Malware Detection using Machine Learning Based Analysis of Virtual Memory Access Patterns

Zhixing Xu Sayak Ray* Pramod Subramanyan Sharad Malik
Princeton University *Intel Corporation

Abstract—**Malicious software, referred to as malware, continues to grow in sophistication. Past proposals for malware detection have primarily focused on software-based detectors which are vulnerable to being compromised. Thus, recent work has proposed hardware-assisted malware detection. In this paper, we introduce a new framework for hardware-assisted malware detection based on monitoring and classifying memory access patterns using machine learning. This provides for increased automation and coverage through reducing user input on specific malware signatures.**

The key insight underlying our work is that malware must change control flow and/or data structures, which leaves fingerprints on program memory accesses. Building on this, we propose an online framework for detecting malware that uses machine learning to classify malicious behavior based on virtual memory access patterns. Novel aspects of the framework include techniques for collecting and summarizing per-function/system-call memory access patterns, and a two-level classification architecture. Our experimental evaluation focuses on two important classes of malware (i) kernel rootkits and (ii) memory corruption attacks on user programs. The framework has a detection rate of 99.0% with less than 5% false positives and outperforms previous proposals for hardware-assisted malware detection.

I. Introduction

Malicious software, referred to as *malware*, is an ever-growing security threat and the detection of malware remains an important area of research. The first step in detection is analysis. This involves either static or dynamic analysis of known malware and is usually performed offline with expert human input. Results of analysis are distilled into a "signature". One technique for malware detection is the use of static signatures to examine programs after they are loaded and before execution [8], [16]. Unfortunately, this can be defeated by malware which uses obfuscation. In response to this, dynamic behavior-based detection has been proposed (*e.g.* [10], [11]). These methods monitor the behavior of the system using operating system or hypervisor-based instrumentation in order to detect malicious behavior. Static and dynamic signatures can be derived using either deterministic or statistical techniques. Statistical techniques based on machine learning are used to find patterns corresponding to malicious behavior (*e.g.* [20], [13]). In contrast, deterministic signatures are typically constructed through human expert analysis [7], [9].

Traditional malware detection techniques – both static and dynamic – are implemented in software. In this paper, we argue against pure software implementations both because of their overhead and the ensuing trusted computing base (TCB) bloat. Software for malware detection is susceptible to the same vulnerabilities that malware exploits during infection and therefore can be, and often is, disabled by malware [24].

Hardware-assisted detection mechanisms are not vulnerable to such disabling and have been proposed for *specific classes* of malware [19], [15], [12], [6], [17]. Petroni et al. [19] introduced Copilot, a PCI-based coprocessor that periodically monitors snapshots of immutable kernel memory. Copilot can detect certain kernel-level rootkits by checking if these snapshots differ from their expected values. Subsequent efforts like Vigilare [15] and KI-Mon [12] improve upon Copilot. However these memory monitors are *outside* the processor, and monitor physical memory addresses "filtered" by on-chip caches. These physical addresses may change from run to run.

Hardware-based control flow integrity (CFI) approaches stand in contrast to these outside-the-processor designs. CFI-mon [23] is an in-processor monitor collecting hardware performance counters available in contemporary processors to detect control-flow deviations. Lucas et al. [5], [4] address the code-reuse attack by enforcing backward-edge CFI with hardware support. These methods rely on expert knowledge of the executable binary and its memory layouts.

The requirement of expert knowledge can be costly. Demme et al. [6] addressed this by using machine learning to spot the differences between "normal" and "malicious" performance counter measurements during execution. Ozsoy et al. [17] improve upon this by analyzing sub-semantic features like address references and the instruction mix of the program. The detection techniques proposed in [6], [17], which we refer to as "traditional" machine learning based detection, aim to find a classifier that separates malicious and benign programs. The trained classifier is a *single model* that attempts to distinguish between the behavior of *all* benign and *all* malicious programs. A major challenge is that most malware is injected in otherwise benign programs. For one particular program, its execution runs could be either benign or malicious depending on whether the malware injected is triggered, making it difficult to label the program during training. The classifier is also extremely sensitive to the choice of benign and malicious examples in the training set and may result in unacceptable false positives and false negatives.

