Semantics-Based Online Malware Detection: Towards Efficient Real-Time Protection Against Malware

Sanjeev Das, Yang Liu, Wei Zhang, and Mahintham Chandramohan

Abstract—**Recently, malware has increasingly become a critical threat to embedded systems, while the conventional software solutions, such as antivirus and patches, have not been so successful in defending the ever-evolving and advanced malicious programs. In this paper, we propose a hardware-enhanced architecture, GuardOL, to perform online malware detection. GuardOL is a combined approach using processor and field-programmable gate array (FPGA). Our approach aims to capture the malicious behavior (i.e., high-level semantics) of malware. To this end, we first propose the frequency-centric model for feature construction using system call patterns of known malware and benign samples. We then develop a machine learning approach (using multilayer perceptron) in FPGA to train classifier using these features. At runtime, the trained classifier is used to classify the unknown samples as malware or benign, with early prediction. The experimental results show that our solution can achieve high classification accuracy, fast detection, low power consumption, and flexibility for easy functionality upgrade to adapt to new malware samples. One of the main advantages of our design is the support of early prediction—detecting 46% of malware within first 30% of their execution, while 97% of the samples at 100% of their execution, with <3% false positives.**

Index Terms—**Malware detection, hardware-enhanced architecture, runtime security, early prediction, reconfigurable malware detection.**

I. Introduction

In RECENT years, malicious software (in short "malware") has skyrocketed in the embedded platforms and resource-constrained systems, such as smart phones and tablets. Recent McAfee report [1] shows that mobile malware samples grew by 16% during third quarter of 2014 with total samples exceeding 5 million and by 112% in 2014. However, effective malware detection has been a challenging task because sophisticated techniques are used by the malware writers to exploit the system vulnerabilities.

Manuscript received May 27, 2015; revised August 25, 2015; accepted September 30, 2015. Date of publication October 15, 2015; date of current version December 9, 2015. This research was supported in part by the National Research Foundation, Prime Minister's Office, Singapore under its National Cybersecurity R&D Program (Award NRF2014NCR-NCR001-30) and administered by the National Cybersecurity R&D Directorate. The associate editor coordinating the review of this manuscript and approving it for publication was Prof. Ozgur Sinanoglu.

S. Das, Y. Liu, and M. Chandramohan are with Nanyang Technological University, Singapore 639798 (e-mail: sanjeevk001@e.ntu.edu.sg; yangliu@ntu.edu.sg; mahintha001@ntu.edu.sg).

W. Zhang is with Hong Kong University of Science and Technology, Hong Kong (e-mail: wei.zhang@ust.hk).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TIFS.2015.2491300

Despite the fact that significant amount of work has been done in malware detection, they have not been successful in combating the ever-evolving and the sophisticated malware. Typical static solutions — such as antivirus, scanners and anti-malware tools — use a signature based method to detect the malware [2]. Unfortunately, adversaries use obfuscation techniques (e.g., code encryption) [3] and write several variants of the same malware to evade the signature based static detection techniques. In response to this, dynamic approaches were proposed, which analyze the program behavior during execution. Principal dynamic techniques include virtual machine inspection [4], function call monitoring [5], [6], dynamic binary instrumentation [7] and information flow tracking [8]. Though these approaches can potentially detect the obfuscated malware and also the malware variants, they require a large amount of resources and have a substantial overhead on the system. These difficulties often limit the malware detection to use a static approach (e.g., antivirus, scanners), however, they can be easily evaded due to the known limitations.

System calls based techniques have been recognized as a promising dynamic approach for malware detection. System call patterns provide effective information about the runtime activities of a program, which can be used to characterize the malicious behavior. Recently, researchers explored clustering techniques to group the system calls of a sample to identify the malicious patterns. To this end, [9] proposed *n*-gram approach to cluster the system calls, whereas other methods grouped them based on common OS resources [10], [11]. However, the current implementations [9]–[11] are mainly software-based, which have the same level of vulnerability as the software. Currently, adversaries use advanced techniques (e.g., anti-virtual machine inspection and anti-debugger [12]) against these protections to hide the malicious activity and finally evade the detection. To give an instance, a malware [13] uses anti-debugger techniques — anti-ptrace, anti-sigtrap and anti-breakpoint — to detect the presence of malware detector in the host and exits cleanly once it finds them. Most importantly, software approaches are infeasible for certain classes of malware; such as rootkits and bootkits achieve their malicious intents by infecting the kernel [14]. Since the rootkits have the similar privilege as the OS (Ring 0), they can disable and evade the software protection. In addition, the software methods are resource demanding, which causes a high energy demand and a substantial performance overhead, when employed on resource-constrained systems.

In this work, we present the first system call based approach using hardware-enhanced architecture that employs machine learning technique for malware detection. We name it GuardOL (guard online). GuardOL uses a novel frequency-centralized model (FCM) for feature construction to learn the malicious behavioral patterns from known malware samples. Our frequency-centralized model takes the frequency of resource-critical system calls into account and constructs features by grouping system calls using comprehensive rules to capture the semantics of malicious behavior. To this end, GuardOL extracts the system calls (with their arguments and return values) during execution on the processor and groups relevant system calls to construct features using FCM. The features obtained from the malware and benign samples are used to train multilayer perceptron (MLP), an artificial neural network model, which is used at runtime to perform the classification of the running program as malware or benign. We develop an architectural design of GuardOL based on our proposed methodology. For the proof of concept, we implement our model in FPGA. We leverage the advantages of FPGA platform to obtain a high performance training and detection at a low cost and reconfigurability for post-fabrication functionality upgrade to adapt to new malware samples. Reconfigurability of FPGA also allows sharing of hardware for classifier training and runtime detection.

Our approach aims to capture the semantics of malicious behavior using the proposed frequency centralized model and therefore has the potential to detect malware variants and even zero-day (previously unseen) attacks. Compared to the software approaches, GuardOL is resistant to aforesaid advanced malware techniques, as it is based on hardware. Unlike software techniques, it cannot be disabled or discovered by sophisticated techniques (e.g., anti-virtual machine and anti-debugger) and thus malicious activities will not be hidden from detection. Moreover, our implementation offers a power-efficient malware detection design, which has low resource demands and negligible performance overhead on the system. The FPGA implementation of GuardOL consumes 0.36 W during training and 0.264 W at runtime. Our results show that GuardOL achieves faster detection with zero performance penalty on the processor.

Our method supports an early prediction of malware samples as it performs the detection during their execution. The early prediction facilitates the real-time capability to the malware detection. In general, malware needs to pass through several phases (such as debugging, environment setup) before they can perform malicious operations in the system. Hence, it is an additional advantage if the malware samples can be detected in their initial stages, before the major damages have been performed in the system. The experimental results on our dataset show that GuardOL can detect 47% of malware samples within first 30% of their execution, while 98% of the samples after their execution, with <3% false positives.

Our contributions in this work are:

- We propose a frequency-centralized model (FCM) for feature construction, which uses comprehensive rules to group the system calls using their arguments and return values, to capture the semantics of malicious behavior.
- We present an architectural design of GuardOL based on the proposed FCM and machine learning classifier (MLP) to detect malware at runtime. We explore the design considerations for GuardOL implementation.
- We explore the early prediction of malware and show that it has the potential to detect malware in its early phase of execution.

GuardOL employs system calls and their arguments to model the behavior of the program, thus it can detect malicious attacks that leave evidence in the system calls. In general, most of the malware invoke system calls to perform malicious activities. Kernel-level malware that does not modify the system call library can be detected by our method. But, some kernel rootkits (e.g., SucKIT, Adore-ng, Sk2rc2) that manipulate the instructions inside system calls or the entries of the system call table [15] cannot be detected by our method. Nevertheless, GuardOL is an extensible architecture that can support instruction-level evidence to detect such rootkits. We reserve this extension for future work. In this article, we assume OS and typically the system call libraries must be trusted.

The rest of the paper is organized as follows: In Section II, we discuss the malware preliminaries. We present our feature construction model in Section III and the offline evaluation in Section IV. Section V describes our proposed hardware design of GuardOL. In Section VI, we present the hardware implementation results. We discuss the related work in Section VII and finally conclude in Section VIII.

II. Malware Preliminaries

Since the early days of computing, malware has evolved from the simple exploits into the complex ones in the forms of — virus, worm, trojan, adware, spyware, backdoor, flooder, botnet, rootkit and bootkits [2]. Over the years, the motivation of malware authors has changed from exploits-for-fun to money-making business. With the arms race between security professionals and malware writers, the present day malware has highly evolved, which uses sophisticated techniques to exploit the vulnerabilities. Malware uses several channels to penetrate in the system, e.g., phising emails, usb, memory card, corrupt downloaded files, click on links, repackaged into normal applications, browser utilities, abusing application stores or exploiting old vulnerabilities [16]. Our work is based on the behavior of the malware in the host system, i.e., after they have penetrated in the system.

A. Dataset

Our dataset contains 472 Linux malware samples collected from Virusshare [17] and VX Heaven [18], which includes trojans, exploits, viruses, worms and rootkits. We used Virustotal [19] online tool for the classification of malware samples and identification of their types, presented in Table I. We selected 371 benign applications on Ubuntu OS 12.04, consisting of programs from several categories — internet, games, system tools, office, sound, video, multimedia and utilities. We divided the dataset into 70%-30%. 70% of the samples (i.e., set-A) were used for classifier training and testing using a standard 10-cross validation approach.

