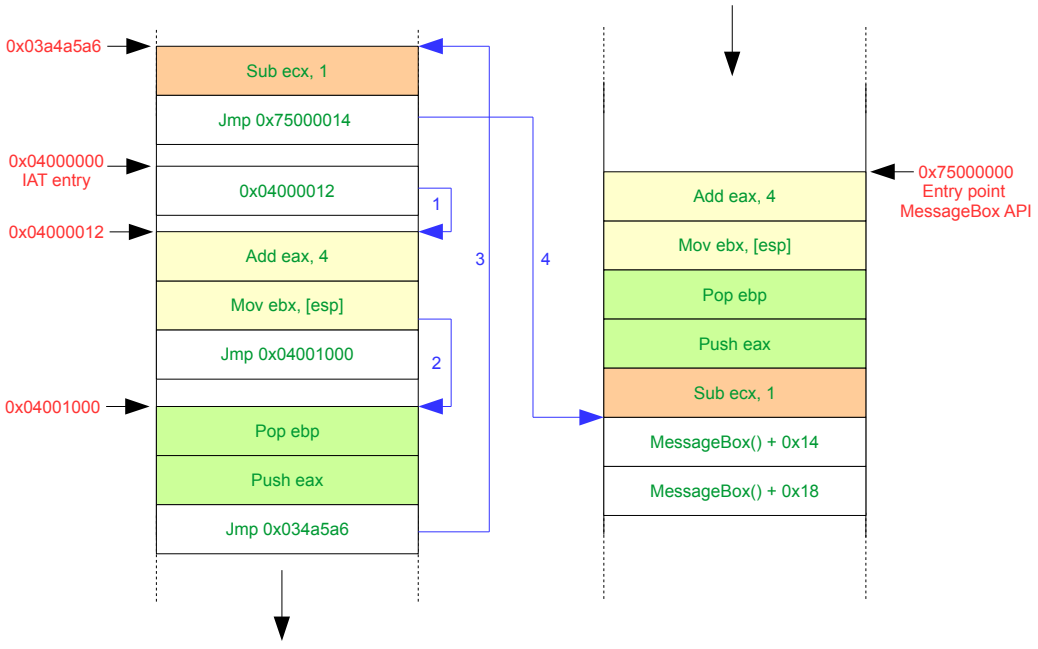

Figure 3.10: Pespin IAT obfuscation technique

1. The execution is redirected to the address inside the IAT entry as usual.
2. The first two instructions of the original API are copied starting from the address pointed by the IAT entry followed by a jump to the next chunk of instructions.
3. The chunk is executed and another jump instruction lead the execution to the next chunk of code.
4. The last instruction copied is executed and finally a jump goes inside the original memory space of the API.

Exploiting the capabilities of our plugin system, we developed an algorithm that can deal with this techniques using static analysis. We made this choice because we noticed that each target of the jump was an absolute address statically known. With this algorithm, for each IAT entry, listed in the list of unresolved import, we statically follow the flow counting the number of instructions different from jumps (instructions belonging to the real API) until the target of a jump is inside a memory region occupied by the DLLs. When the target address and the instructions count are retrieved, the value of the current analyzed IAT entry is patched, using the helper function writeMemoryToProcess, as follow:

$$new\_IAT\_entry\_value = target\_address - instruction\_count \tag{3.1}$$

The whole process can be summarized with the Algorithm 2.

Algorithm 2: IAT deobfuscation

Input: List of *unresolvedImport*. every element has a pointer to the IAT entry to be fixed (*ImportTableAddressPointer*) and its content (*InvalidAddress*)

