Malware Variants Identification in Practice

Marcus Botacin[1], André Grégio[1], Paulo Lício de Geus[2]

[1] Federal University of Paraná (UFPR) – {mfbotacin, gregio}@inf.ufpr.br

[2]University of Campinas (Unicamp) – paulo@lasca.ic.unicamp.br

Abstract. *Malware are persistent threats to computer systems and analysis procedures allow developing countermeasures to them. However, as samples are spreading on growing rates, malware clustering techniques are required to keep analysis procedures scalable. Current clustering approaches use Call Graphs (CGs) to identify polymorphic samples, but they consider only individual functions calls, thus failing to cluster malware variants created by replacing sample's original functions by semantically-equivalent ones. To solve this problem, we propose a behavior-based classification procedure able to group functions on classes, thus reducing analysis procedures costs. We show that classifying samples according their behaviors (via function call semantics) instead by their pure API invocation is a more effective way to cluster malware variants. We also show that using a continence metric instead of a similarity metric helps to identify malware variants when a sample is embedded in another.*

1. Introduction

Malware are persistent threats to computer systems security, causing financial and image losses to public and private organizations. Threats like the `WannaCry` ransomware [Microsoft 2017] caused companies and hospitals to shutdown their operations [Independent 2017]. To handle malware infections, security professionals rely on analysis procedures, which help them to gathering binary information such that proper countermeasures can be developed, such as vaccines [Paleari et al. 2010] and incident response procedures [Souppaya and Scarfone 2013].

Analysis techniques can be classified as dynamic or static [Sikorski and Honig 2012], according whether samples are executed during inspection or not. Although these are distinct in nature, they may produce similar outcomes, such as Call Graphs (CGs) [Egele et al. 2008], our focus in this work. Whereas analysis procedures are effective against individual samples, they face a great challenge while handling the growing number of daily-identified, new samples [Test 2017]. Huge processing capabilities are required to handle this great volume of malware variants, thus making incident response procedures slower and more expensive. Despite such great number, in practice, many samples are variations of an original code [TechNative 2016], created using code morphing procedures or generation kits [TrendMicro 2017], being behaviorally similar, but statically distinct, which suffices for evading AV detection [Borello et al. 2009].

The similar source sample construction approach can be exploited by analysts by leveraging clustering procedures. Analyzing a single sample and extending the result for the whole family reduces the total analysis time and cost, thus making incident-response procedures faster and cheaper. The most often used clustering technique regarding CGs refers to metamorphism and polymorphism identification, taking API functions as features. These solutions are effective on clustering polymorphic samples generated by code reversion or transposition, but cannot assign semantic meaning to them. Therefore, malware variants exploring this drawback still have to be individually processed, thus raising costs. To solve this problem, in this work, we propose a clustering approach based on behavioral classes, allowing polymorphic malware to be effectively clustered. Our solution clusters samples which use distinct functions to implement the same behavior (e.g., using threaded API versions instead of processed ones), thus reducing analysis costs.

The differences of our solution over previous work are threefold: (i) First, unlike previous solutions, which identified similar code by performing static binary disassembly, thus being subject to obfuscation, our solution relies on dynamic binary execution in a hardware-based, transparent sandbox, thus achieving more precise results regarding CG reconstruction even face to evasive samples; (ii) Secondly, we propose to model malware samples as a graph of behaviors and not of functions. By associating functions with their intended behaviors, we can flag samples as similar even when they replace their functions by others having the same semantic goals; and (iii) Finally, instead of relying on a typical similarity metric, which is subject to be defeated by samples which pollute execution with innocuous calls (dead code), we propose comparing samples using a continence metric, flagging samples as similar when one is embedded in another, despite extra function calls present in any of them.

We evaluated the individual impact of each one of the proposed approaches with real malware samples and discovered that: (i) Common, benign function calls (e.g., WriteFile) mask similarity measures and removing them from the CG increased matching in up to 12.5% for Mimail malware samples; (ii) The consideration of a behavior model for malware execution instead of pure function calls helps identifying common constructions, such that Mimail malware samples similarity increased from 10% to 40% only by changing their representation to a behavioral model; (iii) Dead code masks common constructions between two samples, such that our continence metric increased Mimail malware samples similarity from 60% to 100%, thus showing that all behaviors of one sample are present in another, despite implementation differences; (iv) We compared our approach to two other malware clustering solutions [Shang et al. 2010, Carrera and Erdelyi 2004] to show that our behavioral approach outperforms function-based solutions. The most challenging Mimail samples to be classified by the related solutions (40% similarity) were classified as 90% similar by our one; and (v) Finally, we applied our solution to label a set of in-the-wild collected samples, including injectors, backdoors and ransomware. Our solution was able to identify six malware variants, a result confirmed by AV labels and outperforming fuzzy hashing similarity scores.

