# RoAMer: The Robust Automated Malware Unpacker

Thorsten Jenke, Daniel Plohmann, Elmar Padilla
Fraunhofer FKIE
Cyber Analysis and Defense
Zanderstraße 5, 53177 Bonn, Germany
{thorsten.jenke, daniel.plohmann, elmar.padilla}@fkie.fraunhofer.de

## Abstract

Malware Analysis is an essential building block for understanding malware's behaviour and countering the threat posed by malware. One major problem of current malware analysis is most of today's malware occurring in an obfuscated or packed form. Detailed analysis of a malware sample requires unpacking as a first step. To save human analysts' time it would be highly desirable to automate unpacking.

There have been numerous approaches to achieve this, but all of these have been found lacking. They are either not well tested against real-world malware, lack in correctness and speed, are not compatible with most packers, or are not resilient against malware evasion techniques. This makes previous academic malware unpacking techniques unusable for real-world scenarios. We addressed these issues by developing RoAMer, a fast, fully automated, and generic unpacker. It works by natively executing the malware, gathering changes to memory, and detecting the malware's payload amongst the gathered changes. It leverages automation of commonly manually performed unpacking steps and heuristics to determine unpacked malware versions in memory dumps. Based on our analysis of common shortcomings of previous unpacking approaches, we developed four criteria to rate unpackers. These are correctness, speed, generality, and evasion resilience. In our evaluation we rate RoAMer based on these criteria. Our evaluation proves that it unpacks up to 90% of the tested malware.

## 1 Introduction

The amount of uniquely identifiable malware samples in the wild is ever increasing and furthermore massively inflated by the different mechanics of obfuscation and polymorphism which are usually manifested in so-called packers: helper programs designed to wrap malware into an executable shell making them more resistant against detection by security products such as anti-virus scanners. This naturally implies that there are way more packed representations of (near- or fully identical) malware samples than are actually created by the authors of the malware families [17]. In consequence, it is a core task to quickly identify samples of already known families or their derivatives and to gather knowledge about the functionality of unknown samples.

Extracting the payload from a given packed sample is a necessary initial step in order to enable further detailed analysis. This analysis step may pose a significant challenge, as packers may exhibit behaviour for prolonging their unwrapping procedure or use environment detection mechanisms to avoid execution of the payload at all. Hence it is imperative to find a robust, generic, and automated solution for the unpacking problem.

One common approach performed during manual dynamic malware analysis is to natively execute the packed malware and then search for alterations in the system's memory which potentially contains the unpacked payload. While this approach often proves successful in practice, it is uncertain how much of real-world malware can be unpacked in such a way. For this reason one of the main goals of this research is to measure the success rate of this approach against real-world malware. In light of the diverse evasion techniques employed in malware, it is part of the analysis presented in this paper to additionally determine key factors for unpacking success, such as typical execution times needed by packers to reach execution of their payload malware.

Generic unpacking methods for malware have to fulfil certain requirements to be able to be applicable in practical scenarios, as explained in section 2. Ugarte-Pedrero et al. suggest in their work on the complexity of run-time packers that the (un)packer problem has been put prematurely aside by the research community. [20] This gap shall be filled with the method presented in this paper, as we introduce an unpacking method which is able to fulfil the requirements and therefore prove usable in practical scenarios.

In summary, we make the following contributions with

978-0-578-58383-9 

our paper:

- We derived criteria to rate unpacking approaches based on our analysis of previous academic approaches.
- We developed a robust and automated approach to extract payload memory dumps from packed malware. This method is designed to meet all of the necessary expectations and is applicable in real-world scenarios. We implemented this method in a tool called RoAMer and evaluated our method using a high-quality data set consisting of real-world malware.

The remainder of this document is structured as follows. The first section describes the requirements for an unpacking method. We then explain our approach in detail and outline the chosen implementation. This approach is evaluated using one comprehensive data set. After an overview of additional related work a summary concludes the paper.

## 2 Requirements for Unpacker

For unpacking methods to be useful in practice they have to fulfil certain requirements. These requirements ensure the method's correct and efficient functioning.

The first requirement ensures the correctness of the algorithm. Correctness implies that the unpacking method's produced output contains one or more components of the contained payloads protected by the packer. The method should work with an assertable confidence.

