WAVEATLAS: SURFING THROUGH THE LANDSCAPE OF CURRENT MALWARE PACKERS

Joan Calvet
ESET, Canada

Fanny Lalonde Lévesque, José M. Fernandez, Erwann Traourouder & François Menet
École Polytechnique de Montréal, Canada

Jean-Yves Marion
Université de Lorraine, France

Email joan.calvet@eset.com;
{fanny.lalonde-levesque, jose.fernandez, erwann.traourouder}@polymtl.ca;
Jean-Yves.Marion@loria.fr

ABSTRACT

Obfuscation techniques have become increasingly prevalent in malware programs as tools to thwart reverse engineering efforts and evade signature-based detection by security products. Among the most popular methods is the use of packers, which are programs that transform an executable file's appearance without affecting its semantic execution. The use of packers is now widely adopted by malware authors. However, despite the rise in malicious programs distributed with packers, we still lack a global picture of their current use. What kind of packers protect malware nowadays? Is there a common model? Previous attempts, based on static database-signature tools, failed to build an accurate picture of malware packers, their main limitation being that static analysis says nothing about the actual behaviour of the packers and, due to its static nature, it misses run-time features.

In this paper, we present WaveAtlas, a novel framework designed to map the code used by packers. Using a dynamic analysis approach, it reconstructs in a nutshell the structure of the code modification tree, where the root is the packed code and packer, and the nodes represent code snippets extracted in successive 'waves'. We report on a large-scale experiment conducted on a representative sample of thousands of self-modifying malicious code. Our results allowed us to successfully identify common features of malware packers, ranging from their self-modification code usage to exotic choices of machine instructions. In particular, we were able to confirm some commonly held beliefs regarding the use of packers by malware writers. For example, malicious payload (e.g. code including network callbacks) is typically present in the last or one-before-last waves. Furthermore, the number of waves is relatively small and the structure of the trees relatively simple, indicating that malware authors are probably using simpler tools and parameters as a compromise between stealth and efficiency.

1. INTRODUCTION

Over the years, malicious software writers have employed various techniques to protect their creations against detection and analysis. Probably the best known of those techniques is the use of so-called packers. We define a packer as *a program transformation whose output is a program that will decrypt or decompress part of its code during its execution*.

There exists a wide range of different packers, from commercial protections normally sold for legitimate purposes – but also used by malware – to custom packers developed specifically to protect malware. However, we still lack a global picture of the packer landscape. Most of the academic research on this topic is focused on the unpacking problem, i.e. the construction of automatic tools to retrieve payloads from packed samples, while there has been little research focused on giving a clear picture of the use of packers by malware. On the industry side, published work in this area mostly consists of detailed descriptions of specific and unusual packers, and does not provide a global appreciation of the malware packer problem.

A clear picture of malware packers, in particular of their usage trends, would bring new opportunities to defenders. For example, it would help in designing binary analysis methods tailored for them, while most of the existing work in this area only applies to non-protected binary programs.

Extending the knowledge on malware packers is the objective of WaveAtlas, an experimental method we built to study malware packers on a large scale using dynamic analysis (initially published, in French, in [1]). In this paper, we describe the first experiment we performed using this method, in order to propose and check the prevalence of a particular packer model among malware.

2. RESEARCH HYPOTHESIS

In our first experiment with WaveAtlas, we postulate that a certain packer model is particularly common among malware. This packer model is defined on execution traces, i.e. the lists of instructions executed at runtime by a program. Those traces are divided into subsets, called code waves, in the following way:

- Executed instructions that are part of the initial memory image of the program belong to the code wave of index 0.
- Executed instructions that have been written by an instruction belonging to the code wave of index i belong to the code wave of index i +1.

Hence a code wave includes the instructions created at the same 'depth'. We can now formulate our research hypothesis.

Research hypothesis: The packer model defined by the two following characteristics is prevalent among successful malware:

1. The first code waves – all code waves except the last one – serve solely to protect the malware payload
2. The last code wave contains the malware payload, defined here as any code modifying the system state (file creation, process injection, network communications, etc.)