Output: The IAT correctly deobfuscated and patched

```
1  foreach unresolvedImport do
2      insDelta = 0;
3      invalidApiAddress = unresolvedImport->InvalidAddress;
4      IatEntryPointer =
         unresolvedImport->ImportTableAddressPointer;
5      for j = 0; j <1000; j++ do
6          if not isMemoryMapped(invalidApiAddress) then
7              break;
8          end
9          instructionBuffer = readProcessMemory(invalidApiAddress);
10         disassembledInstruction = disassemble(instructionBuffer);
11         if not isValidInstruction(disassembledInstruction) then
12             invalidApiAddress += 1;
13             insDelta += 1;
14             continue;
15         end
16         if isJmpInstruction(disassembledInstruction) then
17             correctAddress = getJumpTarget(disassembledInstruction);
18             if isInsideDllMemoryRegion(correctAddress) then
19                 correctAddress = correctAddress - insDelta;
20                 writeProcessMemory(IatEntryPointer, correctAddress);
21                 break;
22             end
23             else
24                 invalidApiAddress = correctAddress;
25                 continue;
26             end
27         end
28         else
29             insDelta = insDelta + size(disassembledInstruction);
30             invalidApiAddress = invalidApiAddress +
                 size(disassembledInstruction);
31         end
32     end
33 end
```

3.2.6 Heuristics implementation

We use heuristics in our tool in order to evaluate the obtained dump: each heuristic can set a flag in the final report and all the flags contribute to identify the best dump, as explained later in this Section.
We have implemented five heuristics:

- Entropy heuristic: at the beginning of the analysis, when the main module of the binary is loaded, we get its original entropy value. Each time a dump is created we compute again its current entropy and compare it with the initial one. We use the following formula to compute the difference:

$$difference = \left| \frac{current\_entropy - initial\_entropy}{initial\_entropy} \right| \tag{3.2}$$

  If this value is above a given threshold we set the correspondent flag in the output report. In order to estimate an acceptable value for the threshold we did a survey whose results are schematized in Section 4.1.2.

- Jump outer section heuristic: for each executed instruction we keep track of the EIP of the previous one. In this way we can retrieve the section in which the previous instruction was located and compare it to the section of the current one. If these two are not equal we set the *jump outer section* flag.

- Long jump heuristic: as in the previous case we take advantage of tracking the previous instruction's EIP. In this case we simply compute the difference between the previous and the current EIP:

$$difference = |current\_eip - previous\_eip| \tag{3.3}$$

  If this difference is above a given threshold we set the *long jump* flag in the output report.

- Pushad popad heuristic: during the execution of the binary we have two flags indicating if we have encountered a *pushad* or a *popad*. For every instruction we check if it is one of the previous two and if so we set the corresponding flag. Then, when we produce a dump, if both flags are active, we set the *pushad popad* flag in the output report.

- Init function call heuristic: the aim of this heuristic is to search through the dumped code for calls to functions commonly used in the body of the malware and not in the unpacker stub. We achieve this result by using an IDAPython script: using IDA we are able to read the list of the imports of the dump and to confront it with a list of 'suspicious' functions selected by us. Then we count the number of detected functions and write it in the output report.

At the end of the execution of our tool we have a report which contains a line for each dump that lists the results of every heuristic, as well as the dump number, a string that says if the *Import Directory* is probably reconstructed or not, the OEP, the begin and the end addresses of the *Write Interval* considered for the dump. We use these information to choose the best dump as follow:

- First we check if the *Import Directory* is probably reconstructed.

- If so, we count the number of the 'suspicious' functions detected and the number of active heuristics' flags.

- If the previous numbers are the best result until this moment, we save this dump number, eventually rewriting another one saved before.

- At the end of the procedure we choose the saved dump number as the one that has the greatest chance to work.

If no dump has its *Import Directory* reconstructed, we return the value '-1'.

Chapter 4

Experimental validation

This chapter is organized as follow: in Section 4.1 we show some surveys we did to establish some constants (thresholds) that are used by our tool; in Section 4.2 we show the effectiveness of our tool against known packers; finally in Section 4.3 we show the ability of PinDemonium to deal with unknown packed samples.
The goal of our experiments is proving the generality of the unpacking algorithm and the effectiveness of our tool against packed malware spotted in the wild. The first experiment against known packers aims to show that our tool is able to correctly unpack the same binary packed with different known packers, provided that the adequate flags are active, because they are packer-dependent. The second experiment against random samples aims to show that our tool correctly unpack binaries packed with packers not known in advance or even with custom packers.

