Tracking Memory Writes for Malware Classification and Code Reuse Identification (Short Paper)

André Ricardo Abed Grégio[1,2], Paulo Lício de Geus[2], Christopher Kruegel[3], and Giovanni Vigna[3]

[1] Renato Archer IT Research Center (CTI/MCT), Brazil argregio@cti.gov.br
[2] University of Campinas, Brazil paulo@las.ic.unicamp.br
[3] University of California, Santa Barbara, USA {chris,vigna}@cs.ucsb.edu

Abstract. Malicious code (malware) is used to steal sensitive data, to attack corporate networks, and to deliver spam. To silently compromise systems and maintain their access, malware developers usually apply obfuscation techniques that result in a massive amount of malware variants and that can render static analysis approaches ineffective. To address the limitations of static approaches, researchers have proposed dynamic analysis systems. These systems usually rely on a sandboxing environment that captures the system calls performed by a program under analysis. In this paper, we propose a novel approach to capture and model malware behavior that is based on the monitoring of the data values that a certain subset of instructions writes to memory during program execution. We have implemented a malware clustering component and a component to detect code reuse between different malware families. To validate our proposed techniques, we analyzed 16,248 malware samples. We found that our techniques produce clusters with high accuracy, as well as interesting cases of code reuse among malicious programs.

1 Introduction

Malicious software (malware) is a significant threat for cyber security. Current malware operations vary from stealing sensitive data to attacking critical infrastructures. Today's malware employs many different ways to propagate, including social engineering techniques to deceive a user to click on e-mail attachments, and drive-by download attacks that exploit web browsers and their plug-ins In addition, obfuscation techniques are a powerful tool to render static malware analysis approaches ineffective and to decrease detection from signature based scanning. To address this problem, researchers have proposed dynamic analysis systems, which rely on the observed runtime activities (behavior) for detection and classification. To capture and model the behavior of malicious code, dynamic analysis systems typically rely on system calls. They treat the program as a black box and capture activity at a relatively high level. For example, two programs might be very different "inside" but might yield the same, visible effect to the "outside" by invoking the same system calls. While this might not

be an immediate problem for malware detectors, it makes it hard to distinguish between different malware families.

To address the limitations of system-call-based detection and classification, this paper proposes a novel approach to capture and model program (malware) behavior. We record a trace that contains all the values that (a certain subset of) instructions write. These writes can go either to a destination register or a memory location. By looking at the intermediate data values that a computation produces, we analyze the execution of a program at a much finer level of granularity than by simply observing system calls. The main intuition is that by using the data values, we can produce a very detailed profile that captures the activity of individual functions. Also, data values are tied very closely to the purpose (semantics) of a computation, and, hence, are not as easy to disguise as the code that performs the computation. Malware authors have introduced many ways in which code can be altered so that syntactically different instructions implement the same algorithm (e.g., dead code insertion, register renaming, instruction substitution). However, when an algorithm computes something, we would expect that, at certain points, the results (and temporary values) for this computation hold specific values. Our goal is to leverage these values to identify (possibly different) code that "computes the same thing." The main contributions of this paper are the following: (I) we introduce a novel approach to capture and model behavior from dynamically analyzed malware that is based on the sequence of values that a program writes to memory or registers; (II) we describe a two-step procedure to decide wheter two execution traces are similar and leveraged it to implement malware clustering and code reuse identification.

2 Data Value Traces

In this section, we discuss how we build *data value traces* to capture the activity of a (malware) program. To obtain these traces, we developed a prototype system that runs malware samples in an emulated environment. The prototype was developed using PyDBG [13]. This provides us with tight control of the debugging process. Also, PyDBG provides features to hide its debugging activity, which is useful to foil most malware attempts to detect the analysis environment.

We use our prototype to record an ordered sequence of instructions that modify at least one register or memory value; that is, we are only interested in instructions that *write* to memory. For each of these instructions, we store the numeric value(s) of all memory locations and registers to which this instruction writes (typically, this is one). For instance, if a malware sample executes the instruction `sub esp,0x58`, we will log in our trace the line `sub esp,0x58; 0x12ff58`, which corresponds to the instruction and the new value written to register `%esp`, which is 0x12ff58 in this example (assuming that the initial value of `%esp` was 0x12ffb0). When the malware process terminates or a timeout is reached, we also take a snapshot of the content of the malware's executable (code) segments. This information is needed later to identify code reuse between malware samples, but it is *not* required to identify similarity between samples.

