2015 IEEE Symposium on Security and Privacy

SoK: Deep Packer Inspection: A Longitudinal Study of the Complexity of Run-Time Packers

Xabier Ugarte-Pedrero*, Davide Balzarotti†, Igor Santos*, Pablo G. Bringas*
* DeustoTech, University of Deusto
Bilbao, Spain
{xabier.ugarte, isantos, pablo.garcia.bringas}@deusto.es
† Eurecom
Sophia Antipolis, France
{davide.balzarotti}@eurecom.fr

***Abstract*—Run-time packers are often used by malware-writers to obfuscate their code and hinder static analysis. The packer problem has been widely studied, and several solutions have been proposed in order to generically unpack protected binaries. Nevertheless, these solutions commonly rely on a number of assumptions that may not necessarily reflect the reality of the packers used in the wild. Moreover, previous solutions fail to provide useful information about the structure of the packer or its complexity. In this paper, we describe a framework for packer analysis and we propose a taxonomy to measure the runtime complexity of packers.**

We evaluated our dynamic analysis system on two datasets, composed of both off-the-shelf packers and custom packed binaries. Based on the results of our experiments, we present several statistics about the packers complexity and their evolution over time.

I. Introduction

Binary analysis is a time consuming task that requires a considerable amount of effort even for experts in the field. Malware analysts need to deal with different obfuscation techniques that are commonly employed to hinder static and dynamic analysis, delay the reverse-engineering of the samples, and complicate their detection and classification. Run-time packers, originally designed to reduce the size of executables, rapidly became one of the most common obfuscation techniques adopted by malware authors. They are now used by the vast majority of malicious samples to protect and encrypt their data and code sections – which are then restored at run-time by a dedicated unpacking routine.

Run-time packers have been thoroughly studied in the literature, and several solutions have been proposed for their analysis and unpacking [1]–[6]. Most of these solutions are based on the dynamic execution of the sample (e.g., by an emulator or a debugger) and rely on different heuristics to detect the end of the unpacking routine, and therefore the correct moment to dump the content of the process memory. Other solutions [7] have proposed static analysis techniques to extract the unpacking code. Nearly all antivirus software adopt a more or less sophisticated form of these techniques to provide some form of generic unpacking before applying their signatures and heuristics.

Given the early success of these efforts, the research community quickly moved on – turning its attention to other forms of code protection. For instance, several recent studies have focused on *virtualization*-based protectors [8], [9], which involve a new set of challenges, and stand as a completely separate and still unsolved problem.

Unfortunately, traditional packers are still used by the vast majority of the malware in the wild – and the problem of how to perform runtime unpacking of their code is far from being solved. In fact, traditional solutions rely on a number of assumptions that are not necessarily met by common run-time packers. In particular, they often assume that (i) there is a moment in time in which the entire original code is unpacked in memory, (ii) if a sample contains multiple layers of packing, these are unpacked in sequence and the original application code is the one decoded in the last layer, (iii) the execution of the packer and the original application are not mangled together (i.e., there is a precise point in time in which the packer transfers the control to the original entry point), and (iv) the unpacking code and the original code run in the same process with no inter-process communication. These simplifications make previous approaches unsuitable to handle the real challenges encountered in complex run-time packers. Moreover, while there are several tools and on-line services available for malware analysis, there are no equivalent tools for the analysis of run-time packers. Available generic unpackers rely on heuristics that can be easily evaded, and are often tailored to work only for a specific packer family and version.

This brings us to the first of two sets of questions we want to address in this paper. To begin with, we are interested in understanding the level of complexity of the existing packers that are used to protect malware. *How many of them satisfy the simple assumptions made by existing unpacking tools and techniques? What is the maximum level of sophistication that is observed in the wild? And how many malware families are at this end of the spectrum?*

To achieve this goal we present a new fine-grained dynamic analysis system designed to collect a large amount of information from the execution of packed binaries. The collected data is then analyzed and used to build an unpacking graph and a number of indicators that summarize the features and internal characteristics of the packer.

Our experiments with this tool lead to the second open

DOI 10.1109/SP.2015.46

question we address in this paper. It is well known that the malware writer often relies on off-the-shelf packers to protect and obfuscate the code. Tools like Armadillo, ASProtect, and Yodas are well known both in the underground market and by malware analysts. We used our framework to help us understanding the level of sophistication of these tools, covering over 580 different packer configurations in our experiments. However, there is another side of dynamic unpacking that has never been studied before. In fact, malware writers often decide to avoid existing tools, and implement instead their own custom packing routines.

Recent reports [10] claim that new protection engines are discovered every day. Furthermore, 35% of packed malware is protected with routines designed and coded by the author, avoiding commercial (and thus well-known) packers [11], [12].

How widespread are these custom packing routines? How sophisticated are they compared to the ones adopted by off-the-shelf packers? And finally, how is the packing landscape changing and evolving over the years? Are they becoming more diversified? More complex?

To answer this second set of questions we performed the first longitudinal study of malware packing. Using real malware collected over a period of 7 years, from mid-2007 to mid-2014, we performed a comprehensive evaluation of the complexity of known and custom packers. Our results outline for the first time the evolution and trends of packed malware across the last decade.

To summarize, this paper makes the following contributions:

- We propose a taxonomy for run-time packers to measure their structural complexity.
- We develop a complete framework to analyze the complexity of run-time packers.
- We perform a thorough 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.

The rest of the paper is organized as follows. Section II presents our taxonomy of packer characteristics and the technique we designed to analyze and model their complexity. Section III presents the technical implementation of our framework, that allows to measure the different complexity aspects covered by our taxonomy. Section IV describes several interesting packers we found during this research. Section V describes the longitudinal experiments we performed with our tool to collect information about thousands of custom and off-the-shelf packers. Section VI discusses the results obtained and the implications of our findings. Section VII describes the related work on this topic, and finally, Section VIII concludes the paper.

II. A Packer Taxonomy

The most simple form of run-time packer consists of a small routine executed at the beginning of a program to overwrite a certain memory range with either the decompressed, deobfuscated, or decrypted code of the original application. After the unpacking routine has terminated, the execution is redirected to the original entry point (OEP) located in the unpacked region (an operation often called *"tail jump"*).

More complex packers often involve several layers of unpacking routines, in which the first layer unpacks the second one, which in turn unpacks another routine, until the original code is reconstructed at the end of the chain. Others employ several parallel processes, they interleave the execution of unpacking code with the original code, or even incrementally unpack and re-pack the code on-demand before and after its execution.

To model this entire spectrum of different behaviors, we propose a number of features designed to capture the different aspects of an unpacking process. All these metrics are then combined in a single taxonomy that classifies packers into six classes of incremental complexity.

Unpacking Layers

Previous approaches [2]–[4] have proposed different models to capture the self-modifying behavior that is typical of a runtime packer. All models are generally built around the concept of *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 $\mathcal{L}_1$). The concept of layer, under different names, was already used by some of the generic unpackers proposed in the past (e.g., Renovo [4]), but it was not formalized until Debray et al. [13] first (under the name of *execution phases*), and Guizani et al. [14] later (with the name of *code waves*). Unfortunately, execution phases and code waves were designed to model simple packers, and fail to summarize some of the packer properties present in a large fraction of packers used by malware writers. For instance, an instruction-based shifting-decode packer (see the *Code Visibility* section for real examples of this category) would generate a different "wave" for each instruction of the application.

For this reason, our concept of layers is more conservative than the previous definitions, and it is designed to only capture how "deep" a sequence of instructions is into the unpacking process. More formally, we define an execution layer $\mathcal{L}_i$ as a tuple $(\mathcal{X}_i, \mathcal{W}_i)$, where $\mathcal{X}_i$ is the set of instructions executed at that layer, and $\mathcal{W}_i$ represents the memory addresses modified by those instructions. During the execution of a binary, we maintain a set $\mathfrak{L} = \{\mathcal{L}_0, \mathcal{L}_1, \mathcal{L}_2, ..., \mathcal{L}_{maxl}\}$ where $maxl$ is the innermost execution layer (i.e., the deepest unpacked layer) of the binary. When the program is loaded into memory, there is one single execution layer in $\mathfrak{L}$: $\mathcal{L}_0(\emptyset, \emptyset)$. Intuitively, if an instruction is located at a memory address that has been modified by a different layer, its layer is determined by the highest layer (not necessarily the latest) that modified that area of memory. This may seem counter-intuitive at first. In fact, suppose that a region of memory, before being executed, is first written by layer $\mathcal{L}_4$ and then overwritten again from

layer $\mathcal{L}_2$. This behavior is not rare, especially in multi-layer packers. For instance, every layer may unpack the next one, and then transfer the control back to a previous layer ($\mathcal{L}_2$ in our example) that is responsible for fixing some details in the code or for applying a final de-obfuscation pass. For this reason, we place this new area of memory at layer $\mathcal{L}_5$ and not at layer $\mathcal{L}_3$.