4.1 Thresholds evaluation

In this section we are going to explain how we established the two thresholds that our tool needs: the threshold that 'defines' if a jump is long enough to be considered in the *Inter Write Set Analysis* (see Section 2.3.2) and the threshold used in the entropy heuristic to evaluate the disorder of the taken dump with respect to the original program (see Section 2.3.5).

4.1.1 Long jump threshold survey

In order to define the best value for this threshold (see Section 2.3.2) we did a survey analysing how we can distinguish the tail jump to the OEP from the regular jumps inside the *Write Interval*. Compared to the common jumps we have noticed that the tail jump has a bigger length. Moreover, since the number of shorter jumps is higher, choosing a good threshold we can be able to filter a lot of false positives jumps to the OEP without losing the correct one. As we can see from Figure 4.1 we noticed that the length of the jump to the OEP grows about linearly with size of the *Write Interval* in which it is contained and for this reason the threshold has been defined as a percentage over the size of the current *Write Interval*.

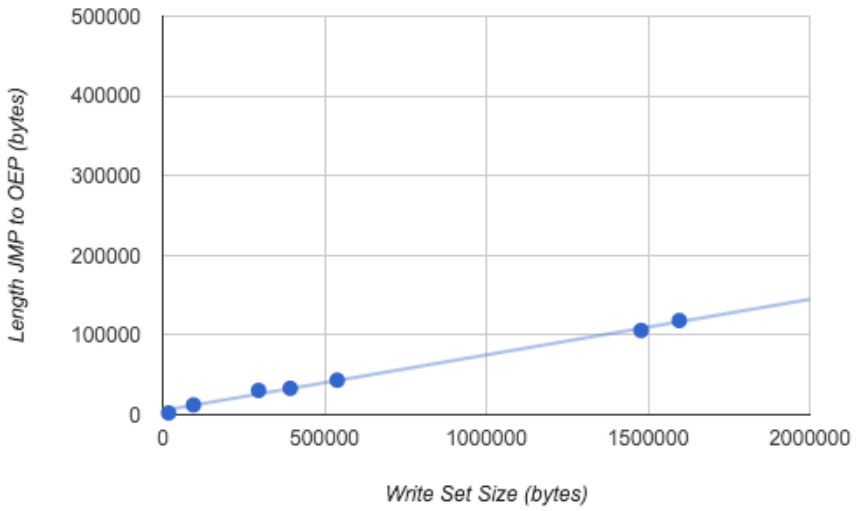

Figure 4.1: Diagram displaying the linear correlation between the length of the JMP to the OEP with the size of the Write Interval in which it is located

In order to determine the best value we packed a set of test programs characterized by different sizes and compared the length of the jump to the

OEP to the *Write Interval* size. The results are summarized in Table 4.1.

Binary	$\frac{\text{OEP jump}}{\text{WI}}$	Write Interval (WI)	OEP jump
write_test	12 %	18258	2301
MessageBox	12 %	95106	12321
7Zip	10 %	296018	30515
PeStudio	8.5 %	394048	33133
Autostarts	8 %	296018	43352
ProcMon	7 %	1478641	105724
WinRAR	7 %	1596762	118207

Table 4.1: Results of long jumps survey

From the results we can conclude that a threshold of 5% of the current *Write Interval's* length is enough to cover all the cases. The maximum number of considered jumps has to be set by command line, as well as the flag that enables the *InterWriteSet* analysis.

4.1.2 Entropy heuristic threshold survey

In order to identify a meaningful threshold used to understand if the current dump is the correct one (see Section 3.2.6), we created a survey analyzing the entropy difference between a known binary and the same binary packed with different packers and crypters. We first calculated both the entropy of the original binary and the entropy of the packed one, and finally we calculated the difference between these two values, trying to derive a constant correlation among the various packers. After this experiment we identify that a deviation of the 40% from the final entropy to the initial one is sufficient to detect the correct dump of almost all packers. The histogram in Figure 4.2 shows the results of our experiment. On the x-axis the various packers employed are listed while on the y-axis the percentage of the difference between the initial and the final entropy normalized to 1 is shown.