In summary, our contribution are the following: **1.** We propose identifying malware variants by classifying samples behaviors instead of function calls to avoid misclassifying similar malware which implement the same behavior using distinct functions; **2.** We propose a new classification for associating function calls to malware behaviors, thus supporting our solution's application; **3.** We propose a new metric for malware similarity identification which considers whether a sample is embedded in another instead of how large their intersection is, thus avoiding dead-code issues; **4.** We evaluate our solution with real malware samples collected in the wild and analyzed in a hardware-assisted, transparent sandbox to show its viability to identify malware variants in practical scenarios; **5.** We pinpoint future directions that might be followed by researchers tackling the malware similarity problem in actual scenarios.

This paper is organized as follows: Section 2 presents related work to better position our research; Section 3 introduces current solution's problems regarding data collection and clustering that limit malware variant identification in practice; Section 4 presents our proposals for a sandbox solution, behavioral classes and matching metrics to mitigate the previously presented malware variant detection limitations; Section 5 presents our solution evaluation against real-world malware samples to demonstrate its advances on malware variants detection; Section 6 discusses our results, limitations and future work; finally, we draw our conclusions in Section 7.

2. Related Work

Morphing Code Generation Code mutations, in a general way, are the procedures used to generate malware variants. Mutations can be classified into either structural and behavioral, or metamorphism and metamorphism. [Borello and Mé 2008] details the possible transformation which can be applied to binaries, such as code replacement, instruction swapping, variable changes, dead code insertion and control flow obfuscation. In this work, we focus on function replacement and permutation.

Morphing Code Identification Many researchers proposed ways of handling graph similarities for malware representation, each one tackling the problem by a distinct perspective. A first class of solutions directly targets the Control Flow Graph (CFG). [Bonfante et al. 2008] presents an architectural solution for detecting metamorphic changes on malware samples, combining syntactic and semantic analysis. [Martins et al. 2014] proposes an identification procedure by analyzing the virtual structures differentiation on dependence graphs. [Christodorescu et al. 2005] presents algorithms and formal foundations for malware detection from behavioral patterns.

A second class of solutions intends to create an intermediary representation (IR) for feature description. None of them, however, address the behavioral issue. [Feng et al. 2014] presents a tainting technique over Android Call Graph (CG) aimed to identify similar flows. The identification relies on the underlying `Inter-Component CG` IR. [Shao and Smith 2009] presents the use of an IR associated to the *Latent Semantic Indexing* technique over the CG to identify malicious portions of code, being able to identify, for example, bugs or vulnerabilities.

A third class of solutions focus on solving the graph matching problem. Such solutions can be applied to this work without major modifications. [Kostakis et al. 2011] presents the use of *simulated annealing* to identify graph similarity. [Wu et al. 2013] presents the use of graph-colouring algorithms and cosine-similarity measure to identify malware variants. Similarly, artificial intelligence-based techniques were also proposed to solve the matching problem. [Kong and Yan 2013] presents a learning algorithm for the structural information distance which describes two samples.

A fourth class of solutions focus on developing metrics and interpretations for the matching samples. Such methods can also be applied to this work. [Jang et al. 2014] handle graph similarity by applying social metrics. [Faruki et al. 2012] leverages `API CG-grams` associated to classification mechanisms for malware detection.

Finally, among all CG-based approaches, [Shang et al. 2010] presents a matching algorithm for malware variation detection which is the closest to ours. However, besides being limited to static disassembling, their work also does not handle the case where distinct functions implement the same behavior, as our work does.

Behavioral Classification Classifying samples according their behaviors is an often employed strategy by malware analysts, being extensively described in the literature. [Grégio et al. 2015], for instance, proposes a malware taxonomy based on dynamic analysis results, labeling each sample according defined behavior classes. This kind of work differs from the hereby proposed because we do not intend to classify whole-samples behavior but its internal parts, making the classification procedure more granular. In a similar way, [Paleari et al. 2010] proposes a high-level feature identification approach aiming to infection remediation. We leverage such kind of classification to allow CG clustering.

3. Current Solutions' Limitations

In this section, we revisit current solution's implementations, highlighting their drawbacks and pinpointing possible mitigations for them. We focused on two major implementation aspects: (i) the CG extraction; and (ii) the evasion problem due to function replacement.

Feature Extraction A critical step of any analysis procedure is data collection, because it gathers the data which will be the input of some analysis algorithm. Generic binary clustering solutions rely on static disassembly procedures for extracting function calls and instructions from the analyzed binaries. However, when analyzing malware, such solutions face the challenge of handling evasive samples able to apply anti-disassembly techniques, which can lead to low detection rates. [Branco et al. 2012] presents a series of anti-disassembly techniques which can fool both linear sweep as well as recursive traversal disassemblers. An incorrect disassembly can omit important function calls as well as include fake calls on the resulting trace.

