IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received November 2, 2020, accepted November 21, 2020, date of publication November 30, 2020, date of current version December 30, 2020.

*Digital Object Identifier 10.1109/ACCESS.2020.3041302*

# SBGen: A Framework to Efficiently Supply Runtime Information for a Learning-Based HIDS for Multiple Virtual Machines

**JIWON SEO[1], INYOUNG BANG[1], JUNSEUNG YOU[1], YEONGPIL CHO[2], AND YUNHEUNG PAEK[1], (Member, IEEE)**
[1]Department of Electrical and Computer Engineering and ISRC, Seoul National University, Seoul 08826, South Korea
[2]Department of Computer Science, Hanyang University, Seoul 04763, South Korea

Corresponding authors: Yeongpil Cho (ypcho@hanyang.ac.kr) and Yunheung Paek (ypaek@snu.ac.kr)

This work was supported in part by the Institute of Information and communications Technology Planning and Evaluation (IITP) Grant funded by the Korea Government (MSIT) (No.2018-0-00230, Development on Autonomous Trust Enhancement Technology of IoT Device and Study on Adaptive IoT Security Open Architecture based on Global Standardization [TrusThingz Project]); in part by the Institute for Information and communications Technology Promotion(IITP) Grant funded by the Korea Government (MSIT) (No.2017-0-00213, Development of Cyber Self Mutation Technologies for Proactive Cyber Defense) and the Ministry of Science and ICT (MSIT), South Korea, under the Information Technology Research Center (ITRC) support program (IITP-2020-2020-0-01602) supervised by IITP and the BK21 Plus program of the Creative Research Engineer Development for IT, Seoul National University, in 2020; and in part by the National Research Foundation of Korea(NRF) Grant funded by the Korea Government (MSIT) (NRF-2020R1A2B5B03095204, NRF-2018R1D1A1B07049870) and the research fund of Hanyang University (HY-2020).

**ABSTRACT** Much compelling evidence urges that the isolation provided by the hypervisor in a virtualized system is not complete at all, and in practice can be neutralized by elaborated adversaries, which consequently emphasizes the need of techniques to detect attacks on the guest VM kernels. In this regard, learning-based HIDSs have received much attention, which inspect the internals of each VM through monitoring models built by machine learning techniques. The inspection capability of learning-based HIDSs depends on the quality of the monitoring models, which in turn can be improved by using rich runtime information reflecting the exact behavior of VMs. However, as extracting such runtime behavior information is onerous on account of its vast quantity, many learning-based HIDSs have resorted to using only fragmentary runtime behavior information. To address this problem, in this paper, we present *SBGen*, a framework for efficient extraction of rich runtime behavior information of VMs, namely the system call traces and the execution paths of the kernel taken to serve system calls. To trace execution of the kernel efficiently, SBGen leverages a salient hardware feature, Intel Processor Trace (PT). Once receiving the execution of the kernel traces from PT, SBGen elaborately decodes and purifies them to extract execution paths of the kernel associated with system calls. The extracted runtime behavior information of VMs is fed into learning-based HIDSs to improve their detection accuracy. Our experiments show that SBGen can extract and supply runtime behavior information efficiently enough for learning-based HIDSs to detect in a timely fashion real-world attacks on the guest VM kernels running in a virtualized system, while incurring a reasonable amount of performance overhead.

**INDEX TERMS** Intel Processor Trace (PT), learning-based HIDS, VM monitoring, extraction of runtime behavior information, guest VM kernel execution traces.

## I. INTRODUCTION

Virtualization allows multiple users to share a single physical machine by installing and executing their own guest virtual machines (VMs) on the machine. Since being introduced, therefore, virtualization has been widely adopted as an enabling technique to establish and manage with high efficiency and low cost modern computing infrastructures, particularly for cloud computing. In such a virtualized system with multiple tenants, *HIDS (Host-based Intrusion Detection System)* is being considered as a promising security approach to protect individual VMs and thereby the entire virtualized

The associate editor coordinating the review of this manuscript and approving it for publication was Chi-Yuan Chen.

system. The HIDS inspects the internals of each VM to detect signs of intrusions into VMs. Depending on the methodology of a VM inspection, the HIDS is divided into *rule-based* and *learning-based*. The rule-based HIDS [1]–[3] relies on monitoring *rules* manually defined by the system developers with expert knowledge on VMs, and the learning-based HIDS [4] leverages a monitoring *model* automatically built by machine learning (ML) techniques. Of these two approaches, the learning-based one has been preferred recently, in particular in terms of low reliance on human effort. Unlike the rule-based one, which requires significant human effort to define monitoring rules, the learning-based one can learn its monitoring model from data with minimal human intervention, and thus has the advantage of being built without being confined to human intuition or expertise.

In running a learning-based HIDS, extracting a runtime behavior of VMs is extremely important because they are used as materials to generate and run a monitoring model, which are directly linked to the inspection capability of the HIDS. Unfortunately, extracting runtime behavior consisting of function calls, system calls, branches, etc. is onerous on account of their vast quantity, which becomes even larger in a system where multiple VMs are running. For this reason, many learning-based HIDSs have resorted to using only fragmentary runtime behaviors such as system call traces to perform monitoring. However, this may be somewhat undesirable, given the intrinsic complexity of running systems whose abnormality cannot be determined through only a fragmentary runtime behavior. Along this notion, we have found in our preliminary experiments, motivating examples, which will be discussed in more detail in Section II, where it is more desirable for a learning-based HIDS to adopt a model with a *multimodal* approach that can associate various inputs together in order to improve its monitoring capability.

As such, to fulfill such improvements, a learning-based HIDS needs to be able to efficiently extract multiple correlated runtime behaviors from VMs, which motivated us to design a framework for efficient runtime behavior extraction, *SBGen*. As most studies on learning-based HIDS focus on modeling and discerning software execution flow [5]–[8] such as system calls or branch sequences, SBGen focuses on providing correlated execution flow information. Specifically, SBGen aims to extract from each VM, *system call traces* (abbreviated as syscall traces), which are the sequences of system calls invoked by a process, and their corresponding *execution paths*, which is a sequence of branches that the kernel takes to serve each system call. Recent HIDS tends to separately examine the execution behavior of each process and therefore, SBGen is built to extract the syscall traces and their corresponding execution paths separately for each unique process. In other words, SBGen differentiates the syscall traces and execution paths of each process and each VM. Through this, SBGen ultimately enables a learning-based HIDS to inspect VMs with high accuracy and low overhead by providing the correlated data of extracted syscall traces and the entailed execution paths as learning material.

To achieve its goal efficiently, SBGen leverages the Intel *Processor Trace* (PT) [9], a hardware feature that the latest (5th and 6th) generations of Intel x86/64 architectures began to support. To be concrete, SBGen uses the Intel PT to efficiently capture execution traces of the system and subsequently, from them extracts the syscall traces and the involved execution paths. To achieve its goal, SBGen faces the following challenges. First, it should be noted that the PT captures the execution traces in the form of a sequence of branches taken during execution, which are low-level data that does not clearly indicate which system calls are invoked. Second, to make matters worse, the execution of each system call can be interrupted or resumed at anytime by the preemptive nature of the kernel; therefore the execution paths of distinct syscalls will be fragmented and mixed with those of other syscalls in the PT-captured execution traces. Besides, this problem can be exacerbated because in a virtualized system being targeted by SBGen, a number of processes created by multiple VMs will execute syscalls simultaneously. To tackle these challenges, SBGen purifies the execution traces captured by the PT. More specifically, SBGen employs a technique of inserting several indicators for the beginning and end of a syscall, process context switching, VM context switching into the execution traces while being captured by the PT, and analyzes the execution traces to extract syscall traces and the involved execution paths capitalizing on the inserted indicators.

In order to evaluate our work, we implement SBGen and incorporate it to our learning-based HIDS, called HIDS-SBGen, which is a renovated version of a conventional learning-based HIDS [10] to employ advanced input features from SBGen. In this setup, SBGen incurs 11.67 % overhead on processes running in VMs and can deliver data to the HIDS within 467 us. However as we found that an average of 45.63 syscalls and 162.873 M branches occur every second, significant processing resource is needed for a HIDS to process this data in real time. This could make SBGen scale poorly for usage in real virtual environments such as cloud systems. Therefore we also built and evaluated SBGen-lite, a relaxed variant version of SBGen. We observed that arguments passed into syscall exerts significant influence on the initial part of the execution path, and thus monitoring the initial part is typically worth more. This observation has also been supported in the literature [11], which argues that execution space can be partitioned into the front small exploitable and the remaining post-exploitable parts due to the attribute of exploit payloads that should be delicately designed based on the strict hypothesis regarding the execution environment. Upon this observation, SBGen-lite extracts and delivers to the HIDS, syscall traces and the first 30 branches of their corresponding execution paths. SBGen-lite incurs 1.78 % overhead on processes running in VMs and can deliver data to the HIDS within 189 us. With the reduction in the amount of delivered data, the computing resource needed to process the data is significantly relaxed and a single GPU could support an HIDS for 4 CPU cores running VMs in real time. And though theoretically the loss of data in SBGen-lite compared to SBGen, i.e., the execution path after its 30th branch, could result in the HIDS being unable to catch malicious behavior in that execution path, we have found that for the attacks used in our evaluation, the HDIS supported by SBGen-lite showed no degradation in its capability of discerning the attacks from normal behavior.

Our contributions in this paper can be summarized as follow:

- We present SBGen, a framework for learning-based HIDS on a virtualized system with multiple VMs.
- We devise a VM-aware mechanism to efficiently extract execution traces by VM using Intel PT.
- We demonstrate the effectiveness of SBGen by developing HIDS-SBGen and performing evaluation against real security threats on VMs.