Parallel Unpackers

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.

However, it is important to differentiate between processes involved in the unpacking operation, and processes that are part of the payload (i.e., the original code) of the protected binary. For this reason, in our model we monitor all the processes involved in the execution of a binary, but we only consider that those processes are part of the packer if we observe an interaction among them – if they write to one another address space. In Section III we explain how this interaction can be performed, and how we monitor different system events to capture it.

We also record the number of threads created for every monitored process. As we detail later in this section, the parallel execution of threads has an impact over the complexity of the packer.

Transition Model

A transition between two layers occurs when an instruction at layer $\mathcal{L}_i$ is followed by an instruction at layer $\mathcal{L}_j$ with $i \neq j$. In particular, forward transitions ($j > i$) bring the execution to a higher layer, while backward transitions ($j < i$) jump back to a previously unpacked layer.

In the simplest case, there is only one transition from each layer (typically at the end of the unpacking routine) to the next one (the beginning of the following unpacking routine or the original entry point of the application). In this case, if a packer has N execution layers, there are obviously $N - 1$ layer transitions. In our taxonomy, we refer to this behavior as a *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, we say that it has a *cyclic transition model*.

An important aspect that can affect the transition model is the scheduling of the operating system. For instance, a packer can create two threads which execute in parallel code located in different layers, one for the original code and one for monitoring the execution and introducing anti-debugging routines. In this scenario, we would observe a layer transition for each thread context switch. We classify these types of packers as *cyclic*, since different layers are indeed interleaved in the final execution (intentionally or not).

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. Note that in some complex cases the execution of the packer and the application code are isolated, even though the line that separates the two is located inside a single layer. For instance a packer may eventually unpack a snippet of bootstrap code which resides at the same layer of the original code, and the jump to the original entry point might take place between these two regions located in the same layer. However, since the bootstrap code does not modify the unpacked code (otherwise they would reside in different layers) the last transition can be considered a *tail transition* without losing any generality.

If a packer does not meet the previous definition we consider its execution model as *interleaved*. In an interleaved packer, the execution of certain parts of the unpacking routine is mixed with the original application code. In some cases, this is achieved by hooking the Import Address Table to point to routines in the packer code. This approach can be used to obfuscate the use of API functions by redirecting them through the unpacking code. It is also used by parallel packers to implement anti-debugging and anti-tampering techniques that get regularly executed even during the execution of the original code. Finally, interleaved layers are the basic blocks required to implement multi-frame unpackers.

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 behavior, we introduce the concept of *Frame*. 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* unpackers. 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

As we explained in the previous paragraph, 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. Based on the amount and on the way the original code is written in memory, we can distinguish three types of unpacking schemes:

- *Full-code unpacking.* These routines first unpack all the original code and data, and then redirect the execution to the original entry point. In this case, there is always

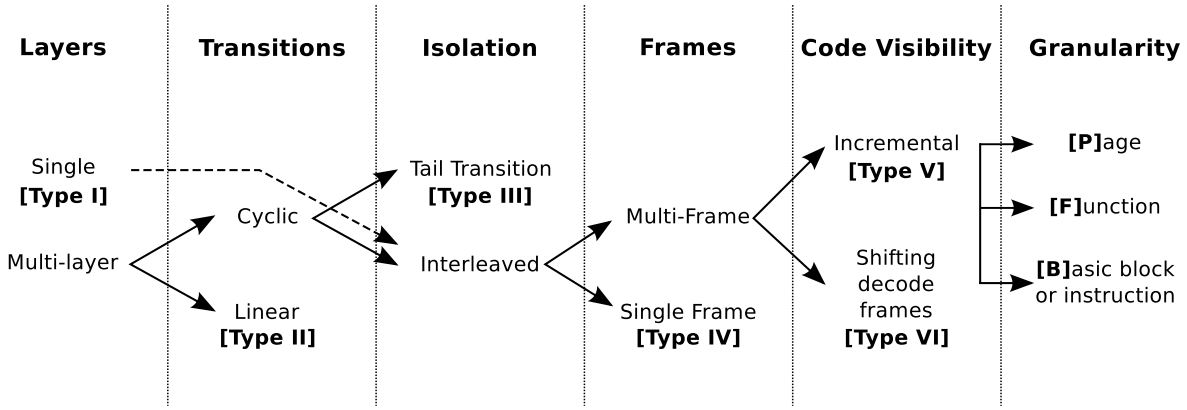

Fig. 1. Packer features and complexity classes

a point in time in which the entire code of the malware can be scanned or retrieved from memory. Single-frame packers always belong to this category.

- *Incremental unpacking.* Incremental unpacking approaches reconstruct the original code on-demand, just before it is executed. In this case, if the content of the memory is dumped at the original entry point of the application, only the first frame of code will be available for analysis. To maximize the amount of collected code, the analyst needs to dump the memory at the end of the program execution. However, if a code path is not executed, the frames that cover that execution path will never be unprotected and will therefore remain hidden to the analyst.
- *Shifting-decode frames.* This is a more complex version of incremental unpacking that involves re-packing each frame of code after its execution. Although this approach is less efficient and may introduce a large overhead, it forces the analyst to extract several memory dumps and join the results in order to reconstruct a more complete view of the original code.

There are several possible solutions to implement incremental and shifting-decode frames unpacking routines. All of them require a way to trigger the packer when a new code frame needs to be unpacked (or re-packed). The following are some of the most common approaches we observed in our study:

- *Exception-based redirection.* A simple approach to redirect the execution back to the packer is to raise an exception. For instance, Armadillo and Beria take advantage of the memory protection mechanisms provided by the operating system to mark memory pages as not executable and then capture the exceptions produced when the execution reaches a protected page. They then overwrite the page with its unpacked content before resuming the original execution.
- *Hooking-based redirection.* Another way to invoke the packer consists of injecting special instructions in the application code to transfer the control to the packer. For instance, ZipWorxSecureEXE replaces original instructions with an interrupt `INT 3` instruction. Whenever the execution reaches the protected address, an exception is generated and the control flow is redirected to the unpacking code, that substitutes the instructions with the original code.
- *Inline packing.* In this case, the code to pack and unpack each frame is inserted directly into the original code. An example of this technique is used by Backpack (an advanced packer proposed by Bilge et al. [15]) which instruments the binary at compile time, using the LLVM framework. Backpack prepends a decryption routine and appends an encryption routine to every region of code that must be individually protected. Themida is another example of this kind of instrumentation. It can be integrated directly into the development environment, allowing the developers to define macros where certain routines of the packer will be placed to protect specific regions of the code. In addition, if this approach cannot be used, Themida also applies binary analysis techniques to discover and instrument functions in the code.

The mechanism adopted to redirect the execution has a large impact on the run-time overhead. For instance, while compile-time instrumentation executes the unpacking code in the address space of the process (without any context switch), exception-based redirection is typically much slower because it requires the operating system to catch and handle the exception each time a new packed block is reached.

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:

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 (either

basic blocks or single instructions).

While fine grained approaches are more difficult to analyze and unpack, they introduce a larger overhead during the process execution. Also, certain packing granularities may require particular instrumentation approaches (such as hooking-based redirection), resulting in an even larger overhead.

Packer Complexity Types

The features we presented so far can be used to precisely characterize the behavior of a packer. In this section, we present a simplified hierarchy to combine all of them together in a single, more concise classification. Figure 1 shows our taxonomy, containing six types of packers with an increasing level of complexity.

[Type I] 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). UPX is a notable example of this class.

[Type II] 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 III] 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 IV] 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 V] 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 analyzed.

[Type VI] packers are the most complex in our 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. A single letter is used to characterize the granularity of Type-V and Type-VI packers. So, a Type-VI-F packer uses the shifting-decode frame technique at the function level.