To collect the sequence of executed instructions, our system runs the sample in single-step debugging mode. More precisely, we single step through the code within the malware executable (and all code dynamically generated by the malware). However, calls that are made to standard operating system libraries are not logged, nor are the instructions executed inside these libraries. Fortunately, the system dynamically-loaded libraries (DLLs) are loaded into the memory region ranging from `0x70000000` to `0x78000000` (in Microsoft Windows XP). This speeds up the monitoring process and makes the resulting traces smaller. Also, it focuses the data collection on the actual malicious code.

To increase the efficiency of the collection process and to minimize the size of the traces, we only log a selected subset of instructions related to logic and arithmetic operations, namely: `add`, `adc`, `sub`, `sbb`, `mul`, `imul`, `div`, `idiv`, `neg`, `xadd`, `aaa`, `cmpxchg`, `aad`, `aam`, `aas`, `daa`, `das`, `not`, `xor`, `and`, `or`. We focus on these instructions because we are mostly interested in characterizing computations that the malware performs. Such computations will almost always involve arithmetic and logic instructions. Other instructions, such as data move or stack manipulation routines, are mostly used to prepare the environment for a computation, and hence, are less characteristic than the values that emerge directly as the result of a computation. We decided to remove the arithmetic instructions `inc` and `dec`, as they are typically involved in simple counters, which reveal little information about the data that is being computed. We also decided to remove instructions from the trace when they write the value 0, as this constant is not very characteristic of a particular computation.

We apply one last transformation to convert a sequence of instructions into the final data value trace. This transformation works by moving a sliding window of length two over the instruction sequence. For the two instructions in the window, we extract the two data values that these instructions write, one value for each instruction, and aggregate then into a pair of values – a bigram. After the bigram is appended to the data value trace, we advance the sliding window by one instruction. The reason for transforming the sequence of instructions (or written) values into bigrams is the following: If we would compare simple traces of individual values, it is more likely that two values in two traces match by accident. By combining subsequent values into pairs, we add a simple form of *context* to individual data values. We found that this extra context significantly lowers the fraction of coincidental matches and improves the separation between different program executions.

3 Comparing Traces

As discussed in the previous section, we capture the activity of a malware program by collecting a trace that consists of a sequence of bigrams of data values that this program has written. For a number of applications (such as malware classification and clustering), we require a technique to determine whether the activities of two malware samples are similar. To perform this comparison, we have developed a two-step algorithm. This algorithm operates on two data value

traces as input and outputs a similarity measure S that ranges from 0 (completely different) to 1 (identical).

3.1 Step 1: Quick Comparison

The goal of the first step is to decide whether two traces are similar enough to warrant a further, more detailed comparison. This step works by creating a small "identifier" for each trace. This identifier is based on the k least-frequent bigrams that appear in a trace. The underlying assumption behind this choice is that if two samples are variants of each other, they should share some specific features or attributes that are particular to their family. Thus, we can discard the most common bigrams, which can appear within many different families, and focus on the specifics of a certain family's fingerprint. We have experimentally determined that a value of $k = 100$ yields good results.

More formally, we state our approach as follows. Let ID_{M_1} and ID_{M_2} be the k least-frequent bigrams from traces produced by malware samples M_1 and M_2. We compare these two malware identifiers by applying the Jaccard index($J(ID_{M_1}, ID_{M_2}) = \frac{ID_{M_1} \cap ID_{M_2}}{ID_{M_1} \cup ID_{M_2}}, 0 \leq J \leq 1$). If, and only if, the Jaccard index (ranging from 0 to 1) is greater than the empirically established threshold of 0.3[4], we move to the second step. Otherwise, the result of this computation is used as the similarity value (which indicates low similarity).