**TABLE 1. Trace packets in Intel PT.**

| Packet Name | Packet Description |
|---|---|
| Paging Information packet (PIP) | Packet Description |
| Flow Update packet (FUP) | Provides the source address for asynchronous events such as those triggering Interrupts |
| Target IP packet (TIP) | Records the target address of control transfers such as indirect jumps and indirect calls |
| TNT packet | Indicates the direction of direct conditional branches and represents the information about returns |
| Time Stamp Counter (TSC) | Provides timing information of packet generation |

## II. BACKGROUND AND MOTIVATION

### A. INTEL PT

Intel PT is a hardware feature that is supported by the latest x86/64 architectures to facilitate efficient capturing and recording of execution flow traces generated by CPU. Execution traces are formed by sequentially concatenating data packets, each encoded with the information about dynamic control flow changes, such as branch targets and branch taken indications. Table 1 describes several types of the data packets constituting an execution trace provided by PT. PIP and FUP are generated by CPU, respectively when the CR3 register is updated and when asynchronous events like interrupts occur. These packets appear relatively rarely in a trace, but are of importance in a sense that they can be used to dissect the sequential trace into multiple execution units (i.e., processes) each with its own context. TIP and TNT take up most of packets in the captured trace, each of which denotes every one of the changes in control flows of a process. TIP records the target of an indirect branch (e.g., register indirect jmp/call and ret) and TNT indicates whether each conditional branch is taken or non-taken (e.g., jz, je, loop, etc). Lastly, TSCs are periodically issued by PT to provide precise timing information about when packets are generated.

For better space and time efficiency, Intel PT applies special optimizations to packet generation. One such optimization aims to minimize the size of packets by logging TIP and TNT in a bit-level compression format. Another is to minimize the number of generated packets by leaving out those for certain types of branches (e.g., direct jmp/call) which has a statically specified target in the program. To enable efficient analysis of data packets, Intel PT offers the *selective tracing* mechanism which is available to selectively enable or disable the packet generation by means of three different hardware features: instruction pointer (IP), CR3 or current privilege level (CPL). First of all, Intel PT can be configured to generate packets depending on whether or not the IP stays within the specified range. Similarly, PT can be configured to enable the packet generation only when CR3 and CPL are equal to specific values, respectively. As every process is assigned a unique CR3 value to use its own page table, the PT trace is enabled only for a process with the specific CR3 value. The PT trace can also be confined to generate packets for either or both of two levels (CPL=0 or 3) of execution where CPL=0 and 3 indicate kernel-level and user-level, respectively. As will be discussed later, the selective tracing mechanism based on CPL has been greatly helpful in our data packet analysis for SBGen. When packets are generated, PT stores them directly into the host machine memory in different ways as directed by its modes: single range (SR) and table of physical addresses (ToPA). In the SR mode, the buffer to store packets is allocated to a physically contiguous region. Contrarily in the ToPA mode, the buffer management is more flexible in that the buffer can be allocated to several non-adjacent regions. In both modes, the buffer is usually used in a circular fashion to improve memory efficiency.

### B. LSTM NETWORK

RNNs are artificial neural networks that are designed to operate in a recurrent manner so that its operation on input $x_t$ is affected by prior inputs $x_1$ through $x_{t-1}$. This is typically accomplished by recurrently using output of the prior iteration $y_{t-1}$ (from processing the prior input $x_{t-1}$) when calculating output of the current iteration $y_t$ from input $x_t$. Long Short Term Memory (LSTM) is a type of RNN which uses, in addition to the prior output $y_{t-1}$, a special memory block to perform such recurrent operations. Through the use of this memory block, LSTM networks can maintain information over long distances (relative to typical RNNs) between inputs which gives it the capability to correlate inputs over large gaps. The LSTM memory block contains a memory cell storing $c_t$, a context vector representing information of prior inputs, as well as three gates regulating the data flow into and out of the memory cell. The input gate $i$ controls how much the current input $x_t$ and prior output $y_{t-1}$ would affect the calculation for $c_t$. The forget gate $f$ decides how much of the prior information represented in $c_{t-1}$ should be kept for the current iteration. From the outputs of these two gates, $c_t$ is calculated. The output gate $o$ controls how the new values stored in $c_t$ should be represented in output of the current iteration $y_t$. And from the output of $o$ and the values in $c$, the current output $y_n$ is calculated. The explicit operations of each gate, which can be found in [12], is omitted here for the sake of brevity.

### *C. ANOMALY DETECTION*

HIDS is a security framework of monitoring the events occurring in the target system and analyzing them for signs of violations or imminent threats to computer security policies. HIDS assume intruder behavior differs from legitimate users and determines an intrusion by generating a profile (or model) of a system's behavior. IDS can be classified by what kind of profile/model it employs for detection: *misuse detection* system and *anomaly detection system*. Anomaly detection creates models for legitimate behavior while misuse detection creates them for malicious behavior. Since misuse detection only alerts known behavior of intruders, recent researchers prefer anomaly detection to misuse detection [13]–[16]. However, though anomaly detection has the potential to detect unknown attacks unlike misuse detection, its detection capability in principle depends on the accuracy of the model it uses. The model of poor quality causes false alarms with a high rate; thus, researchers have tried to improve its quality by applying machine learning [14]–[16].

### *D. MOTIVATION*

As we have briefly mentioned in Section I, most current work on learning-based HIDS focus on only single aspects of runtime behavior such as system call sequences [6]–[8] or branch sequences [10], [17]. Unfortunately, it was shown that adversaries could evade HIDS examining system call sequences [13], [18] with mimicry attacks [19], [20] in which adversaries manipulate the execution of malware or attacks so that their runtime system call sequences appear to be similar to those of benign executions. Recent work [8], [21], [22] argued that, by examining runtime branch sequences, a HIDS could become resilient to mimicry attacks as the branch sequences of manipulated system calls often differ from those of benign system calls and also that branch sequences are much harder for an adversary to mimic while performing malicious activity. Though we agree that branch sequences offer an innate resilience to mimicry attempts, in our preliminary experiments, we have found that a HIDS examining branch sequences alone was not able to detect anomalies that were detectable with system call sequences. This was due to the fact that, when examining branch sequences, the HIDS is flooded with branch information, it was nearly impossible to relate dependency between distant runtime events. On the other hand, as hundreds of branches occur to service a single system call, even runtime events that are thousands of branches apart are only a few system calls apart from each other, which makes it much more likely for a learning-based HIDS to relate the events within a sequence. From this experience, we believe that a HIDS would benefit from relating high level behavior such as system call sequences to their corresponding low level behavior such as branch sequences. The low level behavior would provide resilience against mimicry attempts while the high level behavior would provide a way to relate distant runtime events. We have tested this correlation on a few toy example codes and achieved promising results, however, the collection and processing of system calls and branch events became too time consuming for testing on more real world software. This motivated us to develop SBGen to provide an efficient way to collect and correlate high level runtime behavior alongside its corresponding low level runtime behavior.

## III. RELATED WORK

### *A. INTEL PT USES IN OTHER SECURITY PROBLEMS*

A series of studies [23]–[26] have employed PT for control flow integrity (CFI) solutions that detect control flow hijacking attacks that subvert the control flow of a process. Their solutions use PT to efficiently chase after changes in control flow of the target process in order to ensure that any changes do not deviate from the legitimate control flow defined by the process's control flow graph. Other studies [27] have utilized PT to implement fuzzing solutions for efficient kernel testing. Their solutions obtain from PT a control flow trace of the target kernel which is in turn used as feedback to maximize code coverage in fuzzing. As in our SBGen, both CFI and fuzzing solutions employ Intel PT to gain the advantage of enabling transparent and efficient execution tracing. However, there is a clear difference between the way they and we operate on the outputs of PT. They basically use trace packets in almost the same form as they are decoded from raw compressed data. Contrarily in SBGen, after being decoded, the packets are analyzed elaborately and processed heavily through cascade steps: packet separation, packet filtering and execution path purification. A primary reason for the difference is that they usually need to only track control flows of a specific victim application while we must have the capability of tracking many different kernel execution flows concurrently generated by multiple guest VMs running on a multicore system.

### *B. TRADITIONAL SYSTEM CALL INTROSPECTION SOLUTIONS*

Research on system call introspection (SCI) has been carried out to detect anomalous invocations of system calls to detect attacks originating from the software with user-level privilege. One (or, the most) common approach in this line of research on system call introspection is to check system call sequences made by user applications to interact with their kernels. The rationale behind this approach is two fold. Firstly, for user-level software to leave a lasting effect on the kernel, it must invoke system calls to access and manipulate the system resources under control of the kernel. Secondly, the conventions or practices of making system calls for such a lasting effect tend to differ for malicious and benign software. The work in [13] is, to the best of our knowledge, the most prominent one that learns from system call traces for anomaly detection. They propose that a context-free grammar could be applied to system call traces and generate a dictionary of phrases, contiguous patterns of system calls, from given system call traces and train their model with it. Though their model shows fairly high accuracy as discussed in section I, their strategy that examines only system call traces render their introspection vulnerable to mimicry attacks [19], which as the matter of fact, were born to defeat such previous SCI solutions that focus on system call traces. For instance, a mimicry attack sequence is handcrafted by transforming an existing attack to emit a system call sequence that would be viewed as a normal sequence by SCI solutions. The transformation relies on dummy system calls (e.g., mkdir() invoked with an invalid pointer) which can be made in a way that they would not compromise their original malicious effects on the kernel, and yet disguise their system call footprints of malicious code. This mimicry technique would nullify SCI solutions that are interested only in the invocation pattern of system calls, subsequently demanding researchers to leverage the *contextual* data better characterizing system call sequences in order to defeat attackers by differentiating dummy system calls from genuine ones used in the attack sequence.