TABLE I
LIST OF MALWARE FAMILIES IN OUR DATASET

Types	Family	#Sample
Backdoor	Agent, BO, Bodoor, Boost, Dancer, Divine, Explodor, Fpath, Hydgo, IrcShell, Iroffer, Lala, NetBus, Phobi, Rooter, SitC, Small, SpyEye, SSh, Suffer, Unfstealth	39
Exploit	Acpi, Apache, Bind, Brk, Da2, Epoll, FormatStr, Freeciv, Glc, Ipb, Kmod, Linux, Local, Mysql, Named, Old, OpenSSL, Race, Rpc, ShellCode, Small, SSHD22, Ssl, Vmsplice, WuFtpd, Xpl	114
Flooder	Slice, Small	3
HackTool	Masan, Scanap, Sh, Sshbru, BF, CleanLog, Small, SVScan, Usmel, Vcmer	25
Net-Worm	Kork, Mworm, Ramen, Coptic, Hijack, Lion, Old, Slapper, Sorso, Usmel	20
Rootkit	Agent, Gabitzu, Matrics, R3dstorm	56
Trojan	Generic, Hopbot, Regen2k, Small, Agent, Blitz, Hacktop, Logftp, Ris, Zapchast	16
Virus	Alaeda, Bi, Binom, Caveat, Clifax, Diesel, EthClean, FortyTwo, Grip, Joper, Little, Mandragore, Nuxbee, Orig, Osf, Ovets, Piltot, Quasi, Rike, RST, Satyr, Sickabs, Siilov, Silvio, Small, Snoopy, Spork, Svat, Telf, Thebe, Thou, Winter, Wowood, Xone, Brundle, Dido, Eriz, Grip, Impok, Little, Mais, Mandragore, Osf, Pelf, Piltot, RcrGood, RST, Satyr, Small, Svat, Telf, Thebe, Vit	199
		472

Remaining 30% of the samples (i.e., set-B) were used for the evaluation of early prediction.

To generate system call traces, all malware samples were executed in a virtualized environment, consisting of 32-bit Ubuntu (12.04) OS. We extracted system calls and parameters using *strace* (a Linux debugger to trace system calls and signals). In a similar way, benign samples were executed using *strace* to extract the system calls in a normal environment. Common user operations were performed on the benign programs in order to trace their systems calls. We used WEKA tool [20] for the offline evaluation of machine learning classifier.

B. Low-Level Malware Behavior Observed in Our Dataset

Malware performs a series of actions in order to accomplish their malicious intents, for which they need to use OS resources such as filesystem, memory, process and network. System calls are invoked to access these OS resources. They are commonly used in malware analysis to abstract the malicious behavior, as they accurately describe the runtime activities of the program [5], [10].

We studied the behavior of malware samples to observe the patterns of system calls occurred during their execution. We consider only security-critical system calls, which are those system calls that modify the state of OS resources [11]. In this work, first we listed the most frequent system calls that modify the state of OS resources from malware programs.

```
1.   openat(AT_FDCWD,"/usr/bin",..) = 2
2.   brk(0) = 0x9ec8000
3.   brk(0x9ef1000) = 0x9ef1000
4.   getdents(2,..,32768) = 32536
5.   open("file1.out",O_RDONLY|O_CLOEXEC) = 3
6.   fstat64(3,..) = 0
7.   read(3,.., 1024) = 0
8.   open("../libc.so.6",O_RDONLY|O_CLOEXEC) = 4
9.   fstat64(4,..) = 0
10.  read(4,.., 1024) = 0
11.  mmap2(..,4,0) = 0xb7619000
12.  open("VirusShare_4da7b",O_RDONLY) = 13
13.  open("/tmp/temp0",O_RDWR|O_TRUNC,01) = 14
14.  read(13,..,8000) = 6633
15.  write(14,..,6633) = 6633
16.  chmod("/tmp/temp0",0100770) = 0
17.  utime("/tmp/temp0",..) = 0
18.  execve("/tmp/temp0",..) = 0
```

Fig. 1. Sample malware system call trace.

As our approach is based on determining the common system call patterns found in malware, we neglected those system calls that appear only in few samples. Finally, we selected 64 most frequent system calls.

Table II lists some common actions performed by malware samples on Linux OS, along with the corresponding security-critical system calls. We observe that the actions listed in the table are common to the benign programs. However, the combination of these actions may lead to the stealthy operations by malware. To give a few examples: in order to replicate itself, a virus first searches the executables in the system, copies its content into them and finally executes those infected executables; some exploits first change the signal action (e.g., interrupt, keyboard input) to redirect the execution to their own code upon some signal action. And then they perform malicious activities, which cannot be recognized by the OS, as the signal action has been modified. Therefore, in our malware detection approach (as explained in §III) we aim to learn the semantics of the high-level malicious behaviors, which use low-level actions to perform their intended goal.

III. FEATURE CONSTRUCTION

Feature selection is a crucial step of any machine learning based approach. The performance of any machine learning classifier depends heavily on how closely the features represent the characteristics of the class. In this section, first we brief the two approaches — *n*-gram [9] and BOFM [11] — that have been used by previous approaches for feature construction of malware. Then we propose our frequency-centralized model (FCM).

A. *n*-Gram Technique

An *n*-gram is a popular data mining technique used in feature construction [9]. This approach groups "n" system calls that appear in a consecutive order to form a feature. We illustrate this by an example (Fig. 1 is used throughout the paper to illustrate the feature construction). For system call trace shown in Fig. 1, as listed in Table III, there are 15 features obtained for 4-grams (*n*-gram technique, where n = 4).

TABLE II
LOW-LEVEL ACTIONS OBSERVED IN OUR MALWARE DATASET

Resource	Low-level behavior	System calls
Filesystem	Create/read/write/delete/copy	`read`, `write`, `creat`, `open`, `openat`, `unlink`
	Search executable/library files	`opendir`, `chdir`, `access`, `readdir`
	Modify file/properties	`utime`, `chmod`, `ftruncate`, `rename`
	Get file/directory information	`getdents`, `fstat`, `fstat64`, `fadvise64`
	Execute a file	`execve`
Process	Change signal actions	`rt_sigaction`, `rt_sigprocmask`
	Kill process	`kill`, `tgkill`
	Switch process context	`sched_yield`
Network	Receive/send information	`send`, `bind`, `connect`, `recvfrom`
	Monitor network events	`poll`, `epoll_create`, `select`,
	Receive command from network	`recvfrom`, `ioctl`
Memory	Increase data memory	`brk`
	Map file/device to memory	`mmap`, `mmap2`, `munmap`, `old_mmap`
	Set memory protection	`mprotect`

TABLE III
FEATURES FOR SAMPLE MALWARE TRACE (IN FIG. 1) USING DIFFERENT TECHNIQUES

	4-grams	BOFM	FCM
1.	`openat`, `brk`, `brk`, `getdents`	`openat`, `getdents`	`openat`, `getdents`
2.	`brk`, `brk`, `getdents`, `open`	`brk`	`brk`
3.	`brk`, `getdents`, `open`, `fstat64`	`open`, `fstat64`, `read`	`open`, `fstat64`, `read`
4.	`getdents`, `open`, `fstat64`, `read`	`open`, `fstat64`, `read`, `mmap2`	`mmap2`
5.	`open`, `fstat64`, `read`, `open`	`open`, `read`	`open`, `read`, `write`, `chmod`, `utime`, `execve`
6.	`fstat64`, `read`, `open`, `fstat64`	`open`, `write`, `chmod`, `utime`, `execve`	
7.	`read`, `open`, `fstat64`, `read`		
8.	`open`, `fstat64`, `read`, `mmap2`		
9.	`fstat64`, `read`, `mmap2`, `open`		
10.	`read`, `mmap2`, `open`, `open`		
11.	`mmap2, open, open, read`		
12.	`open, open, read, write`		
13.	`open, read, write, chmod`		
14.	`read, write, chmod, utime`		
15.	`write, chmod, utime, execve`		

TABLE IV
COMPARISON OF n-GRAM, BOFM AND FCM

	#Features			Performance		
Techniques	malware	benign	total	TPR	FPR	AUC
4-grams	5591	17433	22134	1	0.408	0.796
6-grams	9885	47776	56908	1	0.538	0.731
BOFM	132	219	258	1	0.192	0.904
FCM	119	140	186	0.976	0.012	0.997

We performed feature construction using 4-grams and 6-grams techniques, which have better performance as explored in [9] and [21]. The number of features using 4-grams and 6-grams techniques are 22134 and 56908 respectively, as shown in Table IV. Since the feature size is quite large, it requires substantially large memory for storing features and also involves heavy computations during training and runtime detection. Consequently, the high performance overhead limits the practical use of this approach.

B. BOFM

BOFM [11] is a scalable framework, which clusters system calls based on actions on common OS resources. To this end, BOFM uses 3 rules: the same set of actions on OS resource instance is considered as one feature; the sequence of actions is not considered; and identical action sets performed on two OS resource instances are taken as a single feature. Table III lists the features obtained for the trace in Fig. 1.

Using this approach on our dataset, the total feature size reduces to 258 (as shown in Table IV), which is a significant achievement. However, the approach does not consider the frequency of system calls, which can be leveraged by malware. Malware may consume resources of system by executing system calls repetitively, e.g., `brk()` system call is executed repetitively by *Brk.c* exploit. Such a class of malware can lead to the denial of service and CPU starvation attacks in the resource-constrained embedded systems.