Considering this scenario, an efficient feature extraction procedure is essential for properly clustering malware samples. A way of achieving coherent data extraction is to rely on a dynamic, transparent sandbox solution, such as the one presented in the following Section. By using such kind of solution, one can ensure the correctness of extracted features. Besides, it gives one the ability to analyze even evasive samples. This is particularly important because a malware variant can be only an armored version of a previously existing "plain" malware sample.

Same-Behavior Function Replacement A general graph modeling technique for malware applications is to consider each malware function as a graph vertex and relations among the functions, either data-dependency or temporal relations, as edges. An usual way of matching graphs is the strict vertex matching, on which same-label vertices (same function names) are considered as valid matches. Malware creators can bypass such kind of matching by replacing the original functions by others of equivalent behavior. As an example of such kind of replacement, Figure 1 and Figure 2 show CGs of two samples which make the same actions on the system, but using distinct API calls. We highlight that despite of using distinct O.S. abstractions, such as processes and threads, the same basic idea is deployed by both samples: adding a new code piece to the system.

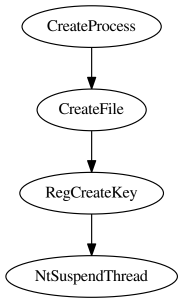


Figure 1. Original sample's CG.

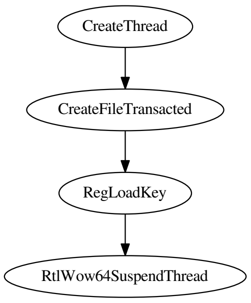


Figure 2. Variant sample's CG.

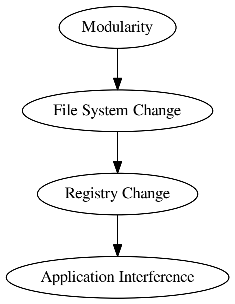


Figure 3. Behavioral graph from both samples.

On the usual approach, the strict similarity between the two graphs would be `null`, because no common functions were found. This false negative detection would require both samples to be analyzed independently by a second step analysis routine, which raises costs and the processing required. Such additional process would be avoided if a semantic-aware approach were adopted, such as considering the behavioral impact of each function instead of simply their labels (names). Figure 3 illustrates how a behavioral model of both samples would look like, thus resulting in a 100% matching. In the following section, we present a proposal for malware behavioral modelling.

4. Malware Variant Identification Proposal

In this section, we introduce our proposal for behavioral malware similarity matching. First, we present the feature extraction solution that we used to generate inputs for the matching algorithm. Second, we present the classes that we defined to model any sample execution. Finally, we discuss graph comparison metrics to evaluate sample's similarity.

Sandbox Solution Our sandbox solution [Botacin et al. 2018] is bare-metal-based, running on physical machines with Intel processors, which allows us to collect binary information at a lower level, using the processor branch monitoring unit (Branch Trace Store—BTS). This way, our solution does not required injecting any code into the monitored object, thus not being detected by malware samples checking for emulation side-effects or virtual-machine detection techniques. The solution captures system wide control flow deviation information (`RET,CALL,JMP`) using the BTS and stores then on a OS-supplied memory page. To isolate processes data, we relied on raising interrupts each time the buffer is written, collecting the provided source and target branch addresses in a step-basis and associating them with the running process.

Our solution is also able to reconstruct whole execution flow by using an introspection procedure, which allows us to bridge the semantic gap, retrieving high level semantic information. As we are interested on function calls, the low level instruction addresses provided by the BTS mechanism are compared against system loaded libraries and executable images, which allows us to discover the base address they point to. After that, the remainder of the calculation is used as an offset on the given library, The offset points to a function name, which will appear on the CG, as the monitor stores addresses of only taken branches. Thus, unlike in static disassembly toosl, no fake function call are included.

Behavioral classes Based on a malware taxonomy [Grégio et al. 2015], we defined the following classes to model sample's behaviors.

`Compression`: APIs related to file compression. They can be used for embedding and extracting files.

`Cryptography`: APIs related to cryptography. They can be used by ransomware samples, fingerprinting by hashing, and for anti-forensic purposes (encoding).

`Debug`: APIs related to debugging. They can be used to monitor and control other processes.

`Delay`: APIs related to execution suspension. They can be used to evade an analysis procedure due to timeouts.

`Environment`: API related to environment variables. They can be used to set general program settings, such as default program paths.

`Escalation`: API related to execution privileges: They can be used to escalate privileges in the system.

`Exfiltration`: API related to user and system information, such as computer name and serial numbers. They can be used for information stealing or fingerprinting.

`Fingerprint`: API related to assure system exclusive access, such as mutexes. They can be used to assure only one malware instance is running at time.

`File System`: APIs related to file system access. They can be used for general filesystem actions, such as storing downloaded data.