As mentioned in Section I, many SCI solutions have adopted system call input arguments for such contextual data because the arguments are believed to well characterize system calls by providing individual execution contexts for kernel routines serving the system calls. In [22], they used a rule-learning system to combine rules for system call arguments and sequences. Though their solution is reported to improve accuracy, it runs up to 10x slower when incorporating system call arguments. In [21], the authors have also used system call arguments to detect mimicry attacks. However, they additionally require hand-crafted specifications of the system to lower false positives in its detection results. All in all, these attempts to use system call arguments have evinced a promise in increasing SCI accuracy especially in the detection of mimicry attacks. But, the requirement is here that the arguments should be parsed into features that could be used by their SCI systems. Unfortunately, to meet this requirement, a developer must have hands-on experience and deep knowledge of real attacks and their invocation pattern of system calls when they provide the rules for parsing. In such a situation, SBGen is designed to glean an execution context of the kernel code serving system calls from kernel execution paths, which eliminates the need to use expensive algorithms for complex data analysis as well as to rely on strenuous human intervention.

### C. VIRTUAL MACHINE INTROSPECTION

With the recent increase in virtualization, virtual machine introspection (VMI) has been an active area of research, emerging as a feasible method for monitoring the runtime state of a guest OS. Most research on VMI [28]–[30] provides an inspector with a capability of intercepting and forwarding guest VM kernel events like system calls to the hypervisor layer where the inspector resides and examines the kernel behavior. The work in [28] could be related to our SBGen in that they interpret VM-internal system call events at the hypervisor layer. They propose a VMI mechanism that collects system calls invoked by all processes running inside a honeypot. To check the honeypot VM internal events from the outside, they inspect each system call by examining its parameters and semantics. Though this solution provides a fine-grained monitoring tool for a virtualization platform, it is difficult to apply in a large-scale virtualization environment, as this monitoring function may only run in an individual target VM. Furthermore, since this solution is only used for later investigation about trace analysis, it cannot detect attacks invoking anomalous system call at the same time the associated guest VM is up and running. We believe that SBGen can be used to relieve these issues by facilitating runtime introspection of system calls from multiple VMs.

## IV. DESIGN

SBGen is a framework to extract runtime behavior of VMs expressed as syscall traces that record invoked system calls in sequence and execution paths that enumerate the taken branches of the VM kernel to serve each system call. SBGen is ultimately aimed to supply the extracted behavior to learning-based HIDSs, enabling better accuracy and efficient monitoring for VMs. In this section, we present a brief overview of SBGen, and then give a detailed description of its operations.

### A. OVERVIEW

Learning-based HIDSs have considered monitoring system calls as an effective approach to stymie attacks on a kernel. The reason is that system calls are the only pathways to reach the kernel from user space, and thus virtually all known kernel attacks could be thwarted effectively if blocking malicious attempts to find and abuse the weakness in such narrow pathways. In light of this, SBGen provides rich information relevant to system calls, i.e., syscall traces and the corresponding execution paths with the learning-based HIDSs to allow them to perform system call monitoring on guest VM kernels in the fight against adversaries who try to penetrate into the kernels via system calls. Figure 1 illustrates the design of SBGen. In our implementation, SBGen runs on a dedicated CPU core, separately from the other cores executing guest VMs, so that they can work concurrently with the VMs. For being fortified against attacks from guest VMs, SBGen runs in the highest privilege mode, namely the *VMX root* (or more conventionally, hypervisor) mode.

SBGen is responsible for obtaining syscall traces and the execution paths that will be used as the input features for learning-based HIDSs. Because SBGen persistently works as long as guest VMs are active, its extraction task must work with low overhead in order to minimize the impact on system performance. To this end, SBGen leverages Intel PT, which is the hardware support for low-cost execution trace. Once receiving trace packets from PT, it converts the raw data in the packets into appropriate formats for use by learning-based HIDSs. For the conversion, the raw data

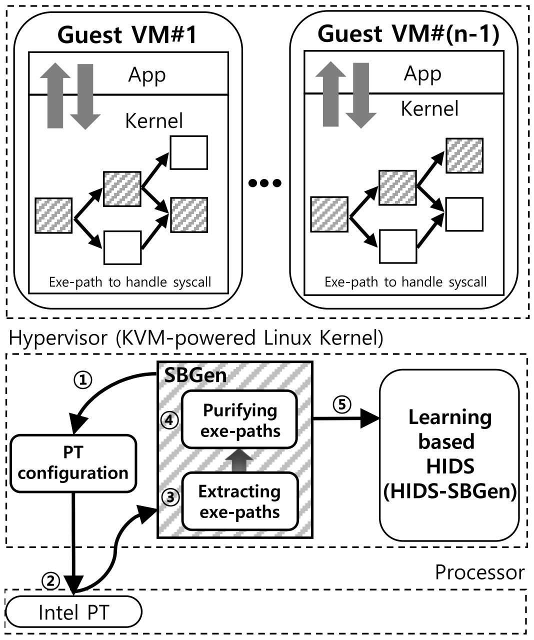

**FIGURE 1. The overall architecture of SBGen.**

originally compressed in a packet must be decoded and represented as an execution path in the form of a sequence of branches. The execution path in the original form is basically a mixture of kernel branch instructions generated by various system events, such as multiple system calls, interrupts, and kernel threads. SBGen, therefore, purifies the execution path to contain only the branches executed in kernel code for each system calls, by getting rid of those executed for other types of events. Now, the resulting execution paths solely reflect the execution context for the kernel routines that handle system calls, such that SBGen can finally organize them in order by system call. Finally, SBGen can offer the extracted syscall traces and execution paths as such to learning-based HIDSs for use in offline model training and online inferencing to detect the existence of anomalies in a series of system calls invoked by guest VMs at runtime. In the following, we will give a more detailed description about the operations of SBGen.

## B. FEATURE EXTRACTION

Every time a system call is invoked by an application, SBGen generates as a training feature for learning-based HIDSs the execution path in kernel code of the target VM made to handle the call. As mentioned above, for minimizing performance degradation, the extraction task must work as efficiently as possible. But not only that, the task must perform with high accuracy as the extracted features are to determine the accuracy of the output results of learning-based HIDSs.

### 1) EXTRACTING EXECUTION PATHS VIA INTEL PT

The greatest benefit of using Intel PT is that it allows SBGen to obtain execution paths efficiently with little interruption to the target guest VM kernels. As explained in subsection II-A, PT records execution traces in various types of packets. Note that guest VMs must be deprived of the control authority over PT because otherwise, they may manipulate several configurable features of PT, particularly selective tracing, to impede the effort of SBGen to acquire proper trace packets. Fortunately, the Intel architecture ensures that guest VMs are forced to be out of control over PT when virtualization is enabled [9].

#### a: PACKET SEPARATION BY VM

As SBGen assumes a virtualized system where multiple guest VMs run simultaneously, it receives trace packets of every guest VM randomly intermingled in the order they are generated from the CPU regardless of their origin VMs. It is noteworthy that each PT packet for one guest VM stores a partial sequence of branches executed in the VM. Thus in our view, a full sequence of such packets with the same origin VM forms an entire execution behavior of the VM woven with the executed branches. Following our view, we have designed SBGen to keep track of packet sequences separately for each and every VM. To fulfill its task, when SBGen receives a new PT packet for one VM, it must be able to incrementally construct a packet sequence exclusively for the VM by appending this packet to an existing sequence of earlier packets for the same VM. To facilitate this individual sequence construction for every VM, we utilize the fact that each VM is allocated a group of `vcpus`, virtual CPUs which will be mapped respectively to physical CPUs by the hypervisor in a virtualized system. As every VM instruction is conceptually executed by vcpus, a sequence of PT packets for one VM will be generated and emitted by a specific vcpu that executes the VM. In our work, we build up a packet sequence dedicated for each vcpu by preparing a trace buffer per vcpu when a guest VM is created and allocated a vcpu (or vcpus if set to be a multicore machine). During VM execution, VM-enter functions to transfer control from the host (i.e., hypervisor) to its guest running on a vcpu. By utilizing this function, whenever a VM-enter occurs in a vcpu, SBGen configures Intel PT to store the succeeding packets emitted by the vcpu into the corresponding trace buffer. As a result, by reading each individual trace buffer, SBGen is able to trace packets in order respectively for all VMs currently running in the system. In order to reduce the likelihood of the overflow of the trace buffers and consequent packet losses, we implement the trace buffers as ring buffers to increase their space utilization. In addition, we empirically allocate the trace buffers large enough to store the packets that are generated during the specific time slice given for a VM by a scheduler. The multi-threaded decoding mechanism that will be described soon virtually empties the trace buffers at every VM-exit by copying the stacked contents to decoder buffers.

#### b: PACKET FILTERING

Recall that SBGen is interested just in the execution paths which the guest VM kernel code follows to serve system calls.

Therefore, to receive PT packets relevant only to kernel code execution, SBGen relies on the CPL-based selective tracing capability of Intel PT by enabling packet generation only when the vcpu runs in kernel mode (i.e., CPL 0). However in a virtualized system, there are two types of kernel that are the host and the guest, and CPL is set to 0 for both cases when the vcpu runs in either host or guest kernel mode. This means that even if PT is configured to trace selectively for CPL=0, SBGen would receive the PT packets generated by both the host and guest kernels. This is undesirable obviously because as mentioned above, SBGen needs only the execution path of the guest kernel. Fortunately, as introduced in subsection II-A, the special PT packet, PIP, conveys the *non-root* (NR) bit that indicates whether each packet originates from the host or guest VM kernel. In consequence, merely by checking the bit, SBGen can drop the packets from the host kernel, and only save those from the guest.

#### c: DECODING

The raw data in trace packets emitted by PT are compressed at bit level for saving space. Hence for substantive uses of the data, SBGen decodes the compressed data into actual execution paths in the form of branch sequences. To decode the packets in time and minimize the performance impact on guest VMs in the system, the decoding task is carried out by the dedicated thread on a separate CPU core. Every time VM-exit occurs, the packets for a guest kernel saved in the corresponding trace buffer are copied to the decode buffer and start to be processed.