The second requirement demands that a low amount of time is needed for unpacking. A short time span ensures that each instance of the unpacker by itself can achieve a high throughput. If the unpacker is using a sandbox approach and the speed of the unpacking process is mostly limited to the execution speed of the malware, then this unpacker is suited for processing large amounts of malware.

The third requirement ensures that the unpacking method is compatible with the diversity found in in-the-wild malware. The unpacking method should be as generic as possible. We use the taxonomy of packer types introduced by Ugarte-Pedrero et al. [20] to describe the different packer types. These different types are explained in the respective paper.

The fourth requirement is the resilience towards evasion techniques. Malware and packers use different evasion techniques to protect themselves against being analysed. These can render an unpacking attempt fruitless. Therefore, the unpacking method has to take different evasion techniques into account. Otherwise the method is not applicable to a vast variety of malware. The evasion techniques addressed here are explained in the next section 2.1.

### 2.1 Evasion Resilience

In order to achieve decent effectiveness for as many malware samples as possible, the method should additionally be resilient to commonly observed and employed evasion techniques. In the following we list a selection of these methods.

Since debuggers are a typical tool used by malware analysts to conduct their analysis, checking for a debugger is a good heuristic for recognising an analysis environment.

Methods of sandboxing are often realised through fully automated systems that do not require or expose ways of user interaction. Therefore, monitoring a system for typical signs of user interaction is another powerful method to distinguish between real systems and potential analysis environments. Malware might also check for the identity of its parent process, which is typically the system shell (in case of Windows, `explorer.exe`) when it was natively started through the GUI, i.e. resulting from a double click.

Occasionally, a packer or payload malware may check for certain process names, upon which execution in an analysis environment is assumed, leading in consequence to termination or altered behaviour. This list of process names includes among others monitoring and analysis tools enabling traffic capture and debugging, or even the Python run-time (in that case primarily targeting the very popular Cuckoo sandbox), or artefacts of virtualization software, such as the Guest Additions of VirtualBox.

Another common technique to evade sandbox environments is the usage of extended sleeps, causing the analysis system to timeout before the sleep is over. This protects the payload from being unpacked in an analysis system.

The malware may check for any untypical system configuration compared to its expected targets like consumer PCs. For example, the screen resolution and the hard drive size should be realistic and timely as well. Methods may also include inspection of the history of running processes and accessed documents. Furthermore, it may be suspicious to malware if no recent internet browser is installed. [5]

To evade the observation of a single process, malware oftentimes uses so-called process injection. The unpacking happens across multiple processes. For example, the unpacked malware is directly written to a different process from the original malware process and is launched there. This renders all of the unpacking methods that focus on a single process useless. Additionally, packers have always the possible capability to allocate additional memory regions to write their unpacked code to. This has to be also considered by unpackers.

## 3 Methodology

In this section, we introduce our proposed methodology for a robust and generic way of unpacking and dumping

the malware samples. This approach is implemented in the Robust Automated Malware unpacker (RoAMer).

### 3.1 Overview

The default procedure for unpacking malware in manual analysis typically relies on dynamic analysis techniques. The overall methodology can be summarised as executing the malware in a confined and closely monitored sandbox environment, focusing on suspicious activity including the start-up of new processes, sudden changes in the memory size of running processes, network connection attempts, or otherwise detectable results of potential behaviour exposed by the payload. All of these examples are good indicators that a packer is still active or has handed over execution to the payload. Either way, these newly launched processes to run the payload or memory being allocated to inject the malware into are elements of interest to an analyst. New memory is allocated in so-called memory regions. A memory region is a continuous array of memory allocated by the operating system. When such a region has been identified, the region is dumped and the analyst has to make a decision based on further analysis whether this region contains the unpacked payload malware. The dumping step is repeated until eventually the desired dump has been found.

This method is based on the assumption that memory regions have to be allocated in order to execute the malicious code. These memory regions are either allocated during the creation of the original malware process, in already existing processes, or new processes. This makes the payload directly observable and therefore extractable. This is also the case for malware that does not need to allocate further memory regions. Our proposed method also regards the memory regions, that have been allocated for the initial packed sample to run, as new memory regions.