In this paper, we propose a different malware detection scenario. Instead of seeking a single model that distinguishes all malicious and benign applications, we learn one model for each application which separates its malware infected executions from legitimate executions. The model is trained on

This work was supported in part by SONIC, one of the six SRC STARnet centers, sponsored by MARCO and DARPA. Sayak Ray contributed to this work while at Princeton University and the views expressed in this paper belong to the authors and do not reflect those of Intel Corporation.

both malicious and benign behavior of the application. When the program is loaded, its associated behavior model is loaded and its execution is monitored. If the process executes in a manner that causes the associated model to flag its behavior as suspicious, a software exception is raised. For example, consider a web browser injected with browser redirecting malware. The browser itself is a legitimate application. Rather than classifying the browser as a malicious/benign as in the "traditional" scenario, our detector distinguishes runs where the infection is triggered from ones where it is not and raises an exception when malicious behavior is detected.

The key insight underlying our work is that an infected application run will modify the control-flow/data structures compared to a benign run. This will be reflected in its memory access pattern. This is obvious for the important class of memory corruption vulnerabilities for code in memory-unsafe programming languages such as C/C++.The same is true for another important class of malware, kernel rootkits, which modify control flow in the operating system. While these two classes are used extensively in this paper, control-flow and/or data structure modification are intrinsic characteristics of malware. Thus, we propose hardware monitoring of memory accesses for classifying individual application runs as being malicious or benign. Since virtual addresses provide for a more consistent signature than physical addresses, we propose obtaining the virtual address trace through in-processor monitoring. A major challenge in monitoring memory accesses is the sheer volume of the data. Our framework addresses this by dividing accesses into *epochs*, *summarizing* the memory access patterns of each epoch into features which are then fed to a machine learning classifier. Experiments show this framework is effective in detecting diverse classes of malware.

The contributions of this paper are as follows:

- We target application-run-specific malware detection.
- We introduce a framework for malware detection that is based on online analysis of virtual memory access patterns in contrast to physical memory access patterns.
- We introduce novel summarization and feature extraction techniques for function/syscall memory access patterns.
- We demonstrate the feasibility of our approach with experiments that consider both kernel-level and user-level malware. We demonstrate its efficacy through extremely low false-positive and false-negative rates.

While the proposed classification methodology is intended to be realized in hardware through in-processor monitoring and classification, the details of hardware design are beyond the scope of this paper. The focus of this paper is demonstrating the value of such a framework, and addressing the data volume concerns in its design. This paper also focuses on offline learning but with recent breakthroughs in the development of machine learning cores, *e.g.* [21], we believe even online learning of the detection model is realizable in hardware.

II. Malware and Memory Access Patterns

We now describe certain common types of kernel and user-level malware and how they affect memory accesses.

Kernel Rootkits: Kernel rootkits modify kernel data structures to redirect control flow in system calls to malevolent code. The two most common ways are: system call table modification, which changes a function pointer in the syscall table, and virtual file system (VFS) function pointer hooking, which replaces function pointers in the VFS file operation structure.

User-level Malware: User-level malware primarily exploits memory vulnerabilities: buffer/heap overflow, return-oriented programming, etc. As with kernel-level rootkits, user-level malware introduces anomalous control flow; *e.g.* in return-oriented programming (ROP), the attack executes a sequence of "gadgets" which are carefully chosen from an existing code base, usually from library functions and chained to implement the malicious objective. In this example, the "signature" is the anomalous control jumps to library functions.

III. Monitoring Memory Accesses: Challenges and Solutions

As contemporary processors execute billions of instructions/second, storing and analyzing all memory accesses is not feasible for online monitoring due to the sheer volume of data. The memory access *trace* needs to be *summarized* while retaining essential characteristics that enable malware detection. Another challenge is the lack of delimiters in the raw memory access trace. In most malware infected program runs, the malicious behavior only occurs at certain phases of the execution, the other phases are normal program behavior. Effective identification of these phases normally requires human expert analysis. We address these as follows.