3.2 Step 2: Full Similarity Computation

In the next step, we compute the overlap of the entire two traces. More specifically, we compute the longest common subsequence (LCS) between them. Suppose that T_1 and T_2 are different data value traces and that L_1 and L_2 are their lengths, respectively. The similarity between the two traces is then calculated as $C(M_1, M_2) = \frac{LCS(T_1,T_2)}{min(L_1,L_2)}$. We chose the longest common subsequence over the longest common substring to tolerate small differences in the computations. Moreover, we note that using a standard LCS algorithm can be computationally expensive. We addressed this by calculating the LCS based on the GNU `diff` tool (`http://en.wikipedia.org/wiki/Diff`) output. Our experiments, evaluating a standard LCS implemented in C++ and our approximate LCS computation showed that we could accomplish faster results using our approach — in some cases $\approx$500$\times$ faster — with no significant loss of accuracy.

The original `diff` tool has the nice property that it inserts "barriers" while computing the longest common subsequences present in a textual input. Our `diff`-based LCS approach, referred from now on as *eDiff*, enhances this capability by (i) marking the regions that differ between two traces and (ii) by mapping the shared subsequences to the original instructions in the respective execution traces. As a result, we know exactly what malware code produced similar memory

[4] To choose this threshold (T), we performed tests with an increment of 0.1 for the range $0.0 < T \leq 0.5$

writes. This will be useful for identifying code reuse, as explained in Section 5.2. To map value traces back to instructions, we simply link the bigram values in the value traces to the raw instructions that produced those values.

4 Applications

In this section, we discuss two applications that we built on top of our malware trace similarity technique. The first application is clustering; the idea is to group samples that show similar activity based on their value traces. The second application uses the data value traces to find cases of code reuse. That is, we want to find cases in which malware samples that belong to different families share one or more snippets of identical code.

4.1 Clustering

The input to the malware clustering application are a set of N data value traces, one trace for each of the N samples to be clustered. The goal is to find groups of malware samples that are similar. Clustering is implemented in two steps: pre-clustering and inter-cluster merging.

Pre-clustering: The goal of the pre-clustering step is to quickly generate an initial clustering and avoid having to perform $N^2/2$ comparisons. To accomplish this, we sequentially process each of the N samples, one after another (in random order), as follows: Each new sample is compared to all cluster leaders (explained below), using the similarity computation described in the previous section. When the trace for the new sample exhibits more than 70% similarity with one or more cluster leaders, this sample is merged with the existing cluster for which the similarity is highest. Otherwise, the sample (and its trace) is put into a new cluster, and this sample also becomes the cluster leader. When merging a trace with an existing cluster, we need to elect a new cluster leader (a cluster leader is basically the trace that is selected to represent the entire cluster). For this, we must make a selection between the existing cluster leader and the new trace. We select the *longer* trace as the new leader. We do this to increase the probability that a sample, whose behavior is similar to the activity of malware in a cluster, is properly matches with that cluster. In other words, by selecting the longest trace as the cluster leader, a new trace has more chances to find a long, common subsequence. By removing from the comparison computation all except one trace for each cluster, we greatly reduce the required number of comparisons.

Inter-cluster merging: The pre-clustering step results in a set of initial clusters whose traces share at least 70% similarity. However, due to the nature of the quick comparison (first step of the similarly comparison), there can be clusters that should be merged but are not. That is, it is possible that two traces are actually quite similar, yet their least-frequent bigrams are too different to pass the threshold. In this case, there are different clusters containing malware from the same family, and it is desirable to merge these clusters. The merging step is applied to the output of the pre-clustering step so as to generate a reduced

amount of clusters. To this end, we perform a pairwise comparison between all cluster leaders, using our *eDiff* algorithm. If their similarity is greater than the same 70% threshold defined previously, the clusters are merged.

4.2 Code Reuse Identification

When comparing two traces, our algorithm not only computes a general similarity (overlap) score but also determines which parts of the traces are identical. When we find a stretch of values that are identical between two traces generated by executing different samples, we might naturally ask the question whether these values were produced by similar code. This would allow us to identify code that is shared between samples that are otherwise different.