It is important to highlight that the complexity in this taxonomy is computed with respect to the difficulty of retrieving the original application code. In other words, it would be possible to have a Type-III packer in which one of the intermediate layers (unpacking code) contains multiple frames (e.g., it is decompressed and executed one function at a time). While this feature is captured by our model, the multi-frame part would only be relevant if the analyst is interested in retrieving the entire code of the packer itself. However, since the focus of a malware analyst is typically to study the packed application, our type-based taxonomy would consider this case equivalent to any other Type-III packer.

Finally, all the presented types of packers can be implemented either in a **sequential** or in a **parallel** fashion. Therefore, it is possible to have, for example, a *"Type-I packer with 4 threads"* or a *"Type-III packer with 5 layers and 2 processes"*.

III. Implementation

Our run-time packer analysis framework is implemented on top of one of the two main components of the Bitblaze project [16]: TEMU, a dynamic analysis tool based on QEMU, which provides an interface to monitor the execution of one or several processes. Our framework consists of 6,000 C/C++ and 2,000 python lines of code.

A. Execution tracing

In order to trace the execution of a binary at a basic block level, we leveraged the binary tracing capabilities present in TEMU – that we extended to properly handle interrupts and exceptions. We then implemented our own set of monitoring techniques to deal with complex run-time packers that employ several processes and inter-process communication.

Our framework is able to track many different techniques that can be used by two processes to interact, including remote memory writes, shared memory sections, disk I/O, and memory-mapped files. It also monitors memory un-mapping and memory deallocation events. In fact, a section un-map or memory free operation on an unpacked region of code can be considered equivalent to overwriting its memory content, since the data that was previously available is not accessible any more. For example, some packers apply page-level protection to their code, mapping a memory page whenever it is needed, and un-mapping it afterwards. To deal with these cases, our framework considers this second step equivalent to re-packing the memory page.

B. Collected information

Apart from the instruction trace and inter-process communication events described above, our system collects many other

information useful to evaluate the complexity of a packer. For instance, for each layer we record the memory type (*module*, *heap*, or *stack*) by analyzing the Process Environment Block (PEB) and the Thread Environment Block (TEB) for each thread executing in each process of the packer, and monitoring several memory allocation API functions such as `ZwAllocateVirtualMemory`. This information is useful for the analysis of the binary, since common applications do not execute code from regions in the heap or stack, while some packers use this kind of memory to place unpacking or (de)obfuscation routines.

We also record every API function called by each layer. This allows an analyst to easily locate important events, such as the use of initialization functions like `GetCommandLine`, `GetVersion` or `GetModuleHandle` – a very common heuristic employed to detect when the execution reaches the original entry point of a binary.

Run-time packers commonly obfuscate the use of API calls to (i) avoid the standard use of the Import Table of the PE file, and (ii), to complicate the reverse-engineering task, hindering the reconstruction of an unpacked and fully functional binary. One of the most common methods employed by packers is to erase the Import Table, and to reconstruct it before the execution of the original code making use of the `LoadLibrary` and `GetProcAddress` functions. In this way, the packed binary is built with an alternative Import Table that declares a different set of functions, or no function at all. Nevertheless, in most of the cases, the original code still uses the same mechanism to call the API functions, which consists in making indirect calls to addresses stored into the Import Address Table, or into other regions in memory that contain indirect jumps to addresses stored in such table. In order to detect potential Import Address Tables in the binary, we instrument the execution of indirect call and jump instructions in the dynamic binary translation routine of the emulator. Once the execution of the binary is terminated, we identify potential tables by grouping the memory addresses used in indirect control flow instructions.

Finally, for each layer we compute the sets of modified and executed memory regions. Additionally, since we record the memory type for every execution block, we label every executed memory region accordingly.

C. Post-processing and Trace Analysis

The instruction trace extracted during the packer execution is automatically analyzed to extract different types of information. In particular, to compute the number of frames for each unpacking layer we define a shadow memory that covers the address space of each layer.

The shadow memory maintains two pieces of information for each byte: its current *State* and an additional *New Frame* bit (NF). The state can be modeled as a finite state machine in which each byte is in one of the following states:

- **Unknown** (O) Indicates the initial state of the memory.
- **Executed** (X) Indicates that the memory has been executed, without being previously written (this can only be true for the first layer of the packer).
- **Written** (W) Indicates that the memory has been written but not yet executed.
- **Unpacked** (U) Indicates that the memory has been executed after being in the Written state.
- **Repacked** (R) Indicates that the memory has been overwritten after being in the Unpacked state. This may sound incorrect, since not all overwritten code is necessarily repacked. However, we will discuss later in this section how we distinguish between repacked code and just new code prepared for execution.

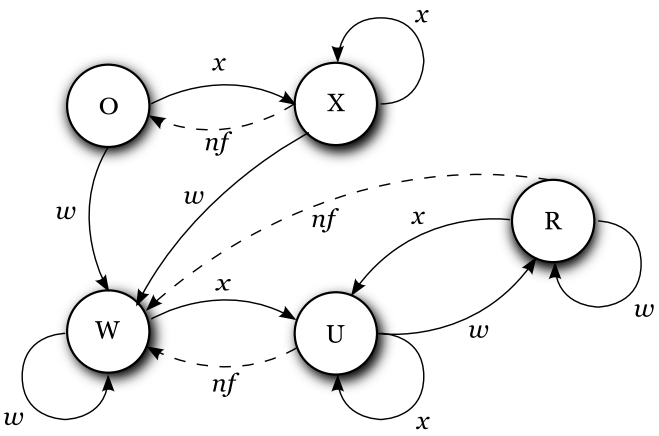

Fig. 2. Finite state machine representing the memory state for each byte.

For every byte executed at layer $\mathcal{L}_j$, there is an execution transition (x) in the finite state machine associated to $\mathcal{L}_j$. Moreover, for every byte written in the address space of layer $\mathcal{L}_j$, there is a write transition (w) in the finite state machine associated to $\mathcal{L}_j$. Figure 2 shows the complete state machine.

The NF bit in the shadow memory is set when a byte has been modified during the execution of the last frame. When there is a transition to the *U* state for a memory region with the NF bit set, we consider that a new frame has been created and we clear the NF bit for every address in the shadow memory. The next frame will not be computed until new writes for a region or layer are followed by an execution of those memory addresses.

Whenever a new frame is created, the state of each memory location is also updated (see the state machine in Figure 2 for more details). In particular, all the locations in the shadow memory in the *U* and *R* state are updated to the *W* state and the bytes in the *X* state are updated to the *O* state. These transitions have a very important implication. Since the unpacked bytes (*U* state) are transformed into written bytes (*W* state), any further write to these bytes will not transform them into repacked bytes. In contrast, these bytes will be considered unpacked memory of the next frame. This is necessary to consider as repacked only the memory which is overwritten *before* starting the next frame of execution, (following strictly the description provided for the *shifting decode frames* technique, detailed in Section II). In fact, it

is possible for a packer to reuse the same memory region as destination to contain the unpacked code of the next frame. In this case, when the code is overwritten it is not repacked, but just prepared for the next layer of execution.

Following the method we just described, once the execution is finished, our system is able to automatically extract the number of execution frames for each unpacking layer.

D. Packer Visualization

Although it is well known that data visualization techniques can help humans in the analysis of an unknown binary, few solutions are focused on the analysis of run-time packers. Vera [17] allows to represent graphically the execution trace of a binary at the basic block level. Unfortunately, it is not clear if this granularity is useful in presence of very complex packers that involve multiple unpacking layers and the interaction between several processes.

To solve this problem, we propose to combine a precise, fine-grained monitoring with a coarse-grained visualization of the execution of the binary in order to provide the analyst a precise but interpretable source of information for the reverse engineering task. The graph generated by our tool displays the different processes and execution layers in each process. Figure 3 shows an example of Obsidium 1.2, a moderately complex packer. Our approach represents every executed memory region (nodes in the graph) enriched with different information such as the memory type ((M)odule, (S)tack, (H)eap, (N)one) and address (first line in each node), size (second line in each node) and number of unpacking frames (third line of the node). These nodes are represented with different colors for a faster identification. The nodes containing instructions that have written some code are displayed in gray. There is a single node painted in red, and it contains the last instruction executed in the binary before the end of the analysis. In case it is part of a system library, we consider the last caller represented in the trace. Nodes containing memory written by another process are painted in green. The rest of the nodes are displayed in yellow.

Also, for each layer we present the total number of frames. Finally, edges in the graph represent transitions and write operations: in red for execution transitions to written memory regions, in green for memory writes to regions that have been executed (i.e., unpacked), in gray for execution transitions, and in blue for interprocess transitions occurred just after an interprocess memory write. While the example in Figure 3 is still relatively small, a packer like Armadillo generates graphs containing hundreds of different nodes and transitions.