C. Frequency-Centralized Model (FCM)

In this work, we propose a novel frequency-centralized feature construction model (FCM) to capture the semantics of malware behavior. We consider the frequency of resource-critical system calls for feature construction in order to detect malicious attacks that execute repetitive system calls to overload the system. Our proposed model is based on the high-level semantics of malicious behavior. We adopt a dynamic approach to monitor security-critical system calls along with their arguments and return values to identify malicious combinations through machine learning. We aim at a generic representation of a malicious behavior using system calls in order to detect the variants of the malware, and even zero-day attacks that have similar semantics. To this end, we propose the following rules to capture the high-level semantics of malicious attacks.

Rule 1: Group System Calls in Sets
The intuition behind this rule is that malicious objectives are accomplished by performing a series of actions on a particular resource. For example, in the above trace, the virus propagates itself by infecting the temporary file. First, it copies its content into the `temp0` file, changes the permission and the accessed time and finally executes the modified file. All these actions performed on a common file reveal malicious intents of the sample. Several variations can be found among viruses to achieve this replication behavior — the required file may be searched in the current/root/parent directories, the target file may be an existing executable, source code, library or temporary file.

We do not consider the *execution order of system calls* in this paper. Malware may perform its actions by calling system calls in a different order, e.g., the accessed time of `temp0` file may be modified first and the permission can be changed later or the same action can be performed in the reverse order, achieving the desired goal. Given these observations, set structure is an effective way to group the system calls for identifying malware, and in general to discover the malware variants. System calls are grouped in sets based on the common OS resource identifier. An OS resource identifier refers to an instance of the OS resource, e.g., a filename is an instance of filesystem resource. In Fig. 1, the directory "/usr/bin" is first opened (using `openat` at line 1) and its information is obtained (using `getdents` at line 4). Hence, `openat` and `getdents` are grouped in a set, as shown in Table III. Similarly, `open` (in line 5), `fstat64` (in line 6) and `read` (in line 7) are grouped into a set based on their action on a common file (i.e., "file1.out").

Rule 2: Trace the Memory Mapping of File/Device
Once the file/device is mapped into the memory (as shown in trace, using `mmap2`), it can be read and written by accessing the user space memory, which does not require the calling of system calls. The detection method based on system calls hence cannot notice the underlying operations on memory-mapped file/device by the malware. In order to detect such behavior, we consider the mapping of file/device into memory as a separate feature. In our example trace (Fig. 1), the file "libc.so.6" is mapped to a user space memory (using `mmap2` at line 11), hence, we take `mmap2` as a feature, shown in Table III.

Rule 3: Include the Source File of the Write Operations
Copying operations involve data propagation, which is a commonly observed behavior of malware. A virus propagates by replicating itself, for which it copies its content into the executable/temporary file partially or wholly, as shown by the trace in Fig. 1. Malware may send out the secret information from the host file system, which may lead to the leakage of private information. In order to capture such behavior, the source of the write operations (in Fig. 1, the source of the write operation is a virus file) needs to be mapped to the destination file. In the above trace (Fig. 1), lines 12-18 reflect the data movement from a virus source file (i.e., "VirusShare_4da7b") into a temporary file (i.e., "tmp/temp0"). In order to capture this behavior, the system calls related to the source and destination files are combined into a set (shown in Table III).

Rule 4: Count the Frequency of Resource-Critical System Calls
Malware often executes resource-critical system calls repetitively in order to overload the system resources (such as RAM, CPU), which may result into the crashing of OS, degrading the processor performance and draining the battery. And this may consequently lead to the attacks such as denial of service and CPU-starvation in resource limited systems. For example, *Brk.c* exploit uses `brk` system call repetitively to increase the data segment of user program and overloads the main memory. Other exploits execute `rt_sigaction` system call repetitively to change the default signal action for all the signals. Once the default action for the signal is modified, they perform malicious actions, which go unnoticed by the OS. Therefore, recording the frequency of these system calls is important to identify similar malicious behavior. In our example trace (Fig. 1), the frequency of `brk` system call is taken into consideration during feature construction.

In comparison to the software based solutions that use system calls directly [9], [21] or group system calls executed on common OS resources [11], we adopt a comprehensive semantics-based approach to extract malware behavior. Here we advocate that the above rules are proposed based on the ground facts of malicious behavior (which was observed in our dataset as mentioned in §II) with justifications as explained above. Our method adopts one rule similar to BOFM, i.e., Rule 1. Because of this, some features in Table III are common between FCM and BOFM. However, our additional Rules 2-4 are novel and help to improve the robustness of extracted malware features. Rule 2 helps to solve the challenge posed by memory-mapped files/devices to the system call based approaches, which may fail to detect memory operations. Similarly, Rule 3 assists in capturing the data propagation, which is a typical malware behavior. As opposed to BOFM [11], our method takes frequency of system calls into account (Rule 4). This is significant to detect malware samples (e.g., Flooder, Botnet), particularly in resource-constrained systems, that may overload the system resources and end up performing attacks such as denial of service. For instance, a malware can execute resource-critical system calls repetitively to consume available computational resources (e.g., network bandwidth, disk space and processor time), causing resource starvation and crashing the OS.

These rules not only provide robustness to the malware detection but also help to identify typical malicious behavior, which is supported by our high detection accuracy and reduced false positives, as shown by the results in §IV. In addition to this, our approach also reduces the feature size, as compared

Feature	n_0	n_1	n_2	n_3	n_4	n_5	n_6	n_7	n_8	n_9	n_{10}	n_{11}	n_{12}	...	n_N
Malware #	0	1	1	1	1	1	1	1	1	1	1	1	0	...	0

(a)

Feature	b_0	b_1	b_2	b_3	b_4	b_5	b_6	...	b_N
Malware #	0	1	1	1	1	1	0	...	0

(b)

Feature	f_0	f_1	f_2	f_3	f_4	f_5	...	f_N
Malware #	0	1	2	1	1	1	...	0

(c)

Fig. 2. Feature vectors using different techniques. (a) 4-grams. (b) BOFM. (c) FCM.

to the aforesaid techniques (as shown in Table IV). In the next section, we will also compare our classifier results with the *n*-gram and BOFM approaches.

IV. Offline Evaluation and Analysis

In this section, we present an offline analysis of our proposed feature selection method using software implementation. We also compare our method with the aforesaid feature construction models — *n*-gram and BOFM — for our dataset. The features are constructed using the complete trace of system calls obtained after the execution of the sample. In §IV-C, we further evaluate the early prediction using our FCM approach. The features for early prediction are constructed using partial system call traces.

A. Evaluation of FCM

Our method uses comprehensive rules proposed in §III to construct features. We utilize these features to construct a feature vector of size N for each sample, where N is the total number of unique features extracted from both malware and benign samples. The feature vectors are used to train the machine learning classifier. A feature vector represents the presence/absence (represented by 1/0) or frequency of a particular feature in the sample. The frequency of a feature in the sample is considered for the resource-critical system calls as explained by Rule 4. For the system call trace in Fig. 1, five features are extracted: f_1: {`openat`, `getdents`}, f_2: {`brk`}, f_3: {`open`, `fstat64`, `read`}, f_4: {`mmap2`}, f_5: {`open`, `read`, `write`, `chmod`, `utime`, `execve`} (shown in Table III). They are embedded in a feature vector as shown in Fig. 2c. Here, we can see that features f_1, f_3, f_4 and f_5 appear only once and f_2 appears twice, while rest of the features are assigned 0.

We use the feature vectors obtained from malware and benign samples to train the machine learning classifiers and to test the samples. Results are obtained using a standard 10-fold cross-validation on Weka tool. To measure the performance of the classifiers, we plot a receiver operating characteristic (ROC) curve for each classifier. An ROC curve is a graphical plot of a true positive rate (TPR) against a false positive rate (FPR) at various thresholds. TPR is the proportion of malware that is correctly predicted as malware and calculated as TPR = TP/(TP + FN); whereas FPR is the proportion of benign that is mis-predicted as malware and calculated as FPR = FP/(FP + TN). In order to select the best classifier for our dataset, we use the area under curve (AUC) measure for the ROC curves. AUC measures the effectiveness of the classifier to differentiate between two classes, i.e., malware and benign programs in our case. The higher value of AUC has a better classification accuracy. We used the following machine learning classifiers in our experiments: C4.5 decision tree (J48), naive bayes (NB), logistic regression (LR), support vector machine (SVM) using LibSVM, sequential minimal optimization (SMO), RIPPER rule learner (JRIP) and multilayer perceptron (MLP) [20]. Fig. 3 depicts the performance of the classifiers. Our results show that multilayer perceptron (MLP) has a higher AUC (0.997) as compared to the other machine learning techniques. With MLP as the classifier, our method has a TPR of 97.6% and an FPR of 1.2% for our dataset. Since MLP has a better classification performance, we choose MLP as a classifier in our method. Another advantage of using MLP is that it is modular and highly parallel algorithm, thus, we can leverage FPGA to perform several levels of parallel computations [22].