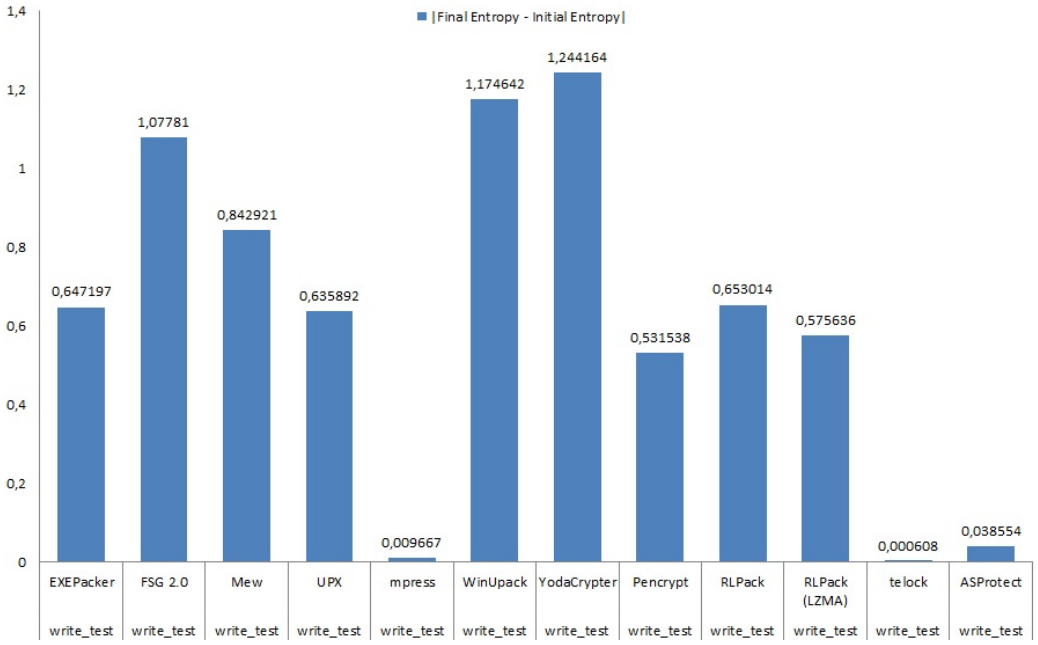

Figure 4.2: Entropy Threshold Survey Results

4.2 Experiment 1: known packers

This experiment has been conducted packing with more than 15 packers available on-line two known binaries with different sizes: a simple MessageBox (100KB) and WinRAR (2MB). This test should confirm that, regardless of the packer employed to pack the same program, our tool used with the correct flags can extract the original program PE. We have decided to use two different programs with different sizes as targets because the behaviour of the packer can be influenced by the size of the program to be packed.
We organized the results in the Table 4.2 using the following columns:

- **Packer**: the name of the packer used.
- **Binary**: the name of the packed program.
- **Flags**: the enabled flags used to unpack the packed binary.

- **Executable**: this field shows if the reconstructed PE is runnable or not.
- **Unpacked**: this fields shows if the original code of the binary is unpacked, not matter if it working or not.
- **Old imports**: This number will represent the number of imports that we can observe with a static analysis of the packed program.
- **Reconstructed imports**: This number will represent the number of imports that we can observe with a static analysis of the reconstructed PE.