Epoch-based Monitoring: Program execution is divided into *epochs* and epochs are separated by inserting *epoch markers* in the memory access stream. We investigate three choices of epoch markers: (i) system calls, (ii) function calls, and (iii) the complete program run. We show experimentally that system calls are effective epoch markers for kernel-level malware. Function calls are effective epoch markers for user-level malware. Using the entire program run as an epoch is not feasible for continuous running programs such as web browsers but can be effective for small programs. It still has limitations and this is discussed further in §VI.

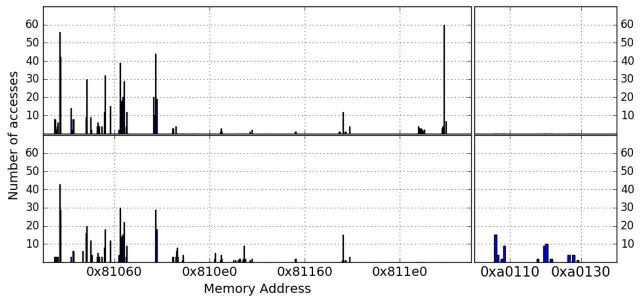

Fig. 1: Summary Histograms Example: benign (top) and malicious (bottom).

Summary Histogram: A memory access histogram summarizes the access pattern in an epoch. The bins of the histogram are virtual memory address regions of memory accesses. Histograms are labeled by the type of memory access, *e.g.* for the system call epoch we consider four types of memory accesses: (i) “far” calls (calls with absolute addresses), (ii) “near” calls (calls using relative addresses) (iii) branch instructions, and (iv) load/store instructions. In each epoch, four histograms corresponding to these types of memory accesses are used as the feature set for the machine learning model. Note that the summary histogram loses some timing information about the accesses. As shown in the evaluation, for most malicious behavior, the deciding features are the location and frequency of memory accesses rather than their sequence, so this reduction in precision is not significant. Figure 1 shows example of “far” call memory access histograms for read system call with and without rootkit attack. The additional accesses in the malicious run are “learned” by the classifier as corresponding to malicious behavior.

IV. Machine Learning for Identifying Malware

A. Feature Selection

For the system call epoch for kernels, we do feature selection using F-score, a commonly used measure of the discriminative power of features. We select the top 10% F-score features for the training phase. For the function call epoch, F-score based feature selection is not needed as the memory accesses in each call are limited to few memory regions resulting in a histogram with very few non-zero bins.

B. Classifier Architecture

Three commonly used classifiers: logistic regression, SVM and random forest are used by our work. We consider two design points for the classifiers.

Direct Classification of Memory Access Histograms: This performs binary classification directly on the summary histograms computed in each epoch. Empirically, this was effective in detecting kernel-level malware. During the training phase, each system call affected by a rootkit is labelled as malicious, while other system calls are labelled benign.[1] Since rootkits corrupt syscall execution, the classifier then learns to distinguish between benign and malicious syscalls.

Weighted Classification of Memory Access Histograms: For user-level malware the epoch boundaries correspond to function calls, which unlike system calls are not limited in number and are different across programs. Therefore, detecting malware by analyzing the histogram of a single function is hard. Our solution is to classify the different functions in a program separately and consider a weighted sum of classification results. Each execution is labeled as either malicious or benign and in the first phase classifiers are trained to recognize these labels. These labels are then assigned to every function in the execution, which may result in a function that was not infected being labeled as malicious. It is precisely this non-correlation that is addressed in the second training phase.

The second training phase uses a different training dataset. Assume we have n models for the n function calls and each model m_1 to m_n provides its classification result. An accuracy rate is assigned to each model. We discard weak models with accuracy below 60% as they have poor correlation between the run being malicious and the function being malicious. Assume that k models, M_1 to M_k are left, and they have accuracy rate r_1 to r_k. We assign a weight: $w_i = \frac{r_i - 0.5}{\sum(r_j - 0.5)}$ to each model. Assume the classification result from each M_i is c_i (0 for benign, 1 for malicious, if f_i is called multiple times, c_i is the average value of all the classification results). The classifier result is defined as the weighted sum of each model $C = \sum c_i w_i$. If C is above a threshold T, decided in the second phase, it is classified as malicious and otherwise it is classified as benign. This classification emphasizes functions whose infection correlates strongly with infected executions. It is important to note that this is done by the algorithm without any expert human analysis.