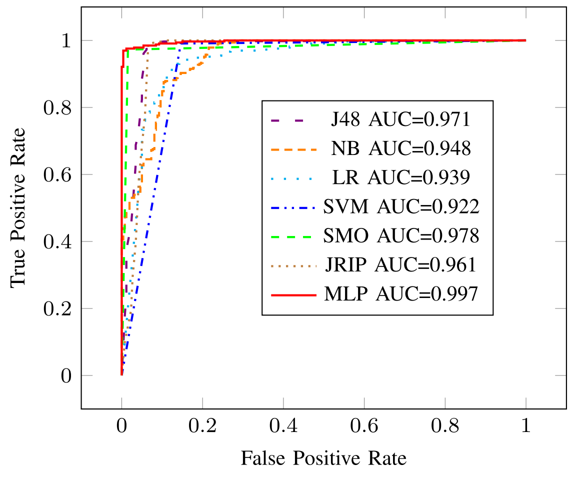
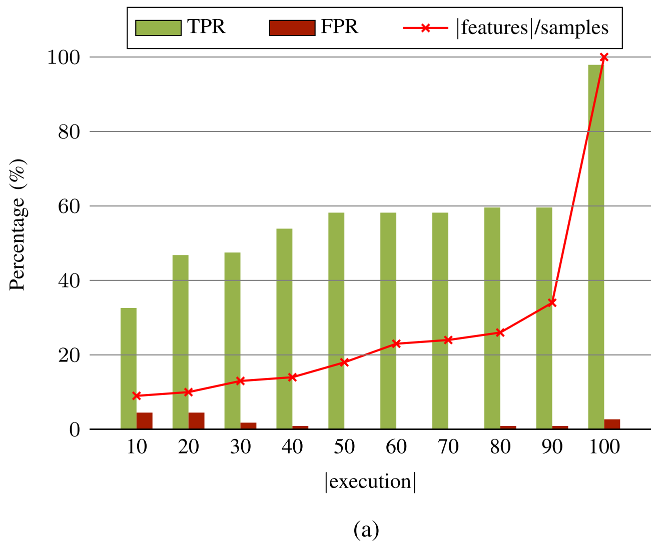


Fig. 3. Performance of classifiers using FCM.

Our approach is based on the system call patterns obtained from malware and benign programs. Similar pattern of system calls executed by malware and benign programs will eventually lead to false positives (i.e., 1.2% using MLP in our case).

B. Comparison of FCM With n-Gram and BOFM

We evaluated the classification performance using *n*-gram (4-grams and 6-grams) and BOFM approaches for our dataset using 10-fold cross-validation. Since these methods use support vector machine (SVM) as a classifier in [9] and [11], we also performed the evaluation using SVM. Table IV compares the performance of our approach with *n*-gram and BOFM methods. The main limitation of *n*-gram techniques is that they



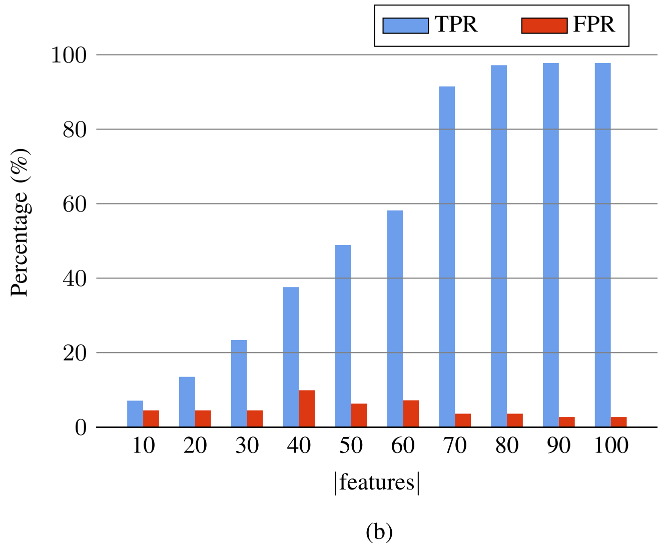


Fig. 4. Performance of Early prediction of malware. (a) With the increasing execution trace. (b) With the increasing features.

have a large number of features. This will introduce substantial memory and performance overheads. In addition, they suffer from high false positives because they consider the strict ordering of system calls and therefore contain more common system calls about the file system operations that are also found in benign programs [9]. BOFM reduces the feature size, however, it does not consider the frequency of system calls and other rules (Rules 2 and 3, as explained in §III), which can be exploited by malware to evade the protection. On the other hand, our FCM approach achieves considerable reduction in the feature size. Our FCM method significantly reduces the false positive rate (1.2%), while achieving marginally less true positive rate (97.6%), as compared to BOFM. Since the lower false positive rate is considered more important in malware detection, our approach has a better performance than the other two approaches. FCM has a lower false positive rate because we consider a more robust semantics of system calls, as explained in §III (i.e., Rules 2-4).

C. Early Prediction Analysis Using FCM

Early prediction is one of the key features of our approach, facilitating the detection of malware in real time. It is known that most of the malware performs malicious operations only at the end of their execution, while performing the debugging to retrieve system information, setting proper environment for malicious attacks in the early phase of execution. As shown in Fig. 1, first the system information is retrieved (lines 1-4), followed by setting up required conditions (lines 5-11) and finally the malicious activity is performed (lines 12-18). Since GuardOL performs the detection during the execution of the sample, based on the system calls traced from the running sample, it has the capability to detect malware during the early phase of execution (such as during the debugging or while setting up the environment). Thus, it may help to detect the malware before the stealthy operations have been performed.

Our approach uses MLP as the classifier. The MLP classifier is trained by a *Backpropagation* learning technique using the samples from set-A. As stated in §II, we test the early prediction using unknown samples, i.e., set-B. First we evaluate the performance of a classifier at different execution stages of the samples, i.e., at different percentage of execution (symbol |*execution*| defines % of execution) of the samples. For this, we collect the complete trace of system calls for the samples during execution. Then we extract features from the initial system call traces by using a certain percentage of the complete trace. As shown by the bar graph in Fig. 4a, our method can detect more than 46% malware samples within the first 30% of their execution, with a false positive rate of about 2%. As the execution continues, more features are collected, which allows the detection of more malware samples. Finally, after the complete execution of the samples, more than 97% malware samples are detected, with less than 3% false positive. The early prediction rate is dependent on the features of the samples traced during the execution and their likelihood of being malware. As shown by Fig. 4a, malware samples are mostly detected either at the beginning or after the complete execution of the samples. The line graph shows the percentage of features (symbol |*features*| defines % of features) per sample at different execution stages. With the increase in |*features*|/sample, the TPR also increases. Between 90% and 100%, there is a significant increase in TPR. This is because the number of features increases at the end of execution of samples, which is shown by the line graph.

Second, we evaluate the performance of early prediction based on different percentage of features (symbol |*features*| defines % of features) of the samples. For this, we collect the complete features from the samples. We test the early prediction using initial features at various percentage of the total features. As shown in Fig. 4b, with the increasing percentage of features, TPR also increases. Our results show that using only 50% of the features of the samples, more than 48% of the malware samples can be detected (with false positive of 6%), while using only 70% of their features, more than 90% of the malware samples can be detected (with false positive of 3%).

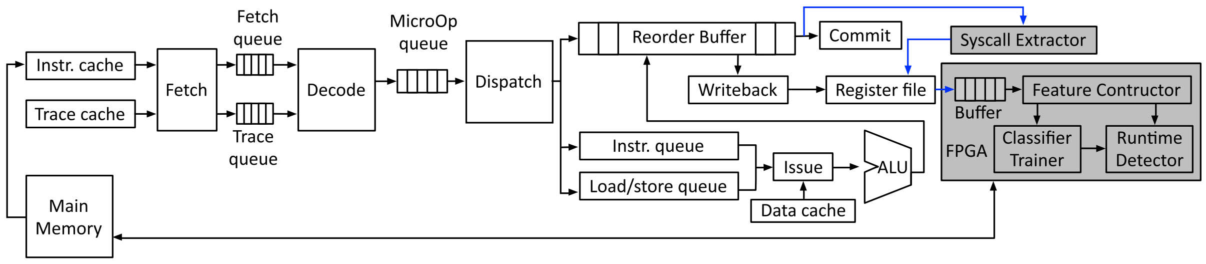


Fig. 5. Architecture of GuardOL.

Overall, the results demonstrate the significance of the early prediction in malware detection.

As illustrated by Fig. 4, the early prediction can effectively detect malware only after the sufficient features are collected from a program, however, it cannot be based on the length of the executed trace. Since machine learning takes a few cycles to perform the classification, it is better to perform the classification operation only when the new features are extracted. This will help to reduce the overall computation and performance overhead. The behavior of malware varies across different malware families. Hence, there could not be a uniform policy for a threshold number of features to be extracted before triggering the early prediction. Therefore, we suggest to trigger the early prediction once a new feature has been extracted.

V. Hardware Implementation

In this section, first we present the overview of our approach. Second, we present the architectural design of GuardOL and discuss its design considerations in detail.

A. Approach Overview

Fig. 5 presents the design of GuardOL. Our approach consists of two phases: *model building* and *runtime detection*. In the model building phase, we train our classifier engine using known malware and benign samples. In the runtime detection phase, the trained classifier is used to detect potential malicious behavior.

Model building phase contains the following three components:

Syscall Extractor logs system calls along with their arguments and return values invoked by the executed program, as shown by the example trace in Fig. 1.

Feature Constructor systematically extracts the meaningful features from system call traces, which are generated by Syscall Extractor, using our frequency-centralized model (FCM). And it embeds them into feature vectors as explained in §IV.

Classifier Trainer uses the feature vectors constructed above to train the classifier engine, i.e., multilayer perceptron (MLP). In our architecture, the classifier training is performed offline and the trained data (including features and updated weights) is stored into the main memory (as shown in Fig. 5). When the system boots up, the trained data has to be fetched from the main memory and stored into the FPGA for runtime malware detection. We allow training of the classifier only when the dataset is updated with new malware samples.

Runtime detection phase leverages the Syscall Extractor to get the system call information from the Processor and uses Feature Constructor and **Runtime Detector** components to do the real-time prediction of the running application, where prediction refers to classification of the unknown binary as either malware or benign. We will elaborate each component of GuardOL in §V-B.

