Extended Abstract : Detection and classification of malware based on symbolic execution and machine learning methods

Charles-Henry Bertrand Van Ouytsel, Axel Legay

July 30, 2021

1 Introduction

These last years, number of malware detected by Antivirus always continue to grow. According to AV-Test [AT21], more than 100 millions of new malware have been observed in 2021 until now. Most of these new malware are just variants of previously observed malware. In fact, malware authors never end to develop new strategies to tweak their malware so that previous detection methods are bypassed. The detection and classification of these new malware samples is a big challenge.

In most cases, malware detection and classification is signature-based. A signature (such as YARA [Vir19]) contains patterns/properties observed in malware of the same family and is generally build by experts and gathered in a database. If a binary matches a signature of the database, it's classified as a malware. Unfortunately, malware creators can get around those detection methods by applying different techniques (polymorphism [WGXW10], metamorphism, packing or other obfuscations) while keeping the malicious behaviour of their program [MKK07].

Another popular approach is dynamic analysis [ESKK08], where malware are executed and observed in a closed environment (recording API calls, network activity, created process,...) to observe how they behave. This type of analysis can sidestep the limitations of signature-based static analyses. However, it focuses on one execution of the program at a time with a given context environment and with no guarantee that the malicious behaviour will be triggered [BY17] (delay, evasion techniques, sandbox detection...).

To overcome these limitations, a promising approach is the use of symbolic execution [BCD+18] which allows to explore the whole behaviour of the malware by considering all paths of execution. With symbolic execution, we can extract a system call dependency graph (SCDG) that represents the main behaviour of a binary. SCDGs have already been used in malware detection and classification [CJK07], as a representation of API calls to the operating system and relations between these calls. These calls are known to be significantly representative of the malware behaviour and its interactions with the system. This can be used to detect malicious behaviour.

Using graph mining techniques like gspan [YH02], it's possible to generalize from several SCDGs of the same malware family a common SCDG that can be used as signature for that family. This kind of approach has already showed accurate detection and classification results [T+16]. Such a toolchain using ANGR [SWS+16] has been proposed in [SBB+20] where the authors manage to get a good classification performance. While using a bigger data set and putting a focus on resources usage, we propose several improvements to this toolchain :

- An open source implementation
- New exploration heuristics to quickly discover different parts of the binary
- New priorities/resources management for states during exploration
- New types of dependency in the SCDG

2 Presentation of our approach

2.1 Overview

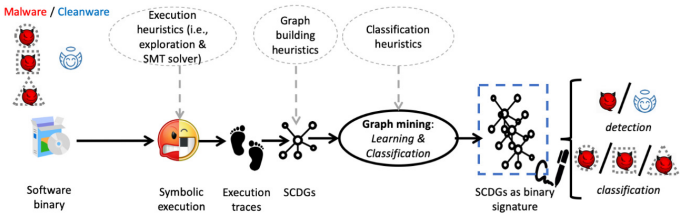

Figure 1: The whole toolchain.

Our toolchain is represented in the figure 1 and works as follow :

1. A collection of labelled binaries of different malwares families is collected and used as the input of the toolchain.
2. ANGR [SWS+16], a framework for symbolic execution, is used to execute symbolically binaries and extract all execution traces. For this purpose, different heuristics have been developed to optimise symbolic execution. Several execution traces (i.e : API calls used and their arguments) corresponding to one binary are extracted with ANGR and gathered together thanks to several graph heuristics to construct a SCDG.
3. These resulting SCDGs are then used as input to graph mining algorithms to extract a common graph for the SCDGs of each families and create the associated signatures.

4. Finally, when a new sample has to be classified, its SCDG is build and compared with SCDGs of known families (thanks to a simple similarity metric).

2.2 Symbolic Execution

We first use symbolic execution on malware binaries, allowing us to explore all possible execution paths. Starting from the entry point address, at each execution step, ANGR records, processes and updates a set of active states representing snapshots of the execution flow of the binary at a given time. Each state stores information like instructions addresses, register values or memory usage. If a variable can take several values at an execution step, a symbolic value is introduced to keep track of possible values and their related constraints. During execution, when a new instruction is encountered, ANGR engine evaluates the next state it will produce. A state can produce a single child state (e.g mov instruction) or two child states if a conditional jump is met. In the case of a conditional jump, the state is forked in two : first child state will consider what happens if the jump is taken , second child will consider what happens if it's not. During exploration, infeasible states can appear(e.g states with a symbolic value which can not have a feasible concrete value). For an efficient exploration, it's important to discard them. For this purpose ANGR use z3, a SMT solver to check satisfiability. This solver has been tuned for malware analysis in previous work [SBB+20].