[1]In this scenario we need some human input to identify the system calls affected by each rootkit used in training. This is relatively easy compared to the manual analysis required for the techniques in [19], [15], [12].

V. Putting It All Together

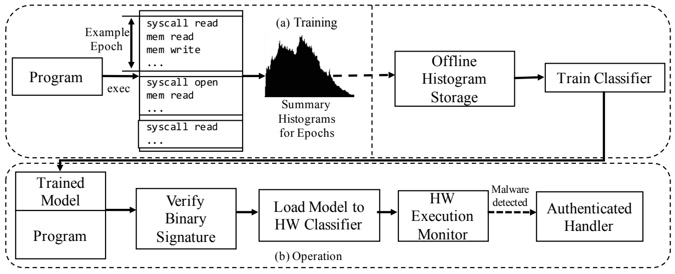

Fig. 2: Overview of the Complete Framework

Figure 2 shows the complete framework. The upper half shows training and the lower half monitoring/detection.

Data Collection and Training: During training, the program is executed and the summary histograms computed and stored for offline analysis. Once the data is collected, each histogram is labeled either “benign” or “malicious” and a classifier is trained to learn these labels. This classifier model is distributed along with the executable for the application/operating system.

Monitoring and Malware Detection: Our application-specific malware detection aims to ensure the integrity of commonly used applications (*e.g.* web browser, email) and the system kernel. There are only a few frequently used applications and kernel system calls, so the model storage space is not large. The classification model for each application is retrieved from the executable and used to program the hardware classifier at load time. Runtime authentication (*e.g.* using [1]) of the malware model and binary is necessary to to protect it from malware. When the classifier detects potential malware, it raises an exception that is processed by an authenticated software handler which executes in a hardware-enforced sandbox and performs detailed analysis of the executing program.

Address Space Layout Randomization (ASLR) introduces noise in the virtual memory traces because it shifts the code and data segments by adding a randomized offset to their initial addresses. Since this offset is known to the operating system, ASLR can be “de-randomized”.

Granularity of Memory Block Size: An important design parameter is the size of each bin in the summary histograms. There is a trade-off here: smaller bins provide more detailed memory access information than larger bins, but require larger net storage. However, from a machine learning perspective, histograms with smaller bins may not always make a better feature vector. Larger bins may be more preferable because the classifier may avoid "confusion" due to unimportant small-scale variations. The experimental evaluation considers bin sizes of 1KB, 4KB and 16KB.

VI. Evaluation

The space of malware is large, targeting different platforms and vulnerabilities, and it is not possible to obtain and experiment with a very large set from this space. Thus, we selected important representatives of both kernel and user mode malware to evaluate the efficacy of our monitoring framework. These are: (i) Linux kernel rootkits and (ii) memory corruption attacks on user code. We used the Scikit-learn [18] machine learning library. Experiments were conducted on a host machine with an Intel Xeon E5645 CPU with 128 GB RAM.

A. Case Study: Detecting Rootkits

Rootkits Analyzed: We experimented with the Linux kernel rootkits shown in Table I.[2] The table summarizes the kernel data structure modified by each rootkit and the system calls this modification influences. Malicious actions performed by these rootkits include file and process hiding, backdoor access to a root shell, interfering with I/O to/from files/network ports.

Rookit	Modified Structure	Infected syscall
avg-coder	VFS pointer	read, write, readdir, ...
kbeast	System Call Table	read, write, getdents, ...
override	System Call Table	write, open, getdents, ...
knark	System Call Table	readdir, getdents, ...
adore-ng	VFS pointer	read, open, readdir, ...
simple-rootkit	System Call Table	read
suterusu	System Call Table	read,write
ivyl	VFS pointer	read, readdir
Diamorphine	System Call Table	getdents
AFkit	System Call Table	getdents

TABLE I: Summary of Rootkit Samples.

