**e**

**An Analysis and Extension of the Paper:   
Malware Detection using Machine Larning Based Analysis of Virtual Memory Access Patterns**

**(Zhixing Xu, Sayak Ray, Pramod Subramanyan, & Sharad Malik)**

**—**

George Marché

Old Dominion University

CS 795: Malware Analysis and Reverse Engineering

Abstract

The authors present a hardware-assisted malware detection framework that applies machine learning concepts to detect anomalous memory access patterns and classify applications as either malicious or benign. To improve on other proposed frameworks, each application would have its own classification model, in which each function call (or system call) is represented as an “epoch”, where the frequency of different memory accesses, calls, and instructions are recorded into a histogram and used for training the model and detecting malware. In this paper, I give a general overview of this model and discuss challenges, machine learning applications, and results. Finally, I introduce an extension to the report, where I present an alternative idea using machine learning for supplementary detection improvement.

1. Introduction

Diagnosing and analyzing malware is challenging and requires an understanding of machine code and computer architecture with careful observation. Static analysis can be used as a preliminary (in rare cases, sufficient) investigation of a program for detecting malware, but malware that uses obfuscation to hide its code makes it difficult to stop there, whether signature-based or anomaly-based (Idika & Mathur, 2007). However, executing each encountered piece of malware in a controlled environment and analyzing it using concepts that include reverse engineering is tedious and labor-intensive (Selçuk et al., 2018). Recent development in machine learning has been used to remove this burden from analysts and make detection more efficient and faster. This framework takes into account several factors that must be taken into consideration when building an effective model while minimizing erroneous results, while addressing several shortcomings in other malware detection projects.

1. Factors of Consideration in Development

The developers of this malware detection faced multiple points of decision-making in making sure that the framework was able to classify malware accurately, with low execution time and consumption of resources. One of the largest factors in selecting the most viable options was assuring that the framework was able to detect both kernel- and user-level attacks.

1. ***Epoch Markers***

The size of epoch markers is arguably paramount in the effectiveness of the framework. Kernel-level functionality is more defined; unlike user-level applications, which can run for a prolonged period of time, kernel interaction is limited and more structured. Kernel rootkits typically modify the system call table (SSDT hooking) which redirects system calls to malicious code. Because of this, epoch markers for kernel interaction are best developed to collect data on each system call. At the user level, epoch markers were considered for the entire program’s execution lifespan, but this would not fare well with larger programs, as small yet significant malware modifications could remain undetected by the model and continuously executing programs would saturate its effect. To make sure that it was universally effective, the developers decided on function calls as epoch markers for user-level applications.

1. *Deciding Features Collected*

Applications run quickly, with the CPU processing code at a rapid pace. Consequently, sequence and timing were difficult to precisely observe, and were not included in the features of the summary histogram. The authors decided to collect data on location and frequency, which resulted in insignificant reduction of precision.

1. *Memory Block Size*

Smaller blocks of memory allowed for more detailed information, but the consumption of storage space was too high. Ultimately, this would also affect the performance, as it would eventually take more time for the framework to find available memory space for new blocks to be added. Larger bins alleviated this by avoiding details that were not necessary in determining the control flow variation. This can be compared to the importance of avoiding single-stepping into every function that is called by malware machine code, as inspection of library functions in detail is unnecessary and can end up continuing to step further into minor functions that give an analyst little to no contributing information about the general mechanics of the application. The developers experimented with block sizes of 1 KB, 4 KB, and 16 KB.

1. Comparison with Other Frameworks
2. *Number of Models in the Classifier*

Many detection techniques use a single model, which covers the broad execution of all programs. One of the techniques included the hardware-assisted detection using performance counters proposed by developers at Columbia University, which had a true positive rate that varied between 53.3% using a FANN classifier and 82.3% using a decision tree classifier for thread detection and 35.7% using a random forest classifier and 83.1% using a decision tree classifier for package detection, both measured when the false positive rate was below 10% (Demme et al., 2013). Conceptually, this would have had better results if there were separate models for each application, like this proposed framework. A single model is effective at detecting malicious programs; however, most malware inserts itself into everyday programs that are otherwise benign. This distinguishes infected executions from “clean” executions.

1. *Software Versus Hardware*

Software that is used for malware detection can be counteracted by malware, as it is vulnerable to exploits that can either cause it to not work at all or give erroneous results. Also, software usage of computing resources produces an overhead and requires a computer to rely on the software’s integrity. Hardware, though not yet determined to detect malware universally, can be relied upon to detect specific classes, including memory corruption attacks. Furthermore, unlike other hardware-assisted malware detection frameworks that monitor physical memory addresses, this framework inspects virtual addresses, since physical addresses can change over time. Disassembly programs like IDA Pro attempt to recreate the machine code for a unique application using the same memory addresses each time in order to not throw off analysis.

1. Insights and Observations

The framework has an excellent balance of accuracy in diagnosis and program segmentation and efficiency. Using a model for each application rather than a single universal model was strategic, and deciding to only select function call models that exceeded the 60% accuracy and determining the classification using a weighted measure allowed for room for error and produced clearly optimal results. Having a two-level classification further made the framework effective with a hierarchical approach.