### 2) PURIFYING EXECUTION PATHS

As been repeatedly revealed, the very information SBGen extracts from each VM is kernel execution paths taken to serve system calls, that is, sequences of kernel branch instructions executed during system call invocation, which will be basically the input features accepted by learning-based HIDSs for training and inference. By decoding trace packets from PT, we now hold kernel execution paths of guest VMs, each corresponding to a sequence of all branches executed in kernel code. In the end, these paths will be fed into the learning-based HIDSs to conduct anomaly detection on each guest VM. However, until we resolve the four design problems listed below, the learning-based HIDSs cannot accept as inputs the just-decoded paths per se. Thus, SBGen should take an additional step for preprocessing these decoded execution paths, namely, *purifying* the paths, to be acceptable by learning-based HIDSs. Ultimately, the paths are grouped by the processes/threads running in our target guest VMs, and each group turns into two acceptable forms of the monitoring model input for learning-based HIDSs that are (1) a sequence of system calls invoked by the corresponding process/thread and (2) the kernel branch sequences (i.e., execution contexts) which are executed to handle the corresponding system calls, respectively. If explained more, in the actual implementation, whenever a system call is discovered in the decoded paths, the associated execution paths are immediately assembled and sent to the learning-based HIDSs in pairs with the system call.

#### a: EXECUTION PATHS IRRELEVANT TO SYSTEM CALLS

A trace packet from PT contains not only execution paths derived from (thus, relevant to) system calls, but also other irrelevant ones because kernel instructions are executed for many different system events including system calls, interrupts, kernel threads and so on. Consequently, the decoded outputs of a PT packet will reflect diversified system events besides system calls. To tackle this problem, SBGen must filter such irrelevant ones out of the original decoded execution paths and leaves only those relevant to system calls.

#### b: PREEMPTIVE KERNEL EXECUTION

The second problem arises from the preemptive nature of the kernel. The kernel execution triggered by a system call is likely to be interrupted anytime by different types of events or even by other system calls. The interrupted execution would certainly resume later. But hereupon, we must notice that the original execution path for the system call is broken into two subpaths before and after the interruption, which are independently delivered, being apart from each other in a stream of PT packets from the CPU running the kernel. It is evident that in order for SBGen to achieve its goal of grasping the whole picture of execution behavior for the system call, the broken subpaths must be bonded together and become a single contiguous path which will be used as the input of the learning-based HIDSs. As such, weaving in an orderly fashion all broken pieces of an execution path derived from the same system call is another mission of SBGen.

#### c: PROCESS/THREAD-WISE SYSTEM CALL CLUSTERING

The third design problem is somewhat an extension of the second one because even if we successfully construct a complete, unbroken execution path for every system call, there still lies another requirement before us. To explain this, we first need to notice that from the perspective of learning-based HIDSs, a system call sequence for each thread/process is considered to be an independent history of its runtime behavior. Therefore when receiving system calls sequentially streaming from PT, the learning-based HIDSs are required to cluster all related system calls invoked by the same process/thread so that they can detect any malicious attempts of the process/thread on the kernel by analyzing the clustered calls, each coupled with the associated execution path. This requirement is boiled down to a problem of pinpointing the exact process/thread that invokes each system call. To unravel this problem, when a system call is loaded into the PT packet stream destined for our learning models, the *process/thread identifier* must be included along with the call in the stream. Now, by receiving each system call together with the identifier for its caller process/thread, our models are able to cluster all call invoked by the identical caller. For the process identifier, the problem is straightforward because, at every process context-switch, Intel PT automatically issues a PIP that carries the new CR3 value identifying the process just switched in. That is, we can solve the problem simply by delivering to the learning models the register value as the identifier for each system call. On the other hand, for the thread identifier, the problem demands a more elaborated solution since PT does not have any means to notify us straight of thread context-switch. Below we will discuss our solution to this problem in the design of our SBGen.

### *d: PROCESS/THREAD MIGRATION*

The last problem applies only when a guest VM runs on multiple CPU cores (i.e., multiple vcpus). In a multicore system, process migration commonly occurs mainly for load balancing between cores. As a process migrates over multiple cores during its life time, its execution would be divided and distributed across those cores. This implies that even if system calls are orderly invoked by the process, the total ordered sequence of them will not be built simply by concatenating all the partial sequences fetched from the trace buffers for different cores unless we do not know the relative order between these partial sequences. As a result, to restore the total order of all system calls invoked by the same caller process, SBGen somehow has to find a way for lining up all the partial sequences arriving (most probably, out of order) from different sources.

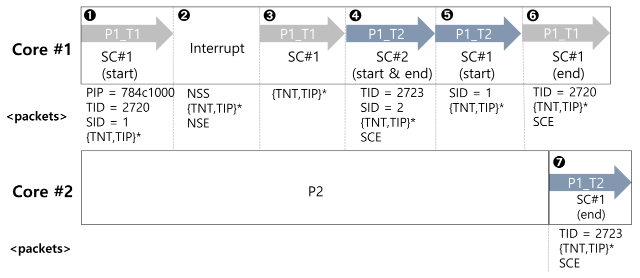

**FIGURE 2. An example of the execution of a dual-core guest VM running two processes `P1` and `P2`. `P1` has two threads `P1_T1` and `P1_T2`. Below each time epoch from ❶ to ❼, trace packets extracted by PT are listed in chronological order.**

Now, let us explain how we design SBGen to tackle the problems stated above. For this, consider Figure 2 where we present an example of the execution of a guest VM that runs two processes, `P1` and `P2`, in a dual-core virtual machine. We assume that `P1` has two threads, `P1_T1` and `P1_T2`. During process execution, the VM experiences several system events that divide the whole execution into seven time epochs, each denoted by the number ❶. Firstly on Core #1, `P1_T1` invokes the system call, `SC1`, which initiates kernel execution (❶). While the kernel is serving the system call, an interrupt is raised, which instantly suspends the current kernel execution and jumps the control to the associated interrupt handler (❷). As soon as the handler finishes its work, the suspended execution for `SC1` resumes. When the time slice of `P1_T1` is complete, it is halted in the middle of execution, and the control is handed over to `P1_T2` (❸).

During the next time epoch ❹, `P1_T2` continues to execute, making the system call `SC2`. As soon as the system call is returned, `P1_T2` makes another call `SC1` (❺). Suppose that the time slice for `P1_T2` is expired while the kernel code is still being executed to handle `SC1`. Then, the control is again back to `P1_T1`, thereby restarting the halted kernel execution for `SC1`. By the end of epoch ❻, the system call finishes. While `P1` is running on Core #1, assume that `P2` occupies Core #2. When the execution of `P2` is ended, `P1` occupies Core #2 and resumes the suspended execution of `SC1` for `P1_T2` (❼).

In this example, when it comes to `P1`, SBGen should be able to obtain three distinct execution paths respectively for `SC1` of `P1_T1` (❶-❸-❻), `SC2` of `P1_T2` (❹), and `SC1` of `P1_T2` (❺-❼). As can be seen, the path stretching over time epoch ❹ is the only one that is not disconnected by any system events. In contrast, the other kernel execution paths have been disturbed by an interrupt or thread context-switches, resultantly being divided into several partial paths. In this example, the mission of SBGen is first to connect together such divided execution paths and form two distinct contiguous paths, respectively relevant to two different invocations to the system call SC1. Then, it is to identify the individual process/thread making each invocation. If the mission is duly complete, we can obtain at once three execution paths (including path ❹) along with their relevant system call invocations and the caller processes/threads making the invocations. Recall that we at this moment have collected all essential input data needed by learning-based HIDSs to perform anomaly detection. Therefore by packing and shipping all these data to learning-based HIDSs, the mission of SBGen is basically complete.

**TABLE 2. Special packets defined by SBGen.**

| Packet Name | Packet Description |
|---|---|
| TID | Reports a thread identifier |
| SID | Reports a system call identifier |
| SCE | Reports a system call exit |
| NSS | Indicates the start point of the execution path not related to system call |
| NSE | Indicates the end point of the execution path not related to system call |

Let us look more closely at the example. In order to solve the first three design problems listed above, we use a technique that injects into a stream of PT packets the special packets delivering all necessary information to the PT recipients. In the current implementation, we provide five special packets for this purpose, as listed in Table 2. The packets, NSS and NSE, are used to distinguish system calls from other types of events. The pair of NSS and NSE marks the beginning and end points for the execution paths irrelevant to system calls so that they are useful to filter out those execution paths. For instance, by injecting NSS and NSE around time epoch ❷, we can eliminate the corresponding path created by an interrupt, thereby resulting in six execution paths relevant to system calls. The packets SID, SCE and TID are used to inform the packet recipient of two important system events, system call and thread context-switch, respectively. Since there are system call invocations at the beginning of time epochs ❶, ❹ and ❺, SIDs are injected at those points. Each SID is not only indicating the system call event, but also carrying the unique number which has been assigned a priori to identify individual system calls. In the example, SIDs for `SC1` and `SC2` are respectively 1 and 2, and so three SIDs 1, 2 and 1 are injected along with the paths ❶, ❹ and ❺, as displayed in Figure 2. Each SID is paired with a SCE that indicates the end of the execution for the associated system call event. In the above trace, two SCEs are injected at the end of time epochs ❹, ❻ and ❼. From Figure 2, we see the thread context-switches at time epochs ❶, ❹, ❻ and ❼. Accordingly in Figure 2, we see that TIDs are inserted three times, 2720, 2723 and 2720, where 2720 and 2723 are the identifiers of the two threads `P1_T1` and `P1_T2`, respectively. Notice here that we do not have a special packet for process context-switch because as said previously, we put the CR3 value from a PT-issued PIP to direct use as the process identifier. In our implementation, execution paths are clustered according to the values of SID, TID and PIP. For instance, the paths covering three epochs ❶-❸-❻ are clustered and arranged to form a contiguous execution path for system call 1 invoked by thread 2720, as shown in Figure 2. To build a contiguous path by clustering the partial paths covering two epochs ❺-❼, we have to solve the last design problem regarding the process/thread migration as the thread `P1_T2` migrates from Core#1 to Core#2 in the middle of its execution. In fact, clustering the two paths is trivial because both have the same SID, TID and PIP. However, arranging ❺ and ❼ in order to form a single full path ❺-❼ requires additional information since we do not know the relative time order of these two paths when they are retrieved respectively from two different trace buffers for Core#1 and Core#2. To supplement this missing information, we rely on a special packet, TSC, issued periodically by Intel PT to provide timing information, as described in section II. By examining TSCs issued around the times when these two partial paths are stored to the trace buffers, SBGen gets to know their relative order in time and arrange them accurately in the full path.