Methodology: Although we envision memory accesses will be collected using specialized hardware, this initial evaluation gathers memory accesses using an instrumented version of QEMU 2.2.0 [2]. QEMU was used to execute a target machine which ran Debian "Squeeze" with Linux kernel v2.6.32.

In a rootkit infected system, the behavior of a system utility is changed to meet the purpose of the rootkit, *e.g.*, `ps` may hide malicious processes. Both benign and malicious traces were collected by running the following system utilities: `ls`, `ps`, `lsmod`, `netstat`. The utilities were executed with different current directories, background processes, arguments, etc. for a total of 50 runs for each rootkit. Both benign and malicious memory traces are summarized using the system call epoch. A detection model is was trained for each kind affected systcall.

[2]`knark_s` and `override` had to be modified to run on our target system.

The training set contained the rootkits: `avg_coder`, `adore-ng`, `kbeast` and `AFkit` (**bold** in Table I). We then test the ability of the learned model to distinguish between infected and benign systems on the remaining rootkits. *The machine learning algorithm is trained on the 4 rootkits, but is asked to detect 6 rootkits it has never seen before. These experiments demonstrate our framework can detect* ***new*** *malware.*

Detection Results: Figure 3 shows the rootkit detection results using our detection framework as a Receiver-Operating Characteristic (ROC) graph. The x-axis shows the false positive rate and the y-axis shows true positive rate. Points on the graph show the achieved detection rate at a certain false positive rate.

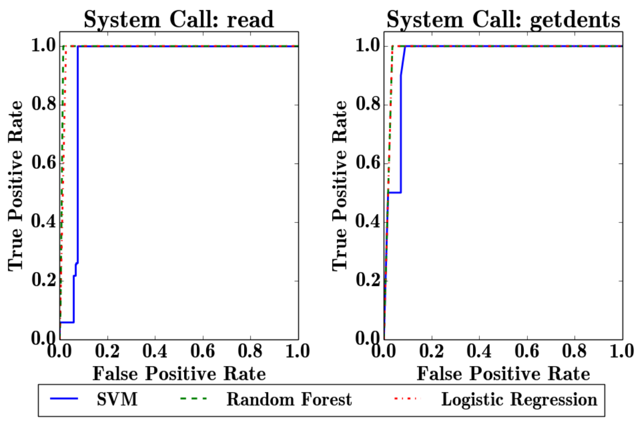

Fig. 3: Rootkit Detection using System Call Epoch

The graphs show the classification results for `sys_read` and `sys_getdents` using different machine learning classifiers with a 4k histogram bin size. Detection performance is not sensitive to histogram bin size: 1k and 16k bin sizes yield similar results. For both system calls, the best performing classifier (random forest) reaches 100% true positive rate, *i.e.*, detects all attacks, at $< 1\%$ false positive rate.

B. Case Study: User Level Memory Corruption Malware

In this section, we focus on user-level applications. We provide a direct comparison with Malware-Aware Processors (MAP)[17] that also targets user level programs. As mentioned in §I, MAP's detection scenario is different from ours. However, their feature sets can still be used in our detection scenario, and provide a good comparison to our feature set of virtual memory access patterns.

Benchmark Suite: We use the RIPE [22] benchmark suite for our experiments. RIPE is a synthetic benchmark which contains a total of 850 different memory corruption attacks in various forms including "modern" attacks such as return-to-libc attacks, return-oriented-programming, etc. This suite was executed on a Linux system running Ubuntu 6.06 distribution with kernel version 2.6.15. We also created a "benign" version of RIPE where each of the attack targets is patched.

Methodology: Among the 850 RIPE attacks, 751 were successful on our target system. For our two-level classification mode, we used 351 attacks and corresponding benign versions for training the first level classifier and 200 attacks and corresponding benign versions for the second level training. The remaining 200 pairs were used in testing. MAP only needs one training phase. Thus, 400 attacks and corresponding benign versions were used for training and the remaining 351 pairs are used for testing. We developed a "pintool" for Pin version 2.14 [14] to collect memory access patterns. As in the case of the rootkit experiments, we expect data gathering and detection will eventually be performed in hardware. *During the testing phase the machine learning algorithm attempts to detect attacks that have not been seen before based on other (somewhat similar) attacks it has seen.*

Detection using MAP's Feature Sets: Table II shows the feature sets used by MAP. There are three kinds of features based on: (i) architectural events, (ii) memory addresses, and (iii) the instruction mix. MAP collects data for every 10K instructions (its epoch) to form feature vectors and each feature vector is labeled as malicious/benign based on the program being executed. The detection model is then trained to label these 10K-instruction epochs as malicious/benign.