`Interference`: APIs related to processes control, such as thread enumeration, suspension, and/or memory writes. They can be used to interfere into another process and eventully hijack them.

`Internet`: APIs related to network communication. They can be used to exfiltrate data, download payloads from internet and perform network attacks.

`Modularity`: APIs related to process creation. They can be used to insert new components into the system, such as instantiating downloader's payloads.

`Monitoring`: APIs related to system monitoring. They can be used trace process, check if is being traced, and/or being alerted when system events happens.

`Registry`: APIs related to the system registry. They can be used for system, writing on Run registry key, for instance.

`Evidence Removal`: APIs related to file deletion. They can be used to remove infection evidence from the system.

`Side Effects`: APIs which cause execution side effects, such as system reboots, often associated to deep modifications on system configuration.

`System Changes`: APIs related to system level configurations, such as driver loading or boot options changes. They can be used by rootkits to hide their intents.

`Target Information`: APIs related to information retrieval. They can be used on the attack recognition phase, by getting memory addresses or process permissions.

`Timing Attacks`: APIs related to time measurement, such as timers and alerts. They can be used on timing attacks against sandboxes.

Matching Metrics The usual graph matching metric used by most solutions is the one presented in Definition 1. This definition considers the universe of both graphs as groundtruth, which makes the metric symmetric. In the case where the two graphs are equal, both union and intersection will be equal, leading to the maximum similarity (1). Totally distinct sets will lead to an empty intersection, thus to the minimum similarity.
Definition 1. *The similarity of two malware, represented as sets, A and B, of vertices or edges of two graphs, is defined as:*

$$Sim(A,B) = \frac{|A \cap B|}{|A \cup B|} \tag{1}$$

The major drawback of this metric is when a sample is embedded in a bigger sample, as shown in Figures 4 and 5. In this case, although the `Sample 1` is totally contained into `Sample 2`, the similarity measure is just 50%, because the additional code surrounding `Sample 1`'s embedding in `Sample 2`. This lack of similarity identification is a drawback for threshold-based approaches, which would fail to recognize samples as similar, causing analysis to occur separately, thus increasing costs.

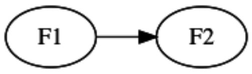


Figure 4. Sample 1. The original sample.

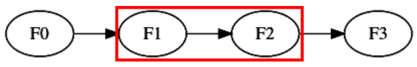


Figure 5. Sample 2. Variant sample embedding the original one.

To give more importance to the continence of a sample into another, we propose the matching metric presented in Definition 2. In this metric, instead the union of both samples, we consider one sample as groundtruth each time, thus solving a maximum subgraph isomorphism problem. In our example, as we consider `Sample 1` as contained into `Sample 2`, their union is the `Sample 1` itself, thus leading to the maximum similarity (100%).
Definition 2. *The similarity of two malware, represented as sets, A and B, of vertices or edges of two graphs, is defined as:*

$$Sim(A,B) = \max\left(\frac{|A \cap B|}{|B|}, \frac{|B \cap A|}{|A|}\right) \tag{2}$$

A drawback of this metric is that, as we are comparing unions to individual samples, the ratio is not symmetric, as shown in Table 1 and 2, for real `Mimail`'s variants. This way, we should consider both directions (1's embedding in 2 and 2's embedding in 1) to compute the maximum continence, as in the example presented on Table 3.

Table 1. Continence of Sample 1 in Sample 2.

CG	A	B	C
I	**0.66**	0.52	**0.64**
J	**0.75**	0.49	**0.50**
K	0.42	**0.80**	**0.44**

Table 2. Continence of Sample 2 in Sample 1.

CG	A	B	C
I	0.57	**0.56**	0.43
J	0.33	**0.51**	0.44
K	**0.76**	0.65	**0.44**

Table 3. Maximum continence of Sample 1 and Sample 2.

CG	A	B	C
I	0.66	0.56	0.64
J	0.75	0.51	0.50
K	0.76	0.80	0.44

5. Evaluation

When applied to binaries, our approach leads to behavioral constructions like the one presented in Figure 6. We here evaluate whether such constructions, along the proposed metric, are suitable for malware clustering in an effective way.

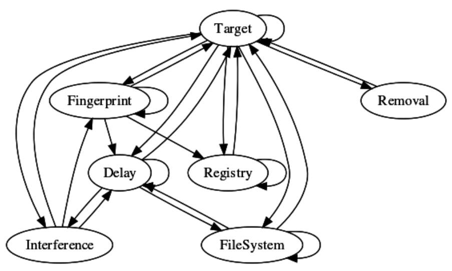


Figure 6. Behavior-based graph for a given sample.

