Incremental clustering of malware packers using features based on transformed CFG

Ludovic Robin, Corentin Jannier, Jean-Yves Marion

Incremental clustering of malware packers using features based on transformed CFG

Ludovic Robin, Corentin Jannier, Jean-Yves Marion

December 10, 2022

Abstract

Packer detection is an important topic because most malware is packed and this allows it to avoid detection based on static analysis. Identifying classes of packers is the key to effective detection because it makes it easier to determine from a static analysis whether further analysis is needed or whether a decision is already possible. Thus in this work we propose new features to cluster packers from their unpacking function. This method makes it possible to effectively cluster packers, and is able, by clustering, to identify packer classes used by malware. It is a step towards a larger data clustering allowing to identify custom packers.

1 Introduction

1.1 Context

The emergence of new malware is a trend that represents an increasingly important issue in our information systems. Detecting malware efficiently is crucial in any information system. Detection engines can use a static analysis that will analyze syntax of programs or a dynamic analysis that will execute programs in a controlled environment. Static analysis has the advantage of being very fast in the general case, while dynamic analysis is more able to bypass obfuscations but much more expensive in terms of analysis time. To counteract these analyses, malware authors will use different obfuscation methods. In the case of dynamic analysis, these obfuscation methods mainly consist in recognizing the fact that the malware is running in a controlled environment and masking its behavior to try to fool the controller. In the case of static analysis, malware will try to hide their malicious code behind a code obfuscation. A classic method to obfuscate the code of a program is to use a packer.

The purpose of these packers is to compress and/or encrypt the program code and create a new program that will consist of a decryption procedure and the original program code in encrypted form. Since static analysis does not run the program, it is stuck with an unpacking function.

Software cannot be considered malicious just because it has been packed. It is quite common to use a packer for legitimate reasons. Packers allow to

compress and therefore optimize the size of files. They also allow to protect the binary code of the file for intellectual property reasons. Therefore, many legitimate files are packed. Malware authors also use the most common packers to obfuscate their malicious program. For example UPX [1] is a packing program that is widely used both in the malware domain and for harmless files, it uses the UCL compression algorithm and provides a decompression procedure that from the packed file can quickly recover the original file. The problem for malware authors using these common packers is that this packers are well known and that their decompression procedure, when not provided directly as in the case of UPX, can be performed by the large malware analysis community. With this in mind, malware authors modify these packers to try to make themselves undetectable. They can also make homemade packers to combine obfuscation and discretion. These homemade packers can be more or less extensive modifications of existing packers or brand new ones. Here too, unpacking methods based on targeted emulation, for example, can be used to unpack malicious packers. The detection of packers thus becomes a challenge of static analysis because it allows, by identifying the class of packers, to propose unpacking methods to push the analysis further. Hence, knowing which packer class we are dealing with when analyzing suspicious files is a crucial issue in malware analysis. Answering this question is also important because it allows decisions to be made about which further analysis methods to use. This requires to understand packers data retrieved from various malware sources and to do so classify this data in some way. Cluster packers into distinct classes allows to make some assumption on a packer class and is a huge help to detection engines.

1.2 Source exploration and clustering

In order to keep up with the constant evolution of malware, new samples must always be collected. The malware analysis community puts a lot of effort into making these samples available and, complemented by the deployment of honeypots, it is possible to actively track the evolution of malware. Thus, our data consists of a large number of files that are constantly evolving and they must be well labeled to be used for detection. The community provides labels for a part of the malware files, for example MalwareBazaar[2] labels with good precision part of malware present in its database. But this labeling only concerns the malware payload, it rarely gives information about the obfuscations or packaging applied to this malware, in particular it does not give information about which packer was used to obfuscate a malware. This is why it is difficult to automatically build packer databases using these sources. Tools based on signature methods such as ClamAV[3] for malware detection or DIE[4] for packer detection, answer to part of this problem, by generally updating their database with manual intervention. This allows malware analysts to identify the characteristic features of the malicious file in order to make a suitable signature and provide some tagging data. But this manual interaction is responsible for delays in taking into account new threats and takes the risk of not detecting an attack in time. Therefore, we place ourselves in the case of automatic gen-