Feature	Description
ARCH	Frequency of memory read/writes, taken & immediate branches and unaligned memory accesses
MEM1	Frequency of memory address distance histogram
MEM2	Memory address distance histogram mix
INS1	Frequency of instruction categories[3]
INS2	Frequency of opcodes with largest difference
INS3	Existence of categories
INS4	Existence of opcodes

TABLE II: MAP's Feature Sets

Using the MAP Features in Our Detection Scenario: MAP provides us with potential features for malware detection. However, their framework is not applicable to our detection scenario as it is designed to label different programs while ours detects whether one particular application is infected by malware. In most malware infected program runs, the malicious behavior only occurs at certain phases of the execution, the other phases are normal program behavior. This makes it hard to label the 10K-instruction epochs correctly without manual intervention. If we label all the epochs in a malware infected run as malicious, most epochs that reflect normal program behavior would be wrongly labeled, resulting in huge training error. Therefore, instead of using the 10K-instruction epoch, we use the entire program run as the epoch. Although it is hard to correctly label the 10K-instruction epochs, any malicious behavior is reflected in the feature vectors of entire malicious runs, so labels can be correctly applied. These feature vectors are collected over the entire run and the classifier is trained to label the entire run. Data collection for the MAP features was also done using our "pintool".

Detection Result and Comparison: Figure 4 shows the malware detection results using our framework and MAP's features with both logistic regression and random forest classifiers. The ROC curve using each individual feature set is marked by the corresponding name in Table II. The curve marked "COMB" is the result of using all of MAP's features combined together. The curve marked "FUNC" shows the detection results of our method with function call as epoch.

[3]The instruction categories are based on Intel XED2 [3] instruction classes

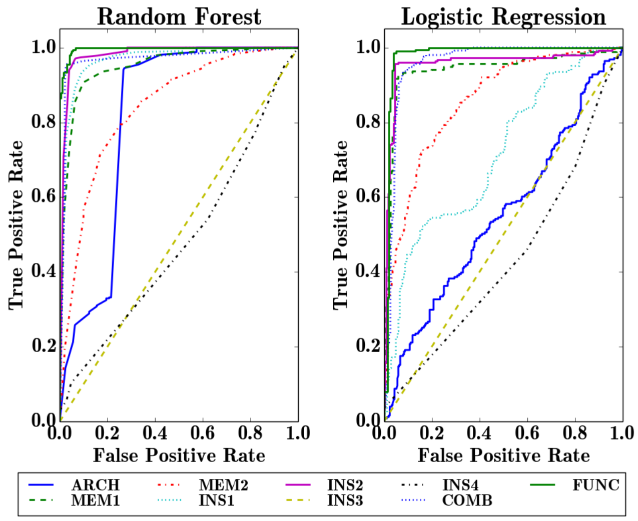

Fig. 4: Detection Performance using Different Feature Sets

Let the standard for good detection be > 95% true positive rate with false positives < 5%. Two feature sets/detection methods meet these criteria. Ours using two-level classification of memory accesses in the function call epoch not only meets the standard but also has the highest accuracy under any false positive rate. Among the MAP feature sets, only INS2 with the logistic regression classifier is slightly above the standard.

Table III shows the comparison between our method ("FUNC") and "COMB", "INS2" and "MEM1" which are the most successful among the MAP features. We see that for each false positive rate (FP), our method has best true positive rate (TP). Also, as the allowed false positive rate decreases, our method maintains a good detection rate while others drop quickly. These results show the strength of our method, especially in the very low false positive rate regime.