## C. COOPERATION WITH HIDS

SBGen aims ultimately to improve the accuracy of learning-based HIDSs by providing salient input features, namely syscall traces along with execution-paths, according to subsection II-D. For an empirical demonstration, we design a learning-based HIDS, HIDS-SBGen, that gets input features from SBGen and performs runtime anomaly detection for the system running multiple VMs. HIDS-SBGen is a renovated version of the conventional HIDS [10], DeePBM, which accepts only execution branch sequences extracted by Intel PT as input feature. We modified DeePBM to get input features processed by SBGen from multiple VMs. Therefore, HIDS-SBGen is able to analyze both syscall traces and their execution paths that are generated from SBGen. The Figure 3 describes the overall structure of HIDS-SBGen.

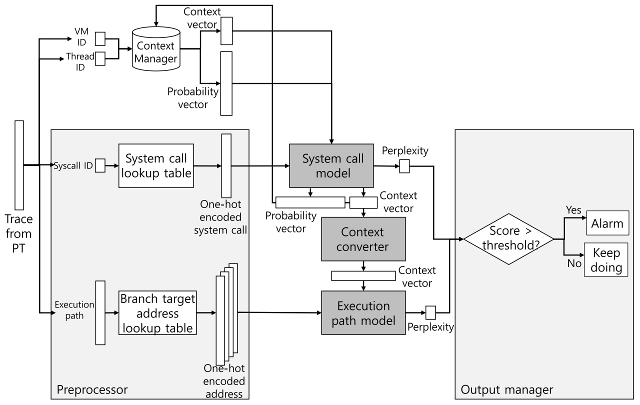

**FIGURE 3.** **The overview of HIDS-SBGen.**

The HIDS-SBGen sequentially receives from SBGen the input data that include system calls and their associated execution paths (as their execution contexts). Given the inputs, it employs two LSTM models: a system call model and an execution path model, which detect any wrongful actions of user applications against our target guest VM kernel by inferring anomaly in the way they interact with the kernel. To be specific, every time a system call is invoked in one of the guest VMs, HIDS-SBGen receives a pair of the invoked the system call and execution path alongside the process/thread identifier and the *VM identifier* that points to the actual VM handling the system call. The data in each pair are then transformed into one-hot encoded vectors by the preprocessor and delivered as input to the LSTM models. The VM and process/thread identifiers are used by the *context manager* to make our model examine the system call and execution path in relation to prior system call invocations from the same VM and process/thread. The LSTM models then consume the input vectors to calculate the degree of the anomaly of the current system call associated with its execution path. Finally, from this anomaly degree, the *output manager* makes a verdict on the malice of the process against the target VM, which is completed by comparing the computed perplexity of and the predefined threshold values of the LSTM models.

## V. IMPLEMENTATION

We have implemented our prototype of SBGen as a part of KVM, which is a hypervisor that virtualizes guest VMs within a host Linux system. In our prototype, we make the modifications required by SBGen to a Linux kernel and perform introspection with our model on guest VMs running the modified kernel. SBGen runs as a single process on the host kernel.

## A. SPECIAL PACKET GENERATION ROUTINES

The main objective of SBGen is to extract from PT packets the execution paths taken to serve each system call. Unfortunately, as the conventional PT packets do not contain enough information to accurately extract execution paths, SBGen devises a technique of injecting special informative packets, such as TID, SID, SCE, NSS, and NSE, into the stream of PT packets as mentioned in subsubsection IV-B2. We have implemented this technique by deploying several types of special packet generation routines into Linux kernel of the guest VM at source code level.

**Algorithm 1** Generating TID

```
1: function generate_TID (thread_id)
2:     indirect_jump_to(0xAAAA)        ⇒ generating TIP to indicate
3:     while each bit in thread_id        the beginning of the TNTs
4:         if bit is 1 then
5:             NOP                     ⇒ generating TNT:N (not taken)
6:         else
7:             NOP                     ⇒ generating TNT:T (taken)
8:         endif
9:     endwhile
10:    indirect_jump_to(0xBBBB)        ⇒ generating TIP to indicate
                                          the end of the TNTs
```

**FIGURE 4.** **Packet generation routine for TID.**

Figure 4 shows the pseudocode of the packet generation routine for TID. The routine is inserted at the thread context-switch routine of the kernel (e.g., `__switch_to()` in Linux). Most kernels already manage a unique number per thread for thread identification, such as `task_struct->pid` in Linux. Therefore, what we need to do is to load the unique value to a packet of Intel PT, but sadly, PT does not provide direct support for this function.[1] To overcome this limitation, the packet generation routine employs TNT that notifies whether a conditional branch was taken or not. The routine first loads a thread identifier and then iterates over the bit-length of the identifier while executing the conditional branch that is either taken or not depending on each bit of the identifier. As the result of this routine, we have the thread identifier encoded as a sequence of TNTs which will be transferred in the trace data of PT. Later, the packet recipient may be able to get the thread identifier by simply decoding the transferred TNT sequence. To accomplish this, however, the TNT sequence should be perceivable within the incoming stream of trace packets. For this purpose, the routine uses TIP that indicates the destination address of an indirect branch. Right before and after generating the TNT sequence, the routine executes two delimiting indirect branches that jump to predefined unique target addresses (0xAAAA and 0xBBBB in the example) and return immediately, thereby marking the beginning and end of the sequence. Later when the packet recipient perceives in the incoming packet stream a TIP with the first target address, it can be aware of the presence of a thread identifier encoded in a TNT sequence immediately following the TIP. It then will read the sequence until it hits another TIP carrying the second target address.

The packet generation routine for SID is located at the entry point of system call handlers. As the system call identifier has a limited range of value depending on the number of types of system calls, it is sufficient for the routine to only use TIP. The routine prepares as many predefined target addresses as the system call identifier can have, and then executes an indirect branch that jumps to one of the predefined target addresses according to the system call identifier, allowing SBGen to discern which system call is invoked through the TIP that will be generated. To generate SCE, NSS and NSE, similar but more simplified packet generation routines are used. The routine for SCE is inserted at the exit point of system call handlers and the routines for NSS and NSE are inserted at entry and exit points that are not related to system calls, such as of interrupts handlers and kernel threads. these routines simply generate TIPs with the predefined unique target addresses (e.g., 0xCCCC and 0xDDDD). As a result, SBGen is able to recognize the end of system call processing and the non-system call related execution paths.

### *B. SBGen-LITE*

SBGen is by default designed to extract full execution paths for all system calls of each VM. However, according to our preliminary investigation for a VM, up to hundreds of system calls can be invoked every second and the associated execution paths to each system call can be comprised of up to millions of branch outcomes that are expressed with TIPs and TNTs. In this situation, it is impossible for learning-based HIDSs to realize online inference while monitoring every system call and the entire execution paths offered by SBGen. To tackle this problem, we implemented a relaxed version of SBGen, SBGen-lite to reduce the performance overhead minimizing losing the monitoring accuracy of learning-based HIDSs. First, SBGen-lite reduces the number of system calls that it extracts by filtering out certain harmless system calls (e.g. alarm, break, poll, time) that are known to be not exploitable [31]. Moreover, SBGen-lite only extracts the front part of each execution path instead of the entire. This strategy was inspired by a rational intuition that an exploit behavior occurs during the beginning of execution after the input containing exploit payloads is provided. This intuition has also been supported in the literature [11], which argues that execution space can be partitioned into the front small exploitable and the remaining post-exploitable parts due to the attribute of exploit payloads that should be delicately designed based on strict hypothesis regarding the execution environment. To see its effectiveness, we run the model with different packets for execution paths, and the results of these experiments will be explained in section VI.

## VI. EVALUATION

In this section, we evaluate SBGen by considering the following points: (1) the performance of SBGen in extracting the syscall traces and the execution paths and (2) the security when using SBGen with learning-based HIDSs. All experiments were conducted on a machine with Intel i5-8500 Coffee Lake 3.00 GHz cores, 8 GB DDR4 RAM and an Nvidia GeForce 1080 graphic card. We have implemented the prototype of SBGen on the KVM hypervisor included in Ubuntu

[1]Intel PT on Atom architecture serves this function through PTWRITE packet, but it is not available to our SBGen targeting Intel x64.