The crux of this simple model is therefore a dichotomy between external code waves serving for protection only, and the innermost code wave implementing the payload. Our research problem is then to test the validity of this hypothesis, and supports our overall research goal of highlighting global trends in malware packers. The hypothesis focuses on *successful* malware – a notion that will be defined precisely later – to indicate that we do not intend to cover all existing

packers, but rather those that were successful from the malware writers' point of view.

3. RELATED WORK

While numerous studies have addressed the problem of unpacking, very few published works have tried to give a global picture of the packer problem. According to Brosh *et al.* [2], more than 92% of malware in 2006 was protected using packers. In 2010, Morgenstern *et al*. [3] reported that 70% of protected malicious programs were using public packers, either available for sale, or freely downloadable on the Internet. More recently, malware writers have started to adopt more and more custom packers made specifically to protect malware. According to Morgenstern *et al.* [3], custom packers represented more than 40% of the cases in 2010.

These statistics show the importance of malware packers, and a recent tendency to favour custom packers. Nevertheless, they do not give any indication as to the internal working of packers, and therefore do not answer our research problem. Additionally, they are based on static signatures tools (e.g. PEiD, SigBuster, YARA), which are prone to false positives and do not analyse the internal code waves.

Daniel Reynaud ran a dynamic analysis on 100,000 malware samples in 2010 for his Ph.D. thesis [4]. He noticed that more than 95% of them were packed. However, no indicators were given on the code wave functionalities, and therefore these results cannot be used to test our hypothesis. Moreover, the results suffered from an important bias: 56% of the samples came from the same malware family, due to the method of collection (honeypot).

A recent concurrent work by Ugarte-Pedrero *et al.* [5] studies malware packers through dynamic analysis, notably by building a taxonomy of packer complexity. This is an important contribution that gives precise concepts to reason about packers. Their experimental conclusion states that packers with a strict boundary between protection layers and payload – but where the payload is not necessarily in the deepest layer – are the most common among malware (so-called 'Type III' packers). Nevertheless, it is unclear to us to what extent those experimental results can be generalized, due to the choice of samples to analyse. Those samples were submitted to a specific sandbox system over the years, and the authors limited the number of sample per malware family to one for each month. Hence the dataset depends heavily on the users submitting to this sandbox, and we have no details of its exact composition. Therefore, the representativeness of the dataset remains unclear: does it represent the threats faced by normal users in the wild or the threats faced by a few corporations using this sandbox system? We also wonder how one could assess a dataset contains only one sample of each family, given the well-known labelling problem from anti-viruses, in particular the presence of generic detection names. The malware sample selection problem is a difficult one to tackle, and we will explain in detail how we dealt with it in the WaveAtlas project. Finally, the authors seem to assume their dynamic analysis environment can monitor the complete execution of all malware samples, without showing any evidence to support that. We will explain how we validated our experiments in the WaveAtlas project.

4. EXPERIMENTAL ENVIRONMENT

We describe here the WaveAtlas framework, which comprises both the experimental set-up and the procedure to analyse malware packers.

Selection of samples

The first experimental step is to choose the malware samples to analyse. More precisely, we need to select malware families, from which we will pick some samples. As stated in our research hypothesis, we want to study successful malware, so how do we define the success of a malware family?

Family name(s)	Activity period	Mentions in industry publications	Mentions in academic publications	Offensive actions
Storm Worm (Nuwar, Peacomm, Zhelatin)	2007–2008	53,300	1,190	Attack against the peer-to-peer protocol in 2008 by independent researchers [6]
Waledac	2008–2010	34,800	249	Attack against the peer-to-peer protocol in 2010 by *Microsoft* and partners [7]
Koobface	2008–2011	103,000	363	Attack against the command-and-control infrastructure in 2010 by independent researchers [8]
Conficker (Downadup, Kido)	2008–2011	324,000	1,390	In 2000, *Microsoft* offered a bounty of $250,000 for any information related to the operators' identity [9]
Cutwail (Pushdo)	2008–2015	48,300	231	Attack against the command-and-control infrastructure in 2010 by academic researchers [10]
Ramnit	2010–2015	129,000	44	No known attacks

Table 1: Selected malware families.