In this work, we choose the platform consisting of Linux OS running on a 32-bit Intel x86 processor for generality. GuardOL utilizes OS-specific details for semantics of system calls and also to support multitasking operations. Our approach can also be ported to other OS and processor with the modification of the platform specification. Our architecture can support multitasking operations by tracing the control register CR3, as done in [6] and [23]. Linux OS assigns a unique page directory for each process. Intel x86 architecture stores the physical address of base of page directory for the current running process in control register CR3. Upon context switching, the entry in CR3 is overwritten by the physical address of the new process. This CR3 register can be monitored for multitasking support. System calls can be appended with a unique identifier to differentiate the system calls obtained from different processes.

In this article, we prototype the GuardOL design in FPGA. As compared to ASIC, FPGA implementation offers lower cost, fast prototyping and the flexibility to adapt to the new malware samples.

B. Micro-Architecture Design

In this work, we assume a System-on-Chip (SoC) architecture with a single core processor and an FPGA on one chip, as shown in Fig. 5. The interconnection between the FPGA fabric and commit stage of the processor is done at the design time of SoC. The FPGA logic can be updated in a trusted environment using a secured way, such as using bitstream encryption techniques [24]. Bitstreams can be stored along with the OS in main memory, as this region of the memory is secured and not accessible to the application programs. Our method assumes that the processor and the OS are trusted, but the programs running on them are not trustworthy.

As shown in Fig. 5, Syscall Extractor is integrated with the commit stage of the processor pipeline whereas, Feature Constructor, Classifier Trainer and Runtime Detector are implemented on FPGA fabric. A special design consideration

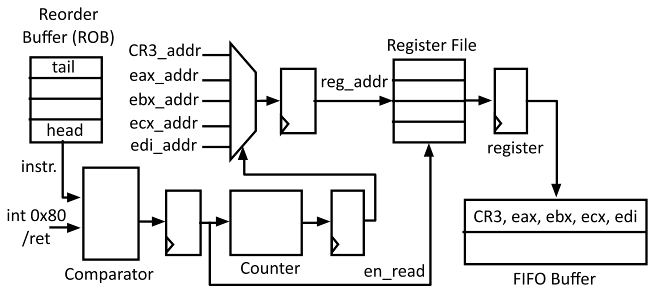


Fig. 6. Syscall Extractor design.

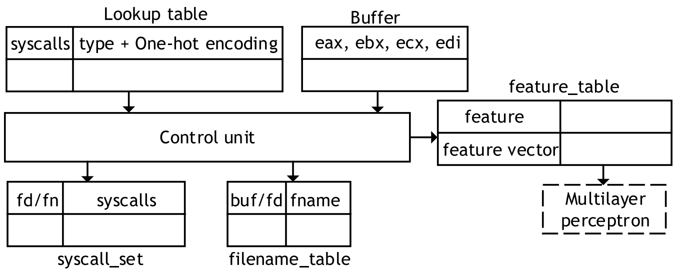


Fig. 7. Feature Constructor design.

is to introduce a Buffer component. Because the processor operates at a higher speed than FPGA, and in addition, Feature Constructor also takes several cycles to perform its operations, adding a buffer can avoid the processor to stall and reduce the performance penalty.

Now we describe the implementation of each component in detail:

1) Syscall Extractor: Syscall Extractor traces the system calls augmented with their parameters and return values from the running program. In Intel ISA, system calls are invoked by using trap instruction `int 0x80` (or `sysenter`). As shown in Fig. 6, the system calls are traced from the reorder buffer (ROB) at the commit stage of the processor pipeline to identify the trap instruction. In ×86 processor, for each of the instructions that enters the pipeline, ROB contains four fields: instructions, temporary register storage, result and validity of results. Once a trap instruction is committed in ROB, Syscall Extractor will read and store the system call type from register `eax` and other parameters from registers `ebx`, `ecx`, `edi`. For supporting multitasking operations, each entry in the buffer will be appended by the CR3 register value to identify the process. These register values will then be passed to the Buffer in FPGA. For some system calls, we also need to store their return values. Hence, once a following `ret` instruction is executed, its return value (given by `eax` register) will also be stored into the Buffer. The system call information stored in the Buffer will be further processed by Feature Constructor. Note that since the extractor only needs to read specific registers from the register file, it will not interfere with the normal instruction execution. The extractor is pipelined to catch up with the processor speed for instruction checking.

2) Feature Constructor: Feature Constructor involves the identification of relevant system calls to construct system call sets (a.k.a. *features*), followed by feature vector construction. Feature Constructor consists of control unit, lookup table (LUT) and three data tables — syscall_set, filename_table and feature_table with each table implemented as key-value pairs, as shown in Fig. 7. LUT maps system calls into their types (represented by 4-bits) followed by one-hot encoded representation of system calls (64-bit). *Note that we only need to consider 64 security-critical system calls in our method to provide satisfying classification accuracy.* The syscall_set table stores relevant system calls with their corresponding filename/file descriptor, whereas filename_table stores filename with their corresponding file descriptor/buffer pointer. The system calls use the pointer to the filename, which is unique for each filename in a program. We use this pointer to represent the filename in our design. We implemented the first column of each of three data tables as Content Addressable Memory (CAM) to expedite the searching process whereas the second column as conventional memory. The control unit is implemented as a state machine, which performs the fetching of system calls and parameters from the Buffer.

System call sets are constructed by grouping relevant system calls based on the rules introduced by FCM (in §III). To this end, we divide the security-critical system calls into 10 types, based on the semantics of system calls, their arguments and return value.

System calls with:

1. *first argument pointing to filename* (e.g., `chmod`),
2. *return value representing file descriptor* (e.g., `socket`),
3. *first argument as file descriptor* (e.g., `fstat`),
4. *no filename/file descriptor in arguments* (e.g., `uname`),
5. *resource-critical system calls* (e.g., `brk`),
6. `open`,
7. `read`,
8. `write`,
9. `close`, and
10. `mmap`

As stated above, the `eax` register is used to extract system calls (when the instruction is `int 0×80`) and return value (when the instruction is `ret`). Similarly, filename (fn) and file descriptor (fd) are given by `ebx` register; source/destination file (buf) for `read/write` system calls is given by `ecx` register; and file descriptor (fd) for `mmap` system call is given by `edi` register. System calls with common filename are grouped together and stored in syscall_set table (Rule 1). For memory mapping of a file or a device (Rule 2), `mmap` is separately stored corresponding to the file or the device in syscall_set table. filename_table maps the filename for the corresponding file descriptor. For `read` system call, filename_table stores the mapping of the content to the source file (Rule 3), which is used by `write` operation. feature_table stores features along with the corresponding feature vector, which consists of '1'/'0' or the frequency of a feature, as discussed in §IV.

We emphasize that one-hot encoding representation of a system call is an important design consideration. As said in §III, we do not consider the order of system calls for feature construction. To store the set of system calls, either the sorting technique or the hashing technique is needed. However, they require a large amount of computation and resources. Using one-hot encoding technique, system call sets can be constructed using the *ORing* technique, yet

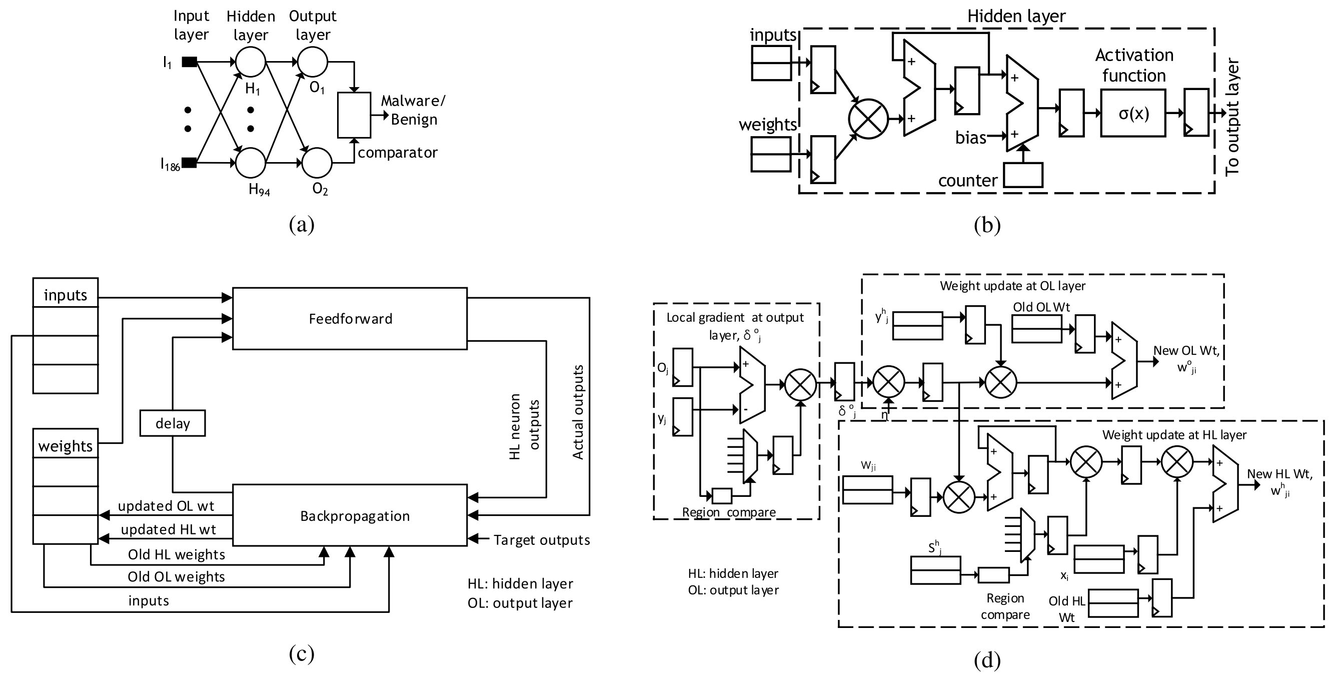


Fig. 8. Implementation of Classifier Trainer and Runtime Detector. (a) Runtime Detector design. (b) Implementation of feedforward logic. (c) Classifier Trainer design. (d) Implementation of backpropagation logic.

meeting the requirement of unique representation and order independency.