The proposed methodology closely follows the described procedure in which manual unpacking is conducted. It automates all of the steps that are necessary when performing manual unpacking. The flow of the algorithm is depicted in figure 1. First a snapshot of every memory space of every process on the system is taken. Then the malware is executed and the snapshot from the previous step is periodically repeated. These new snapshots are then compared to the previously made snapshot and all of the changed memory regions are recorded. After a predefined time, a series of filters are applied to the recorded sections and the remaining memory regions are dumped.

### 3.2 Filters

By definition, the proposed algorithm returns a set of dumps that is comprised of all the memory regions whose sizes have changed (new or re-allocated) during the execution. This also includes system libraries or additional data sections like stack and heap regions. These dumps may have little relevance up to being undesired and should not be part of the output. Therefore, the set of potential dumps is run through a series of filters to eliminate dumps that will most likely not contain the payload malware.

Before the filters are applied, directly subsequent memory regions are combined to a single dump. Now, two filters are applied to this dump.

The first filter eliminates the dumps that hold images, which have been already present on the system before the execution of the malware, i.e. system libraries. These can be filtered through the use of a whitelist. The hashed PE-headers with normalised imagebases of the images present on the system offer a cheap and expressive way for whitelisting.

The second filter has to eliminate all of the dumps, that do not include at least one executable section. This is based on the assumption, that at least one section of the malware has to be loaded to an executable section in order to be executed. Therefore, including an executable section in a dump is a strong indicator that this image includes code. This filter is able to eliminate stack and heap regions, because these are not executable most of the time.

After these filters have been applied, only executable images are extracted, which have not been on the system previously.

### 3.3 Limitations

The approach is based on a simplified packer model, which is compatible to type I, type II, type III, type IV and type V as defined by Ugarte et al. [20]. Each of these types have a point in time, at which the whole image is fully unpacked in memory. Type VI, however, cannot be reduced to the previously mentioned model because there is no point in time when the whole image is unpacked in memory.

However, for types I to V, the approach outlined in this section is completely agnostic to the actually used packer. Therefore, the methodology is generally compatible with existing and future packers, as the algorithm only relies on common inner workings of processes in modern operating systems.

### 3.4 Implementation

The method has been implemented for Windows 7 32/64 bit and uses VirtualBox [14] for virtualization in Python3 in a tool called RoAMer. It consists of two parts, one residing in the host system and one in the client system. The part in the host system controls the virtual machine and provides an interface to the user. The part in the client system performs

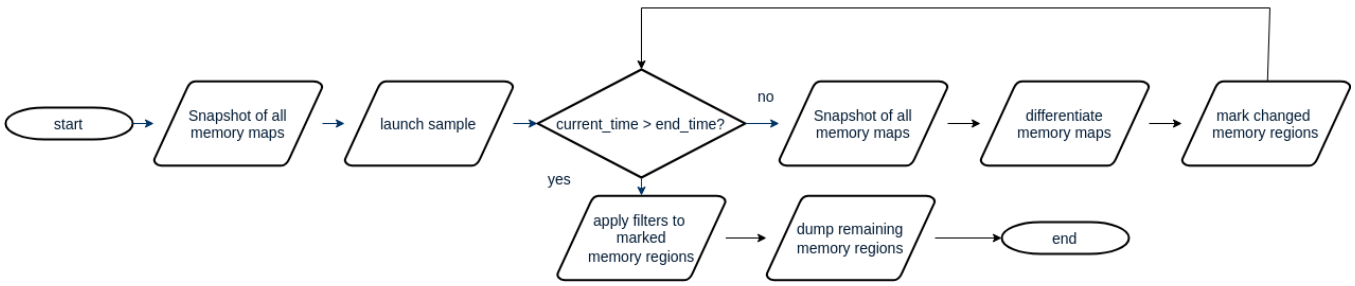


Figure 1: Flow graph of unpacking method

the unpacking. The interaction with the OS has been implemented through the use of the Windows API and without debugger functionality. Thus RoAMer satisfies the requirements described in section 2. The timeout of the algorithm can be leveraged to counter extended sleeps, that are used to avoid analysis systems. We additionally equipped RoAMer with the functionality to generate user interaction through the Windows API. The unpacker also considers all of the processes and therefore memory maps, so that an injection into a different process can be observed. In order to prevent the malware from fingerprinting the virtual machine or the system configuration, an especially hardened VM has been used. The creation of this virtual machine has been described in [5]. With the help of `AppInit_DLL` it is possible to basically system-wide hook NtTerminateProcess with code that lets the caller sleep indefinitely. This prevents the process from being terminated and the memory from being cleared.