Several questions do arise that would make for great further clarification and research. Can some malware code hide in the function calls in which the models did not meet the required accuracy rate? Can it be determined that these classes of malware do and will continue to comprise a large proportion of cumulative attacks now and in the future? The evolution and re-editing of malware, along with new developments in computing technology, may pose new challenges for this framework. I must further inspect this scenario, and hopefully am not inaccurate in disregarding significant factors when inquiring, but could malware avoid detection by, as displayed in a basic example Figure 1, producing a massive amount of noise to retain the memory access proportionality of a “benign” classified program?

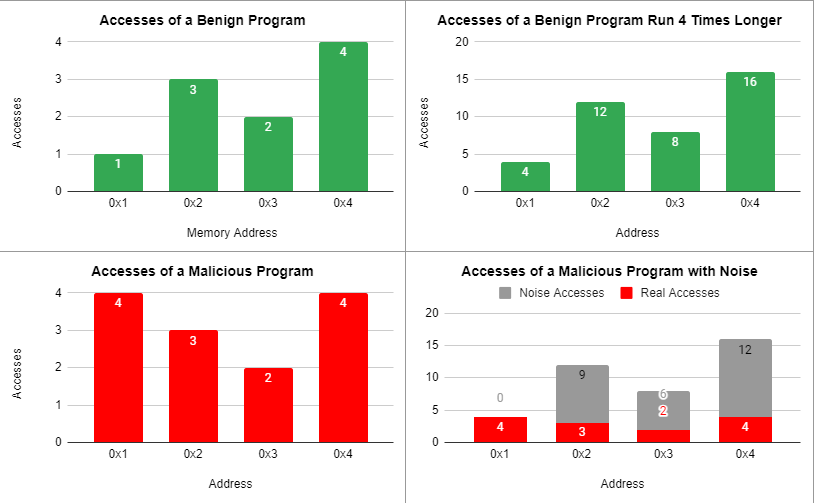


Fig. 1: Comparison between a benign program, a benign program that runs four times longer, a malicious program, and a malicious program with “noise” memory accesses.

1. Conclusion

The framework is a revolutionary idea that combines techniques in machine learning with running knowledge of the requirements of malware analysis and effective processing. Accuracy of the model cannot be easily obtained by many practical models, and the considerations were important to optimizing this. Other models have attempted to pursue a solution to the labor-intensive problem of human-powered malware detection with machine learning, but have missed the mark when it came to the design choices that were made. There are many questions that beg for discussion, but are likely addressed should one examine data collected and functionality applied into the program that are visible outside the scope of the paper I have analyzed. All in all, the hardware-assisted malware detection with per-application models with epoch-based monitoring is a promising addition to everyday computer architecture and a potentially feasible solution to assuring that our computers can operate without risk of malware infestation.

1. Extension

Learning about the application of machine learning into malware detection has helped to arise the idea of possibly applying natural language processing into the detection process. Most malware code contains several string values that have distinguishing qualities and patterns that are currently best analyzed by a human analyst. It can be wondered if, along with an initial input of supervised learning, there can be some type of framework (whether software- or hardware-assisted) that can detect (or contribute to the detection of) malware by inspecting the strings that exist in the code. This is currently done by opening a command prompt and typing the argument “strings” along with the name of the application.

For example, certain websites or access to abnormal resources through insertion of string arguments could happen that may possibly be detected by this proposal. Also, strings that are used in arguments in a certain order to create malicious activity may be noticed. However, several criticisms arise. Strings are difficult to use to detect malware even by some of the most skilled analysts, with a high likeliness of false positives. It may be difficult, if not impossible, for the model to be trained to detect malware with the variability of strings. Also, the amount of initial research necessary to introduce the training process to malicious control flow may not be worth dismal results, and malware evolution may outpace any effectiveness. Even so, this may be effective in helping to categorize the malware and its purpose, while the framework proposed by the authors can initially detect that it is malware to begin with. There are many patterns that exist in the strings of malware that can shed light on what the functionality of the malware is. For example, noticing strings that resemble key values can imply that it is a keylogger malware. Strings that look like a URL may indicate that it is a network-based malware, and an IP address may indicate that it will access a remote computer. This could be an interesting application to either this framework or an independent development all in itself.

References

Idika, N., & Mathur, A. P. (2007). *A Survey of Malware Detection Techniques*, 8–9.

Selçuk, A. A., Orhan, F., & Batur, B. (2018). Intractable Problems in Malware Analysis and Practical Solutions. *Journal of Internet Technology and Secured Transactions*, *6*(2), 588–595. https://doi.org/10.20533/jitst.2046.3723.2018.0072

Demme, J., Maycock, M., Schmitz, J., Tang, A., Waksman, A., Sethumadhavan, S., & Stolfo, S. (2013). On the Feasibility of Online Malware Detection with Performance Counters. *Proceedings of the 40th Annual International Symposium on Computer Architecture*. https://doi.org/10.1145/2485922.2485970