eration of signature databases and we must answer the problem of identifying classes of packers for weakly labeled data. To identify these classes of packers we can cluster our data using clustering algorithms. Clustering is the process of dividing data into subsets (clusters) according to relevant criteria. The elements to be clustered can be considered only once, at the initialisation step, which is the standard procedure for most clustering algorithm. In our case we have to consider our data as a continuous flow in real time because of malware packer evolution. Thus, we have to apply incremental clustering methods. In this paper we use the incremental DBScan clustering algorithm that we detail in Section 2.2. To distinguish these classes of packers we focus on the static analysis of their unpacking function. We apply transformations on this unpacking code to avoid changes in the code that would not be characteristic. This step of extracting relevant data from our samples is described in Section 2.1.

1.3 Ambition

This is a preliminary work that introduces a method to distinguish classes of packers. By clustering large amounts of data we can determine crucial information about packer classes without labeling them precisely. This way, we can interpret clustering results with the aim of improving a detection engine. To do this, we can take advantage of our trust in the sources from which the malware and goodware are fetched. Indeed, a clustering on this scale makes it possible to distinguish classes of packers which concern both goodware and malware. The composition of these classes of packers is a good indicator to notice that they are common packers, such as UPX or MPRESS. On the contrary, if these classes of packers are only composed of malware then this gives a good indication that they are characteristic of the malware world. For example it can be packers purchasable on the darknet which can be used by many different malwares. Also, considering the labeling of the malware community and in particular the most serious sources, we can consider our classes of packers through the labeling of their payloads. If in the composition of a class of packers we always find the same payload labeling, then we may have to deal with a packer specific to a particular malware. The reasoning is also true for groups of malware authors. Figure 1 illustrates this general reasoning. This characterization is the goal of our work and the case studies of Section 3 goes in this direction.

1.4 Related work

There are works on the packer classification problem in literature. Most studied techniques are based on yara-like signatures [5], and DIE [4] (Detect It Easy) is a classic tool using this method, also PEiD [6] is well known but does not evolve anymore. These tools are used in major malware detection platforms. Yara-like signatures in these tools are generally manually wrote which is not adapted to rapidly evolving malware. As a result these needed manual intervention are responsible for delaying detection of new packers. Machine learning methods are also used for packer classification, particularly supervised learning methods as in

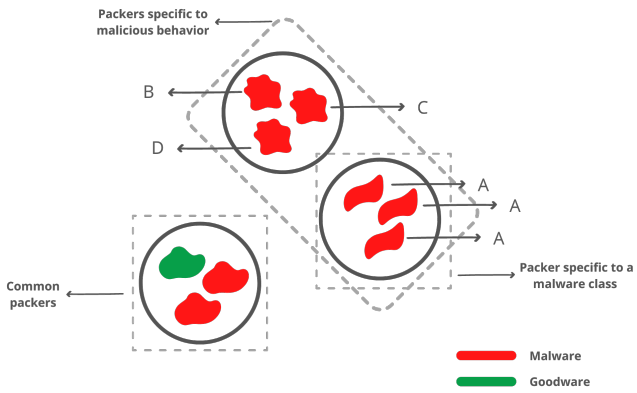

Figure 1: Packer classes identification

[7]. These packer classifier models perform well for classifying packers belonging to classes that the model trained from, but it is not adapted to evolving and new packers that appear in the wild. The method described in [8] is the closest to our work in terms of transformations on the graph (Section 2.1), but they do not validate their model experimentally. Moreover, their way of considering the graph as a whole seems less suitable to clustering approaches. The work of Noureddine et al.[9] uses the same incremental clustering method and performs very well on their custom set of packers. It has demonstrated the relevance of incremental clustering in the context of highly evolving packers. However, more tests on custom packers used by malware are needed to have more confidence in this model. Also, it might be necessary to weight the different features by their relevance.