To identify code reuse, when *eDiff* compares two data value traces, it stores for each element (bigram) in the traces whether this element is unique to the trace or shared between both traces. For instance, let us assume that we have two traces. One contains the three bigrams: `(0x1,0x2)`, `(0x2,0x4)`, and `(0x4,0x5)`; the other contains the four bigrams: `(0x1,0x2)`, `(0x2,0x7)`, `(0x7,0x4)`, and `(0x4,0x5)`. In this case, *eDiff* would find that the first and last element in each trace are shared, while the middle one(s) are unique (to each trace). To find code reuse, we check both traces for the presence of at least four consecutive elements that are shared. The threshold of four was empirically determined and allows us to find shared code roughly at the function level. A higher threshold would be possible when we want to find longer parts of shared code. A lower threshold often yields accidental matches that do not reflect true code reuse.

Next, we require a mechanism to "map back" values in a data value trace to the instructions that produced them. This can be done easily because we retain the original instruction sequences that were recorded during dynamic analysis. To find the code in the malware program that contains the "shared instructions" we generate a regular expression pattern, which is then matched against the dumped code segment. When a match is found, we consider the resulting code block as a candidate for reuse. All matches that are found for each trace are compared, and when we find a sequence of identical code of a minimum length, we identify the code snippet as reused between malware samples.

5 Preliminary Experiments

We performed an initial set of experiments using 16,248 execution traces that produced promising results. These traces were obtained from the analysis of Windows PE32 executable programs and they represent a diverse and recent set of different malware families that are currently active in the wild.[5]

5.1 Malware Clustering

The evaluation of the quality of a clustering algorithm is a complicated task [5], as clustering results are often not objectively right or wrong but depend on a

[5] For a complete list of MD5 sums of the samples, please contact the authors.

number of factors, such as the metrics used to calculate the distances among samples and clusters, the final amount of clusters generated, the chosen heuristics, etc. We used three different ways to obtain a reference clustering, based on [9]: one from the static analysis of malware [4], one from the dynamic analysis of malware [2], and the last one based on anti-virus (AV) labels from AVG (`http://www.avg.com`), Avira (`http://www.avira.com`) and F-Prot (`http://www.f-prot.com`). These AV labels were relabeled so that only the general identifier for each family remains (e.g., Trojan.Zbot-4955 became "zbot").

After we generated the reference clustering sets, we borrowed the precision and recall metrics from [2] to measure the quality of our clustering results. The product of the values obtained from the overall precision and recall can be used to measure the overall clustering **quality** ($Q = P \times R$). For the reference clustering based on AV labels, we measured the quality of our clustering scheme by defining the level of agreement related to the labels assigned to each sample in a cluster. Perdisci et al. [11] proposed to use two indexes (cohesion and separation) to validate their HTTP-based malware behavioral clustering. However, their approach "attenuates the effect of AV label inconsistency" due to the way the Cohesion Index is defined (there is a "gap" and a "distance" value that causes a boost in cohesion). To avoid this boost, we define a simpler level of agreement A for a cluster j, calculated as:

$$A_j = \frac{\sum_{N \in AV} max_{lbl \in Labels_N}(frequency(lbl))}{[|T_j| * |AV|]}$$

where AV is the number of AV vendors, $Labels_N$ is the set of the assigned labels and their related frequencies for each AV engine for each cluster ($N = avg, avira, fprot$), T_j is the total amount of samples in the cluster.

Reduced dataset. Before we applied our clustering technique to the entire dataset, we ran preliminary tests using a smaller subset, which consisted of 1,000 random samples. Those initial tests were important to experiment with and determine different threshold parameters. In particular, we varied the similarity threshold for the second step of the algorithm from 0 to 100% (incrementing by 10% after each iteration) and observed the highest quality, i.e., the average between the obtained static and behavioral quality values, for the similarity threshold of 70%. Moreover, the AV labels' level-of-agreement value for this threshold is also very high (0.894).

Full dataset. Based on the results of the preliminary tests, we defined a similarity threshold of 70% for the *eDiff* process. We continued to use the initially-established Jaccard index threshold of 0.3 for the quick comparison. The amount of clusters produced by the two reference clustering sets for the 16,248 samples with traces were 7,900 clusters for the static approach and 3,410 for the behavioral one. Our approach produced 7,793 clusters that were compared to the reference clustering sets, generating the precision values of 0.758 and 0.846 and the recall values of 0.81 and 0.572 for the static and behavioral reference, respectively. Calculating the AV labels' level-of-agreement for our clustering yielded 0.871. These values yield average results of 0.843, 0.652, and 0.656 for precision, recall, and quality, respectively.