The first criterion could be its prevalence, i.e. the number of samples of the family found in the wild. However, this would artificially inflate the importance of file infectors, which produce an enormous number of samples that hardly correlates with the actual number of victims. Therefore, we decided to consider different criteria. We started from the assumption that the security community demonstrates a strong interest in specific malware families only when they pose an important threat. Consequently, those malware families have very likely been successful from their operators' point of view. We then defined two measurements of success for a malware family:

- **The number of publications from the security community on the malware family**. We measured this indicator using a customized search engine for security industry publications [11] and *Google Scholar* for academic publications. This measurement is intended to show the global interest in a malware family.
- **The fact that some offensive actions have been taken against the malware family**. The security community sometimes launches attacks against certain malware families, either technical ones [6, 8] or law enforcement ones [7]. These operations require a lot of effort, and are therefore only done when the malware family represents a significant threat.

Based on these criteria we selected the six malware families (excluding targeted attacks), as described in Table 1.

For each family, we picked 100 samples recognized as members of the family by at least four different major anti-virus vendors. We collected those samples from the public malware database *malware.lu* [12] and, as explained later, we made sure they were not corrupted files.

Experimental set-up

The experimental set-up we built to execute malware and analyse their execution traces is described in Figure 1.

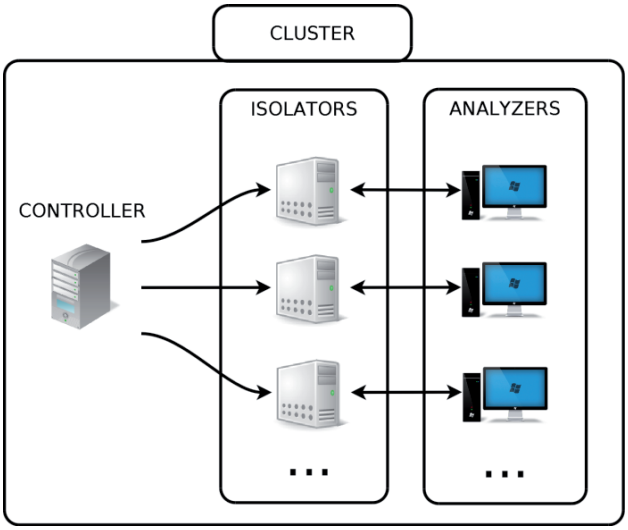

Figure 1: WaveAtlas experimental set-up.

This environment was created in the Sécurité des Systèmes d'Information (SecSI) laboratory of the École Polytechnique de Montréal. A smaller version was also built in the High Security Laboratory (HSL) in Nancy [13]. The set-up at the Polytechnique includes a cluster of 100 physical machines on which we deploy virtual machines. For the needs of our experiment, we defined three types of virtual machines:

- **Analysers** execute malware samples on *Windows XP* under the control of a tracer for a period of 20 minutes maximum. The tracer was built with Pin [14] and provides us a detailed execution trace. The analysers are also in charge of processing the execution traces and collecting the measurements we will present later. Each analyser is connected to a single isolator.
- **Isolators** execute the network simulator INetSim [15]. This software answers to network requests coming from the analysers with standard messages, in order to make them believe they are connected to the Internet. The isolators also control the experiments on the analysers, and collect the results.
- **The experience controller** deploys the samples to analyse, and centralizes the results coming from the isolators.

In total, we have 388 isolator/analyser tandems in our WaveAtlas set-up. Our tracer monitors only the main thread of execution, which is enough to analyse the payload of most samples (see experiment validation below).