3) Runtime Detector: It performs feedforward computation in a MLP network (as shown in Fig. 8a), using the weights updated during model building phase, to classify a sample as a malware or a benign. The computation involved is given by Equation 1-2.

$$y_j^l = \sigma^l(s_j^l) = \sigma^l\left(\sum_{i=1}^{N_l} w_{ji}^l x_i + \theta_{ji}^l\right) \qquad (1)$$

$$\sigma^l(x) = 1/(1 + e^{-x}) \qquad (2)$$

where j is a neuron in layer l; s is the weighted sum; w_{ji}^l is the weight for the input i at neuron j; x_i is the input, θ_{ji}^l is the bias value; N_l is the total number of neurons in layer l.

At runtime, weights from the training are stored in the FPGA on-chip memory. Feature vector generated by Feature Constructor is taken as the input to the MLP network. In our design, hidden and output layer neurons as well as multiply and addition (MAC) operations in each neuron are pipelined, as shown in Fig. 8b. The output layer implements the same logic as the hidden layer. Now we detail our design choices for MLP implementation:

Network structure: Based on our dataset, the MLP network has 186 inputs in the input layer, 94 neurons in the hidden layer and 2 neurons in the output layer, as shown in Fig. 8a. Two neurons of the output layer determine the likelihood of the given sample being either malware or benign. We adopt the *winner takes all* strategy, i.e., if the output value at malware neuron (O_1) is greater than that of benign neuron (O_2), then the running sample is classified as a malware, otherwise as a benign program.

Activation function: Activation function is used for compressing the MAC output in each neuron between 0 and 1, given by $\sigma(x)$, as shown in Fig. 8b. In general, *sigmoid* or *tanh* function is used as an activation function (Equation 2), which requires a large number of resources due to the exponential computation. We implement activation function using *piece-wise linear approximation* (Equation 3) [22], for both hidden and output layers neurons, to save the hardware resources.

$$\sigma(x) = \begin{cases} 0 & \text{if } x \le -8 \\ (8 - |x|)/64 & \text{if } -8 < x \le -1.6 \\ (x/4 + 1/2) & \text{if } |x| < 1.6 \\ 1 - (8 - |x|)/64 & \text{if } 1.6 \le x < 8 \\ 1 & \text{if } x \ge 8 \end{cases} \qquad (3)$$

Number format, Data size, Precision: Arithmetic representation of inputs, weights and outputs of neurons have significant impact on hardware resource utilization [22]. Our design uses the fixed point implementation to save the hardware resources [22] while still maintaining the same classification accuracy. We implement our design using 1-4-11 format, with 1 bit for sign, 4 bits for integer and 11 bits for fraction. It has a high precision of 2^{-11}, which does not compromise the classification accuracy.

4) Classifier Trainer: Model building phase involves the training of the MLP network using feature vectors, which are obtained from the training data set. We use *Backpropagation* learning technique for finding appropriate weights for different inputs at each neuron. Two types of computations — feedforward and backpropagation — are performed during training (as shown in Fig. 8c), whereas only feedforward computation is performed during runtime using the trained weights. The detailed design of backpropagation computation is illustrated in Fig. 8d, whereas feedforward computation in Fig. 8b.

All the weights are updated as:

$$w^l_{ji}(n+1) = w^l_{ji}(n) + \eta\delta^l_j y^{l-1}_i \quad (4)$$

$$\delta^{l+1}_j = \varepsilon^{l+1}_j \sigma^l_1(s^{l+1}_j) \qquad \text{for } l = 1, 2, \ldots, L \quad (5)$$

$$\varepsilon^l_j = \begin{cases} O_j - y^l_j & \text{for } l = L \\ \sum_{i=1}^{N_{l+1}} w^{l+1}_{ji}\delta^{l+1}_j & \text{for } l = 1, 2, \ldots, L-1 \end{cases} \quad (6)$$

where, δ^{l+1}_j is the local gradient calculated based on the error ε, η is the learning rate, s and y^{l-1}_i are obtained from the feedforward computation in Equation 1, σ^l_1 is the first derivative of $\sigma(x)$ given by Equation 3 in layer l, O is the target output at the output layer neuron and L is the final output layer.

During training, error is calculated at the output layer and propagated backwards towards the hidden and input layers, as shown in Fig. 8c. The design is fully pipelined to improve its throughput. The computation of local gradients at output layer neurons (Equation 5) takes few cycles and thereafter, the weights at the output and hidden layers are updated in parallel (as depicted by Fig. 8c). Feedforward computation runs in parallel to the backpropagation, once the hidden layer weights begin updating. Backpropagation reads the old weights and updates the new weights into FPGA on-chip memory.

VI. Evaluation

A. Experimental Setup

To evaluate our design, we implemented Feature Constructor, Classifier Trainer and Runtime Detector on FPGA device of Virtex-5 LX110T using Xilinx ISE 14.5. Power consumption for FPGA logic was estimated using Xilinx Xpower Analyzer tool. Note that we use Virtex-5 device only to demonstrate GuardOL architecture. The real design of GuardOL has very small footprint with only required reconfigurable fabric. Syscall Extractor requires changes in the processor core (see Fig. 6). We used Synopsis Design Compiler tool to evaluate the area and power for Syscall Extractior at 65 nm technology.

To perform the function verification and performance estimation of the whole design, we integrated the models for the hardware components (frequency and number of cycles) into a cycle-accurate processor simulator, Multi2sim, which was configured as 32-bit, ×86 @2.66GHz to mimic the real processor. Then the instruction trace with extracted system calls was given as the input to the modified simulation platform.

B. Experimental Results

1) Syscall Extractor Cost Evaluation: The results show that Syscall Extractor has a negligible area overhead of 0.003% (3588.839 μm^2) for the processor, while consuming a very small power of 2.0762 mW. This reflects that the modification in the processor is minor. Moreover, since the logic connected to the register file is simple as shown in Fig. 6, i.e., two registers to drive a 4-to-1 multiplexer (MUX) and a small first in first out buffer (FIFO), the extra delay added to the processor pipeline is also small (less than 2%).

TABLE V
MEMORY REQUIREMENT

	#entries	width (bits)	size (bytes)
Buffer	10	160	200
Lookup table	64	100	800
syscall_set	300	96	3600
filename_table	225	64	1800
feature_table	186	80	1860
Weight	17768	16	35536
Total			43796

2) FPGA Implementation: Table V summarizes the overall memory requirement including Feature Constructor and MLP design (Classifier Trainer and Runtime Detector), which is less than 43 KB. Weights require more memory compared to other components (>34 KB) as the number of weights required by our design is 17768, each of 16-bits. The estimation has been given for one process. While monitoring multiple processes concurrently, the memory usage will be increased for Buffer, syscall_set, filename_table and feature_table. As per our results, the usage for these components is 7 KB per process, while weight can be shared by all the processes. Assuming the memory on modern FPGA to be in MB, it will be sufficient for multiple processes (e.g., 1 MB will be sufficient for 140 concurrent processes, including weight). Table VI presents the results for resource utilization in hardware and latency for Feature Constructor and MLP in FPGA. It can be seen that the resource requirement of the logic implementation is also low.

As mentioned before, we placed a buffer between Syscall Extractor and Feature Constructor to reduce the performance penalty due to the slow speed of FPGA. Since we trace only system calls, the size of the buffer is dependent on the gap between appearance of two successive system calls in the program and also on the processing speed of Feature Constructor. As per our results, the gap between system calls in the instruction stream is large, such that even a small buffer size (set to 10 in our design) can hold the system call information while FPGA continues its processing.

About the performance, our Feature Constructor takes 3-12 cycles (an average of 10 cycles) to complete its operations, depending on the type of system calls as discussed in §V-B.2. The MLP takes 17502 cycles on FPGA for one sample, both for training (running at 232 MHz) and detection (running at 250 MHz). As shown in Table VI, the dynamic power consumption for FPGA design, including Feature Constructor, Classifier Trainer and Runtime Detector is very low. Note that as more digital signal processing blocks (DSPs) are used in our design, the performance can be significantly increased due to the parallel processing of the samples with the cost of increase in area and power. The DSP blocks occupy around 10% of the total logic area (i.e., area of 2 DSPs≈5 CLBs)[1] and consume 1% of the total dynamic power on FPGA. Hence, when doubling the number of DSPs, the logic area and power increase around 10% and 1% respectively, but the throughput of MLP can be almost doubled.

[1]CLB: Configurable logic block.

TABLE VI
PERFORMANCE OF FPGA LOGIC UNITS

	Metrics	Feature Constructor	Classifier Trainer	Runtime Detector
Resources	Flip flops	625	537	253
	LUTs[2]	1254	770	396
	DSPs	-	8	2
Performance	Cycles	10 (avg.)	17502	17502
	Frequency (MHz)	200.56	232	250
	Execution time (μs)	0.0498	75.44	70
Power	Dynamic Power (W)	0.17841	0.18170	0.08638

Based on our understanding of system calls, we selected small buffer size (of 10 entries) in our experiments. We found no overflow for this buffer size because each system call execution takes large number of cycles, while the feature construction takes only about 10 cycles for its operation. For a processor running at 2.66 GHz and FPGA at 200 MHz, 10 FPGA cycles correspond to 130 processor cycles. The syscall invocation (using trap instruction) is followed by syscall handling subroutine and appropriate syscall handler, which take more than 130 cycles for execution. Thus, the buffer size of only one entry is required. But, to leave a safe margin, we selected the buffer size of 10 entries, which has no overflow in our experiments. Therefore, a small buffer size can hold the information while feature construction operation is performed.