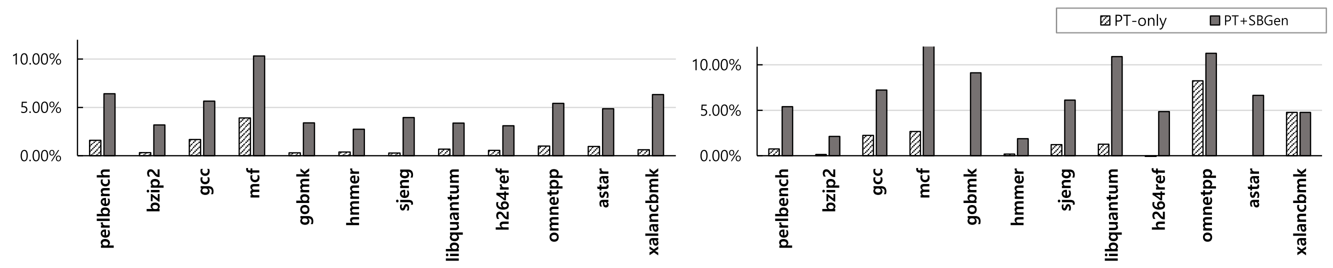

**FIGURE 5.** The slowdown for SPEC CPU2006 by components of SBGen when one VM (left) or two VMs (right) are running.

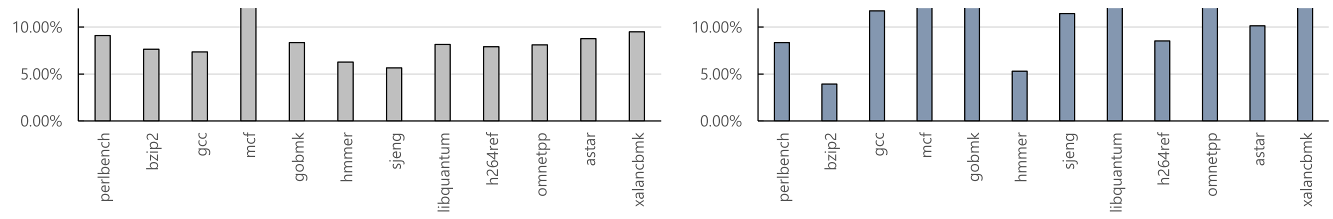

**FIGURE 6.** Runtime performance impact of HIDS-SBGen with SBgen-lite when one or two guest VMs are running.

**TABLE 3.** Summary of generated datasets for SBGen.

| The size of data | System call | Exe-path |
|---|---|---|
| Lookup table | 283 | 4096 |
| Training data | 3733 | 3343194 |
| Validation data | 414 | 371466 |
| Normal test data | 5560 | 4390484 |
| Attack test data | 285 | 75715 |

16.04 with Linux kernel 4.8.0. SBGen is designed to allow each guest VM to run a heterogeneous OS. For simplicity in this paper, however, we assume that all guest VMs run the same OS of Ubuntu 16.04 with Linux 4.8.0. The LSTM models of HIDS-SBGen is implemented using Tensorflow[2] v1.11 with GPU support, one of the most popular frameworks for machine learning.

## A. PERFORMANCE OF SBgen

To show the efficiency of SBGen, we conducted experiments to measure its runtime performance. For this, we ran the SPEC CPU2006 benchmark suite, respectively when one or two guest VMs are running. We measured the impact on runtime performance by examining how much the benchmarks slow down on guest VMs when SBGen is enabled. Figure 5 shows the measurement results of activating Intel PT and the feature extraction function one by one. On average, PT alone incurs 4.90 % (with one VM) and 7.28 % (with two VMs) slowdown. After the feature extraction function runs additionally, it introduces 8.51 % (with one VM) and 11.67 % (with two VMs) slowdown on average.

[2]https://www.tensorflow.org

## B. SECURITY OF LEARNING-BASED HIDSs WITH SBGen

The HIDS-SBGen described in subsection IV-C empirically shows how SBGen is helpful to improve the effectiveness of HIDS. Below, we evaluate it in terms of both performance and security. Table 5 shows the detailed parameters for model configuration and the hyperparameters for training.

### 1) RUNTIME ANOMALY DETECTION

By cooperating with SBGen, HIDS-SBGen can perform runtime anomaly detection on a multi-tenant VM system. To see its effectiveness, we conducted the performance and security assessments. First, as performance assessment, we measured the amount of performance degradation of the system when HIDS-SBGen is running. Next, as a security assessment, we confirmed that HIDS-SBGen can detect major security attacks threatening VM kernels, such as privilege escalation attacks and mimicry attacks.

### 2) PERFORMANCE IMPACT ON THE SYSTEM

HIDS-SBGen can potentially work with either SBGen or SBGen-lite. Given the limited processing capacity of the LSTM models HIDS-SBGen having, we experimented HIDS-SBGen with SBGen-lite. As shown in Figure 6, when all HIDS-SBGen run with SBGen-lite, the average slowdown of the system is 5.47 % (with one VM) and 8.01 % (with two VMs). The performance impact appears to grow depending on the number of running VMs, which is due apparently to the increasing number of packets that HIDS-SBGen and SBGen-lite have to handle. We believe the measured results show the efficiency of SBGen-lite. Particularly noteworthy in the performance evaluation is that HIDS-SBGen with SBGen-lite achieves online inference for the runtime behavior of guest VMs.

**TABLE 4. Detection accuracy of HIDS-SBGen with SBGen-lite, given as the area under the receiver operating characteristic curve (AUC), equal error rate (EER), mean values ($\mu_n$: normal data, $\mu_a$: attack data) of the perplexity and standard deviations ($\sigma_n$: normal data, $\sigma_a$: attack data) of the perplexity. SC-LSTM and EP-LSTM mean the system call model and execution path model, respectively.**

| Attack | System call model only | | | | | | Types of Exe-path | Execution path model only | | | | | | Both models w/ fixed SC-LSTM | | Both models w/ fixed EP-LSTM | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | AUC | EER | $\mu_n$ | $\sigma_n$ | $\mu_a$ | $\sigma_a$ | | AUC | EER | $\mu_n$ | $\sigma_n$ | $\mu_a$ | $\sigma_a$ | AUC | EER | AUC | EER |
| MySQL | 0.9633 | 0.0416 | **9.0703** | 5.0406 | **18.8338** | 0.8423 | Top 10 TIPs | 0.9828 | 0.0171 | **6.6921** | 8.9 | **42.6655** | 0.011 | 0.9746 | 0.0483 | 0.9895 | 0.0322 |
| | | | | | | | Top 10 packets | 0.9828 | 0.0172 | **3.3272** | 1.2881 | **9.4320** | 0.0021 | 0.9742 | 0.0490 | 0.9902 | 0.0200 |
| | | | | | | | Top 30 TIPs | 0.9922 | 0.0079 | **2.2399** | 1.1818 | **7.5265** | 0.0067 | 0.9781 | 0.0416 | 0.9960 | 0.0078 |
| | | | | | | | Top 30 packets | 1.0000 | 0.0000 | **1.5666** | 1.3555 | **8.3367** | 0.0142 | 0.9784 | 0.0416 | 1.0000 | 0.0000 |
| | | | | | | | Top 50 TIPs | 0.9820 | 0.0179 | **13.4720** | 30.9662 | **144.6756** | 0.6697 | 0.9774 | 0.0420 | 0.9900 | 0.0200 |
| | | | | | | | Top 50 packets | 0.9836 | 0.0200 | **2.3025** | 2.7803 | **14.9158** | 0.4422 | 0.9866 | 0.0219 | 0.9904 | 0.0200 |
| GNU Screen | 0.9949 | 0.0063 | **8.1248** | 4.7441 | **20.2001** | 1.4153 | Top 10 TIPs | 0.9451 | 0.0800 | **8.8645** | 14.4369 | **53.5781** | 13.6822 | 0.9968 | 0.0063 | 0.9568 | 0.0790 |
| | | | | | | | Top 10 packets | 0.9903 | 0.0600 | **4.6462** | 1.6735 | **15.6253** | 5.3937 | 0.9968 | 0.0063 | 0.9716 | 0.0538 |
| | | | | | | | Top 30 TIPs | 0.9859 | 0.0800 | **1.9049** | 1.6352 | **7.0758** | 1.0087 | 0.9968 | 0.0063 | 0.9564 | 0.0806 |
| | | | | | | | Top 30 packets | 0.9839 | 0.0800 | **1.4097** | 1.1178 | **6.1845** | 1.1514 | 0.9968 | 0.0063 | 0.9559 | 0.0814 |
| | | | | | | | Top 50 TIPs | 0.9546 | 0.0800 | **57.3161** | 201.8759 | **653.0443** | 223.6096 | 0.9968 | 0.0063 | 0.9578 | 0.0800 |
| | | | | | | | Top 50 packets | 0.9693 | 0.0800 | **8.9046** | 30.7609 | **117.8157** | 56.6203 | 0.9968 | 0.0063 | 0.9579 | 0.0800 |
| BPF | 0.9746 | 0.0463 | **24.1608** | 10.7867 | **48.9005** | 6.9342 | Top 10 TIPs | 0.9615 | 0.1127 | **11.9058** | 3.8320 | **18.5696** | 1.7747 | 0.9760 | 0.0456 | 0.9384 | 0.1093 |
| | | | | | | | Top 10 packets | 0.9149 | 0.1800 | **14.3649** | 4.1852 | **18.3741** | 2.9480 | 0.9760 | 0.0458 | 0.9081 | 0.1800 |
| | | | | | | | Top 30 TIPs | 0.9563 | 0.0800 | **11.5844** | 3.3405 | **15.9058** | 3.6798 | 0.9760 | 0.0456 | 0.9560 | 0.0800 |
| | | | | | | | Top 30 packets | 0.9365 | 0.1288 | **12.7027** | 4.0323 | **18.712** | 3.0832 | 0.9760 | 0.0456 | 0.9258 | 0.1284 |
| | | | | | | | Top 50 TIPs | 0.9863 | 0.0292 | **11.6743** | 3.5821 | **16.7481** | 2.3213 | 0.9760 | 0.0456 | 0.9849 | 0.0292 |
| | | | | | | | Top 50 packets | 0.9164 | 0.1222 | **13.4069** | 3.7322 | **17.2987** | 4.5319 | 0.9760 | 0.0456 | 0.9377 | 0.1106 |
| Netlink socket | 0.9792 | 0.0800 | **22.3541** | 10.6255 | **60.5400** | 13.7846 | Top 10 TIPs | 0.9789 | 0.0600 | **11.0623** | 3.2144 | **18.5696** | 1.7747 | 0.9550 | 0.0799 | 0.9654 | 0.0600 |
| | | | | | | | Top 10 packets | 0.9635 | 0.0480 | **15.2260** | 4.2874 | **20.2858** | 3.4978 | 0.9545 | 0.0800 | 0.9729 | 0.0479 |
| | | | | | | | Top 30 TIPs | 0.9877 | 0.0530 | **10.9526** | 3.1133 | **14.7328** | 2.4357 | 0.9543 | 0.0799 | 0.9734 | 0.0504 |
| | | | | | | | Top 30 packets | 0.9773 | 0.0422 | **14.8705** | 4.2134 | **20.1404** | 4.9962 | 0.9545 | 0.0799 | 0.9766 | 0.0421 |
| | | | | | | | Top 50 TIPs | 0.9814 | 0.0800 | **10.6403** | 2.9636 | **13.6755** | 3.2490 | 0.9559 | 0.0799 | 0.9567 | 0.0800 |
| | | | | | | | Top 50 packets | 0.9942 | 0.0400 | **13.8503** | 3.8450 | **17.5891** | 4.2100 | 0.9565 | 0.0799 | 0.9794 | 0.0400 |
| sudo | 0.9859 | 0.0598 | **47.6893** | 25.7166 | **77.9265** | 4.1966 | Top 10 TIPs | 0.9572 | 0.1111 | **11.6395** | 3.3062 | **13.2536** | 0.1833 | 0.9641 | 0.0670 | 0.9570 | 0.1111 |
| | | | | | | | Top 10 packets | 0.9901 | 0.0177 | **3.7701** | 1.2335 | **6.4948** | 4.4615 | 0.9641 | 0.0670 | 0.9911 | 0.0174 |
| | | | | | | | Top 30 TIPs | 0.9740 | 0.0494 | **12.9517** | 3.3062 | **13.2536** | 0.2901 | 0.9641 | 0.0670 | 0.9849 | 0.0519 |
| | | | | | | | Top 30 packets | 0.9482 | 0.1362 | **5.9862** | 2.3984 | **10.5974** | 6.8960 | 0.9641 | 0.0670 | 0.9192 | 0.1342 |
| | | | | | | | Top 50 TIPs | 0.9468 | 0.1111 | **11.2450** | 3.1914 | **12.9546** | 0.1505 | 0.9641 | 0.0670 | 0.9380 | 0.1110 |
| | | | | | | | Top 50 packets | 0.9855 | 0.0283 | **4.8228** | 1.8214 | **8.6975** | 5.8806 | 0.9643 | 0.0666 | 0.9909 | 0.0291 |
| Rootkit | 0.9997 | 0.0143 | **15.4255** | 9.1038 | **39.1315** | 2.2312 | Top 10 TIPs | 0.9722 | 0.0333 | **17.3101** | 10.1487 | **91.2013** | 65.3277 | 0.9826 | 0.0331 | 0.9830 | 0.0324 |
| | | | | | | | Top 10 packets | 0.8655 | 0.1680 | **7.2042** | 3.1343 | **15.3141** | 14.7254 | 0.9775 | 0.0421 | 0.9147 | 0.1456 |
| | | | | | | | Top 30 TIPs | 0.9866 | 0.0161 | **1.1085** | 0.8705 | **4.4418** | 1.2635 | 0.9930 | 0.0135 | 0.9919 | 0.0160 |
| | | | | | | | Top 30 packets | 0.9977 | 0.0071 | **1.1206** | 0.6161 | **2.3387** | 1.0335 | 0.9931 | 0.0135 | 0.9959 | 0.0081 |
| | | | | | | | Top 50 TIPs | 0.9846 | 0.0185 | **32.3415** | 7.5349 | **336.3354** | 175.3800 | 0.9925 | 0.0142 | 0.9906 | 0.0183 |
| | | | | | | | Top 50 packets | 0.9870 | 0.0153 | **10.1095** | 10.1095 | **89.3687** | 53.5788 | 0.9926 | 0.0142 | 0.9923 | 0.0152 |