Remark: The duration of 20 minutes was chosen such that we could get for each malware family 100 experiments where the sample modifies the system (see experiment validation below). It corresponds to an execution trace of maximum 110 million instructions in our set-up.

Experiment validation

A malware sample belonging to a selected malware family does not necessarily provide valid experimental results. For example, the file could be corrupted, or it could detect our analysis environment and refuse to execute normally. To filter out such invalid experiments, we define two checks to be made after each experiment in order to verify its validity.

First, we consider an experiment – a run in an analyser – to be *technically valid* only if the tracer terminates its execution normally, and if the execution trace contains more than 500 instructions. This filters out cases where a technical problem arose during the experiment.

Second, we consider an experiment to be *behaviourally valid* only if the sample modified the system state during its execution. We defined a set of API functions that modify the system and we verified that the malware successfully called one of those functions (file creation, registry modification, etc.). This is a strong criterion that indicates that the malware likely started executing its payload and trusts the environment to be a real machine to infect.

All experimental results presented in the following section concern experiments both technically and behaviourally valid. We made sure this corresponds to 100 experiments for each selected malware family.

5. EXPERIMENTAL RESULTS

The execution traces collected allowed us to extract a series of measurements that will be presented in the following subsections. Before discussing the purpose of code waves, we first assess the importance of packers among our sample set.

Figure 2 shows the number of code waves measured during the execution of our samples, which concerns only the main thread of execution.

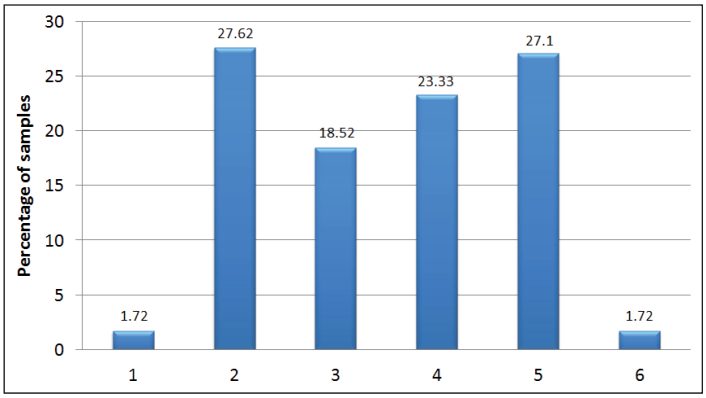

Figure 2: Number of code waves.

As seen in Figure 2, almost all malware samples are packed: 98.28% of them possess at least two code waves and there is no clear trend in the number of waves.

We now present measurements to test our research hypothesis, namely to check if code waves at various depths serve to protect and/or to interact with the system.

Machine instruction choice

Looking at which machine instructions are used in a program can give a hint as to the code purpose. In particular, it can help to distinguish efficient code produced by a compiler from obfuscated code produced by a packer. To analyse that, we looked at assembly mnemonics. An assembly mnemonic is the English abbreviation of a machine instruction; for example jmp, push or nop are mnemonics of x86 instructions.

We first established a list of 82 'classic' mnemonics by executing standard *Windows* programs (e.g. Calculator, Notepad) and extracting their mnemonics. Those classic mnemonics are the ones chosen by modern compilers for performance reasons. We then considered a code wave to have 'exotic' code if it executes more than 10 non-classic mnemonics. We also counted the number of different mnemonics present in each wave, as presented in Table 2.

	…the first code wave	…the middle code waves	…the penultimate code wave	…the last code wave
Part of samples with exotic code in…	10.12%	0%	0%	0%
Number of different mnemonics in…	110	63	80	77

Table 2: Machine instructions.