## 4 Evaluation

This section presents an evaluation of the proposed method and its implementation as described in section 3. As the method generally resembles the current best practices of unpacking in manual malware analysis, the results serve representatively for what can be generally achieved by the described procedure.

The evaluation is constructed to show that the methodology fulfils all of the requirements defined in section 2. The correctness and speed requirements of the algorithm are asserted along two metrics. The third and fourth requirement are satisfied through the use of a high-quality and representative data set, which is comprised of practical relevant malware. Describing these metrics, the data set, the setup, and the results is subject of this section.

### 4.1 Data Set

The choice of the data set is critical for the expressiveness of the evaluation results and not a trivial task. It has to prove that the methodology is compatible with a high diversity of packers and evasion techniques. Several previous studies on unpacking used off-the-shelf packers and a known binary to evaluate their unpacking techniques. This is a very clinical approach, as it does not reflect the reality, where the analyst has to fight with evasion techniques and unknown payloads. Therefore, the main focus of this evaluation is to use representative data that provides better demonstration of the presented method's effectiveness.

We have decided to use Malpedia [16] as our data set, which ensures a high diversity in malware families. According to [16], the Malpedia data set adheres to the prudent practices designed by Rossow. [18]

According to documentation, almost all of the samples have been confirmed through peer review of the Malpedia community and the majority of samples have an unpacked and dumped version. This means that the data provided by Malpedia can be considered very reliable. On top of that, Malpedia offers a set of Yara-Rules, that have been proven to work reliably on the Malpedia data set as shown in [15].

In the context of the evaluation, this means that the correctness of the corresponding output of RoAMer is simplified since reference data can be used for comparison. Although the diversity in malware samples is only ensured along the families and not the packers, Malpedia is still a great resource to evaluate RoAMer against a wide range of malware families.

Malpedia is organised in a git repository and the state of the data set used in this paper is identified by commit hash `3b47d6` as of the 27th of March 2018.

### 4.2 Metrics

The first metric is the correctness of the output. This metric determines whether one or more of the produced dumps contain the unpacked target payload. To determine the correctness, a similarity-preserving hash named Trend Micro Locality Sensitive Hash (TLSH) [13] is used. TLSH is robust to the effects of address relocation and can provide a good means to determine the similarity between the output and a reference file. However, this still faces the inherent problem that dumps may differ from run to run. Therefore,

the second criterion is the application of identification signatures, in this case YARA-Rules [1]. The previously mentioned Yara-Rules provided by Malpedia are used for that purpose. When the Malpedia Yara-Rules for that dump hit, then this dump is considered to be correctly unpacked.

The proposed method for unpacking implies that the output consists likely of more than one dump instead of the correct dump only. As a consequence, the post-processing of all the dumps may become so extensive, that our proposed unpacking method would become unfeasible. For example, from an analyst's point of view, an automated unpacking method becomes impractical, when the amount of output dumps is so high that the potential effort of reviewing them surpasses the expected manual effort. Therefore, the precision of the method and thereby the quality of the output is measured as the recorded overhead in storage space in addition to the malware sample itself. This means, that the lower the amount of additional undesired dumps is, the more precise the method becomes. The precision is presented through a factor, which transform the amount of wanted data into the amount of unwanted data.

The third metric is the time, needed by the method to unpack the malware. This metric proves that the algorithm fulfils the requirement of being able to process large amounts of malware in a timely manner.

### 4.3 Setup

The previously mentioned prudent practices by Rossow [18] also entail ones designed to improve experiments. For this experiment, the realism aspect is the most important as the other aspects are covered by Malpedia. However, it is not mandatory to remove moot samples from the data set, as even older ones should be compatible with RoAMer. But the measurement of true positives and negatives is part of the main goal of this evaluation as explained in the next section. The experiment is tested against a significant number of malware families, of which Malpedia offers a vast variety. But there is no internet connection for the samples in the VM as the packers generally do not require an internet connection. Furthermore, the potential danger from launching samples and granting access to the Internet overshadows the potential benefit. The user interaction described by Rossow is an integral part of RoAMer, but only Windows Malware is regarded in our current setup.