E. Complexity Analysis

The last step in the analysis of a packer is to measure its complexity. First, our system extracts the values for all the features mentioned in Section II. With this information it is possible to precisely distinguish, in an automated fashion, between packers of Type-I, Type-II, and Type-III and between Type-V and Type-VI. However, to properly distinguish Type-IV from the adjacent complexity classes we need to be able to clearly separate the code of the packer from the code of the original application. Unfortunately, this task does not always have a complete and sound solution.

In fact, one may intuitively think that the application code would always reside in the deepest layer. However, our experiments confirm that this is not necessarily true. There are cases in which the packer extends the original application code with some special routines. In this case, both the application and the packer code reside in the same layer and, if they are mangled together, it is practically impossible to tell them apart.

For this reason, we implemented a set of heuristics, based on the assumption that the original application code and the unpacking routines are not mangled together. It is possible for them to be co-located in the same layer, but only if they are not in contiguous areas of memory. In other words, it is possible for a certain code to unpack part of the original application and some additional unpacking routine that will later unpack the remaining parts. However, these two distinct functionalities cannot be located in adjacent memory pages but need a minimum distance between one another (a threshold that can be configured, and was set to 10 pages in our experiments). Otherwise, it would be impossible to separate them and our system would flag both of them as part of the packer. Under this assumption, we can safely classify any code that performs write operations to the memory of another layer as belonging to the packer – while the remaining code is temporarily flagged as potential candidate for the application code.

Separating Type-III from Type-IV:

Once we discriminate between the original and the packer code, distinguishing between interleaved and cyclic packers is simple. Starting from the end of the instruction trace, we move backward and consider the transitions between the original code and the packer code. If we only find a transition from the unpacking routine to the original code (i.e., tail transition), then the packer is considered cyclic (Type-III), otherwise it is interleaved (Type-IV or higher). The only uncertain case happens when all the code is flagged as belonging to the packer. This means that the original code and some unpacking routine are located at the same layer and stored in memory at a distance closer than our threshold. In this case, it is not possible to distinguish if the packer and application code are interleaved or not. Therefore, for the lack of evidence, we assume that the packer belongs to Type-III.

Separating Type-IV from Type-V and Type-VI:

If the majority of the code identified as potentially belonging to the original application contains multiple frames, we conclude that the packer belongs to either Type-V or Type-VI. To tell the two classes apart, we analyze the unpacked and repacked regions overlapping the original code. If there are repacked blocks, we consider the sample as shifting-decode-frames. Otherwise, we consider it is incremental.

Finally, we consider the size of the memory written on the unpacking frames. If the majority of the frames present an unpacked size multiple of 4K (the size of a page), we consider the packer to have a page-granularity. If the average

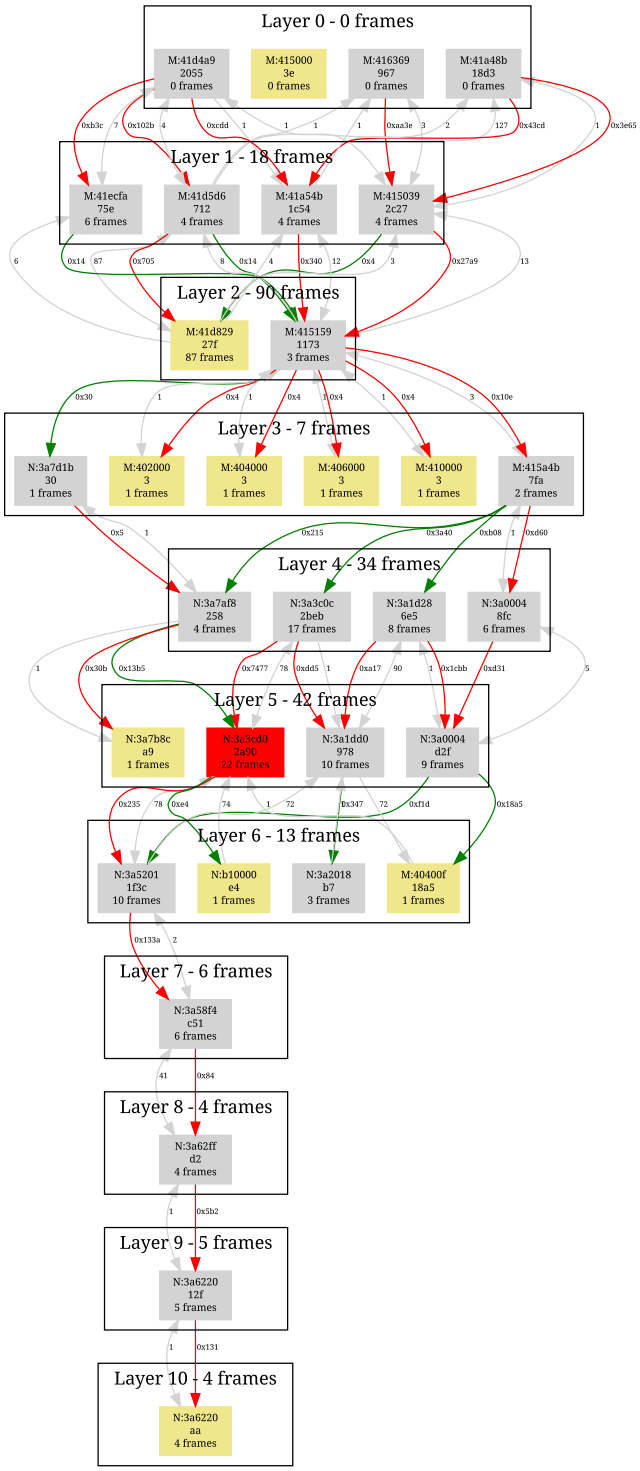

Fig. 3. Graph generated for Obsidium 1.2, a Type-IV packer with several threads, interleaved execution, and a multi-frame unpacking routine. The original code, located at layer $\mathcal{L}_6$, starts at address `0x0040400f`, is `0x18a5` bytes long and is unpacked in one single frame (i.e., it is fully unpacked before being executed). The last executed instructions belong to the unpacking routine (red node at layer $\mathcal{L}_5$).

number of basic blocks in each frame is one, then we assign a basic-block granularity. In any other case (e.g., when the size of the unpacked frame is always different) we mark that as a more generic *block* granularity. This category includes the packers that unpack one function at a time, as well as those that unpack one functionality at a time (for instance, based on the command they receive from their C&C server).

IV. Case Studies

During our experiments, we found many different types of packers that attracted our attention. Moreover, the development of the proposed taxonomy and the described run-time packer analysis framework are the result of an iterative process of analysing, representing and understanding packer structures.

In this section we describe the characteristics of three interesting run-time packers that belong to different complexity classes: UPolyX, ACProtect, and Armadillo.

UPolyX 0.4: A Type-III packer

UPolyX is a well-known UPX scrambler which obfuscates a binary already packed with UPX[1]. The decryption engine of this packer is polymorphic. Part of the unpacking routine of this packer is located at layer $\mathcal{L}_0$, while another part is decrypted at run-time, and therefore located at layer $\mathcal{L}_1$. When the execution begins, the control flow jumps from layer $\mathcal{L}_0$ to layer $\mathcal{L}_1$ back and forth while it unpacks the original code. Interestingly, the different parts of the original code are decrypted alternatively by the unpacking routines located at layer $\mathcal{L}_0$ and layer $\mathcal{L}_1$. As a consequence, the original code is split in two layers ($\mathcal{L}_1$ and $\mathcal{L}_2$) generating many transitions between these two layers at run-time. This packer is classified by our taxonomy as a Type-III packer. It presents a cyclic transition model not only in the unpacking routines, but also in the original code. Nevertheless, it has a clear tail-transition – both parts of the code are not interleaved. Also, there is an interesting aspect about this structure. Layer $\mathcal{L}_1$ contains part of the unpacking routine, and part of the original code. According to the definition of unpacking frame in our model, this layer is unpacked and executed in two different times. At t_0, the packer reveals part of the unpacking routine. At t_1, the unpacking routine, located in layer $\mathcal{L}_0$ and layer $\mathcal{L}_1$ decrypts the original code (that will be assigned to layer $\mathcal{L}_1$ or layer $\mathcal{L}_2$). At t_2, the tail-jump occurs, and the original code starts executing at layer $\mathcal{L}_1$ and layer $\mathcal{L}_2$. Consequently, the code at layer $\mathcal{L}_1$ is written at t_0, executed and modified at t_1, and then these last modifications are executed at t_2. Layer $\mathcal{L}_1$ is unpacked in two frames.