In order to compare execution traces with different numbers of code waves, we present our results based on the relative positions of code waves. For example, the first row of Table 2 can be interpreted as follows:

- There are 10.12% of samples with exotic code in their first code wave, which is the one with index 0. This code wave is present in all samples.
- There are 0% of samples with exotic code in at least one of their middle code waves, which is the set of code waves whose index is strictly superior to 0 and strictly inferior to the index of the penultimate code wave. Therefore, only samples with at least four waves have middle code waves, namely 52.15% of our sample set (see Figure 2).
- There are 0% of samples with exotic code in the penultimate code wave. Only the programs with at least three code waves have a penultimate code wave, namely 70.67% of our sample set.
- There are 0% of samples with exotic code in the last code wave, which is the one of maximal index. Only the samples with self-modification code have a last wave, namely 98.28% of our sample set.

Regarding the results, the first code wave is the only one to have exotic code, and it contains significantly more mnemonics than the others. Hence it was probably not created by the same kind of program transformation as the others, and the objective of this transformation was likely not performance, but rather protection.

Remark: As we just explained, not all the samples possess the four categories of code waves. Still, all percentages are calculated on the whole sample set, in order not to artificially overemphasize the importance of code waves that are not always present.

Function call convention

There is no formal definition of a function call in x86 machine language. Function calls are usually implemented with the machine instruction call, whose semantic is to redirect the execution flow and to save the caller address onto the stack (in order to come back to it later, usually with a ret instruction). However, this is just a convention followed by compilers, and nothing prevents the use of calls to redirect the execution flow without ever coming back to the caller location. Such a technique puts most binary analysis tools in a difficult situation, because they assume the execution flow returns after each call.

We checked the usage of the function call convention in our malware samples, by counting:

- the number of good function calls, defined as when the execution flow comes back after a call instruction
- the number of bad function calls, defined as when the execution flow never comes back after a call instruction.

As there could be a few legitimate cases where the execution flow does not come back (for example because an exception is thrown inside the function), we set a threshold of 10 function calls.

	…the first code wave	…the middle code waves	…the penultimate code wave	…the last code wave
Part of samples with more than 10 *good* function calls in…	15.88%	11.17%	62.13%	60.38%
Part of samples with more than 10 *bad* function calls in…	10.82%	0%	0%	0.17%

Table 3: Function calls.

Based on our results presented in Table 3, we can observe that:

- The first and the middle code waves have significantly fewer good function calls than the others. The penultimate and the last code waves employ the usual function implementation intensively, probably because they tend to be created by a classic compiler.
- The first code wave contains bad function calls in a significant number of samples, whereas the other code waves have none. This non-typical usage of function calls is likely employed to make code analysis harder.

Exception throwing

An exception is usually thrown by a program when there is a software problem, like a division by zero, an invalid memory access, etc. Alternatively, it can be used as a means of protection to redirect the execution flow to a custom exception handler, and thus make the execution flow harder to follow.

We measured the number of exceptions thrown during our malware execution (see Table 4). We first observe that a significant number of samples throw exceptions in their first code wave. As all our experiments have been validated, those exceptions likely do not correspond to software problems, but rather to protection tricks.

	…the first code wave	…the middle code waves	…the penultimate code wave	…the last code wave
Part of samples that thrown an exception in…	20.24%	0.52%	8.9%	0.35%

Table 4: Exception throwing.

There are also a surprisingly high number of samples with an exception thrown in the penultimate code wave. After manual analysis, we discovered that these samples belong to the Ramnit malware family, where an invalid memory access is made when the last code wave has been fully decrypted (which is the expected behaviour).

System modification

As previously explained, we considered an experiment run to be valid only when the system was modified during the malware execution. We measured in which code wave this system modification takes place (see Table 5).

	…the first code wave	…the middle code waves	…the penultimate code wave	…the last code wave
Part of samples that modifies the system in…	2.09%	1.05%	24.96%	72.95%

Table 5: System modification.

For more than 70% of samples the last code wave is the location where interaction with the system takes place. This is a strong indicator that the last code wave usually implements the malware payload. On the other hand, very few samples modify the system in the first or the middle code waves. The objective of those first code waves is therefore not to implement the payload.