2 Methodology

2.1 Part of transformed CFG as features

We want to capture similarities between packed files. Extracting features from our files that will match this ambition is necessary and is the main difficulty. We must choose features that correspond to relevant characteristics for clustering our files. In related work [9] features selection is focused on metadata and pondered with the first few instructions extracted through `radare2` [10]. We decide to match packers through similarities of their unpacking code. To extract characteristics of this code, we disassemble statically this unpacking procedure, building a Control Flow Graph (CFG). We could compute a distance from

this CFG but it may pose a problem since it is not resistant to few syntactic modifications of a program code or modifications implied by new versions of code compilers. To handle this problem, after static disassembly of the program code we apply some transformations on the resulting Control Flow Graph (CFG). These transformations are based on previous work on the Gorille tool[11]. They are made for catching similarities between binaries and to be resistant to code modification. Principle is to focus the analysis on the CFG shape, in particular on conditional branching in the program. From this transformed CFG graph we compute several sub-graphs of 12 nodes, a sub-graph is computed for each node of the graph and correspond to a breadth-first exploration of the graph.

A sample can be represented as a set of sub-graphs, and each sub-graph can be signed in a unique way. In our case we chose to sign these sub-graph by computing a md5 hash. We can now compare this new form of samples by comparing the number of common sub-graphs signature between two samples. It is the problem of comparing two sets, and one notion of distance that can be used, and will be used in this paper is Jacquard distance which is a classic distance for set comparison. Jaccard distance of two sets A and B, $J_\delta(A, B)$ can be described as:

$$J_\delta(A, B) = 1 - \frac{|A \cap B|}{|A \cup B|}$$

This distance has the advantage of comparing the ratio of common elements in two sets, it is a good choice in our opinion when applied in our context since packers does not hide their extraction procedure among a larger code. If we want to take this hypothesis into account, for more obfuscated packer extraction procedure, the overlap distance[12] may be a better choice and will be investigated in future works.

2.2 Incremental DBScan

Since the files we need to cluster are constantly fetched from multiple sources, static clustering methods would be a large waste of time and computing resources since the entire clustering needs to be calculated again each time new points are added to the dataset. Another practical problem with static clustering is that many methods rely on a pre-computed distance matrix that would be of a prohibitive size when working with large datasets. We picked a DBScan[13] incremental clustering algorithm. DBScan is a density-based clustering method that is widely used. Density-based approaches usually consider elements in sparse areas as noise, which correspond to our context where we want to cluster files from the wild. The percentage of these files that will be unique and impossible to cluster should be important, hence DBScan will consider it as noise and won't be bothered. Its pairwise comparison also match our features (described in sub-section 2.1), because we do not have any order notions on our data and only pairwise comparison has a meaning is this context. Also incremental DBScan has already been used with some success in the work of Noureddine et al.[9] for metadata features.

Incremental DBScan, when clustering a new sample, will first find the closest cluster to the incoming data point, then check if the new point respect the density criterion to be integrated into the cluster (e.g. there are enough cluster points close to it). If the incoming point cannot be integrated into an existing cluster, the algorithm will try to create a new cluster by finding enough close neighbours in the data point currently classified as noise. Finally, if no new cluster could be created the incoming point is classified as noise and the procedure repeats for the next data point. See Figure 2.

