2018 IEEE International Conference on Software Quality, Reliability and Security

BehaviorKI: Behavior Pattern Based Runtime Integrity Checking for Operating System Kernel

Xinyue Feng[12], Qiusong Yang[12], Lin Shi[12], Qing Wang[123]
[1]Laboratory for Internet Software Technologies, Institute of Software Chinese Academy of Sciences, Beijing, China
[2]University of Chinese Academy of Sciences, Beijing, China
[3]State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
{xinyue, qiusong, shilin, wangqing }@nfs.iscas.ac.cn

***Abstract*—Kernel rootkits pose a serious threat to system security by tampering with the state of operating system inconspicuously. To ensure operating system kernel integrity, Virtual Machine Monitor (VMM) based approaches have been proposed. Most of these approaches use snapshot-based or event-triggered techniques. However, snapshot-based techniques have been suffering from missing transient attacks or significant performance overhead, while event-triggered methods are facing with heavy workload as integrity checking might be triggered by any suspicious actions.**

In this paper, we propose a novel solution which is a behavior-triggered integrity checking approach named BehaviorKI. By analyzing attacking processes, BehaviorKI can extract a set of behavior patterns which characterize malicious behaviors. BehaviorKI will trigger integrity checking with kernel invariants when a malicious behavior pattern detected. In this way, our approach can alleviate the performance burden by reducing the frequent kernel integrity checking. The experiment results show that BehaviorKI outperforms existing snapshot-based and event-triggered approaches.

Keywords-Behavior Patterns; Kernel Integrity Checking; Kernel Rootkit Analysis

I. INTRODUCTION

Kernel rootkits are malwares that subvert the security of the machine and cause malicious functions to be executed. They can load themselves in the highest privilege level, as well as hiding from the traditional antivirus software. Kernel rootkits usually attack kernel code and data structures during runtime by deceiving the detecting and protecting mechanisms running on the kernel layer or application layer. Many researches [4, 5, 7, 8] address this issue by adding the detecting and protecting mechanisms in the Virtual Machine Monitor (VMM) layer which is considered as a safe execution environment. When deploying rootkits detecting mechanisms on the VMM layer, they take the advantages of hardware-assisted virtualization technology to intercept system events and track system states. They cannot be perceived and destroyed by rootkits deployed on operating system.

The integrity of a program is a binary property that indicates whether the program and/or its environment have been modified in an unauthorized manner [11]. The integrity [20] of operating system refers to that the operating system should run as expected, instead of being modified by rootkits. Integrity checking will inspect whether the critical components in operating system are modified illegally, including static components and dynamic components.

In order to check whether there are rootkits that threaten kernel integrity during runtime, Copilot [1] and SBCFI [2] present snapshot-based solutions, which monitor kernel states by periodically executing monitor processes and comparing the collected snapshots of memory contents of the kernel static regions. Recently, researchers realize that dynamic data integrity is also crucial to the security of computer system for modern rootkit used dynamic data as their attacking target [21, 22]. Gibraltar [10] introduced data invariants to define kernel data integrity and checked those data invariants at regular intervals. However, the transient rootkits are likely to be missed if they finish the attacking between two snapshots. One way to deal with this situation is increasing the frequency of snapshots. However, it will increase performance overhead significantly [15].

To detect transient rootkits and reduce performance overhead, event-triggered monitoring techniques are proposed. The trigging events include hypercall interception, page fault interception [4, 5], and etc. However, once some registered event occurs, an integrity checking will be triggered without considering the event context. As a result, there will be too many trigging events that are irrelevant to kernel integrity violation. For example, a writing operation to the particular regions of memory will trigger an unnecessary integrity checking, and thus reduces system performance. To reduce performance overhead of monitoring memory access, some researches [3, 23] only trap events that imply definite misuse. KI-Mon [3] uses a whitelist-based filter to avoid unnecessary kernel integrity checking. But essentially, these approaches do not consider the context when events occurred.

The goal of our work is to reduce kernel integrity checking overhead while ensuring the checking efficiency. Behavior models have been widely used in traditional intrusion detection technologies to monitor malicious behavior [6, 9, 16]. The basic idea of BehaviorKI is to characterize malicious behavior by behavior model, and use malicious behavior to capture potential events that may destroy kernel integrity. In this way, BehaviorKI filters unnecessary kernel integrity checking with events context. It will reduce the frequency of kernel integrity checking and relieve system overload.