Finally, there are a surprisingly high number of samples that modify the system in the penultimate code wave. This observation contradicts our research hypothesis, as we postulated that only the last code wave would interact with the system. To understand that behaviour, we took the following steps:

- We measured the number of samples modifying the system *both* in the penultimate *and* the last code wave. We found that none of the samples actually modified the system in both code waves.
- We manually analysed the samples with system modification in the penultimate code wave. The concerned samples belong either to the Conficker family, or to the Ramnit family. In both cases, we observed the same behaviour in the last code wave: hooks. This technique consists of writing small code snippets at the beginning of library functions, in order to redirect the execution flow to the malware code when those library functions are called. Having hooks in its own process can make the monitoring of the execution flow harder.

  As hooks are dynamically written code, they belong to a new code wave according to our definition. In both families the malware payload puts the hooks in place, and hence this payload becomes the penultimate code wave, while the hooks code becomes the last one. This is the reason we have system modification in the penultimate code wave in these samples, and no system modification in the last code wave.

Payload functionalities

To establish more precisely the participation of each code wave in the payload, we analysed the library function calls.

To do so, we classified *Windows* library functions in the following four groups related to malware payload:

- Network communications
- *Windows* registry manipulations
- Process manipulations
- File manipulations

We then measured which code wave makes more than three calls to the functions of each group. The results are presented in Table 6.

	…the first code wave	…the middle code waves	…the penultimate code wave	…the last code wave
Network communications in…	0%	0%	0.17%	19.55%
Windows registry manipulations in…	0%	0%	1.05%	28.62%
Processes manipulations in…	0.87%	0.35%	7.33%	44.15%
Files manipulations in…	10.99%	0.35%	7.33%	36.47%

Table 6: Functionalities.

We observe that the last code wave participates significantly more than the others in all four functionalities. Therefore, the last wave is usually the one that implements the malware payload. The others waves tend not to implement the monitored functionalities, and hence are not part of the malware payload.

6. HYPOTHESIS VALIDATION

We established that more than 98% of our malware samples are packed. Based on the previous experimental measurements, we can now check the validity of our research hypothesis, namely the prevalence of a packer model with the following two characteristics in successful malware:

1. '*The first code waves – all code waves except the last one – serve solely to protect the malware payload.*'

   This has been **partially validated** due to the following observations:

   a. The first code wave is particularly aggressive:

      i. It contains exotic machine instructions in more than 10% of the samples, whereas the other code waves only have standard instructions.

      ii. It misused function call conventions in more than 10% of the samples, whereas the other code waves do not have such bad function calls.

      iii. It throws exceptions in more than 20% of the samples, whereas the other code waves rarely do.

   b. The middle code waves do not modify the system (less than 2% of the samples). In particular, they do not implement payload functionalities (less than 1% of the samples). This is the same for the first code wave.

   We conclude that the first and the middle code waves do not participate in the payload implementation, but are rather here to protect the malware.

   However, the purpose of the penultimate code wave is not always protection. As we have seen, it can be the location of the malware payload in specific cases. In those cases, the boundary between the protection and the payload is before the penultimate code wave and not before the last one.

2. '*The last code wave contains the malware payload.*'

   This has been **partially validated** due to the following observations:

   a. The last code wave tends to be standard code. It contains no exotic machine instructions, very few exceptions thrown, and a lot of classic function calls. This code wave was therefore likely produced by a standard compiler.

   b. The system is modified in the last code wave in more than 70% of the samples.

   c. The last code wave is the only code wave where a significant number of samples implement their payload functionalities.

We conclude that the last code wave implements the malware payload, with the exception of some special cases where it just contains hooks.

Our experiment hence shows that packers used by successful malware tend to simply add code layers around the payload. Also, the payload is not necessarily in the last code wave, but can also be in the penultimate code wave.

7. CONCLUSION AND FUTURE RESEARCH

This paper presented the first large-scale experiment we conducted with the WaveAtlas framework. We analysed malware packers and partially validated that a certain packer model was prevalent among successful malware.