TP Rate		Logistic Regression			
		FUNC	COMB	INS2	MEM1
FP Rate	1%	87.1%	13%	41.4%	32.7%
	5%	98.0%	83.3%	95.6%	86.5%
TP Rate		Random Forest			
		FUNC	COMB	INS2	MEM1
FP Rate	1%	88.4%	31.8%	70.8%	53.6%
	5%	99.0%	94.6%	94.5%	85.4%

TABLE III: Performance with Function Epoch and MAP Features

Function Call vs Entire-Program Epoch: Since using the entire program run as an epoch gives reasonable results with MAP's feature sets, it is worth comparing this with the function call epoch. The main problem with using the entire-program-run epoch is that several applications run for an indeterminate amount of time (*e.g.* web browser). Summarizing over the entire program run would require the program to finish which, if not making the training phase impossible, would likely add error as the malicious part could be a small part of the run. In contrast, by summarizing over a function call, the training data is easier to collect and more accurate. For applications that do not execute continuously, summarizing over the entire run is feasible. RIPE has this characteristic. To evaluate the value of using function call as epoch for it, we built the memory access histograms for the entire program run, and trained them using different classifiers.

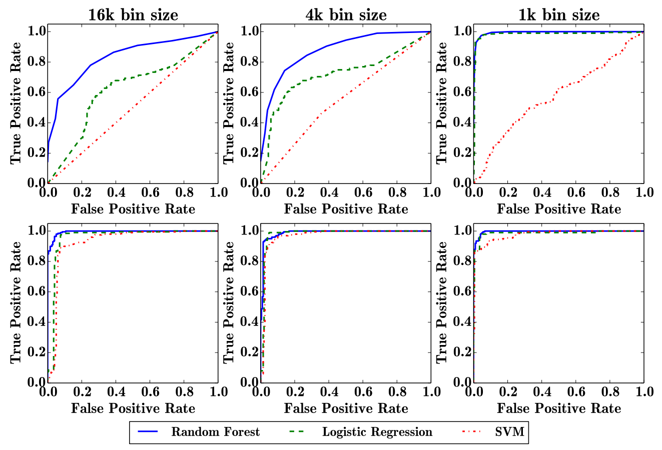

Fig. 5: Detection Performance with Different Memory Block Sizes (Upper Row: Entire-Program Epoch; Lower Row: Function Call Epoch)

Figure 5 shows the performance of using the function call as epoch and using the entire-program epoch under different memory histogram bin sizes. The results are presented using ROC graphs. From the graphs, we see that when using small histogram bin sizes, the two epochs perform similarly. But when the bin size is larger, the results of the function epoch stay mostly the same while the detection rate using the entire-program epoch deteriorates. Thus, the function call epoch is resilient to changes in histogram bin size, while with the entire-program epoch, the histogram bin size needs to be chosen carefully. This may need human input and may possibly differ between applications, thus limiting automation.

VII. Conclusions

In this paper, we introduced a framework for malware detection based on online analysis of virtual memory access patterns using machine learning. This framework was applied to the application-specific malware detection scenario which targets detecting malware infected runs of known applications. We addressed the challenge of online memory data collection using a system/function-call epoch based memory access summary. We experimentally covered both kernel and user-level threats and demonstrated very high detection accuracy against kernel level rootkits (100% detection rate with less than 1% false positives) and user level memory corruption attacks (99.0% detection rate with less than 5% false positives). A key value of the proposed methodology is using machine learning to determine malware signatures for classification in contrast to the traditional reliance on human insight – a major step in automating this critical analysis problem.

References

[1] Apvrille, A., Gordon, D., Hallyn, S., Pourzandi, M., and Roy, V. Digsig: Runtime authentication of binaries at kernel level. In *18th USENIX Conference on System Administration* (2004).

[2] Bellard, F. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track* (2005), pp. 41–46.

[3] Charney, M. Xed2 user guide. http://software.intel.com/sites/landingpage/pintool/docs/56759/Xed/html/main.html., 2011.

[4] Davi, L., Hanreich, M., Paul, D., Sadeghi, A.-R., Koeberl, P., Sullivan, D., Arias, O., and Jin, Y. Hafix: hardware-assisted flow integrity extension. In *Proceedings of the 52nd Annual Design Automation Conference* (2015), ACM, p. 74.