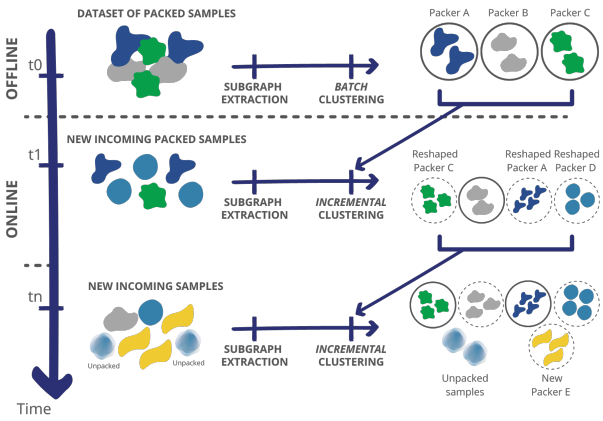

Figure 2: Incremental clustering process for packed files

3 Case studies

3.1 Manually packed samples

For initial testing and evaluation of the clustering method, it is necessary to have a selection of samples that are packed with known packers and to know specifications that can act as a ground truth when judging cluster coherence and quality. But a problem when dealing with packed files from an outside source is that it can be very difficult to verify which packer was used on the file, even when it is a given information, since errors in packer identification are common and hard to detect and would greatly pollute a selection of test samples. As such, a good way to obtain a controlled set of samples is to manually pack a control set of unpacked executable files using different packers. For this experiment the starting set of unpacked files was created from approximately 600 files obtained from a freshly installed Windows 10 distribution that were subsequently packed using the following twelve packers: aspack, mew, packman, pecompact, pelock,

petite, rlpack, telock, themida, upack, upx and yoda. This operation yielded 5912 packed files (Table 1), with the difference from the expected 7200 being explained by not all packers being able to pack every original file, with the most common limitation being x64 executables not being supported by the packer.

packer name	**successfully packed count**
pecompact	461
upack	661
rlpack	636
upx	600
telock	413
yoda	427
themida	560
petite	426
aspack	429
mew	429
packman	432
pelock	421

Table 1: Manually packed data-set

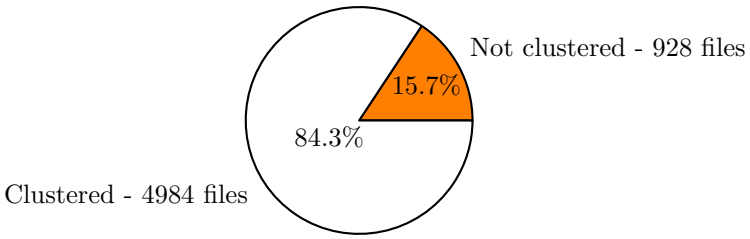

Figure 3: Manually packed samples clustering ratio

Results Our clustering method managed to cluster 85% of packed files (Figure 3), since our criteria is a minimal of 4 elements with appropriate distance by cluster, 15% of our samples could not be clustered with 3 others similar samples. This ratio of 85% of clustered samples is an encouraging result. To focus first on well clustered results (Table 4), we successfully clustered 11 of the 12 packers we initially picked for this study with an average success of 86%. We have 5 packers that are fully well-clustered meaning that we regroup in one cluster all samples of the same packer. One of these packed files (telock) have been clustered into 2 separate clusters (clusters 3,4), we suspect that it is due to variations in the unpacking procedure depending on the file that is packed.

Only one of these packers has not been clustered and it is due to limit of static analysis approach. We use standard static analysis approach to disassemble programs and pecompact jumps to register values in its unpacking procedure. This limit can be surpassed by using a static analysis tool that tracks register values by symbolic analysis.

packer name	**unclustered count**	**unclustered ratio**
pecompact	459	99.6%
upack	215	33%
rlpack	181	28%
upx	42	7%
telock	20	5%
yoda	1	0%
total	928	15.7%

Table 2: Manually packed unclustered samples

cluster №	**packer**	**count**
14	pecompact	2
	upack	1
	upx	2

Table 3: Manually packed wrongly clustered samples

3.2 Zeus malware case

In this use case we want to show how this clutering allows us to match our ambition (see Section 1.3) on a reduced set of malware samples. To do so we want to cluster some goodware that are packed (here they are part of the previous set of goodware that we pack by ourselves) with some malware that is packed in unknown ways. By doing this we want to show that we can cluster Zeus packed samples with goodware packers. This is to conclude that we have successfully identified a class of packers that is used to pack both harmless programs and Zeus malware. We also want to use this test to compare our clustering method with the DIE packer detection tool.