### 3) PRIVILEGE ESCALATION ATTACKS

Privilege escalation attacks [32]–[34] are a major threat to the kernel in that malicious applications, out of the control of the kernel, can compromise other applications or even the kernel. To test that HIDS-SBGen can detect this type of attacks, we first trained the LSTM models with data, namely the syscall traces and execution-paths, from benign VMs running various benign applications including MySQL, GNU Screen, BPF and a Netlink socket [35]–[38]. Four applications were specially selected as they were the target victims of publicly available privilege escalation attacks: CVE-2016-6663 and CVE-2016-6664 [39], [40] against MySQL, EDB-ID-41154 [41] against GNU Screen, CVE-2017-16995 [38] against BPF program, CVE-2017-11176 [37] against a Netlink socket, and CVE-2019-18634 against sudo with pwfeedback enabled.

The data for training is obtained in a controlled environment, that is, without any attacks. In order to induce the program to show normal behavior, we provide a mix of human input, statements or commands to be exact, and simulated human input, a random combination of records of actual human input activities.

For our inference with HIDS-SBGen, we give a mixture of normal and attack inputs to MySQL, GNU Screen, BPF and Netlink socket as well as execute benign applications (that were not included in the training of a model) and rootkits (Trojan.Elf32.Ganiw.dirahq, Backdoor.Linux.Gates.B, virus.elf.rootkit.i, ELF/DDoS.BD!tr, HEUR:Backdoor.Linux.Ganiw.d, Linux.Xorddos, Trojan.Xorddos.Linux.34).

In Table 3 we report the total number of syscalls and execution paths collected to train and test our LSTM models. For model training, we report the number syscalls and the sum of their corresponding execution paths lengths used to train and validate our model alongside the lookup table size built from the training data. For inference, we report the amount of data collected from normal benign VM activity and attack events.

**TABLE 5. Parameter for model configuration and hyperparameters for training.**

| Name of parameter | System call model | Execution path model |
|---|---|---|
| the number of LSTM cell | 8 | 64 |
| embedding vector size | 8 | 64 |
| maximum lookup table size | 283 | 4096 |
| minimum occurrence to be added in lookup table | 0 | 10 |
| dropout probability | 0.5 | 0.5 |
| learning rate | 0.001 | 0.001 |
| batch size | 512 | 512 |

Table 4 comprehensively shows the detection accuracy of HIDS-SBGen with SBGen-lite against the privilege escalation attacks and rootkits. As mention in subsection IV-C, HIDS-SBGen embeds two LSTM models, i.e., system call model and execution path model, whose relevant parameters are listed in Table 5. Therefore, we experimented on the following cases according to the types of the activated models: system call model (SC-LSTM) only, execution path model (EP-LSTM) only, and both models. The case of using both models together is divided again into two sub-cases depending on which model's predefined threshold is fixed. Apart from the system call model only, we ran the experiments while varying the number of the initial packets to be examined of execution paths by tuning SBGen-lite: 'Top 10/30/50 TIPs' examine only the initial 10/30/50 TIPs and 'Top 10/30/50 packets' examine the initial 10/30/50 packets regardless of TIP or TNT. Note that one TNT packet may contain from one to six conditional branch outcomes while one TIP contains a single target address of an indirect branch.

As can be seen in Table 4, though all cases show adequate accuracy, the AUC of using both models is typically the greatest, which indicates that it best differentiates attack events from normal behavior. This high accuracy is contributed by the clear difference between the perplexity values of normal ($\mu_n$) and attack behavior ($\mu_a$). Obviously, it can be seen that SBGen-lite, a relaxed SBGen in a way of extracting only the initial packets of an execution path, provides enough accuracy when being used with the learning-based HIDS. The optimal number of initial packets for the best accuracy varies on attacks. Generally, HIDS-SBGen gains a higher detection accuracy when examining a bigger number of initial packets. However, the more monitoring packets, the more run-time overheads. Therefore, to achieve online inference, examining only top 30 initial packets seems adequate for HIDS-SBGen.

In the above experiments, we tested HIDS-SBGen mainly with privilege escalation attacks and rootkits. We deem that our experiments are practical because these attacks are considered the most critical threats from malicious applications within VMs. In addition, since these attacks exploit various common security vulnerabilities, such as race condition, improper arithmetic/sign-extension, buffer overflow, and code injection, HIDS-SBGen would be effective as well against other types of attacks based on these vulnerabilities.

*Mimicry Attacks:* As discussed in subsection II-D, system call mimicry attacks are types of attacks that evade SCI by exploiting security vulnerabilities while invoking a system call sequence that appears to be in a normal pattern. Fortunately, system call mimicry attacks can be thwarted by monitoring execution paths regarding system calls. This is because no matter how carefully crafted system call sequences are, the detailed execution paths must be revealed as abnormal when exploiting vulnerabilities. Therefore, we tested HIDS-SBGen against a syscall mimicry attack to evaluate the usefulness of employing execution paths as another input feature along with syscall traces. The mimicry attack is a modified version of the privilege escalation attack against MySQL which mimics normal MySQL syscall invocation behavior. As can be seen in Table 6, the perplexity of syscalls in mimicry attacks becomes close to that of normal syscall behavior. On the other hand, the perplexity of execution paths stays high which indicates that execution paths can reveal the malicious behavior hidden with mimicry attacks.

**TABLE 6. The usefulness of input features.**

| | syscall perplexity | exe-path perplexity |
|---|---|---|
| Normal | 9.0703 | 25.2631 |
| Attack | 18.3883 | 4174.9115 |
| Mimicry | 11.4239 | 4581.2765 |

## VII. CONCLUSION

In this paper, we presented SBGen that can improve learning-based HIDSs that detect any abnormal activities against guest VM kernels in a virtualized system, by extracting the two types of rich information reflecting execution context in which system calls are invoked and processed: (1) system call traces and (2) execution paths of the guest VM kernel taken to serve each system call. To obtain such rich information efficiently and accurately, SBGen is designed to leverage both Intel PT, a hardware-supported low-overhead execution tracing feature, and several software techniques devised for elaborately decoding and purifying the traced PT packets. As a result, SBGen can perform with reasonable performance, and cooperate with learning-based HIDSs to improve their detection accuracy.