We used RoAMer to process the data set two times each time with one of two different configurations. For both configurations, the run time has been chosen to be ten minutes but only during the second run is process termination prevented by hooking `ntdll.dll!NtTerminateProcess`.

The run time of ten minutes ensures that the majority of packers should have had a chance to unpack the payload. This provides a good impression on how much time needs to be allocated for a significant amount of malware to enter an unpacked stage. Therefore, the statistics on dump timing are highly relevant to practitioners. Our hypothesis is that most samples are in fact unpacked after a short amount of time.

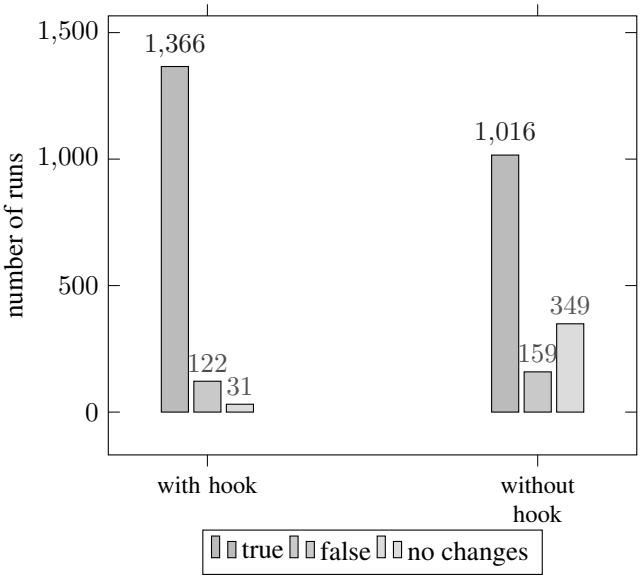


Figure 2: Correctness

The comparison between the runs with and without the hook will provide insights on the benefits and disadvantages introduced by it. The hypothesis is that the hook leads to a higher coverage but also to a lowered precision, because one natural way of freeing memory (through process termination) is eliminated.

### 4.4 Results

The correctness of the output determines whether the run has been successful for the respective dump. A run is considered successful (or true), if there is at least one correct dump among all of the output dumps. Contrarily, a run is considered failed (or false), if there is no correct dump in the output. These two results may only occur when there is a least one output dump. When the output is empty, the run is tagged as "no changes". "No changes" is used to describe the runs in which the malware did not commit any observable changes to the system that were present at the end of the run. This applies to the malware samples that stopped execution without leaving a trace, for example, as result of a detected analysis environment or crash of the whole system.

#### 4.4.1 Correctness

90% of the malware in the run with the hook enabled has been unpacked correctly and only 67% of the malware have been correctly unpacked in the run without the hook. The amount of malware, that did not commit any changes is more than ten times higher in the run without the hook. This suggests that most evasion techniques are located in the payload of the malware and not in the packers themselves. This can be assumed because the malware is not able to close itself, when the hook is enabled, and evasion techniques mostly result in the termination of the malware.

The significant amount of coverage suggests that RoAMer is able to generically work with a high and diverse amount of samples. Enabling the hook improves the correctness of the algorithm significantly. More than 300 samples have been additionally unpacked due to enabling the hook. As a result, it is strongly suggested to make sure to always scrape memory at the end of the process lifetime or, in the case of RoAMer, to always enable the termination hook.

#### 4.4.2 Precision

The data overhead in table 1 describes the data that is not part of the correct output dumps. The displayed numbers are the sizes of data not containing the target payload in comparison to the size of correct dumps. The result is therefore the relative overhead. This metric is a fine-grained way of rating the output quality and it presents the overall amount of unwanted output. The minimum, first quartile, and median results are very similar across both runs. Only the third quartile has a value of 1.03 for the run without hook and 1.94 for the run with hook. Similar to the number of output dumps, the maximum values are extremely high, with over 3322 for the run without hook and over 2071 for the other. Again, these are heavy outliers, which is also reflected by the average, which is 21 and 9,77 times higher than the third quartile value for their respective runs.