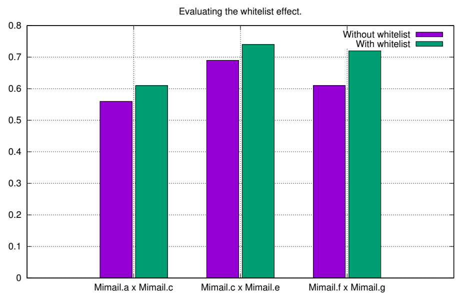


Figure 7. Evaluating whitelisting effect. Similarity scores are higher when using the whitelist-based approach.

General-purpose functions pollution As most malware clustering approaches do not consider function semantics, they end up considering general-purpose functions that are present both in goodware as well as in malware samples (e.g., OpenFile) and do not contribute for characterizing malware similarity. In practice, these functions only contribute for increasing CG size and making similar edges sparser. Malware creators may exploit this fact by intentionally adding general-purpose functions as dead-code to their variants, thus lowering sample's similarity rates. Our solution mitigates this problem by discarding these constructions via a whitelist of general-purpose functions.

To evaluate the impact of this choice, we first conducted a comparison among samples from the `Mimail`[1] family by evaluating them with and without function whitelisting and applying the usual comparison metric (still not applying behavior classification). Some comparison results[2] are shown in Figure 7. We notice that the whitelist approach outperforms the original strategy. In our tests, all `Mimail` samples presented higher scores when using the whitelist approach. We highlight the significance of this result by remarking that such improvement was achieved when considering Mimail samples, that are very

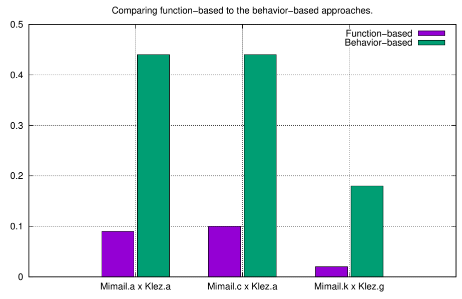
[1]We chose Mimail and Klez samples for our exploratory tests because they are available on the Internet and thus our results can be compared to other published research work.

[2]We present only some cases due to page length constraints

similar among themselves by construction. Therefore, we hereafter consider function whitelisting as basis for comparison.

Behavioral vs. Function-based classifications The major contribution of this work is to propose the use of behavioral classes to improve sample similarity detection. We here contrast the function-only approach (already whitelisting general-purpose function calls, as previously justified) to the behavior-based approach.

We evaluated the behavioral strategy by comparing the samples of `Mimail` family and the ones from the `Klez` family. The samples belong to distinct families and although they present the same infection goal, they are not very similar in terms of called functions, thus we can observe any change in similarity detection rates when applying distinct approaches. The tests were performed using the usual comparison metric and the results are presented in Figure 8.



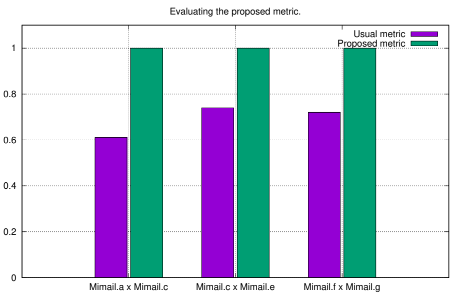


Figure 8. Function vs. Behavior-based approaches. Scores are higher when considering behavioral patterns.

Figure 9. Proposed metric. Scores are higher when using it in comparison to the usual one.

The behavioral approach presented higher scores compared to the function-only one for all the 180 compared samples, which corroborates our expectations that even samples presenting distinct functions can be considered as behaviorally similar. Therefore, we hereafter consider behavioral models for malware similarity evaluation.

Metrics Comparison The previous tests, using the usual similarity metric, showed that our proposed behavior-based approach improved samples comparison accuracy. However, this measurement still provides limited information, thus being unable to identify, for instance, whether a given sample is embedded in another. To fill this gap, we propose using the continence metric.

To verify the effectiveness of the proposed metric, we compared the results of the usual metric to the new proposed one when applied on `Mimail` family samples using the behavioral approach. The results presented in Figure 9 show that although the usual metric indicated samples shared significant snippets of code (more than 50% similar), it does not indicate whether one was embedded in the other. In turn, the continence metric is able to provide such information and, for the presented cases, all samples contained a complete variant of another behavior inside them.

Approaches comparison To verify whether our solution was able to improve malware similarity detection im comparison to the existing approaches, we compared our results against the ones from other works (here named Solution 1 [Shang et al. 2010] and Solution 2 [Carrera and Erdelyi 2004]). We evaluated all solutions using the same sample's families (`Mimail` and `Klez`) used in these work. All samples were retrieved from Vx-Heaven [VxHeaven 1999].

Our solution presented the same overall score levels as reported in these work. None of our scores were lower than the ones from the other approaches. Moreover, our solution scored greater values on specific cases, such as on the ones presented in Figure 10. Therefore, we hereafter consider our solution as basis for malware variant identification.