VII. Related Work

In this section, first we overview the common software-based malware detection approaches. Second, we discuss the detection techniques that utilize hardware features. Third, we summarize the general hardware-based approaches that help to protect from the exploitation of vulnerabilities.

A. Software-Based Malware Detection

A considerable number of works have been done in malware detection. Malware detection techniques can be broadly classified into two categories — Static and Dynamic. Static approaches (e.g., antivirus, scanners) analyze a structure of the program by inspection, without executing the program. They use the signatures of the known malware samples. But these techniques can be easily evaded by the simple program transformation or code obfuscation (e.g., polymorphic malware) [25]. Malware authors write several malware variants, which have similar functionality but different signatures, to evade static protections. Besides, it requires a lot of manpower and time to extract the signatures of each malware. On a contrary, dynamic techniques analyze the program behavior during execution. The principal techniques include function call monitoring [5], [6], [9], [11], [26]–[28], virtual machine introspection [29], information flow tracking [8], [30], instruction trace monitoring [31]–[33]. Since they monitor the program behavior during the execution, they can potentially detect malware variants as well as the obfuscated malware. However, most of the previously proposed techniques have a high false positive rate, a substantial performance overhead and high resource demands. In addition, recent malware employs sophisticated concealment strategies (such as anti-debugging, anti-virtual machine) to hide its stealthy operations and evade the protection [12].

[2]LUT: Lookup Table.

System call patterns provide effective information about the runtime activities of the program. In the past, many system call based techniques have been proposed. Forrest *et al.* [5] first proposed anomaly detection using small sequences of system calls. Maggi *et al.* [34] used system call arguments and sequences to capture interrelations among different arguments of a system call, which were utilized to study time correlations to detect abnormal behaviors. Canali *et al.* [9] explored the data-mining techniques (such as n-gram, m-bag, k-tuples) to group the system calls and used machine learning techniques to capture the malicious behavior. However, these data-mining techniques generate large number of features. As a result, they have high resource demands and a substantial performance overhead. In response to this, Chandramohan *et al.* [11] proposed a scalable clustering approach, BOFM, to group the system calls based on the common OS resource identifier. Though BOFM significantly reduces the feature size, it does not consider the frequency of system calls for feature construction. This can be exploited by malware to perform attacks (such as denial of service and CPU starvation) by executing system calls repetitively. In contrast to this work, we adopt a frequency-centralized feature construction model to group the system calls and capture malware behavior based on more comprehensive rules (as described in §III). Compared to BOFM, our method significantly reduces the false positive rate. Furthermore, all these works consider offline analysis of malware, whereas our work focuses on online malware detection.

Software-based solutions have known limitations. They offer the same level of vulnerability as the software and thus, they can be bypassed by using sophisticated techniques [12], [13]. Moreover, certain class of malware (e.g., kernel-mode rootkits) can easily evade them, and thus making them undesirable for advanced malware detection. In addition, these solutions are resource demanding causing high energy consumption and a substantial performance overhead, when employed on resource-constrained systems.

B. Hardware-Based Malware Detection

A number of previous works leverage architectural features for malware analysis and detection. Bilar [31] used the difference of opcodes between known malware and benign programs for malware prediction. Similarly, other methods employ frequency of opcodes [32] and sequences of opcodes [33] to model the malicious behavior. Runwal *et al.* [35] proposed a graphical technique to find the similarity of the opcode sequence. However, these techniques require significant amount of work to model each program based on instructions. As the code size increases day by day, modeling program based on opcodes becomes a time-consuming process. Moreover, with the increment in code size, the memory requirement also increases. This will also result in significant performance overhead on the system, as each instruction of the program has to be traced.

Demme *et al.* [36] proposed the use of hardware performance counter to monitor the lower level micro-architectural parameters such as instruction per cycle, cache miss rate. Tang *et al.* [37] built baseline models of benign program execution using unsupervised machine learning to detect the deviations that occur as a result of malware exploitation. NumChecker [15] detects malicious modifications to a kernel function (system call) by checking the hardware events including total instructions, branches, returns and floating-point operations. Following the similar approach as in [36], Ozsoy *et al.* [38] proposed the design of a malware-aware processor using architectural events (e.g., frequency of memory read/writes, immediate branches taken, frequency of opcodes) as the features and machine learning for the classification of malware. The authors in [39] applied singular value decomposition technique to reduce the feature size. All the above methods use low-level hardware features to model the malicious behavior. They have low performance overhead, however, they are limited by a high false positive rate (about 10% [36], [38]). In comparison, our technique uses high-level features (i.e., system calls and arguments) to model the program behavior and has lower false positive rate, which is more significant in malware analysis.

Rahmatian *et al.* [23] proposed host-based intrusion detection using FPGA, which uses system call sequences to characterize the correct system behavior. The approach of system calls extraction from the processor is similar to ours (i.e., hardware modification to trace system calls using trap instruction), however, they use finite state machine (FSM) of the program generated offline to do runtime monitoring. The main limitations of this approach are: each program needs to be assisted by system calls based FSM; it does not allow unknown programs to run; and FSM of system calls would be significantly large for complex programs, thus demanding large memory size. In contrast, GuardOL uses semantics of malicious attacks and machine learning approach, which results in small feature models and applicability to unknown programs.

C. Hardware-Based Protection Approaches

Malware also exploits the system vulnerabilities by performing attacks such as buffer overflow and code reuse. A number of hardware approaches aim to protect from these attacks. Consequently, these hardware techniques also help to protect from the malicious operations. In this section, we brief the relevant hardware approaches that make it harder for malware to exploit the vulnerabilities.

Buffer overflow remains the most prevalent exploit till date. In the past, several architectural techniques [40]–[43] have been proposed to prevent from such attacks. With the introduction of DEP or W$\oplus$X [44], [45] protection, which allows memory page to be either writable or readable but not both at the same time, the traditional code injection attacks can be prevented. However, the adversaries developed a new type of attacks, code reuse attacks (CRAs), which use the existing code sequences instead of code injection. Return Oriented programming (ROP) [46] and Jump Oriented Programming (JOP) [47] are two common types of CRAs. Effective solutions based on control flow integrity [48]–[51] have been proposed to defend against CRAs. A complimentary line of research includes architectural mechanisms that aim to secure embedded systems by verifying the integrity of the code [52].

VIII. Conclusion

We presented GuardOL, a hardware-enhanced architecture to detect malware at runtime. Our approach first extracts the system calls and constructs the features based on the high-level semantics of malicious behavior. To this end, we propose a novel frequency-centralized model for feature construction. The features obtained from the benign and malware samples are then used for training the machine learning classifier, multilayer perceptron, which is used to detect the malware samples at runtime. The evaluation results show that GuardOL is fast, effective and has a marginal performance overhead on the processor. In future, we will extend our approach to multicore systems.

References

[1] (2014). *McAfee Labs Threats Report Quarter 3*. [Online]. Available: http://www.mcafee.com/us/resources/reports/rp-quarterly-threat-q3-2014.pdf, accessed Jan. 2014.

[2] M. Egele, T. Scholte, E. Kirda, and C. Kruegel, “A survey on automated dynamic malware-analysis techniques and tools,” *ACM Comput. Surv.*, vol. 44, no. 2, 2008, Art. ID 6.

[3] I. You and K. Yim, “Malware obfuscation techniques: A brief survey,” in *Proc. Int. Conf. BWCCA*, 2010, pp. 297–300.

[4] T. Garfinkel and M. Rosenblum, “A virtual machine introspection based architecture for intrusion detection,” in *Proc. NDSS*, vol. 3. 2003, pp. 191–206.

[5] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, “A sense of self for unix processes,” in *Proc. IEEE Symp. S&P*, May 1996, pp. 120–128.

[6] U. Bayer, A. Moser, C. Kruegel, and E. Kirda, “Dynamic analysis of malicious code,” *J. Comput. Virol.*, vol. 2, no. 1, pp. 67–77, 2006.

[7] J. Newsome and D. Song, “Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software,” NDSS, Tech. Rep., 2005.

[8] M. Christodorescu, S. Jha, and C. Kruegel, “Mining specifications of malicious behavior,” in *Proc. 6th Joint Meeting ISEC*, 2008, pp. 5–14.

[9] D. Canali, A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda, “A quantitative study of accuracy in system call-based malware detection,” in *Proc. ISSTA*, 2012, pp. 122–132.

[10] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda, “Scalable, behavior-based malware clustering,” in *Proc. NDSS*, 2009, pp. 8–11.

[11] M. Chandramohan, H. B. K. Tan, L. C. Briand, L. K. Shar, and B. M. Padmanabhuni, “A scalable approach for malware detection through bounded feature space behavior modeling,” in *Proc. IEEE/ACM 28th Int. Conf. ASE*, Nov. 2013, pp. 312–322.

[12] X. Chen, J. Andersen, Z. M. Mao, M. Bailey, and J. Nazario, “Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware,” in *Proc. IEEE Conf. DSN*, Jun. 2008, pp. 177–186.