In our approach, we propose BehaviorKI, a novel integrity checking system based on behavior-trigger method. BehaviorKI introduces behavior patterns to decide when to trigger nec-

978-1-5386-7757-5/18/$31.00 ©2018 IEEE
DOI 10.1109/QRS.2018.00015

essary kernel integrity checking. Once malicious behaviors are detected, BehaviorKI will trigger to check kernel data invariants to verify whether kernel integrity has been tampered. Data invariants [10, 24] are used to describe the integrity property of critical data structures. The integrity property should be kept all the time, and violations of integrity property indicate that the system has been attacked by rootkits directly or indirectly.

We implement the BehaviorKI prototype on Xen hypervisor [17] which is an open-source hypervisor-based environment. We evaluated the effectiveness of BehaviorKI with experiments. In our experiments, we compared BehaviorKI with snapshot-based methods at 1ms, 10ms, 100ms and 1000ms intervals. To illustrate the decline of performance overhead, we compared BehaviorKI and event-triggered methods with 2610 kernel invariants. At last, we did experiment of BehaviorKI with 24954 invariants to illustrate that our approach can perform well with large numbers of invariants. We use the STREAM benchmark [29] to measure the performance cost on memory bandwidth of the monitored system. The results show that performance overhead of integrity checking is positively correlated with the frequency of integrity checking and the scale of checking contents. In our evaluation, BehaviorKI can detect transient attacks that missed by snapshot-based method. Comparing with event-triggered methods, BehaviorKI reduces performance overhead with lower integrity checking frequency and can detect the same number of rootkits in our experiments

In this paper, we make the following contributions:

- We model the malicious behaviors with behavior patterns by extracting frequent event sequences from historical attacks. Those patterns can be used to detect future integrity violations.
- We propose a novel approach based on the identified malicious behavior patterns to trigger kernel integrity check, which can reduce performance overhead by reducing the frequency of integrity check. It triggers the integrity check only when identified malicious behaviors pattern occurs.
- We implement the prototype of BehaviorKI based on hardware-assisted virtualization technology and an open sourced virtualization platform Xen. It is portable to other VMM-based platforms that support hardware-assisted virtualization.
- We conduct experiments to evaluate the effectiveness of our approach. The results show that BehaviorKI can detect more dynamic data integrity violations than snapshot-based methods and outperform event-triggered methods in performance overhead.

The remainder of the paper is organized as follows. Section II presents some background. Section Ⅲ describes our design of BehaviorKI. We evaluate BehaviorKI deployed on Xen in section Ⅳ, discuss the results and future works in section Ⅴ and Ⅵ. We describe the relative works in section Ⅶ and conclude in section Ⅷ.

II. Background

In this section, we describe the background of hardware-assisted virtualization and the performance problem of integrity checking with hardware-assisted virtualization firstly. Then, we describe the thread model and our assumptions.

A. Hardware-assisted Virtualization

In order to monitor behaviors and measure the integrity of operating system in the privileged level, we utilize hardware-assisted virtualization (HAV), which supports unmodified guest OS with small performance overhead. In this paper we use Intel virtualization technology VT-x [34].

To support CPU virtualization, VT-x provides two mode: VMX (Virtual-Machine eXtension) root mode and non-root mode. The two mode can switch between each other through a set of instructions as VM Exit and VM Entry. Once a privileged instruction is executed in the VMX non-root mode, processor control switches to the root mode through the VM exit instruction. After the hypervisor in root mode takes certain actions to handle VM exit, it switches back to virtual machine through the VM entry instruction [12].

Extended Page Table (EPT) is designed for virtualizing Memory Management Unit (MMU) with hardware. MMU is used for translating guest virtual addresses to guest physical addresses. Guest virtual address is the virtual address used by guest OS, Guest physical address is the physical memory address of guest VM. When EPT is enabled, guest physical addresses are translated to real machine addresses by traversing a set of EPT page structures. EPT can also specify the privileges of software when they access the address, e.g., read, write and execute. Any attempts that access disallowed memory will trigger EPT violations, and then cause VM exits [12].