We focus on Zeus/Zbot malware which is a trojan designed to steal banking information. It is also known to be packed by different packers. In this experiment we won't control which packers are used to pack the Zeus malware payload, we will pick some Zeus samples randomly from multiple sources and only check if they are packed. Checking if Zeus samples are packed is not a trivial issue and we use dynamic analysis to ensure this.

cluster №	packer name	count	packer family ratio
1	upack	446	67%
2	themida	560	100%
3,4	telock	388 5	94% 1%
5	petite	426	100%
6	aspack	429	100%
7	upx	556	93%
8	mew	428	99.8%
9	packman	432	100%
10	pelock	421	100%
11	rlpack	455	72%
12	yoda	426	99.9%

Table 4: Manually packed well clustered samples

Confidence in packing properties The notion of wave is developed here [14] and is built to identify self-modifications of a program's code. It consists in running a program in a sandbox and trace its execution. That is, if a program allocates memory or rewrites its own code in order to then execute these newly instantiated instructions, this modification operation is written into a what is called a wave. These waves describe exactly what kind of obfuscation a packer apply, the concept of wave is more inclusive than just considering packers but it considers any packer-like behavior. Thus, by analyzing our Zeus malware with such a process, we can identify whether they use packer-type procedures or not by the fact they have at least one rewriting wave. We also completed this approach by adding files detected by DIE as packers.

Samples selection We have recovered 1000 randomly picked unique samples of the Zeus malware from MalwareBazaar and VxUnderground, these malware are labeled as Zeus on these sites. This labeling has been confirmed by VirusTotal which is an online antivirus aggregator. We therefore consider with a high degree of certainty that our test samples all match the Zeus malware. These Zeus files were analyzed by the DIE[4] packer detection tool in order to compare the DIE labeling with our clustering. Additionnaly, with these files we clustered our 600 manually packaged UPX to see if some cluster contains both goodware packers and packed Zeus samples. Among these data, 685 were named by DIE, our 600 manually packed UPX and 85 Zeus samples.

Results We clustered 93% of our elements (see Figure 4) into 10 clusters. These 10 clusters are decomposed into 3 clusters of more than 400 elements (clusters 8,9,10) and 7 much smaller clusters (see Table 5). Clusters 8 and 10 are each composed of more than half of the copies of UPX that we had packaged ourselves. It seems that we have clustered, with these manually packed UPX

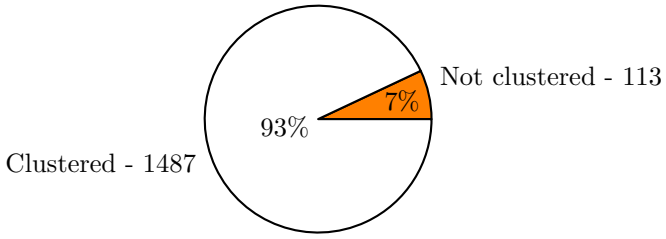

Figure 4: Zeus and UPX packed samples clustering ratio

samples, some Zeus samples which are packed using a packer very similar to UPX, if not UPX itself. This conclusion is reinforced by DIE's labeling of about 25% of the Zeus samples in cluster 8 as UPX, as well as some Zeus copies as UPX for cluster 10. Looking at these clusters we can see that while DIE agrees in general for most elements, it does not draw the same conclusions as our clustering algorithm, even on classical packers like UPX. It is difficult to draw more conclusions at this stage of our work because we were not able to obtain labels with more confidence for these elements. In any case, we can show here that for labeling packers on malware DIE performs poorly. Cluster 4 is smaller but it is interesting to see that our clustering here is very consistent with DIE labeling. It seems that we have identified here with our clustering a class of packers corresponding to MPRESS with 100% success according to DIE.