Symbolic execution faces an *accuracy/performance tradeoff* : it consumes a lot of resources (time, memory,...) but more resources lead to better performances. Given the huge amount of possible execution states (also known as the problem of state explosion), it's really important to optimize resources usage and help ANGR to explore state space as efficiently as possible.

Initially, ANGR explores binaries with a Breadth-First Search (BFS) approach, i.e it considers at each execution step all paths that are not finished yet and explore them simultaneously. This number of paths grows quickly at each step, leading to an important resource consumption (time and memory). For a better resources usage, only a subset of paths are considered at each step for execution, others are put in a stash for future exploration when more space is available. Four methods of exploration were compared in our work, allowing us to choose in which order we explore states :

- *Breadth-First Search (BFS)* : When a new path has to be considered, the shortest path not yet explored is chosen. It means that even if previous paths were "deep" in the exploration, a path early put aside can still be considered relatively early anyway.

- *Depth-First Search (DFS)* : When a new path has to be considered, the longest path not yet explored is chosen. Contrary to BFS, exploration will quickly go deep in the binary but if an interesting path begins early and we don't take it directly, it will take a lot of time to go back to it.

- *Custom Breadth-First Search (CBFS)* : At each step, a check is performed to verify if there exists a path which leads to an instruction not yet executed. If such a path exists, it is chosen, otherwise the shortest path is considered.

- *Custom Depth-First Search (CDFS)* : At each step, a check is performed to verify if there exists a path which leads to an instruction not yet executed. If such a path exists, it is chosen, otherwise the longest path is considered.

When executing symbolically a binary, different states representing different paths have to be kept in memory. To spare memory consumption, we choose to not explore all available states at the same time as it was done in previous work. Instead, states are put in different stashes corresponding to different priorities :

- *Active* (default in ANGR) : Stash where states currently used for exploration are stored. To decrease memory consumption, an option has been added to limit number of states in this stash : *simul_state* which decide how many states can be present simultaneously in this stash. If this limit is exceeded, some states are moved to *Pause* stash.

- *Pause* : The stash where states are moved to wait until some space becomes available in *Active* stash. The size of the space in this stash is a parameter of the toolchain. If new states appear and there is no space available in the *Pause* stash, some states are dropped.

- *ToExplore* : The stash where states leading to new instruction addresses (not yet explored) of the binary are kept. If CDFS or CBFS are not used, this stash merges with the pause stash.

- *ExcessStep* : The stash where states exceeding the threshold related to number of steps are moved. If new states are needed and there is no state available in pause stash, states in this stash are used to resume exploration (their step counter are put back to zero).

- *ExcessLoop* : The stash where states which exceed the threshold related to loops are moved. If new states are needed and there is no state available in pause or ExcessStep stash, states in this stash are used to resume exploration (their loop counter are put back to zero).

- *Deadended* (default in ANGR) : The stash where states are moved when their execution is over (e.g : exit function call).

  Finally, another important problem of symbolic execution is a proper modelisation of the OS and external libraries called during execution. An improper modelisation of these libraries can lead to incorrect state creation or failures to explore important states.

2.3 Building SCDGs and signatures

After the exploration of a binary with symbolic execution, several execution traces are obtained. Each

trace consists of a list of API calls, their arguments, their resolved addresses and their calling addresses. All this information is used to build a SCDG. In SCDGs, vertices correspond to system calls, edges to information flows between them and the directions of the edges represent the orders in the traces. To be more precise, two types of edges are considered in this work:

- *Argument - Argument* relationship : An edge is built if an argument of a system call is the same as the argument of another system call. Each argument is given a number corresponding to its location in the argument list of the system call, the return value of the system called is given the index 0. Finally, each edge takes a label related to argument position of the first and the second call. Note that we also allow the creation of an edge if two arguments are strings included in each other (typically, it occurs with path names or register values). These particular cases were not considered in previous works.
- *Argument - Address* relationship : An edge is build if an argument of a system call (often the return value) corresponds to the resolved address of another system call. This usually occurs when a binary load a function or a module dynamically. This was also not considered in previous works.

From different SCDGs corresponding to a same malware family, we are able to construct a signature by observing similarities between SCDGs. This signature has to represent the main behaviour of the malware family. For this purpose, Gspan algorithm is used to extract the biggest SCDG common subgraphs to a majority of those SCDGs. We take advantage of previous results related to Gspan and choose to extract biggest subgraphs supported by at least 80% of the malware in the samples.