5.2 Code Reuse

To look for code reuse in samples that likely belong to different malware families, we only check pairs of malware clusters that are sufficiently different. More precisely, based on the clustering results obtained in the previous step, we look for pairs of clusters that have a similarity score between 10% and 30%. We identified 974 pairs of clusters that fulfill this requirement. We discovered 15 pairs (involving ten different clusters) that share code between them. More precisely, we found seven different blocks of code that seem to be reused among samples. We sent the ten representatives (one for each cluster) to VirusTotal (`http://www.virustotal.com`). Looking at the results, we noticed that the most common label assigned to them refers to different Trojan malware that all seem to attack online games (and, apparently, shared code to do so).

6 Related Work

Dynamic malware analyzers, such as [8] and [15], operate at the system call level and currently do not log the low-level values of an execution (memory and registers). Ether [3] performs both instruction and system call tracing to analyze malware in a transparent way by using hardware virtualization extensions, but it has several of prerequisites on the type of operating system, architecture and platform, which can limit its use. Indeed, we can divide malware classification techniques according to how the traces were obtained — i.e., through either static or dynamic analysis — and to the type of behavior gathered — i.e., either lower-level or assembly-related data or higher-level or system call information.

Static Analysis Approaches. Shankarapani et al. [14] propose two detection methods to recognize known malware variants without the need of new AV signatures: SAVE, which generates signatures based on a malware sample API calls sequence through the static analysis of its executable, and MEDiC, in which the signature of a malware sample is part of its disassembled code. SAVE performs an optimal alignment algorithm before applying similarity measure functions (cosine, extended Jaccard, and Pearson correlation). This kind of algorithm does not scale well if there are too many sequences or if the sequences are very large. For files whose size is among ≈500 Bytes to ≈1000 Bytes, the detection time can be in a range of few seconds (considering just one executable). Kinable and Kostakis [7] performed a study of malware classification based on call graph clustering, where they measured the similarity of call graphs that were extracted from malicious binaries through matches that try to minimize the edit distance between a pair of graphs. The authors conclude that it is difficult to partition malware samples in well-defined clusters using k-means based algorithms, chosing the DBSCAN algorithm to cluster some sets, the larger having 1,050 samples. They state that this larger set had 72% correct clusters. Zhang and Reeves [16] propose a method to detect malware variants that uses automated static analysis to extract the executable file semantics. These semantic templates are characterized based on the system calls executed by a malware

sample and used in a weighted pattern matching algorithm that computes the degree of similarity between two code fragments.

Dynamic Analysis Approaches. Park et al. [10] present a classification method that uses a directed behavioral graph extracted from system calls through dynamic analysis. The generated directed graphs from two malware samples are compared computing the maximal common subgraph between them and the size of this subgraph is used as the similarity metric. Bailey et al. [1] developed a method based on the behavior extracted from a malware sample after executing it in a virtualized environment. This behavior is considered the malware's fingerprint and represents the set of actions that changed the system state, such as files written, processes created, registry keys modified, and network connection attempts. The tests were performed on a dataset of 3,700 samples from which the fingerprints were extracted. The normalized compression distance (NCD) was used as a similarity metric, and the pairwise single-linkage hierarchical clustering algorithm was used for classification purposes. Rieck et al. [12] propose the use of machine learning techniques on malware behaviors composed of changes that occurred in a target system in term of API function calls. They ran their experiments on more than 10,000 malware samples divided into 14 families labeled by AV software. The behavioral profiles obtained from dynamic analysis serve as a basis to feature extraction and use the vector space model and the bag of words techniques. After that, the Support Vector Machines method is applied to the feature sets for classification purposes. Bayer et al. [2] present a scalable clustering approach to classify malware samples based on the behavior they present while attacking a system. Dynamic analysis is used to generate the behavioral profiles — sequences of enriched and generalized information abstracted from system call data. The similarity metric used in their work is the Jaccard index, which is then used as an input to the LSH clustering method. Very recently, Jang et al. [6] introduced BitShred, an algorithm for fast malware clustering. In this paper, the authors present a new way to efficiently simplify and cluster features from inputs such as (static) code bytes and (dynamic) system call traces.