Despite of this elaborate behavior, the packer is still a Type-III packer because (i) there is a clear tail-jump, and (ii) the unpacking frames do not affect the visibility of the original code. Furthermore, from an unpacking point of view, it is fairly easy to find the original entry point.

[1]http://blog.trendmicro.com/trendlabs-security-intelligence/some-bits-about-upolyx/

ACProtect 1.09: A Type-IV packer

ACProtect is a complex protector that incorporates a metamorphic engine, several layers of encryption, and the ability to interleave the original code with the packer code in order to achieve a higher degree of protection. All these obfuscations complicate the task of recovering a clean version of the original code.

We analyzed a sample protected by this packer using our framework, and found that it contains up to 216 layers of code. Surprisingly, the original code is present at the second layer, while the rest of the layers contain obfuscated self-modifying routines that belong to the packer code.

In order to interleave the execution of both types of code, the packer performs IAT hooking. When the original program calls to certain API functions, it performs an indirect jump through the Import Address Table. The addresses corresponding to these functions are replaced by an address pointing to the packer code. Each time one of these APIs is called, the packer will take control of the program executing some anti-debugging routines. Then, it will redirect the execution to the called function, finally returning to the original code.

This is an example of a moderately complex Type-IV packer that interleaves the execution of the protected code and the packer code.

To summarize, this protection scheme presents several characteristics: (i) the original code is not located in the last layer, (ii) the last executed instruction is not part of the original code, and (iii), the interleaving of both types of code can complicate the task of finding the original entry point.

Armadillo 8.0: A Type-VI packer

Armadillo is a well-known protector that implements numerous anti-reverse-engineering techniques. This packer is commercialized as a tool to protect legitimate software and allows the user to precisely configure the desired protection level. One of the options available is `CopyMem-II`, which produces Type-VI protected binaries.

This packer employs two separate processes during the unpacking procedure. This scheme is implemented both as an anti-debugger technique (avoiding another process to attach to the child process) and to intercept the execution of the original code, unpacking new frames on demand.

When the process starts, it first creates a child process and attaches to it. Then, the child process starts executing until it reaches the original entry point of the binary. The permissions of the memory pages in the module address space of this binary are modified to trigger an exception when they are executed. This exception is captured by the parent process, which writes to the debugged process memory using the `WriteProcessMemory` function.

Finally, the packer resumes the execution of the child until it reaches another protected page. Using this technique, Armadillo manages to reveal only one frame of code at a time. In this multi-process execution, the code of the packer (parent process) is interleaved with the execution of the original code (child process). Nevertheless, since the original code is

unpacked and executed in different frames (i.e., at different times), Armadillo is a clear example of a Type-VI packer.

During our experiments, we found samples protected by Armadillo that expand 2 processes with up to 100 layers (in the case of the parent process) and 102 layers of code (in the case of the child process). In both cases, the first 99 layers corresponded to relatively small obfuscation routines. The original code was located at layer $\mathcal{L}_{101}$ in the child process, which also contained part of the unpacking routines. Several of the layers in both processes were unpacked in different frames. Our approach allowed us to automatically identify the original code, observing that it was unpacked and repacked in different frames with a page granularity.

This structure has several implications: (i) the original code is not contained in the last layer, (ii) the code of the packer and the original code are interleaved in a multi-process scheme, and (iii), the original code only presents one visible frame of code each time.

V. Longitudinal study of the Complexity of Run-time packers

Our approach allows us to measure the complexity of run-time packers based on the information collected in our analysis platform. In our experiments we use our system to study two different datasets: (i) a set of off-the-shelf packers, and (ii) a set of malware samples packed with custom techniques.

The off-the-shelf packers dataset contains 685 samples, covering 389 unique packers. Some popular packers are present multiple times with different versions of the tool or configured with different parameters/options to obtain different packing behaviors. The second dataset was instead extracted from the samples submitted to the Anubis malware analysis sandbox. It contains malicious samples (recognized as such by at least three antivirus products) that had at least a PE section with entropy higher than 7 but that were not recognized as packed by Sigbuster, PEiD or F-Prot. The idea behind this dataset is to represent malware binaries which adopt custom unpacking routines, often as part of a polymorphic engine or of a protection/obfuscation layer. More concretely, we retrieved over 1000 samples per year, between 2007 and 2014. The samples are equally distributed for each month of the year, based on their submission time. Moreover, to avoid biasing the dataset towards very common polymorphic families such as the Zeus botnet, we ensured that the dataset did not contain more than one sample per malware family per month (the monthly-based time window allows us to catch the evolution of large malware families that adopted different packing techniques over time).

A. Analysis Infrastructure

We analyzed every sample in our framework using 20 virtual machines configured with two CPUs and 4 GB of RAM each. The analysis of the samples was automated and each sample was run until one of the following condition was satisfied:

- All the processes under analysis terminated their execution.
- An exception was produced and not recovered within two minutes. In order to detect exceptions, we monitored the execution of the `KiUserExceptionDispatcher` function in the `ntdll.dll` system library.
- The monitored processes were idle for more than two consecutive minutes.
- A maximum time-out of 30 minutes was reached.

TABLE I
Summary of the packer complexity of the studied samples.

Type	Off-the-shelf	Custom packers
Type I	173 (25.3%)	443 (7.3%)
Type II	56 (8.2%)	752 (12.4%)
Type III	352 (51.4%)	3993 (65.6%)
Type IV	86 (12.6%)	843 (13.8%)
Type V	6 (0.9%)	46 (0.8%)
Type VI	12 (1.8%)	11 (0.2%)

B. Analysis of Off-the-shelf Packers

The first interesting result we observed in the collection of off-the-shelf packers is the fact that 559 of them (81.6%) are single-process, 121 (17.6%) use two concurrent processes, and only five adopted more than two. The number of layers is summarized in Figure 4. While the majority of packers adopt less than four layers, there is a significant number of packers (4.4%) that use more than 50 layers. Even more interesting, almost 10% of the packers in this dataset did not have the protected code in the last unpacked layer. In these cases, the last layer contained part of the routines of the packer, only revealed at run-time.

Regarding the number of transitions to shallower layers, one third of the packers did not present any cycle (and thus, presented a linear transition model). At the other end of the scale, for 15 packers we observed more than 1 million backward transitions.

Finally, Figure 5 shows the prevalence of the different interprocess communication techniques observed for this group of packers. In this dataset, we did not find any sample using shared memory mapping, writing to shared memory regions, or deallocating memory of unpacked regions. The most common interprocess interaction techniques are the creation of shared files (that are afterwards executed), and the use of the Windows API (`ReadProcessMemory` and `WriteProcessMemory`) for process interaction. Five samples were reported to use injection techniques to processes not directly created by the process itself. We can also notice that a significant number of samples created several processes or threads during their execution.

C. Packers Distribution

In the previous section we have analyzed the complexity of different off-the-shelf packers. In order to understand the relevance of these results, we obtained access to the Anubis malware sandbox database (containing over 60 Million unique

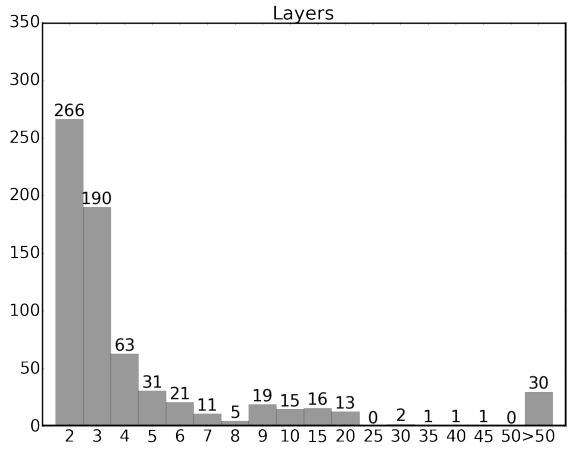

Fig. 4. Number of layers of the packer.