Measuring the precision emphasises that the hook generally decreases precision. Therefore, it may become unfeasible to enable the hook in certain scenarios with a constraint of post-processing time or storage capacity.

#### 4.4.3 Speed

This metric describes the time spent after the execution, when the correct dump was first observed. The run time of the samples is one of the most important configuration options for an unpacking or even sandboxing system. It introduces a trade-off between throughput and coverage, because the longer a sample runs the more likely it is for the malware to be unpacked. Therefore, finding the best trade-off between run time and coverage is the goal of this section.

Due to the discrete nature of the observation intervals, the corresponding memory regions have not been actually allocated at that point in time. The snapshots of the memory maps are done in intervals of five seconds each, in which the memory maps of the system are recorded. If a memory region first appeared after eleven seconds, then this dump was first recorded in the third recording window. Therefore, the dump timing describes the least amount of time, which could have yielded the respective dump containing the payload.

All values up to the third quartiles are the same for both runs at eleven seconds as seen in table 1. Their averages and maxima are also pretty similar. The averages are 22.14 seconds for the run without hook and 17.78 seconds for the run with hook and the maxima are 641 seconds and 626 seconds respectively. The maxima are higher than ten minutes because the packers put too much load on the VM so that the unpacker missed its timings and therefore ran longer than ten minutes.

The result shows that eleven seconds each would suffice to unpack 75% of all the samples. This is also consistent with the hypothesis that the most prominent packers have a similar but fast unpacking time.

Enabling the hook therefore very slightly reduces the time spent until the first dump of the malware is observable. This is reflected in the overall slightly lowered averages for the runs where the hook is enabled.

After that, the algorithm also has to apply filters to the dumps and perform the extraction of the dumps. This takes seven seconds for 50% of the samples without and with the hook. Even the third quartile is still seven seconds for the run without the hook but nine seconds for the run with the hook. The maximum values are again heavy outliers, at 98 and 159 seconds. This is reflected in the averages being at 7.47 and 8.73 seconds. Thus it takes 18 seconds without the hook and 20 seconds with the hook to unpack, extract, and filter 75% of the malware

### 4.5 Discussion

The interpretation of the data reveals a trade-off between the correctness, speed, and the precision of the approach. When the process termination hook is enabled, the correctness and speed increase but the precision decreases and vice versa. Therefore, it may be better under certain circumstances to refrain from enabling the hook, for example, when hard disk space or post-processing capacity are constraints. When a high coverage is the goal, it is absolutely recommended to enable the hook.

The evaluation has shown that RoAMer is able to fulfil all of the four requirements. The correctness is shown by the high coverage rate. The speed is shown by the relatively low amount of time needed to unpack most malware. The

| | OH w/o Hook | OH w/ hook | speed w/o hook | speed w/ hook | filtering & dumping w/o hook | filtering & dumping w/ hook |
|---|---|---|---|---|---|---|
| min | 0 | 0 | 11 | 11 | 4 | 4 |
| 25% | 0 | 0 | 11 | 11 | 6 | 6 |
| Average | 21.99 | 18.96 | 22.14 | 17.78 | 7.47 | 8.73 |
| Median | 0.01 | 0.41 | 11 | 11 | 7 | 7 |
| 75% | 1.03 | 1.94 | 11 | 11 | 7 | 9 |
| max | 3322.9 | 2071.43 | 641 | 626 | 98 | 159 |

Table 1: Number of result dumps, unpacking overhead (OH), unpacking speed in seconds, and time spent filtering and dumping in seconds

generality and evasion resilience are shown by the conjunction between the high correctness and the diverse data set Malpedia.

## 5 Related Work

This chapter mentions related work that was crucial to our work.

Ugarte-Pedrero et al. [20] propose a framework for packer analysis and a taxonomy to measure the run time complexity of packers. The complexity is determined along different classes and features, for example, whether the packer is multi-layered or if code is only decrypted on-demand. The accompanying unpacking framework presented in the work gathers various information during the execution, like the execution trace and inter-process communication. Based on these information the complexity and functionality of the packer are determined. Their work has been an important influence on our works.

Rossow et al. [18] described best practices for designing malware experiments and surveying related work for their conformance with these requirements. Their work laid the foundations on how the data set was chosen and the evaluation has been conducted.