Packer	Binary	Flags	E[1]	U[2]	Old imports	Recon. imports/Total imports
Upx	MessageBox	-unp	yes	yes	8	55/55
FSG	MessageBox	-unp	yes	yes	2	55/55
Mew	MessageBox	-unp	yes	yes	2	55/55
Mpresss	MessageBox	-unp -iwae 2 -adv-iatfix -nullify-unk-iat	yes	yes	2	55/55
Obsidium	MessageBox	-unp -iwae 2	no	yes	4	4/55
PECompact	MessageBox	-unp -iwae 2 -adv-iatfix	yes	yes	4	55/55
EXEpacker	MessageBox	-unp	yes	yes	8	55/55
WinUpack	MessageBox	-unp	yes	yes	2	55/55
ezip	MessageBox	-unp	yes	yes	4	55/55
Xcomp	MessageBox	-unp	yes	yes	5	55/55
PElock	MessageBox	-unp -iwae 2 -adv-iatfix	no	yes	2	3/55
Asprotect	MessageBox	-unp -iwae 2 -adv-iatfix	no	yes	7	46/55
Aspack	MessageBox	-unp	yes	yes	3	55/55
eXPressor	MessageBox	-unp -iwae 10 -adv-iatfix	no	yes	14	42/55
exe32packer	MessageBox	-unp	yes	yes	2	55/55
beropacker	MessageBox	-unp	yes	yes	2	55/55
Hyperion	MessageBox	-unp	yes	yes	2	55/55
Upx	WinRAR	-unp	yes	yes	16	433/433
FSG	WinRAR	-unp	yes	yes	2	433/433
Mew	WinRAR	-unp	yes	yes	2	433/433
Mpress	WinRAR	-unp -iwae 2 -adv-iatfix -nullify-unk-iat	yes	yes	12	433/433
Obsidium	WinRAR	-unp -iwae 2	no	yes	2	0/433
PEcompact	WinRAR	-unp -iwae 2 -adv-iatfix	yes	yes	14	433/433
EXEpacker	WinRAR	-unp	yes	yes	16	433/433
WinUpack	WinRAR	-unp	yes	yes	2	433/433
ezip	WinRAR	-unp	yes	yes	12	433/433
Xcomp	WinRAR	-unp	yes	yes	5	433/433
PElock	WinRAR	-unp -iwae 2 -adv-iatfix	no	yes	2	71/433
Asprotect	WinRAR	-unp	yes	yes	16	433/433
Aspack	WinRAR	-unp	yes	yes	13	433/433
Hyperion	WinRAR	-unp	yes	yes	2	433/433

[1] Executable

[2] Unpacked

Table 4.2: Results of the test against known packers

This experiment shows the effectiveness of our generic unpacking algorithm and PE reconstruction system against different kinds of packers and crypters. For some packers as ASProtect, eXpressor and Obsidium we managed to take a dump at the correct OEP, but due to the presence of IAT obfuscation technique not yet handled we cannot reconstruct a working PE.

4.3 Experiment 2: unpacking wild samples

In this Section we are going to explain all the phases of the evaluation of our tool against random samples.

4.3.1 Dataset

We built our dataset with random samples collected from VirusTotal, but since is not available a tag aimed to download only packed programs we have decide to collect a very large pool of malicious PEs with the only constraint that they must be 32 bits binaries. After that we have classified them with the packer detection tool *Exeinfo PE* [12] in order to rule out not packed samples and keep the packed ones. In this set we can find binaries protected by commercial packers and also possibly custom packers tagged by our detection tools as 'Unknown packer'. This is the final set that we have used in order to test the effectiveness of PinDemonium against malicious samples spotted in the wild protected with both known and unknown packers/crypters.

4.3.2 Setup

In order to automatize the analysis we have created a python script on the host machine that using the *VBoxManage* commands does the following steps:

1. Restores the virtual machine used for testing the samples in a clean state.

2. Start the machine and once started launches the sample instrumented with PinDemonium with a time out of 5 minutes.

3. After the timeout expiration or after the program has terminated its execution, the results collected inside the guest are moved in a shared folder on the host in order to avoid losing them in the next restoring process.

4. Closes the guest and returns to point 1.

4.3.3 Results

After the analysis performed by our tool, all the results are finally validated manually in order to classify them in three category:

- **Fully Unpacked**: in this category we have all the samples that has been reconstructed and with a behavior identical to the original one.
- **Unpacked**: in this category we put all the samples that are correctly reconstructed, but with a behavior different from the original one.
- **Unpacked but not working**: here we have all the samples for which we have reconstructed the import directory and have a possible dump at OEP, but they crash when we try to execute the reconstructed program.
- **Not Unpacked**: in this category we put all the samples for which we do not have a reconstructed PE.