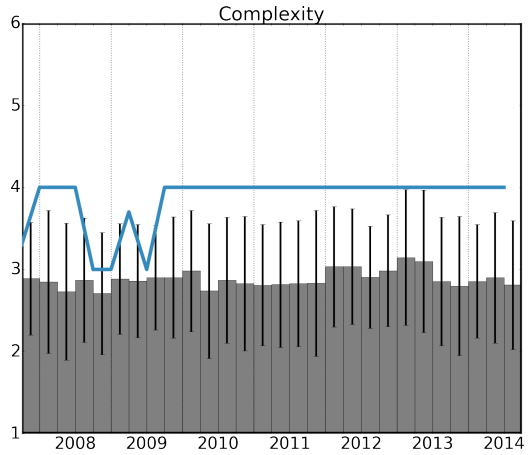

Fig. 6. Average complexity used by custom packers over time.

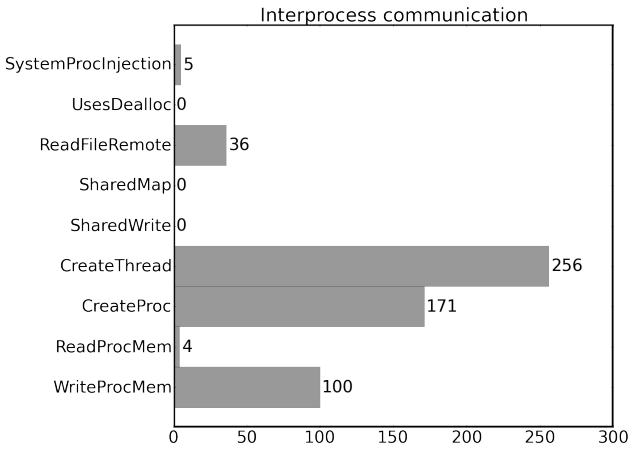

Fig. 5. Interprocess communication techniques observed.

samples) to measure the number of malware samples protected with each packer.

Table II shows the distribution of the most common packers over the years according to Sigbuster, the signature-based packer detection tool present in the sandbox. The table also presents the highest complexity observed among the different packer versions tested during the experiments. Since the tool is capable of identifying the polymorphic protection engines adopted by famous worms, we indicate in the first row the percentage of those samples. The second row of the table shows the percentage of samples which are packed with an off-the-shelf packer. This set is divided into different categories in the lower part of the table, one row for each of the 9 most prevalent packer tools in the database. Finally, we report the percentage of well-known packers not listed in the table (Others), the percentage of packers labeled by the tool as *Unknown* packers, and finally the percentage of tools labeled as Windows installer tools.

We can observe that the number of samples detected as packed with well-known packer tools significantly decreased over the years. This trend might be the consequence of the fact that the packer signature database is not up-to-date with recent packer families, or that malware writers now prefer to employ their own custom unpackers or to rely on simple packers (such as UPX) that do not raise suspicion on their programs. With the exception of Themida, a complex protector that uses virtualization technology, the most commonly employed packers are relatively simple, ranging from Type-I to Type-III.

D. Analysis of Custom Packers

The packers analyzed in the previous section are well-known tools that can be recognized by signature-based detection tools such as PEiD or Sigbuster. Nevertheless, a significant number of malware samples are protected by custom protection engines in an effort to complicate reverse engineering.

To study the complexity and the characteristics of these packers we run our analysis tool on 7,729 malware binaries uniformly distributed over the past seven years (based on the date they were first submitted to the public analysis sandbox). Despite all having a section with entropy higher than 7, only 6,088 samples presented an unpacking behavior during our analysis. Table I shows the packer complexity classes in both datasets, off-the-shelf and custom packers. Custom packers show a higher prevalence for Type-II and Type-III, while very simple packers (Type-I) and very complex ones (Type-VI) are more common in the off-the-shelf dataset. Table III shows the evolution of custom packer complexity over the years, and Figure 6 summarizes it by plotting the average packer complexity, together with its standard deviation and the 90^{th} percentile of the distribution. The data shows no clear trend, with all the complexity classes remaining roughly constant over the past eight years.

TABLE II
DISTRIBUTION OF KNOWN PACKERS OVER THE YEARS

	2007	2008	2009	2010	2011	2012	2013
Polymorphic worms	11.75%	4.58%	4.01%	2.56%	0.80%	0.73%	0.06%
Packed samples	14.98%	17.35%	9.54%	14.59%	17.94%	15.31%	1.25%
UPX (Type-I)	29.99%	49.15%	56.78%	41.45%	52.69%	55.26%	55.38%
PE Compact (Type-III)	6.83%	5.52%	3.02%	4.70%	8.84%	6.26%	3.67%
Aspack (Type-III)	3.81%	4.55%	5.68%	4.64%	5.71%	4.95%	8.71%
FSG (Type-III)	11.42%	9.55%	0.90%	0.76%	0.86%	0.64%	0.77%
Asprotect (Type-III)	4.71%	1.53%	1.57%	3.22%	2.22%	2.27%	2.22%
NSPack (Type-III)	3.95%	2.06%	1.19%	1.30%	1.13%	1.02%	1.42%
Themida (Virt.)	6.00%	2.40%	0.78%	0.92%	0.62%	0.57%	0.57%
Upack (Type-III)	2.85%	2.31%	1.41%	0.35%	0.32%	0.44%	0.88%
Xtreme (Type-III)	0.50%	0.57%	0.27%	1.32%	0.46%	0.25%	0.24%
Others	19.41%	8.54%	5.24%	4.85%	3.82%	2.89%	3.67%
Unknown	2.75%	4.64%	5.29%	3.90%	2.39%	2.05%	2.27%
Installer	7.79%	9.18%	17.88%	32.60%	20.94%	23.40%	20.21%

Figure 7 and Figure 8 plot the overall number of layers and inter-process communication methods found in the dataset of custom packers. Like in the case of off-the-shelf packers, the majority of custom packers use few layers. Nevertheless, in contrast to off-the-shelf packers, there is a significant number of samples that present between 3 and 6 layers. From the total of 6,088 samples, 826 (14%) did not have the original code in the last level. Regarding interprocess communication, we observed an increment in the use of system process injection and unpacking by using external files, caused by the presence of malware that injects the code to other processes or drops files that are afterwards executed.

Finally, Figure 9 and Figure 10 show the evolution (average and 90^{th} percentile of the distribution) of the number of processes and the number of layers over time. The standard deviation of the number of processes over the years was between 1.08 and 1.66. Regarding the number of layers, it was between 5.46 and 35.32. In both cases we did not observe any significant variation in our experiments. Combining all this information, we can conclude that the *average* custom packer in our dataset presents a multi-layer unpacking routine with a cyclic transition model.

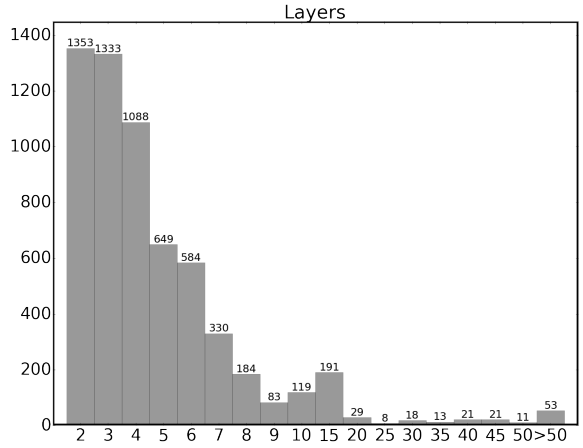

Fig. 7. Number of layers used by custom packers.

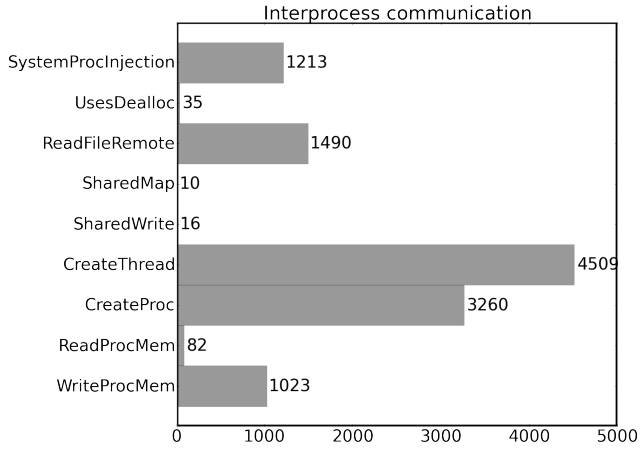

Fig. 8. Interprocess communication techniques used found in custom packers.