B. Performance Overhead Problem

When we use technology of hardware-assisted virtualization, many events could cause the guest VM trapping into VMM. When these events trigger VM exit, they will cause a penalty of 2000 to 7000 CPU cycles approximately [14]. This is one of the reasons why virtualizations increase performance overhead. Moreover, the usage of EPT technology will increase the burden of this kind of performance overhead. By setting appropriate access permission to monitored pages, EPT technology can monitor the memory access events. It will trigger VM exit if there exits any access to the specific pages. Monitor events will cause the VM trapped into VMM frequently in consequent.

Besides VM trapping, memory addresses translation from virtual addresses to machine addresses is another reason that increases performance overhead. The hardware will first search for memory pages from translation lookaside buffer (TLB), which cache the most recently used page table entries. If the pages are not found, the hardware will trigger a 2D page walk which is a long latency operation to fetch the virtual-to-machine mapping [13]. Such 2D page walk will cause the performance overload significantly. When we check the kernel integrity in VMM level, we have to reconstruct the data structures from operating system memory space to VMM via memory address translations. It will increase performance overhead significantly, especially when the scale of translations is large. Our work will try to reduce this part of performance overhead by reducing the frequency of integrity checking.

C. *Threat Model*

In our work, we focus on attacks that exploit kernel-level vulnerabilities instead of hardware-level vulnerabilities such as vulnerabilities in CPU, memory controller, system memory chips and system bus. These attacks consists of transient attacks which have shot living duration in memory and persistent attacks. Besides, hardware attacks and attackers whose target is the hypervisor are not considered in this work. The primary threat that BehaviorKI try to avert is kernel rootkits. Kernel rootkits have privileges on operating system kernel and try to mask their presences. These rootkits can modify both static and dynamic kernel components to achieve their attacking goals. Instead of preventing these types of attacks from violating the integrity of operating system kernel, BehaviorKI only detect whether the integrity of operating system is violated in an unauthorized manner. Data privacy is out of our study scope.

D. *Assumptions*

There are two assumptions we set in our approach.

1) We assume that the underlying hardware and hypervisor can be trusted, as assumed by most virtualization security architectures [4, 30]. Some researchers even proved the trust of tiny trusty hypervisors with formal methods, such as XMHF [16] and BitVisor [19]. Our approach only relies on hardware virtualization technology and can be applicable on these hypervisors.

2) We assume that the kernel integrity will not be attacked during booting time. BehaviorKI that starts kernel integrity checking after the operating system booting and reaching a stable state. This is because the kernel code and data structures change significantly during the system initialization, and there already exists mature technologies, such as secure boot [35], to ensure kernel integrity before being loaded.

III. OUR APPROACH: BEHAVIORKI

BehaviorKI is a behavior-pattern based integrity checking system. It launches the integrity checking when malicious behavior is detected. BehaviorKI is designed in hypervisor level to be totally transparent to the monitored operating system without modifying the operating system kernel. In order to monitor the states of operating system and check the kernel integrity without interference from kernel rootkits, BehaviorKI requires a higher level authority to access the states and memory of the kernel. The goal of BehaviorKI is to reduce the performance overhead when checking integrity. By filtering out events that are irrelevant to malicious behaviors, it can reduce the frequency of kernel integrity checking. Therefore, BehaviorKI could further minimize performance overhead during integrity checking.

The architecture of BehaviorKI is shown in Fig.1. It consists of four modules in the hypervisor layer which are malicious behavior modeling, behavior monitoring, kernel invariants extracting and integrity checking. In "malicious behavior

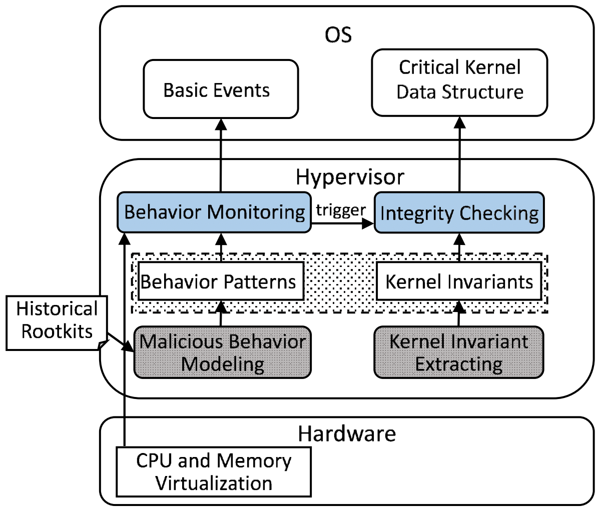


Fig. 1. The architecture of BehaviorKI. Behavior monitoring module and integrity checking module run in hypervisor level to check the kernel integrity.