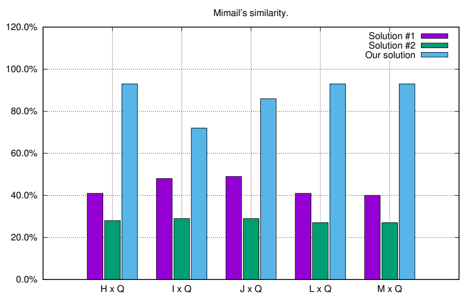


Figure 10. Mimail's sample similarity. Our solution's scores are higher when compared to other ones.

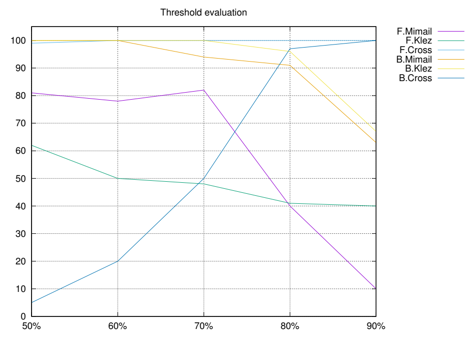


Figure 11. Threshold evaluation. This should be higher than `80%` in order to proper label the `cross` dataset.

Variant Identification The previously presented results showed that our approach outperforms related work regarding identifying whether the behavior of a sample is embedded in another. However, to classify samples in families, a similarity threshold must be defined. We here discuss how such threshold can be defined.

Figure 11 shows the results of samples classification using distinct thresholds. The `F.` lines correspond to results obtained using the function-based approach whereas the `B.` lines correspond to results obtained using the behavior-based approach. We evaluated `Mimail` and `Klez` families individually and also performed a `Cross`-verification, comparing samples from one family against the samples from the other one. From samples from the same family, the result is considered correct whether the sample's similarity is higher than the threshold, because we know *a priori* that they belong to the same family. On the `Cross` case, the result is considered correct whether samples' similarity is lower than the threshold, because we know *a priori* that they belong to distinct families. The graph shows the percentage of samples correctly labeled.

We observe that a lower threshold, such as `50%`, is not suited to classify samples as family variants or not. It happens due to the fact that the proposed behavioral model naturally increases sample's similarity, thus, at the same time that it increases the rate of correctly identified variants, it also labels distinct family samples as variants. Therefore, we need a tighter threshold so that we can properly classify samples as family variants. As the threshold increases, the number of correct labels on the behavioral approach also increases. It is important to notice that, in all cases, the number of same-family samples labeled correctly is higher on the behavior-based approach than in the function-based one.

In the `80%` threshold, the number of `cross-family` samples labeled correctly is very similar on both approaches whereas the `same-family` is greater on the behavior one, which constitutes a suitable threshold value. An increase to a `90%` threshold presents us the same accuracy on both approaches for the `cross-family` class, but the lowest value of `same-family` cluster similarity. Therefore, we hereafter consider a 80% threshold as basis for malware variant identification.

Study Case We here showcase the application of our solution to a set of 18 in-the-wild malware samples collected from a honeypot in the day before submiting this document. From all samples, our solution identified 6 as variants (33%), which is according to the average number of variants identified in the wild (**blinded reference**). The identified sample variants are shown in Table 4.

We notice that two variant families (cluster) were identified, presenting three samples each. To give a bit more information about them, we cross-checked our detection results to AV detection labels[3]. The first family is a generic one, having `backdoor` and `injector` characteristics. The second family is a ransomware-like/cryptor.

Table 4. Identified variants among unknown, wild-collected samples.

Family	Sample	Hash	Label
	A	c2ef1aabb15c979e932f5ea1d214cbeb	Generic_vb.OBY
1	B	747b9fe5819a76529abc161bb449b8eb	Generic_vb.OBO
	C	39a04a11234d931bfa60d68ba8df9021	Generic_vb.OBL
	A	96d13246971e4368b9ed90c6f996a884	Atros4.CENI
2	B	e23588078ba6a5f5ca1c961a8336ec08	Atros4.CENI
	C	31a2b6adc781328cb1d77e5debb318ff	Atros4.CENI

To provide more insights about our solution's efficiency, we compared our continence results to the fuzzy hash similarity scores provided by the `ssdeep`[4] tool, as presented in Figure 12. Our solution scored higher than the similarity measure provided by `ssdeep` in all cases. Besides, all `in-family` comparison results are higher than the previously established threshold of `80%`.

In addition to these results, we also evaluated the portion of reused code on each samples—what we called coverage. In most samples, the coverage is as high as the continence itself. In two cases from the second family, a small portion of code was reused. These cases illustrate how our approach contributes to the development of better clustering analysis procedures.