The simplicity of the packer model we exposed may come as a surprise. Why are malware still using such straightforward packers after all these years? It seems that, as defenders, we have failed to deal with the problem. A possible explanation is that packers are actually posing problems for anti-virus products, which is likely due to their very limited resources on users' machines. In others words, packers following the standard model may seem easy to unpack for a human analyst with unlimited execution time, but are much more difficult to deal with in the constraints of real-time protection on end-user machines. Therefore, malware writers do not need to adopt more complex packers, as their primary goal is to infect their targets.

An important observation made during our experiments is that the purpose of a code wave is either to protect or to implement the payload, but rarely both at the same time. We believe this should lead to tailored program analysis, either focused on code produced by classic compilers, where for example functions can be defined in the usual way, or on code

produced by packers, where we need abstractions of the code other than functions.

As a first research avenue with WaveAtlas, we could specifically study some long-running malware families, like Sality, which has existed since 2003, to check the evolution of their protection over the years. This would give us an idea of the impact of security products, e.g. by answering questions such as whether malware writers had to change their protections at some point.

Another interesting possibility is to establish the real prevalence and importance of public packers (commercial or free) among malware. The usual way to detect the usage of those packers is through the use of static tools (e.g. PEiD, SigBuster) which only check the first code wave. However, public packers can also be used in internal code waves, and even when they are present in the first code wave they can be combined with other packers. Thanks to its dynamic approach, WaveAtlas could provide a clearer picture of the usage of those packers.

REFERENCES

[1] Calvet, J. (2013). Analyse Dynamique de Logiciels Malveillants. Ph.D. Thesis. Université de Lorraine & École Polytechnique de Montréal.

[2] Brosch, T.; Morgenstern, M. (2006). Runtime packers: the hidden problem. BlackHat USA.

[3] Morgenstern, M.; Pilz, H. (2010). Useful and useless statistics about viruses and anti-virus programs. Proceedings of the CARO Workshop.

[4] Reynaud, D. (2010). Analyse de codes auto-modifiants pour la sécurité logicielle. Ph.D. Thesis, Université de Lorraine.

[5] Ugarte-Pedrero, X.; Balzarotti, D.; Santos, I.; Bringas, P. G. (2015). SoK: Deep Packer Inspection: A Longitudinal Study of the Complexity of Run-Time Packers. Proceedings of the IEEE Symposium on Security and Privacy.

[6] Wicherski, G.; et al. (2008). Stormfucker: Owning the storm botnet. 25th Chaos Communication Congress (CCC).

[7] Microsoft (2010). Deactivating botnets to create a safer, more trusted internet. http://www.microsoft.com/mscorp/twc/endtoendtrust/vision/botnet.aspx.

[8] Villeneuve, N.; Deibert, R.; Rohozinkski, R. (2010). Koobface: Inside a crimeware network. Technical Report, Munk School of Global Affairs.

[9] Microsoft (2013). Microsoft collaborates with industry to disrupt conficker worm. http://www.microsoft.com/en-us/news/press/2009/feb09/02-12ConfickerPR.aspx.

[10] Stone-Gross, B.; Holz, T.; Stringhini, G.; Vigna, G. (2011). The underground economy of spam: A botmaster's perspective of coordinating large-scale spam campaigns. Proceedings USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET).

[11] Hanel, A. (2013). Malware analysis search. http://www.google.com/cse/home?cx=011750002002865445766%3Apc60zx1rliu.

[12] Malware.lu (2013). Malware Sample Database. https://avcaesar.malware.lu/.

[13] High Security Laboratory (2015). http://www.lhs.loria.fr/.

[14] Luk, C.-K.; Cohn, R.; Muth, R.; Patil, H.; Klauser, A.; Lowney, G.; Wallace, S.; Reddi, V. J.; Hazelwood, K. (2005). Pin: building customized program analysis tools with dynamic instrumentation. Proceedings of the 26th ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI).

[15] Hungenberg, T.; Eckert, M. (2010). INetSim. http://www.inetsim.org/index.html.