Branco et al. [4] investigated the evasion techniques employed in malware. They use a live system to constantly monitor incoming malware samples on used evasion techniques based on different analysis modules.

OmniUnpack by Martignoni et al. and Justin by Guo et al. are malware unpackers. Their central requirement is a working malware detector. They both work on the basis of the write-then-execute pattern to detect unpacking. They, at least, violate the second requirement, because the AV scanner slows down execution. [12] [7]

Renovo by Kang et al. traces the execution of the malware on instruction level. Based on the information of the monitoring, Renovo determines the point at which a newly written piece of memory is executed. The malware is run in an emulated environment and it only emulates one process space. This heuristic is supposed to point out the time when the malware has been unpacked. It violates the fourth requirement, because it does not work with process injection. [10]

The scope of the unpacker Pandora's Bochs by Lutz Boehne is narrowed down to malware where the unpacking process does not cross process boundaries, where unpacking does not happen in kernel mode, the unpacked code stays inside the image boundaries and the jump to the OEP is done via a branch instruction. Pandora's Bochs is not capable of handling process injections. It therefore violates at least the fourth requirement. [2]

RePEconstruct by David Korczynski is a malware unpacker and rebuilds the unpacked binaries for further analysis. RePEconstruct's main idea is to use dynamic analysis to unpack self-modifying code. It also gathers information about the execution, which can be used to rebuild the import address table and identify jumps into obfuscated code. However, this method is also not capable of detecting process injections and therefore violates at least the fourth requirement. [11]

Ether by Dinaburg et al. focuses heavily on a completely unobtrusive approach by locating the analysis software outside the virtual machine. It serves as a successor to PolyUnpack [19]. The actual unpacking procedure is again done through the monitoring of memory writes and system calls. Due to the monitoring, the execution of the malware is slowed down and it therefore violates the second requirement. [6]

The approach of Joeng et al. leverages entropy analysis to find the OEP. Every time some kind of jump is encountered, the program is paused and an entropy analysis is conducted to determine the end of the unpacking process. The analysis is conducted on the sections of the malware. However, this does not take process injection into account. Therefore, it violates the second and fourth requirement. [9]

One unpacking approach based on PANDA has been introduced by Haq et al. [8]. Their unpacking approach also yields promising results. Unfortunately, a real assessment of the requirements is not possible because their work does not focus on the unpacking. Their methodology for unpack-

ing malware is based on the works of Bofante et al. [3]

Rambo by Ugarte-Pedrero et al. suggests a way of unpacking malware that has been packed by type VI malware. Ugarte-Pedrero et al. propose a set of optimisations and heuristics to improve the multi-path exploration approach. These improvements involve, for example, circumventing blocking API calls through restoring the instruction pointer, dumping unpacked memory regions, and identifying important execution paths. [21]

## 6 Conclusion

In this paper we made two contributions.

First, we derived criteria to rate unpacking approaches based on our analysis of previous academic approaches.

Second, we created RoAMer, a fully automated unpacker. RoAMer operates by natively executing the malware, taking memory dumps and heuristically determining the unpacked malware from the candidate dumps. We evaluated it using our criteria to rate unpacking approaches. By using a representative, high quality malware corpus we could prove RoAMer's usefulness in practical scenarios. This differentiates it from most previous academic approaches. In our evaluation RoAMer achieved good results regarding all evaluated criteria. Thus, it is a solution for the unpacker problem, which is still very relevant and has been prematurely put aside by the research community according to Ugarte-Pedrero et al. [20].

One possible direction for future work could be to further research the potential post-processing of extracted memory dumps, e.g. ordering them in terms of importance or reconstructing the dumps to properly unpacked PE files and making them available for static or even dynamic analysis. The evasion technique using extended sleep should also be adressed in future iterations. One possibility of achieving this is by hooking the sleep and other related functions. Another way to improve RoAMer is by relocating the client part into the kernel or conducting the monitoring outside of the virtual machine, e.g. by using introspection.

## References


[1] ALVAREZ, V. M. yara: The pattern matching swiss knife for malware researchers (and everyone else). `http://virustotal.github.io/yara/`. Accessed: 2019-07-24.