VI. DISCUSSION

Generic unpackers rely on a number of assumptions about the packer structure that, as we have seen in our study, are not always true for the samples observed in the wild. First of all, there is a significant number of packers with more than 2 layers. This implies that every unpacked code is not necessarily part of the original code. Despite some approaches like Renovo [4] consider cases with several layers of protection, not many studies have focused on determining which layer contains the original code, and it is not clear how to proceed in these cases. Moreover, things are complicated by the fact 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.

Many unpackers try to identify the *tail-jump* to dump the protected code at the appropriate moment. Type-IV, Type-V and Type-VI packers complicate this operation. Even though Type-V and Type-VI packers exist in our off-the-shelf dataset, they are not very common in the wild. These packers require

TABLE III
CUSTOM PACKER COMPLEXITY OVER THE YEARS.

	2007	2008	2009	2010	2011	2012	2013	2014
Type-I	5.2%	8.1%	6.1%	8.1%	9.2%	4.6%	8.3%	8.0%
Type-II	18.3%	15.6%	10.2%	15.4%	10.5%	8.9%	11.5%	15.0%
Type-III	63.8%	64.6%	71.2%	62.5%	64.4%	69.0%	63.7%	61.3%
Type-IV	11.3%	11.4%	12.1%	13.7%	15.7%	15.1%	15.0%	15.0%
Type-V	-	0.2%	0.1%	-	-	2.3%	1.7%	0.7%
Type-VI	1.4%	0.1%	0.3%	0.2%	0.2%	-	-	-

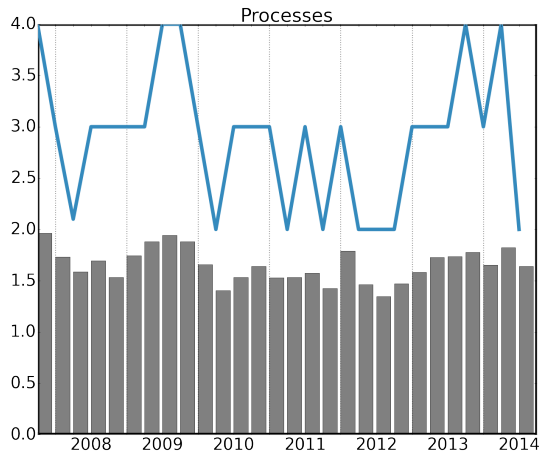

Fig. 9. Average number of processes used by custom packers over time.

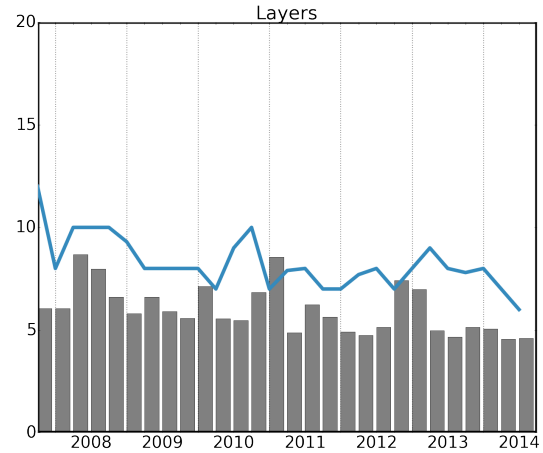

Fig. 10. Number of layers used by custom packers over time.

a significantly more complex development and also impose a run-time overhead that may not be desired by malware writers. None of the generic unpackers proposed to date has dealt with this kind of packers.

Based on our results, we can conclude that the average packer has a Type-III complexity. While Type-I packers are very common for off-the-shelf packers, in the case of custom packers there is a relatively low number of samples in this category - showing that malware writers look for more complex protection schemes.

It is quite interesting the lack of a clear evolution in the packers characteristics and complexity over time. In the past eight years, we did not observe any trend that shows that malware writers are moving toward more complex techniques. The fact that more sophisticated packing techniques are widely available, but malware writers do not use them is not necessarily a good sign. In fact, it may be the consequence of the fact that an average Type-III packing routine is already complex enough to protect against automated scanners.

In order to measure the run-time packer complexity, we propose a taxonomy that combines several metrics to classify each packer in one of six categories with incremental complexity. This taxonomy is focused on common run-time packers, and does not cover *virtualization*-based protectors. These sophisticated tools belong to a different family of protectors that do not recover the original code by overwriting a region of memory. We plan to extend our taxonomy in the future in order to cover these tools.

The presented framework was developed over a whole-emulation solution: TEMU. While it is true that some malware samples may implement specific anti-QEMU techniques, other approaches such as debugging or binary instrumentation are also susceptible of being detected, and do not provide a system-wide point of view of the execution. Although different authors [5], [18], [19] have proposed virtualization based approaches for binary tracing, we believe that the transparency of the analysis environment is beyond the scope of this study.

Finally, the approach presented in this paper was designed following an iterative process, by analyzing interesting packers and manually verifying the validity of the results. On the one hand, some of the properties considered in the taxonomy (e.g., number of layers, number of frames, repacking of memory, and transition model) can be measured precisely by our model. On the other hand, the distinction between Type-III and Type-IV, and between Type-IV and Type-V/Type-VI require to locate the memory regions where the original code resides. In order to confront this problem we designed a heuristic and manually verified its effectiveness in a number of real examples. Unfortunately, due to the lack of labeled datasets, and therefore of a ground truth, it is not possible to measure the accuracy of this heuristic beyond the manual analysis already conducted.

We believe that the lack of data-sets and ground truth in this domain is caused by the lack of tools for packer behavior

analysis. As a result, numerous authors [20]–[22] have built custom data-sets in order to conduct experiments on packers. However, in order to label these data-sets, the authors resorted to signature based detection tools (known to raise too many false negatives), dynamic generic unpackers – that do not report information about the packer behavior, and even manual analysis.

Tools like the framework proposed in this paper can help the analyst in the reverse engineering process, allowing the collection and labeling of run-time packer datasets.

VII. Related Work

Run-time packers have been widely used by malware authors for a long time. When these protection tools became problematic for malware analysis, the community proposed different solutions to generically recover the code of the binary. Most of these approaches are based on the dynamic execution of the sample in a controlled environment monitoring events at different granularity levels. These solutions differ in the heuristics and statistical methods employed to determine the right moment to dump the unpacked memory content.

Polyunpack [2] is based on the comparison of the statically observable disassembled code and the trace obtained after the execution of the binary. Omniunpack [1] is a real-time unpacker that monitors memory writes and execution at a page-granularity level using memory protection mechanisms provided by the operating system. Its focus is on efficiency and resilience, and is intended to trigger the analysis of an anti-virus scanner whenever new code is ready to be executed.

Renovo [4], in constrast, instruments the execution of the binary in an emulated environment and traces the execution at an instruction granularity level. This approach is capable of dealing with several layers of unpacking code, providing a memory snapshot for each new layer of code discovered. Eureka [3] focuses on coarse-grained granularity analysis, but instead of monitoring page-level protection mechanisms, it intercepts system calls and decides the moment in which the unpacked content has been revealed based on heuristics and statistical analysis.

Other approaches have focused on different techniques for monitoring the execution of the binary. For instance, Cesare and Xiang [6] proposed an application-level emulation unpacker, providing a method to determine the appropriate moment to dump the memory by analyzing the entropy of the binary. Ether [5], in contrast, proposes an unpacking framework based on instrumentation techniques in a virtual-machine environment making use of the Intel VT extensions.

All the publications mentioned above focus on dynamic analysis techniques. Nevertheless, a few authors have studied the use of static and hybrid analysis techniques in order solve the problem. Coogan et al. [7] proposed a solution based on control flow and alias analysis to identify potential transition points from the unpacking routine to the original code. From that point, the authors applied *backward slicing* to extract the packing routine and execute it as part of an unpacking tool. Caballero et al. [23] proposed a mixed dynamic and static approach consisting on hybrid disassembly and data-flow analysis to extract self-contained *transformation functions*, identifying code and data dependencies, and extracting the function interface (input and output parameters) in order to reuse it for the unpacking of the sample. Finally, other authors have focused on virtualization-based obfuscators [8], [9], a very complex type of packer that represents a different challenge.

While Bayer et al. [24] presented a short overview of off-the-shelf packers used by malware in 2008, to the best of our knowledge we are the first ones to present a longitudinal study of the packer prevalence and complexity.