7 Conclusions

In this paper, we empirically demonstrated that the values stored in memory and registers after write operations can be used to detect and cluster malware in families. We also presented a different approach to perform the similarity score calculation that is simple and effective when applied to the malware problem. We compared the results from more than 16 thousand malware samples executed and processed in our prototype system to three reference clustering sets — static, behavioral (dynamic), and AV labeling — and our produced clustering reached an average precision value of 0.843 for the first two sets and a level of agreement value of 0.871 for the last one. Finally, we showed that our classification process can also be used to verify for code reuse, which helps to investigate the sharing of functions in different families of malware.

References

1. Bailey, M., Oberheide, J., Andersen, J., Mao, Z.M., Jahanian, F., Nazario, J.: Automated Classification and Analysis of Internet Malware. In: Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection (RAID '07). pp. 178–197. Gold Coast, Australia (September 2007)
2. Bayer, U., Milani Comparetti, P., Hlauscheck, C., Kruegel, C., Kirda, E.: Scalable, Behavior-Based Malware Clustering. In: 16th Symposium on Network and Distributed System Security (NDSS) (2009)
3. Dinaburg, A., Royal, P., Sharif, M., Lee, W.: Ether: malware analysis via hardware virtualization extensions. In: Proceedings of the 15th ACM conference on Computer and Communications Security. pp. 51–62. CCS '08 (2008)
4. Jacob, G., Neugschwandtner, M., Comparetti, P.M., Kruegel, C., Vigna, G.: A static, packer-agnostic filter to detect similar malware samples. Tech. Rep. 2010-26, UCSB (November 2010)
5. Jain, A.K., Murty, M.N., Flynn, P.J.: Data clustering: a review. ACM Comput. Surv. 31, 264–323 (1999)
6. Jang, J., Brumley, D., Venkataraman, S.: BitShred: Feature Hashing Malware for Scalable Triage and Semantic Analysis. In: ACM Conference on Computer and Communications Security (CCS) (2011)
7. Kinable, J., Kostakis, O.: Malware classification based on call graph clustering. J. Comput. Virol. 7(4), 233–245 (Nov 2011)
8. Kruegel, C., Kirda, E., Bayer, U.: Ttanalyze: A tool for analyzing malware. In: Proceedings of the 15th European Institute for Computer Antivirus Research (EICAR 2006) Annual Conference (4 2006)
9. Neugschwandtner, M., Comparetti, P.M., Jacob, G., Kruegel, C.: Forecast: skimming off the malware cream. In: Proc. of the 27th Annual Computer Security Applications Conference. pp. 11–20. ACSAC '11, ACM (2011)
10. Park, Y., Reeves, D., Mulukutla, V., Sundaravel, B.: Fast malware classification by automated behavioral graph matching. In: Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research. pp. 45:1–45:4. CSIIRW '10, ACM, New York, NY, USA (2010)
11. Perdisci, R., Lee, W., Feamster, N.: Behavioral clustering of http-based malware and signature generation using malicious network traces. In: Proceedings of the 7th USENIX conference on Networked Systems Design and Implementation. pp. 26–26. NSDI'10 (2010)
12. Rieck, K., Holz, T., Willems, C., Düssel, P., Laskov, P.: Learning and classification of malware behavior. In: Proceedings of the 5th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment. pp. 108–125. DIMVA '08, Springer-Verlag, Berlin, Heidelberg (2008)
13. Seitz, J.: Gray Hat Python: Python Programming for Hackers and Reverse Engineers. No Starch Press, San Francisco, CA, USA (2009)
14. Shankarapani, M., Ramamoorthy, S., Movva, R., Mukkamala, S.: Malware detection using assembly and api call sequences. J. Comput. Virol. 7, 107–119 (2011)
15. Willems, C., Holz, T., Freiling, F.: Toward automated dynamic malware analysis using cwsandbox. IEEE Security and Privacy Magazine 5(2), 32–39 (2007)
16. Zhang, Q., Reeves, D.: Metaaware: Identifying metamorphic malware. In: Proc. of the 23rd Annual Computer Security Applications Conference. pp. 411 –420. ACSAC '07 (December 2007)