[5] Davi, L., Koeberl, P., and Sadeghi, A.-R. Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation. In *Proceedings of the 51st Annual Design Automation Conference* (2014), ACM, pp. 1–6.

[6] Demme, J., Maycock, M., Schmitz, J., Tang, A., Waksman, A., Sethumadhavan, S., and Stolfo, S. On the Feasibility of Online Malware Detection with Performance Counters. In *40th International Symposium on Computer Architecture* (2013).

[7] Ellis, D. R., Aiken, J. G., Attwood, K. S., and Tenaglia, S. D. A behavioral approach to worm detection. In *Proceedings of the 2004 ACM Workshop on Rapid Malcode* (2004), WORM '04.

[8] Idika, N., and Mathur, A. P. A survey of malware detection techniques. *Purdue University 48* (2007).

[9] Ilgun, K., Kemmerer, R. A., and Porras, P. A. State transition analysis: A rule-based intrusion detection approach. *IEEE Transactions on Software Engineering 21*, 3 (Mar. 1995).

[10] Jacob, G., Debar, H., and Filiol, E. Behavioral detection of malware: from a survey towards an established taxonomy. *Journal in computer Virology 4*, 3 (2008), 251–266.

[11] Kolbitsch, C., Comparetti, P. M., Kruegel, C., Kirda, E., Zhou, X.-y., and Wang, X. Effective and efficient malware detection at the end host. In *18th USENIX Security Symposium* (2009), pp. 351–366.

[12] Lee, H., Moon, H., Jang, D., Kim, K., Lee, J., Paek, Y., and Kang, B. B. Ki-mon: A hardware-assisted event-triggered monitoring platform for mutable kernel object. In *Proceedings of the 22nd USENIX Conference on Security* (2013).

[13] Lee, W., and Stolfo, S. J. Data mining approaches for intrusion detection. In *7th USENIX Security Symposium* (1998), SSYM'98.

[14] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. Pin: building customized program analysis tools with dynamic instrumentation. In *ACM Conference on Programming Language Design and Implementation* (2005).

[15] Moon, H., Lee, H., Lee, J., Kim, K., Paek, Y., and Kang, B. B. Vigilare: Toward Snoop-based Kernel Integrity Monitor. In *ACM Conference on Computer and Communications Security* (2012).

[16] Moser, A., Kruegel, C., and Kirda, E. Limits of static analysis for malware detection. In *23rd annual Computer Security Applications Conference* (2007).

[17] Ozsoy, M., Donovick, C., Gorelik, I., Abu-Ghazaleh, N., and Ponomarev, D. Malware-aware processors: A framework for efficient online malware detection. In *IEEE 21st International Symposium on High Performance Computer Architecture* (2015).

[18] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., et al. Scikit-learn: Machine learning in python. *The Journal of Machine Learning Research 12* (2011), 2825–2830.

[19] Petroni, Jr., N. L., Fraser, T., Molina, J., and Arbaugh, W. A. Copilot - a Coprocessor-based Kernel Runtime Integrity Monitor. In *13th USENIX Security Symposium* (2004).

[20] Wang, K., and Stolfo, S. J. Anomalous payload-based network intrusion detection. In *Recent Advances in Intrusion Detection* (2004), Springer.

[21] Wang, Z., Lee, K., and Verma, N. Overcoming computational errors in sensing platforms through embedded machine-learning kernels. *IEEE Transactions on VLSI Systems 23*, 8 (2015).

[22] Wilander, J., Nikiforakis, N., Younan, Y., Kamkar, M., and Joosen, W. Ripe: runtime intrusion prevention evaluator. In *27th Annual Computer Security Applications Conference* (2011), ACM.

[23] Xia, Y., Liu, Y., Chen, H., and Zang, B. CFIMon: Detecting violation of control flow integrity using performance counters. In *IEEE/IFIP International Conference on Dependable Systems and Networks* (2012).

[24] Xue, F. Attacking Antivirus. In *Black Hat Europe Briefings* (2008).