We consider two metrics to evaluate our results, the SUCCESS metric take into consideration the sum of Fully Unpacked and Unpacked, while the FULL SUCCESS metric only the Fully Unpacked results. We have decided to introduce the SUCCESS metric because these results can also be used to study the original program besides the fact that they are not perfectly identical to it. In Table 4.3 and Table 4.4 we can find the results obtained by PinDemonium.

Result	Num
Fully Unpacked	519
Unpacked	150
Unpacked but not working	139
Not unpacked	258

Table 4.3: Results obtained by unpacking random samples collected from Virus Total

Metric	%
FULL SUCCESS	49%
SUCCESS	63%

Table 4.4: Metrics about the collected results

As we can see, for 63% of the samples we successfully reconstruct a working de-obfuscated binary, given one packed with a not known at priori packer. In the Not Unpacked category we have all the programs that cause problems to PinDemonium for different reasons, such as:

- evading the virtual environment in which we are instrumenting them;
- detecting the presence of PIN and do not proceed with the unpacking;
- messing with the environment in a way that our script cannot manage to move the collected results from the guest to the host;
- employing IAT obfuscation techniques not handled by the program
- exploiting packing techniques with a level of complexity out of scope (i.e., greater than 4 as specified in the survey in Section **??**)

List of Acronyms

API Application Programming Interface

AV Anti-Virus

DBI Dynamic Binary Instrumentation

DLL Dynamic Loaded library

EIP Extended Instruction Pointer

FSA Finite State Automata

IAT Import Address Table

JIT Just in time

OEP Original Entry Point

PE Portable Executable

PEB Process Environment Block

PID Process Identifier

RVA Relative Virtual Address

TEB Thread Environment Block

VM Virtual Machine

WI Write Interval

WxorX Write xor Execution

Bibliography

[1] *CHimpREC*. https://www.aldeid.com/wiki/CHimpREC.

[2] *DynamoRIO is a Dynamic Instrumentation Tool Platform*. http://www.dynamorio.org/.

[3] *IDA is a Windows, Linux or Mac OS X hosted multi-processor disassembler and debugger*. https://www.hex-rays.com/products/ida/.

[4] *Imports Fixer*. https://tuts4you.com/download.php?view.2969.

[5] *ImpREC*. https://www.aldeid.com/wiki/ImpREC.

[6] Malwr. https://malwr.com/.

[7] *PEiD*. https://www.aldeid.com/wiki/PEiD.

[8] *PinDemonium*. https://github.com/Seba0691/PINdemonium.

[9] *UPX is a free, portable, extendable, high-performance executable packer for several executable formats*. http://upx.sourceforge.net/.

[10] Swinnen Arne and Mesbahi Alaeddine. *One packer to rule them all: Empirical identification, comparison and circumvention of current Antivirus detection techniques*. https://www.blackhat.com/docs/us-14/materials/us-14-Mesbahi-One-Packer-To-Rule-Them-All-WP.pdf.

[11] LRohit Arora, Anishka Singh, Himanshu Pareek, and Usha Rani Edara. A heuristics-based static analysis approach for detecting packed pe binaries. 2013.

[12] A.S.L. *Exeinfo PE*. `http://exeinfo.atwebpages.com/`.

[13] Piotr Bania. Generic unpacking of self-modifying, aggressive, packed binary programs. 2009.

[14] Leyla Bilge, Andrea Lanzi, and Davide Balzarotti. Thwarting real-time dynamic unpacking. 2011.

[15] BromiumLabs. *The Packer Attacker is a generic hidden code extractor for Windows malware*. `https://github.com/BromiumLabs/PackerAttacker`.

[16] Juan Caballero, Noah M. Johnson, Stephen McCamant, and Dawn Song. Binary code extraction and interface identification for security applications. 2009.

[17] Seokwoo Choi. *API Deobfuscator: Identifying Runtime-obfuscated API calls via Memory Access Analysis*.

[18] Kevin Coogan, Saumya Debray, Tasneem Kaochar, and Gregg Townsend. Automatic static unpacking of malware binaries. 2009.