[13] (Nov. 2014). *An Unofficial Analysis of the Retaliation Virus*. [Online]. Available: http://vxheaven.org/lib/vrn01.html

[14] J. Bickford, R. O’Hare, A. Baliga, V. Ganapathy, and L. Iftode, “Rootkits on smart phones: Attacks, implications and opportunities,” in *Proc. 11th HotMobile*, 2010, pp. 49–54.

[15] X. Wang and R. Karri, “NumChecker: Detecting kernel control-flow modifying rootkits by using Hardware Performance Counters,” in *Proc. 50th Annu. DAC*, 2013, Art. ID 79.

[16] (Jan. 2015). *Shellshock: A Technical Report*. [Online]. Available: http://www.trendmicro.com/cloud-content/us/pdfs/security-intelligence/white-papers/wp-shellshock.pdf

[17] (May 2014). *VirusShare*. [Online]. Available: http://virusshare.com/

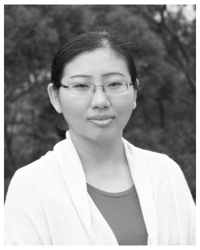
[18] (Jan. 2014). *VX Heaven*. [Online]. Available: http://vxheaven.org/

[19] *VirusTotal-Free Online Virus, Malware and URL Scanner*. [Online]. Available: https://www.virustotal.com

[20] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The WEKA data mining software: An update," *ACM SIGKDD Explorations Newslett.*, vol. 11, no. 1, pp. 10–18, 2009.
[21] S. B. Mehdi, A. K. Tanwani, and M. Farooq, "IMAD: In-execution malware analysis and detection," in *Proc. 11th Annu. Conf. GECCO*, 2009, pp. 1553–1560.
[22] A. W. Savich, M. Moussa, and S. Areibi, "The impact of arithmetic representation on implementing MLP-bp on FPGAs: A study," *IEEE Trans. Neural Netw.*, vol. 18, no. 1, pp. 240–252, Jan. 2007.
[23] M. Rahmatian, H. Kooti, I. G. Harris, and E. Bozorgzadeh, "Hardware-assisted detection of malicious software in embedded systems," *IEEE Embedded Syst. Lett.*, vol. 4, no. 4, pp. 94–97, Dec. 2012.
[24] L. Bossuet, G. Gogniat, and W. Burleson, "Dynamically configurable security for SRAM FPGA bitstreams," *Int. J. Embedded Syst.*, vol. 2, nos. 1–2, pp. 73–85, 2006.
[25] A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection," in *Proc. 23rd ACSAC*, Dec. 2007, pp. 421–430.
[26] G. Creech and J. Hu, "A semantic approach to host-based intrusion detection systems using contiguousand discontiguous system call patterns," *IEEE Trans. Comput.*, vol. 63, no. 4, pp. 807–819, Apr. 2014.
[27] A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda, "AccessMiner: Using system-centric models for malware protection," in *Proc. 17th ACM Conf. CCS*, 2010, pp. 399–412.
[28] Y. Xue, J. Wang, Y. Liu, H. Xiao, J. Sun, and M. Chandramohan, "Detection and classification of malicious javaScript via attack behavior modelling," in *Proc. ISSTA*, 2015, pp. 48–59.
[29] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: Malware analysis via hardware virtualization extensions," in *Proc. CCS*, 2008, pp. 51–62.
[30] H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck, "Structural detection of android malware using embedded call graphs," in *Proc. ACM Workshop AISec*, 2013, pp. 45–54.
[31] D. Bilar, "Opcodes as predictor for malware," *Int. J. Electron. Secur. Digit. Forensics*, vol. 1, no. 2, pp. 156–168, 2007.
[32] I. Santos *et al.*, "Idea: Opcode-sequence-based malware detection," in *Engineering Secure Software and Systems*. Berlin, Germany: Springer-Verlag, 2010, pp. 35–43.
[33] I. Santos, F. Brezo, X. Ugarte-Pedrero, and P. G. Bringas, "Opcode sequences as representation of executables for data-mining-based unknown malware detection," *Inf. Sci.*, vol. 231, pp. 64–82, May 2013.
[34] F. Maggi, M. Matteucci, and S. Zanero, "Detecting intrusions through system call sequence and argument analysis," *IEEE Trans. Dependable Secure Comput.*, vol. 7, no. 4, pp. 381–395, Oct./Dec. 2010.
[35] N. Runwal, R. M. Low, and M. Stamp, "Opcode graph similarity and metamorphic detection," *J. Comput. Virol.*, vol. 8, nos. 1–2, pp. 37–52, 2012.
[36] J. Demme *et al.*, "On the feasibility of online malware detection with performance counters," in *Proc. 40th Annu. ISCA*, 2013, pp. 559–570.
[37] A. Tang, S. Sethumadhavan, and S. J. Stolfo, "Unsupervised anomaly-based malware detection using hardware features," in *Proc. 17th Int. Symp. RAID*, 2014, pp. 109–129.
[38] M. Ozsoy, C. Donovick, I. Gorelik, N. Abu-Ghazaleh, and D. Ponomarev, "Malware-aware processors: A framework for efficient online malware detection," in *Proc. IEEE 21st Int. Symp. HPCA*, Feb. 2015, pp. 651–661.
[39] M. B. Bahador, M. Abadi, and A. Tajoddin, "HPCMalHunter: Behavioral malware detection using hardware performance counters and singular value decomposition," in *Proc. 4th ICCKE*, 2014, pp. 703–708.
[40] D. Arora, S. Ravi, A. Raghunathan, and N. K. Jha, "Architectural enhancements for secure embedded processing," *NATO Secur. Through Sci. D, Inf. Commun. Secur.*, vol. 2, p. 18, 2006.
[41] A. M. Fiskiran and R. B. Lee, "Runtime execution monitoring (REM) to detect and prevent malicious code execution," in *Proc. IEEE ICCD*, Oct. 2004, pp. 452–457.
[42] A. K. Kanuparthi, R. Karri, G. Ormazabal, and S. K. Addepalli, "A high-performance, low-overhead microarchitecture for secure program execution," in *Proc. IEEE 30th ICCD*, Sep./Oct. 2012, pp. 102–107.
[43] D. L. C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, "Architectural support for copy and tamper resistant software," in *Proc. 9th Int. Conf. ASPLOS*, 2000, pp. 168–177.
[44] S. Andersen and V. Abella. (2004). *Data Execution Prevention: Changes to Functionality in Microsoft Windows XP Service Pack 2, Part 3: Memory Protection Technologies*. [Online]. Available: http://technet.microsoft.com/en-us/library/bb457155.aspx
[45] PaX Team. (2003). *PaX Non-Executable Pages Design & Implementation*. [Online]. Available: http://pax.grsecurity.net/docs/noexec.txt
[46] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proc. 14th ACM CCS*, 2007, pp. 552–561.
[47] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: A new class of code-reuse attack," in *Proc. 6th ACM ASIACCS*, 2011, pp. 30–40.
[48] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Transparent ROP exploit mitigation using indirect branch tracing," in *Proc. 22nd USENIX Conf. Secur.*, 2013, pp. 447–462.
[49] Y. Xia, Y. Liu, H. Chen, and B. Zang, "CFIMon: Detecting violation of control flow integrity using performance counters," in *Proc. 42nd Annu. IEEE/IFIP Int. Conf. DSN*, Jun. 2012, pp. 1–12.
[50] S. Das, W. Zhang, and Y. Liu, "Reconfigurable dynamic trusted platform module for control flow checking," in *Proc. ISVLSI*, Jul. 2014, pp. 166–171.
[51] L. Davi, P. Koeberl, and A.-R. Sadeghi, "Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation," in *Proc. 51st ACM/EDAC/IEEE DAC*, Jun. 2014, pp. 1–6.
[52] A. K. Kanuparthi, R. Karri, G. Ormazabal, and S. K. Addepalli, "A survey of microarchitecture support for embedded processor security," in *Proc. ISVLSI*, Aug. 2012, pp. 368–373.

Sanjeev Das received the B.Tech. degree in electronics from the National Institute of Technology, Surat, India, in 2010. He was a Software Engineer with IBM, India, from 2010 to 2012. He is currently pursuing the Ph.D. degree with Nanyang Technological University, Singapore. His research interests include embedded systems security, hardware-enhanced architecture for software security, control flow integrity, memory attacks, and malware analysis.

Yang Liu received the bachelor's degree in computing and the Ph.D. degree from the National University of Singapore (NUS), in 2005 and 2010, respectively. He continued with his postdoctoral work in NUS. Since 2012, he has been with Nanyang Technological University, as an Assistant Professor. His research focuses on software engineering, formal methods, and security. In particular, he specializes in software verification using model checking techniques. This work led to the development of a state-of-the art model checker, Process Analysis Toolkit.



Wei Zhang received the Ph.D. degree in electrical engineering from Princeton University with the Wu Prize for research excellence. She joined The Hong Kong University of Science and Technology, in 2013, and established the Reconfigurable System Laboratory. She was an Assistant Professor with the School of Computer Engineering, Nanyang Technological University, Singapore, from 2010 to 2013. Her research interests include reconfigurable system, FPGA-based design, low-power high-performance multicore system, electronic design automation, embedded system, and emerging technologies. She currently serves as the Associate Editor of the IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION SYSTEMS, and the Area Editor of Reconfigurable Computing for *ACM Transactions on Embedded Computing Systems*. She has authored over 60 technical papers in refereed international journals and conferences, and authored three book chapters.

Mahintham Chandramohan received the B.Eng. degree from Nanyang Technological University, Singapore, where he is currently pursuing the Ph.D. degree with the School of Electrical and Electronic Engineering. His research interests include malware analysis and vulnerability detection at machine code level and machine learning with applications in software security.