2.4 Classifying new samples

When a new binary has to be classified, its SCDG is first build as described previously. Then, Gspan is applied to extract the biggest common subgraph G'_i between this SCDG and each of the signature of known malware families G_i (where i denotes the index of the family signature). A similarity score is evaluated to decide if the graph is considered as part of this family or not.
The similarity score S between the graphs G'_i and G_i is computed as follow (this similarity score was first used in [SBB+18])

$$S(G'_i, G_i) = \frac{num\_edges(G'_i)}{num\_edges(G_i)}$$

Since G'_i is a subgraph of G_i, this is calculating how much G'_i appears in G_i. We classify the new sample by finding the family i with the highest similarity score :

$$\arg\max_i S(G'_i, G_i)$$

3 Experiments

For our experiments, we used a dataset of 20 families of malware (Wabot, Sodinokibi, Simbot,...) with samples of 30 malware for each family. These samples were gathered by Cisco during 2019 and labelled thanks to AVClass [SRKC16]. A 3-fold cross validation is used to evaluate classification performance and avoid overfitting.

We first launch extraction of SCDGs with a timeout of one hour for each of the binary, recording extracted calls at 20 minutes, 40 minutes and 1 hour. Using the same parameters as [SBB+20] and our different exploration methods, we compare the extracted API calls, the number of edges/nodes of our SCDG as well as classification accuracy.

Then we launch extraction with a timeout of 10 minutes for each sample. We also limit the number of states handled simultaneously to reflect limited resource usage. We compare again API calls extracted, number of edges/nodes of our SCDG as well as classification accuracy.

Finally, we re-use traces previously extracted and use different graph building methods for our SCDGs to observe which one allows the better classification accuracy with Gspan algorithm.

We observe that our improvements generally increase the number of calls extracted as well the size of our SCDGs while allowing a better control of resource usage.

4 Conclusion

We propose an implementation of a toolchain using symbolic execution and machine learning to detect and classify malware. On one hand, symbolic analysis avoids the inherent problems of dynamic analysis: thanks to symbolic analysis, all execution paths can possibly be explored. On the other hand, machine learning avoids the problem of rigidity (e.g. pattern matching like YARA) and obfuscation of the signatures encountered with static analysis. Our new exploration heuristics and priority/resources management techniques allow to improve malware binary exploration while reducing required resources. We also put the focus on proper environment simulations to improve the consistency of the exploration. Our results show that it is possible to explore more execution paths of the binaries and achieve a greater classification performance with our improvements of the toolchain.

References

[AT21] AV-Test. Malware statistics & trends report, 2021.

[BCD+18] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 51(3):1–39, 2018.

[BY17] Alexei Bulazel and Bülent Yener. A survey on automated dynamic malware analysis evasion and counter-evasion: Pc, mobile, and web. In *Proceedings of the 1st Reversing and Offensive-oriented Trends Symposium*, pages 1–21, 2017.

[CJK07] Mihai Christodorescu, Somesh Jha, and Christopher Kruegel. Mining specifications of malicious behavior. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 5–14, 2007.

[ESKK08] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. A survey on automated dynamic malware-analysis techniques and tools. *ACM computing surveys (CSUR)*, 44(2):1–42, 2008.

[MKK07] Andreas Moser, Christopher Kruegel, and Engin Kirda. Limits of static analysis for malware detection. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, pages 421–430. IEEE, 2007.

[SBB+18] Najah Ben Said, Fabrizio Biondi, Vesselin Bontchev, Olivier Decourbe, Thomas Given-Wilson, Axel Legay, and Jean Quilbeuf. Detection of mirai by syntactic and behavioral analysis. In *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*, pages 224–235. IEEE, 2018.

[SBB+20] Stefano Sebastio, Eduard Baranov, Fabrizio Biondi, Olivier Decourbe, Thomas Given-Wilson, Axel Legay, Cassius Puodzius, and Jean Quilbeuf. Optimizing symbolic execution for malware behavior classification. *Computers & Security*, page 101775, 2020.

[SRKC16] Marcos Sebastián, Richard Rivera, Platon Kotzias, and Juan Caballero. Avclass: A tool for massive malware labeling. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 230–253. Springer, 2016.

[SWS+16] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 138–157. IEEE, 2016.

[T+16] Tayssir Touili et al. Automatic extraction of malicious behaviors. In *2016 11th International Conference on Malicious and Unwanted Software (MALWARE)*, pages 1–10. IEEE, 2016.

[Vir19] VirusTotal. Yara: The pattern matching swiss knife for malware re- searchers (and everyone else), 2019.

[WGXW10] Zhenyu Wu, Steven Gianvecchio, Mengjun Xie, and Haining Wang. Mimimorphism: A new approach to binary code obfuscation. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 536–546, 2010.

[YH02] Xifeng Yan and Jiawei Han. gspan: Graph-based substructure pattern mining. In *2002 IEEE International Conference on Data Mining, 2002. Proceedings.*, pages 721–724. IEEE, 2002.