[19] Aldo Cortesi. *Visual analysis of binary files*. `http://binvis.io/`.

[20] Gritti Fabio and Fontana Lorenzo. Pinshield: an anti-anti-instrumentation framework for pintool. Master's thesis, Politecnico di Milano, 2015/2016.

[21] Francisco Falcon and Nahuel Riva. *I know you're there spying on me*. `http://www.coresecurity.com/corelabs-research/open-source-tools/exait`.

[22] Star force technology. *ASPack is an advanced Win32 executable file compressor*. `http://www.aspack.com/`.

[23] Jason Geffner. *Unpacking Dynamically Allocated Code*. `https://github.com/NtQuery/Scylla`.

[24] Fanglu Guo, Peter Ferrie, and Tzi-cker Chiueh. A study of the packer problem and its solutions. 2008.

[25] Hex-Rays. *IDA Pro*. `https://www.hex-rays.com/products/ida/`.

[26] Hex-Rays. *Ida Universal Unpacker*. `https://www.hex-rays.com/products/ida/support/tutorials/unpack_pe/index.shtml`.

[27] Immunity Inc. *Immunity Debugger*. `http://www.immunityinc.com/products/debugger/`.

[28] Intel. *Pin User Manual*. `https://software.intel.com/sites/landingpage/pintool/docs/71313/Pin/html/`.

[29] Min Gyung Kang, Pongsin Poosankam, and Heng Yin. Renovo: A hidden code extractor for packed executables. 2007.

[30] Julien Lenoir. *Implementing Your Own Generic Unpacker*.

[31] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. 2005.

[32] Robert Lyda and James Hamrock. Using entropy analysis to find encrypted and packed malware. 2009.

[33] Milkovic Marek. Generic unpacker of executable files. 2015.

[34] Lorenzo Martignoni, Mihai Christodorescu, and Somesh Jha. Omniunpack: Fast, generic, and safe unpacking of malware. 2007.

[35] mmiller@hick.org. *Using dual-mappings to evade automated unpackers*. `http://www.uninformed.org/?v=10&a=1&t=sumry`.

[36] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. 2007.

[37] NTCore. *CFF Explorer*. `http://www.ntcore.com/exsuite.php`.

[38] NtQuery. *Scylla - x64/x86 Imports Reconstruction.* `https://github.com/NtQuery/Scylla`.

[39] Mario Polino, Andrea Scorti, Federico Maggi, and Stefano Zanero. Jackdaw: Towards automatic reverse engineering of large datasets of binaries. 2015.

[40] Danny Quist. *Circumventing software armoring techniques.* `https://www.blackhat.com/presentations/bh-usa-07/Quist_and_Valsmith/Presentation/bh-usa-07-quist_and_valsmith.pdf`.

[41] Paul Royal, Mitch Halpin, David Dagon, Robert Edmonds, and Wenke Lee. Polyunpack: Automating the hidden-code extraction of unpack-executing malware. 2006.

[42] Isawa Ryoichi, Kamizono Masaki, and Inoue Daisuke. Generic unpacking method based on detecting original entry point. 2009.

[43] Yu San-Chao and Li Yi-Chao. A unpacking and reconstruction system - agunpacker. 2009.

[44] Monirul Sharif, Vinod Yegneswaran, Hassen Saidi, and Wenke Lee. Eureka: A framework for enabling static malware analysis. 2015.

[45] Michael Sikorski and Andrew Honig. *Practical Malware Analysis.* No Starch Press, 2012.

[46] MATCODE Software. *MPRESS compresses PE32 (x86), PE32+ (x64, AMD64) programs and libraries.* `http://www.matcode.com/mpress.htm`.

[47] Xabier Ugarte-Pedrero, Davide Balzarotti, Igor Santos, and Pablo G. Bringas. Sok: Deep packer inspection: A longitudinal study of the complexity of run-time packers. 2015.

[48] Babak Yadegari, Brian Johannesmeyer, Benjamin Whitely, and Saumya Debray. A generic approach to automatic deobfuscation of executable code. 2015.