6. Discussion

Solution Limitations Besides implementation issues, which are more related to the algorithm complexity field, an inherent limitation of our solution is that it only considers imported API functions. An evasive sample could implement its own functions, so a given behavior would not be identified. A solution which considers internal function calls is an extension from this work. In addition, we also have to manually determine the behavior

[3]We chose AVG as groundtruth because it was the only AV which detected all samples
[4]`http://ssdeep.sourceforge.net/`

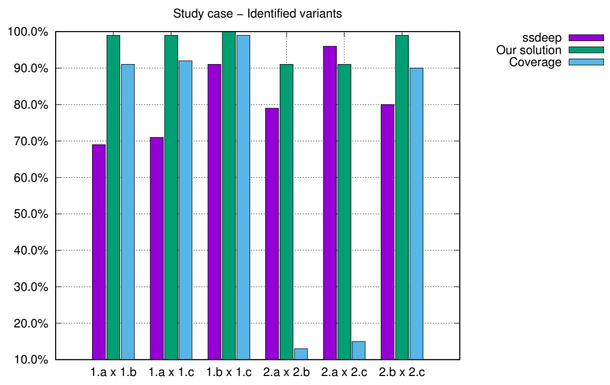


Figure 12. Study case: variant identification. Our approach outperforms others even on low coverage scenarios.

classes, so our results are dependent of these. If new functions and libraries were required to be included in the sample characterization procedure, a manual update would be required. Therefore, our approach would benefit from solutions which try to automatically identify malicious behaviors, such as Jackdaws [Polino et al. 2015].

Library Organization Proposal The function-based, behavioral classification procedure could be eased if functions whose behaviors are similar could be clustered on the same library. A typical Windows library may present a myriad of behaviors. The `ntdll.dll`, for instance, presents I/O functions, such as `printf` and `scanf`, file system functions, such as `CreateFile`, process functions, such as `CreateProcess`, and so on. This organization requires manual processing to understand how each function behaves. Modern Windows APIs, such as `crypto` API, pack all behavior-related APIs on the same DLL, which makes an automated procedure development easier. If all APIs were presented in this same way, we could automate classes generations by just understanding the DLL "subject" and then parsing its exported functions. We know this proposal is not practical right now, since legacy support should be provided by OS vendors. However, this can be considered on newer systems versions.

Open Research Questions Whereas our proposed metric brings new possibilities, it also brings new questions. For instance, in our definition, the metric does not consider edges location in code. Therefore, a given behavior present in the binary begin and end have the same impact. Additional research is required to identify its impact over classification. Similarly, our metric is not weighted. Therefore, a sequence of 2 common behaviors is scored the same way as an n-long. Further investigation is required to integrate such information in the metric without losing generality and detection effectiveness.

Future Work As a future work, we are interested on identifying more complex behaviors. As an example, consider the DLL injection procedure identification problem. The injection is composed by many calls, which should appear on a given sequence (`OpenProcess` + `VirtualAlloc` + `WriteProcessMemory` + `CreateRemoteThread`). On a real scenario, these APis can be interleaved by many others, either legitimate or dead code ones, as shown on Figure 13.

By identifying such complex behaviors, we could build a DLL injection behavior class, as shown on Figure 14, which is much more semantically meaningful for classification.

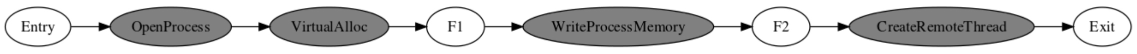


Figure 13. DLL injection functions among other function calls.

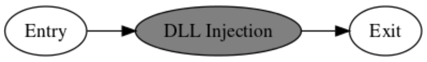


Figure 14. Proposed DLL injection class.

The development of such kind of detector is associated to complex graph matching algorithms, which may require heuristic procedures to be solved.

7. Conclusion

Clustering malware samples is essential to speed up analysis procedures face a scenario of multiple malware variants created every day. Therefore, we presented strategies for handling malware variants in practice. We tackled the problem from three complementary perspectives: i) by leveraging transparent data-collection techniques to mitigate the impact of malware variants created by adding evasive code; ii) by performing behavioral-based instead of function-based clustering to mitigate the effects of malware variants created by function replacement; and iii) by applying a continence instead of similarity metric to mitigate the effect of malware variants created by embedding a malware samples and adding dead-code to it. We first discussed the effects of each one of these factors individually and then presented the whole effect when combining the use of all of them. Our variant identification experiments on a set of real, unknown samples showed that the combined application of the three approaches outperforms other solutions. Finally, we discussed weak points and presented insights on how system libraries could be organized in the future to ease behavioral clustering procedures.

Reproducibility. All code developed for this research work is available at `https://github.com/marcusbotacin/Malware.Variants`.