As part of the malware-analysis process, run-time packers represent a *moving target* that implement many different obfuscation techniques to prevent generic unpackers from recovering the code, avoid debuggers, emulators, disassemblers, or memory dump tools. Some studies [25]–[27] have focused on documenting or measuring the prevalence of these techniques in malware or common packers. Other studies [28] highlight the fact that, although current anti-virus systems implement some of the generic unpacking techniques proposed to date, these approaches can be evaded with sufficiently complex packers.

Other approaches, in contrast, have focused on measuring the complexity of the packer by considering the number of phases, waves, or layers. First, Debray et al. [13], [29] proposed a formalization of the semantics of self-unpacking code, and modeled the concept of execution phases. In their model, a phase involves all the executed instructions written by any of the previous phases. This concept is related to our definition of execution frames, with the difference that, in our model, execution frames only occur in the context of a single unpacking layer. Guizani et al. [14] proposed the concept of waves. This first definition of waves is equivalent to our concept of execution layers. Later, Marion et al. [30] proposed a different formalization in which they modified the semantics of waves, proposing a model similar to the phases proposed by Debray et al.. Some of these publications have measured the number of layers present in malware samples. Nevertheless, they do not cover other complexity aspects such as the transition model, execution frames, or code visibility. Moreover, our model differentiates between the concept of unpacking layers and unpacking frames, allowing us to compute different properties that can be combined to provide a complexity score based on the class of the packer.

VIII. Conclusions

In this paper we have presented a packer taxonomy capable of measuring the structural complexity of run-time packers. We also developed an analysis framework that we 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 behavior of packers suggests that the

(un)packing problem has been put prematurely aside by the research community.

The results of our experiments show that, while many run-time packers present simple structures, there is a significant number of samples that present more complex topologies. We believe that this study can help security researchers to understand the complexity and structure of run-time protectors and to develop effective heuristics to generically unpack binaries.

ACKNOWLEDGMENTS

We would like to thank the reviewers for their insightful comments and our shepherd Michael Bailey for his assistance to improve the quality of this paper. This research was partially supported by the Basque Government under a pre-doctoral grant given to Xabier Ugarte-Pedrero, and by the SysSec Researcher Mobility Program for System Security, funded by the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement 257007.

REFERENCES

[1] L. Martignoni, M. Christodorescu, and S. Jha, "Omniunpack: Fast, generic, and safe unpacking of malware," in *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pp. 431–441, IEEE, 2007.

[2] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee, "Polyunpack: Automating the hidden-code extraction of unpack-executing malware," in *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC)*, pp. 289–300, 2006.

[3] M. Sharif, V. Yegneswaran, H. Saidi, P. Porras, and W. Lee, "Eureka: A Framework for Enabling Static Malware Analysis," in *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, pp. 481–500, 2008.

[4] M. Kang, P. Poosankam, and H. Yin, "Renovo: A hidden code extractor for packed executables," in *Proceedings of the 2007 ACM workshop on Recurring malcode*, pp. 46–53, 2007.

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

[6] S. Cesare and Y. Xiang, "Classification of malware using structured control flow," in *Proceedings of the Eighth Australasian Symposium on Parallel and Distributed Computing-Volume 107*, pp. 61–70, Australian Computer Society, Inc., 2010.

[7] K. Coogan, S. Debray, T. Kaochar, and G. Townsend, "Automatic static unpacking of malware binaries," in *Reverse Engineering, 2009. WCRE'09. 16th Working Conference on*, pp. 167–176, IEEE, 2009.

[8] R. Rolles, "Unpacking virtualization obfuscators," in *3rd USENIX Workshop on Offensive Technologies.(WOOT)*, 2009.

[9] K. Coogan, G. Lu, and S. Debray, "Deobfuscation of virtualization-obfuscated software: a semantics-based approach," in *Proceedings of the 18th ACM conference on Computer and communications security*, pp. 275–284, ACM, 2011.

[10] McAfee Labs, "Mcafee threats report: Fourth quarter 2013," 2013. Available on-line: http://www.mcafee.com/sg/resources/reports/rp-quarterly-threat-q4-2013.pdf.

[11] M. Morgenstern and H. Pilz, "Useful and useless statistics about viruses and anti-virus programs," in *Proceedings of the CARO Workshop*, 2010.

[12] F. Guo, P. Ferrie, and T.-C. Chiueh, "A study of the packer problem and its solutions," in *Proceedings of the 2008 Conference on Recent Advances in Intrusion Detection (RAID)*, pp. 98–115, 2008.

[13] S. Debray, K. Coogan, and G. Townsend, "On the semantics of self-unpacking malware code," tech. rep., Dept. of Computer Science, University of Arizona, Tucson, July 2008.

[14] W. Guizani, J.-Y. Marion, and D. Reynaud-Plantey, "Server-side dynamic code analysis," in *Malicious and unwanted software (MALWARE), 2009 4th international conference on*, pp. 55–62, IEEE, 2009.

[15] L. Bilge, A. Lanzi, and D. Balzarotti, "Thwarting real-time dynamic unpacking," in *Proceedings of the Fourth European Workshop on System Security*, p. 5, ACM, 2011.

[16] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "BitBlaze: A new approach to computer security via binary analysis," in *Proceedings of the 4th International Conference on Information Systems Security*, (Hyderabad, India), 2008.

[17] D. A. Quist and L. M. Liebrock, "Visualizing compiled executables for malware analysis," in *Visualization for Cyber Security, 2009. VizSec 2009. 6th International Workshop on*, pp. 27–32, IEEE, 2009.

[18] L.-K. Yan, M. Jayachandra, M. Zhang, and H. Yin, "V2e: combining hardware virtualization and softwareemulation for transparent and extensible malware analysis," *ACM SIGPLAN Notices*, vol. 47, no. 7, pp. 227–238, 2012.

[19] Z. Deng, X. Zhang, and D. Xu, "Spider: Stealthy binary program instrumentation and debugging via hardware virtualization," in *Proceedings of the 29th Annual Computer Security Applications Conference*, pp. 289–298, ACM, 2013.

[20] R. Perdisci, A. Lanzi, and W. Lee, "Mcboost: Boosting scalability in malware collection and analysis using statistical classification of executables," in *Computer Security Applications Conference, 2008. ACSAC 2008. Annual*, pp. 301–310, IEEE, 2008.

[21] M. Z. Shafiq, S. M. Tabish, F. Mirza, and M. Farooq, "Pe-miner: Mining structural information to detect malicious executables in realtime," in *Recent advances in intrusion detection*, pp. 121–141, Springer, 2009.

[22] X. Ugarte-Pedrero, I. Santos, I. García-Ferreira, S. Huerta, B. Sanz, and P. G. Bringas, "On the adoption of anomaly detection for packed executable filtering," *Computers & Security*, vol. 43, pp. 126–144, 2014.

[23] J. Caballero, N. Johnson, S. McCamant, and D. Song, "Binary code extraction and interface identification for security applications," in *Proceedings of the 17th Annual Network and Distributed System Security Symposium*, pp. 391–408, ISOC, 2009.

[24] U. Bayer, I. Habibi, D. Balzarotti, E. Kirda, and C. Kruegel, "A view on current malware behaviors," in *USENIX workshop on large-scale exploits and emergent threats (LEET)*, LEET 09, April 2009.

[25] P. Ferrie, "Anti-unpacker tricks–part one," *Virus Bulletin*, p. 4, 2008.

[26] K. A. Roundy and B. P. Miller, "Binary-code obfuscations in prevalent packer tools," *ACM Computing Surveys (CSUR)*, vol. 46, no. 1, p. 4, 2013.

[27] G. Negreira and R. Rubira, "Prevalent characteristics in modern malware.," *BlackHat USA*, 2014. Available on-line: https://www.blackhat.com/docs/us-14/materials/us-14-Branco-Prevalent-Characteristics-In-Modern-Malware.pdf.

[28] A. Swinnen and A. Mesbahi, "One packer to rule them all: Empirical identification, comparison and circumvention of current antivirus detection techniques.," *BlackHat USA*, 2014. Available on-line: https://www.blackhat.com/docs/us-14/materials/us-14-Mesbahi-One-Packer-To-Rule-Them-All-WP.pdf.

[29] S. Debray and J. Patel, "Reverse engineering self-modifying code: Unpacker extraction," in *Reverse Engineering (WCRE), 2010 17th Working Conference on*, pp. 131–140, IEEE, 2010.

[30] J.-Y. Marion and D. Reynaud, "Wave analysis of advanced self-modifying behaviors," *Groupement De Recherche CNRS du Génie de la Programmation et du Logiciel*, pp. 137–152, 2013.