[2] BOHNE, L., AND HOLZ, T. Pandoras bochs: Automated malware unpacking. *Master's thesis, RWTH Aachen University* (2008).

[3] BONFANTE, G., FERNANDEZ, J., MARION, J.-Y., ROUXEL, B., SABATIER, F., AND THIERRY, A. Codisasm: medium scale concatic disassembly of self-modifying binaries with overlapping instructions. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015), ACM, pp. 745–756.

[4] BRANCO, R. R., BARBOSA, G. N., AND NETO, P. D. Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti-vm technologies. *Black Hat* (2012).

[5] BYTEATLAS. Knowledge fragment: Hardening win7 x64 on virtualbox for malware analysis. `http://byte-atlas.blogspot.com/2017/02/hardening-vbox-win7x64.html`. Accessed: 2019-07-24.

[6] DINABURG, A., ROYAL, P., SHARIF, M., AND LEE, W. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on Computer and communications security* (2008), ACM, pp. 51–62.

[7] GUO, F., FERRIE, P., AND CHIUEH, T.-C. A study of the packer problem and its solutions. In *International Workshop on Recent Advances in Intrusion Detection* (2008), Springer, pp. 98–115.

[8] HAQ, I., CHICA, S., CABALLERO, J., AND JHA, S. Malware lineage in the wild. *Computers & Security 78* (2018), 347–363.

[9] JEONG, G., CHOO, E., LEE, J., BAT-ERDENE, M., AND LEE, H. Generic unpacking using entropy analysis. In *2010 5th International Conference on Malicious and Unwanted Software* (2010), IEEE, pp. 98–105.

[10] KANG, M. G., POOSANKAM, P., AND YIN, H. Renovo: A hidden code extractor for packed executables. In *Proceedings of the 2007 ACM workshop on Recurring malcode* (2007), ACM, pp. 46–53.

[11] KORCZYNSKI, D. Repeconstruct: reconstructing binaries with self-modifying code and import address table destruction. In *Malicious and Unwanted Software (MALWARE), 2016 11th International Conference on* (2016), IEEE, pp. 1–8.

[12] MARTIGNONI, L., CHRISTODORESCU, M., AND JHA, S. Omniunpack: Fast, generic, and safe unpacking of malware. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual* (2007), IEEE, pp. 431–441.

[13] OLIVER, J., CHENG, C., AND CHEN, Y. Tlsh–a locality sensitive hash. In *Cybercrime and Trustworthy Computing Workshop (CTC), 2013 Fourth* (2013), IEEE, pp. 7–13.

[14] ORACLE, V. Virtualbox, 2015.

[15] PLOHMANN, D. Malpedia. `https://malpedia.caad.fkie.fraunhofer.de/stats/yara`. Accessed: 2019-07-24.

[16] PLOHMANN, D. ApiScout: Painless Windows API information recovery, April 2017. Blog post for ByteAtlas: `http://byte-atlas.blogspot.de/2017/04/apiscout.html`.

[17] PLOHMANN, D., CLAUSS, M., ENDERS, S., AND PADILLA, E. Malpedia: A collaborative effort to inventorize the malware landscape. *The Journal on Cybercrime and Digital Investigations 3*, 1 (2018).

[18] ROSSOW, C., DIETRICH, C. J., KREIBICH, C., GRIER, C., PAXSON, V., POHLMANN, N., BOS, H., AND VAN STEEN, M. Prudent Practices for Designing Malware Experiments: Status Quo and Outlook . In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (S&P), San Francisco, CA* (2012).

[19] ROYAL, P., HALPIN, M., AND DAGON, D. Polyunpack: Automating the hidden-code extraction of unpack-executing malware, dec. 2006. *ACSAC, pp289-300*.

[20] UGARTE-PEDRERO, X., BALZAROTTI, D., SANTOS, I., AND BRINGAS, P. G. Sok: deep packer inspection: a longitudinal study of the complexity of run-time packers. In *2015 IEEE Symposium on Security and Privacy* (2015), IEEE, pp. 659–673.

[21] UGARTE-PEDRERO, X., BALZAROTTI, D., SANTOS, I., AND BRINGAS, P. G. Rambo: Run-time packer analysis with multiple branch observation. In *DIMVA* (2016), Springer, pp. 186–206.