## REFERENCES

[1] S. Jha and M. Hassan, "Building agents for rule-based intrusion detection system," *Comput. Commun.*, vol. 25, no. 15, pp. 1366–1373, Sep. 2002.

[2] U. Kanlayasiri, S. Sanguanpong, and W. Jaratmanachot, "A rule-based approach for port scanning detection," in *Proc. 23rd Elect. Eng. Conf.*, Chiang Mai Thailand, 2000, pp. 485–488.

[3] P. Lichodzijewski, A. Nur Zincir-Heywood, and M. I. Heywood, "Host-based intrusion detection using self-organizing maps," in *Proc. Int. Joint Conf. Neural Netw. (IJCNN)*, vol. 2, May 2002, pp. 1714–1719.

[4] Y.-J. Ou, Y. Lin, Y. Zhang, and Y.-J. Ou, "The design and implementation of host-based intrusion detection system," in *Proc. 3rd Int. Symp. Intell. Inf. Technol. Secur. Informat.*, Apr. 2010, pp. 595–598.

[5] H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, and W. Gong, "Anomaly detection using call stack information," in *Proc. 19th Int. Conf. Data Eng.*, May 2003, pp. 62–75.

[6] C. Kruegel, D. Mutz, F. Valeur, and G. Vigna, "On the detection of anomalous system call arguments," in *Proc. Eur. Symp. Res. Comput. Secur.* Springer, 2003, pp. 326–343.

[7] 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. 2010.

[8] D. Mutz, F. Valeur, G. Vigna, and C. Kruegel, "Anomalous system call detection," *ACM Trans. Inf. Syst. Secur.*, vol. 9, no. 1, pp. 61–93, Feb. 2006.

[9] Intel. (2016). *Intel 64 and IA-32 Architectures Software Developer's Manual*. [Online]. Available: https://software.intel.com/en-us/articles/intel-sdm

[10] H. Yi, G. Kim, J. Lee, S. Ahn, Y. Lee, S. Yoon, and Y. Paek, "Extended abstract: Mimicry resilient program behavior modeling with LSTM based branch models," 2018, *arXiv:1803.09171*. [Online]. Available: http://arxiv.org/abs/1803.09171

[11] Y. Kwon, B. Saltaformaggio, I. L. Kim, K. H. Lee, X. Zhang, and D. Xu, "A2C: Self destructing exploit executions via input perturbation," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2017, pp. 1–15.

[12] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, 1997.

[13] 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.

[14] G. Kim, H. Yi, J. Lee, Y. Paek, and S. Yoon, "LSTM-based system-call language modeling and robust ensemble method for designing host-based intrusion detection systems," 2016, *arXiv:1611.01726*. [Online]. Available: http://arxiv.org/abs/1611.01726

[15] Y. Shen, E. Mariconti, P. A. Vervier, and G. Stringhini, "Tiresias: Predicting security events through deep learning," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2018, pp. 592–605.

[16] K. Xu, K. Tian, D. Yao, and B. G. Ryder, "A sharper sense of self: Probabilistic reasoning of program behaviors for anomaly detection with context sensitivity," in *Proc. 46th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, 2016, pp. 467–478.

[17] H. Oh, H. Yi, H. Choe, Y. Cho, S. Yoon, and Y. Paek, "Real-time anomalous branch behavior inference with a GPU-inspired engine for machine learning models," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2019, pp. 908–913.

[18] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A sense of self for unix processes," in *Proc. IEEE Symp. Secur. Privacy*, May 1996, pp. 120–128.

[19] D. Wagner and P. Soto, "Mimicry attacks on host-based intrusion detection systems," in *Proc. 9th ACM Conf. Comput. Commun. Secur. (CCS)*, 2002, pp. 255–264.

[20] C. Parampalli, R. Sekar, and R. Johnson, "A practical mimicry attack against powerful system-call monitors," in *Proc. ACM Symp. Inf., Comput. Commun. Secur. (ASIACCS)*, 2008, pp. 156–167.

[21] R. H. Yap, "Improving host-based ids with argument abstraction to prevent mimicry attacks," in *Proc. Int. Workshop Recent Adv. Intrusion Detection*. Springer, 2005, pp. 146–164.

[22] G. Tandon and P. K. Chan, "On the learning of system call attributes for host-based anomaly detection," *Int. J. Artif. Intell. Tools*, vol. 15, no. 6, pp. 875–892, Dec. 2006.

[23] R. Ding, C. Qian, C. Song, B. Harris, T. Kim, and W. Lee, "Efficient protection of path-sensitive control security," in *Proc. 26th USENIX Secur. Symp. (USENIX Secur.)*, Vancouver, BC, Canada, 2017, pp. 131–148.

[24] X. Ge, W. Cui, and T. Jaeger, "Griffin: Guarding control flows using intel processor trace," *ACM SIGOPS Operating Syst. Rev.*, vol. 51, no. 2, pp. 585–598, 2017.

[25] Y. Gu, Q. Zhao, Y. Zhang, and Z. Lin, "PT-CFI: Transparent backward-edge control flow violation detection using intel processor trace," in *Proc. 7th ACM Conf. Data Appl. Secur. Privacy*, 2017, pp. 173–184.

[26] D. Kwon, J. Seo, S. Baek, G. Kim, S. Ahn, and Y. Paek, "VM-CFI: Control-flow integrity for virtual machine kernel using Intel PT," in *Proc. Int. Conf. Comput. Sci. Appl.* Springer, 2018, pp. 127–137.

[27] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz. *KAFL: Hardware-Assisted Feedback Fuzzing for os Kernels*. Accessed: Aug. 10, 2017. [Online]. Available: https://www. usenix.org/system/files/conference/usenixsecurity17/sec17-schumilo.pdf

[28] X. Jiang and X. Wang, "'Out-of-the-box' monitoring of VM-based high-interaction honeypots," in *Proc. Int. Workshop Recent Adv. Intrusion Detection*. Springer, 2007, pp. 198–218.

[29] Y. Fu and Z. Lin, "Space traveling across VM: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection," in *Proc. IEEE Symp. Secur. Privacy*, May 2012, pp. 586–600.

[30] R. Wu, P. Chen, P. Liu, and B. Mao, "System call redirection: A practical approach to meeting real-world virtual machine introspection needs," in *Proc. 44th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, Jun. 2014, pp. 574–585.

[31] S. N. Chari and P.-C. Cheng, "BlueBox: A policy-driven, host-based intrusion detection system," *ACM Trans. Inf. Syst. Secur.*, vol. 6, no. 2, pp. 173–200, 2003.

[32] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy, "Privilege escalation attacks on Android," in *Proc. Int. Conf. Inf. Secur.* Springer, 2010, pp. 346–360.

[33] N. Provos, M. Friedl, and P. Honeyman, "Preventing privilege escalation," in *Proc. USENIX Secur. Symp.*, 2003, p. 16.

[34] L. Xing, X. Pan, R. Wang, K. Yuan, and X. Wang, "Upgrading your Android, elevating my malware: Privilege escalation through mobile os updating," in *Proc. IEEE Symp. Secur. Privacy*, May 2014, pp. 393–408.

[35] F. S. Foundation. *GNU Screen*. [Online]. Available: https://www.gnu.org/software/screen/

[36] Oracle. *MySQL*. [Online]. Available: https://www.mysql.com/

[37] (2017). *Privilege Escalation*. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2017-11176

[38] (2017). *Privilege Escalation*. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2017-16995

[39] (2016). *CVE-2016-6663 MySQL/MariaDB/Percona-Privilege Escalation/Race Condition PoC Exploit*. [Online]. Available: http://legalhackers.com/advisories/MySQL-Maria-Percona-PrivEscRace-CVE-2016-6663-5616-Exploit.html

[40] (2016). *CVE-2016-6664 MySQL/MariaDB/Percona-Root Privilege Escalation PoC Exploit*. [Online]. Available: http://legalhackers.com/advisories/MySQL-Maria-Percona-PrivEscRace-CVE-2016-6663-5616-Exploit.html

[41] X. R. LTD. (2017). *EDB-ID:41154 GNU Screen 4.5.0–Local Privilege Escalation*. [Online]. Available: https://www.exploit-db.com/exploits/41154/

**JIWON SEO** received the B.S. degree in electrical and computer engineering from Seoul Women's University, South Korea, in 2016. She is currently pursuing the Ph.D. degree in electrical and computing engineering with Seoul National University, South Korea. Her research interest includes system security against various types of threats.

**INYOUNG BANG** received the B.S. degree in electrical and computer engineering from Seoul National University, South Korea, in 2017, where he is currently pursuing the Ph.D. degree in electrical and computing engineering. His research interest includes system security against various types of threats.

**JUNSEUNG YOU** received the B.S. degree in electrical and computer engineering from Seoul National University, South Korea, in 2019, where he is currently pursuing the Ph.D. degree in electrical and computing engineering. His research interest includes system security against various types of threats.

**YEONGPIL CHO** received the B.S. degree in electrical engineering from POSTECH, South Korea, in 2010, and the Ph.D. degree in electrical and computer engineering from Seoul National University, South Korea, in 2018. He is currently a Professor with the Department of Computer Science, Hanyang University. His research interest includes system security against various types of threats.

**YUNHEUNG PAEK** (Member, IEEE) received the B.S. and M.S. degrees in computer engineering from Seoul National University, South Korea, in 1988 and 1990, respectively, and the Ph.D. degree in computer science from the University of Illinois at Urbana-Champaign, in 1997. He is currently a Professor with the Department of Electrical and Computer Engineering, Seoul National University. His research interests include system security with hardware, secure processor design against various types of threats, and machine learning based security solution.