cluster №	**DIE information**	**Manually packed upx**	**count**
1	∅	0	41
2	∅	0	17
3	upx: 2	0	42
4	mpress: 13	0	13
5	∅	0	10
6	upx: 1	0	7
7	∅	0	6
8	upx: 434	381	436
9	upx: 1 spoon studio: 3	0	508
10	upx: 221 spoon studio: 11	219	407

Table 5: Zeus and UPX well clustered samples

4 Conclusion

Our method based on transformed CFG sub-graphs seems promising. Relying on the unpacking function extracted statically from the files to cluster similar packers gives very good results for our controlled test set (Section 3.1). Our method performs matching on packers in a completely generic way. Thus we can apply it to the case of custom packers used by malicious files. The case study in Section 3.2 is a clustering of packers used by Zeus malware. We managed to cluster goodware manually packed by us with Zeus packed malware files. It seems that our verification with the DIE tool supports our clustering of UPX for Zeus. Moreover, if we believe the DIE results we have clustered another sample of Zeus which is packed by MPRESS. It would be now a question of studying the clusters of Zeus not recognized by DIE, which could mean that we identified another common packer that neither us nor DIE have exposed (by lack of clustered data in our case). These clusters could also signify that it is a custom packer used by Zeus as in the article [15]. We are now thinking of clustering larger datasets to achieve our ambition presented in Section 1.3 which would improve the decision of malware detection engines.

References

[1] "Upx: the ultimate packer for executables - homepage." https://upx.github.io/. (Accessed on 12/07/2022).

[2] "Malwarebazaar — malware sample exchange." https://bazaar.abuse.ch/. (Accessed on 12/07/2022).

[3] "Clamavnet." https://www.clamav.net/. (Accessed on 12/08/2022).

[4] "horsicq/detect-it-easy: Program for determining types of files for windows, linux and macos.." https://github.com/horsicq/Detect-It-Easy.

[5] "plusvic/yara: The pattern matching swiss knife." https://github.com/plusvic/yara.

[6] "wolfram77web/app-peid: Peid detects most common packers, cryptors and compilers for pe files.." https://github.com/wolfram77web/app-peid. (Accessed on 12/07/2022).

[7] F. Biondi, M. A. Enescu, T. Given-Wilson, A. Legay, L. Noureddine, and V. Verma, "Effective, efficient, and robust packing detection and classification," *Computers & Security*, vol. 85, pp. 436–451, 2019.

[8] M. Saleh, E. P. Ratazzi, and S. Xu, "A control flow graph-based signature for packer identification," in *MILCOM 2017-2017 IEEE Military Communications Conference (MILCOM)*, pp. 683–688, IEEE, 2017.

[9] L. Noureddine, A. Heuser, C. Puodzius, and O. Zendra, "Se-pac: A self-evolving packer classifier against rapid packers evolution," in *Proceedings of the Eleventh ACM Conference on Data and Application Security and Privacy*, pp. 281–292, 2021.

[10] "radare2." https://www.radare.org/n/.

[11] P. Antoine, G. Bonfante, and J. Marion, "Gorille: Efficient and relevant software comparisons," *ERCIM News*, vol. 2016, no. 106, 2016.

[12] M. Vijaymeena and K. Kavitha, "A survey on similarity measures in text mining," *Machine Learning and Applications: An International Journal*, vol. 3, no. 2, pp. 19–28, 2016.

[13] M. Ester, H.-P. Kriegel, J. Sander, X. Xu, *et al.*, "A density-based algorithm for discovering clusters in large spatial databases with noise.," in *kdd*, vol. 96, pp. 226–231, 1996.

[14] G. Bonfante, J. Fernandez, J.-Y. Marion, B. Rouxel, F. Sabatier, and A. Thierry, "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*, pp. 745–756, 2015.

[15] "Zeus/zbot unpacking : analyse d'un packer customisé — connect - editions diamond." https://connect.ed-diamond.com/MISC/misc-051/zeus-zbot-unpacking-analyse-d-un-packer-customise. (Accessed on 12/09/2022).