Acknowledgments. This project was partially financed by the Brazilian National Counsel of Technological and Scientific Development (CNPq, PhD Scholarship, process 164745/2017-3) and the Coordination for the Improvement of Higher Education Personnel (CAPES, Project FORTE, Forensics Sciences Program 24/2014, process 23038.007604/2014-69).

References

Bonfante, G., Kaczmarek, M., and Marion, J.-Y. (2008). Architecture of a morphological malware detector. *JICVHT.*

Borello, J.-M., Filiol, E., and Mé, L. (2009). Are current antivirus programs able to detect complex metamorphic malware? an empirical evaluation. *EICAR*.

Borello, J.-M. and Mé, L. (2008). Code obfuscation techniques for metamorphic viruses. *JICVHT.*

Botacin, M., Geus, P. L. D., and Grégio, A. (2018). Enhancing branch monitoring for security purposes: From control flow integrity to malware analysis and debugging. *ACM Trans. Priv. Secur.*

Branco, R. R., Barbosa, G. N., and Neto, P. D. (2012). Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti-vm technologies. `https://tinyurl.com/y5f8kb3j`.

Carrera, E. and Erdelyi, G. (2004). Digital genome mapping - advance binary malware analysis. `https://tinyurl.com/y3klja7y`.

Christodorescu, M., Jha, S., Seshia, S. A., Song, D., and Bryant, R. E. (2005). Semantics-aware malware detection. In *2005 IEEE Sec. & Priv.*, pages 32–46.

Egele, M., Scholte, T., Kirda, E., and Kruegel, C. (2008). A survey on automated dynamic malware-analysis techniques and tools. *ACM Comput. Surv.*, 44(2):6:1–6:42.

Faruki, P., Laxmi, V., Gaur, M. S., and Vinod, P. (2012). Mining control flow graph as api call-grams to detect portable executable malware. In *Proc. Int. Conf. Sec. of Inf. and Net.*, SIN '12. ACM.

Feng, Y., Anand, S., Dillig, I., and Aiken, A. (2014). Apposcopy: Semantics-based detection of android malware through static analysis.

Grégio, A. R. A., Afonso, V. M., Filho, D. S. F., de Geus, P. L., and Jino, M. (2015). Toward a taxonomy of malware behaviors. *The Computer Journal*, pages 1–20.

Independent (2017). Nhs cyber attack: Int. manhunt to find criminals behind wannacry ransomware that crippled hospital systems. `https://tinyurl.com/mzsvkua`.

Jang, J.-w., Woo, J., Yun, J., and Kim, H. K. (2014). Mal-netminer: Malware classification based on social network analysis of call graph. In *WWW*. ACM.

Kong, D. and Yan, G. (2013). Discriminant malware distance learning on structural information for automated malware classification. In *SIGKDD*. ACM.

Kostakis, O., Kinable, J., Mahmoudi, H., and Mustonen, K. (2011). Improved call graph comparison using simulated annealing. In *SAC*. ACM.

Martins, G. B., Souto, E., de Freitas, R., and Feitosa, E. (2014). Estruturas virtuais e diferenciação de vértices em grafos de dependência para detecção de malware metamórfico. *Anais do XIV SBSEG*.

Microsoft (2017). Wannacry ransomware. `https://tinyurl.com/ljaz72z`.

Paleari, R., Martignoni, L., Passerini, E., Davidson, D., Fredrikson, M., Giffin, J., and Jha, S. (2010). Automatic generation of remediation procedures for malware infections. In *USENIX Sec.*

Polino, M., Scorti, A., Maggi, F., and Zanero, S. (2015). Jackdaw: Towards automatic reverse engineering of large datasets of binaries.

Shang, S., Zheng, N., Xu, J., Xu, M., and Zhang, H. (2010). Detecting malware variants via function-call graph similarity. In *MALWARE Conf.*

Shao, P. and Smith, R. K. (2009). Feature location by ir modules and call graph. In *ACM-SE 47*.

Sikorski, M. and Honig, A. (2012). *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. No Starch Press, San Francisco, CA, USA, 1st edition.

Souppaya, M. and Scarfone, K. (2013). Guide to malware incident prevention and handling for desktops and laptops. `https://tinyurl.com/kh4mnjv`.

TechNative (2016). Ransomware variants are now the top three most common malware types. `https://tinyurl.com/y5suauy9`.

Test, A. (2017). Malware statistics and trends report. `https://tinyurl.com/ycxdzkmz`.

TrendMicro (2017). Exploit kit. `https://tinyurl.com/yxgl3hf9`.

VxHeaven (1999). Vxheaven. `http://vxheaven.org/`.

Wu, L., Xu, M., Xu, J., Zheng, N., and Zhang, H. (2013). A novel malware variants detection method based on function-call graph. In *Conf. Anthology, IEEE*, pages 1–5.