modeling", we analyze the behavior of historical rootkits, extract its frequent events sequences, and model them as malicious behavior patterns. Kernel invariant extracting gets kernel invariants automatically by analyzing kernel source code and kernel memory snapshots. Malicious behavior modeling and kernel invariant extracting are established in advance, while behavior monitoring and integrity checking are running online. Behavior monitoring module monitors state changes of operating system by intercepting basic events with hardware-assisted virtualization. It collects these events and matches them with behavior patterns. When the behavior monitoring module detects malicious behavior, it will trigger the integrity checking module to check kernel integrity. Then the integrity checking module extracts critical kernel data structures from OS level, and checks whether kernel invariants are violated or not. The details of the four modules are introduced as follows.

A. *Malicious Behavior Modeling*

BehaviorKI describes the attacking malicious behavior as the sequences of events, and introduce behavior patterns to describe the relationship between events. BehaviorKI identifies malicious behaviors patterns by analyzing historical rootkits. We characterize attacking behaviors of rootkits with behavior patterns compositing of operating events such as register accesses, memory accesses and system calls, etc.

1) Basic event

VMM can intercept a wide range of events from hardware relevant events to OS level events. The hardware relevant events are registers accessing, modifications to exception handler, I/O access interceptions and memory accessing. Operating system level events mainly refer to system calls. In our work, we focus on register accessing events and memory accessing events at hardware level, and intercept system calls at operating system level. We refer the above events as basic events. The formal presentations of basic events are listed in TABLE Ⅰ.

TABLE I. FORMAL PRESENTATIONS OF BASIC EVENTS

system operation	formal presentation
reg_op	<r, op>
mem_op	<m, l, op>
system_call	<syscall_num, arg_1, arg_2, … , arg_n >

Register accessing operation: For register relevant operations, BehaviorKI focuses on control register (CR), debug register (DR), model-specific register (MSR), and global descriptor table register (GDTR). By intercepting register access events, BehaviorKI can perceive the state changes of the operating system. For example, CR3 records the Page Directory Base Address for the virtual address space of the running processes. Once a process is switched, contents of CR3 will be changed. By intercepting CR3 writing event, BehaviorKI can monitor the process switching behavior. In BehaviorKI, the register operation is represented as reg_op: <*r*, *op*>, where *r* represents the register that is being accessed and *op* represents the type of the access (read or write).

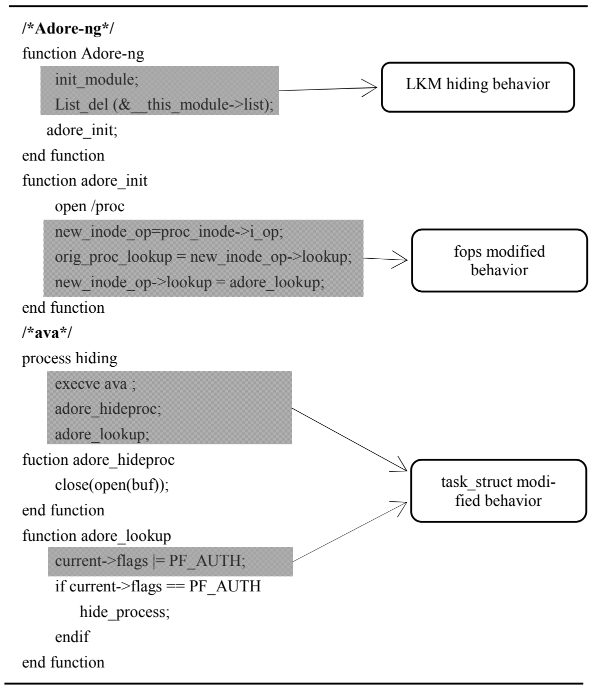
Memory access operation: Because rootkits will violate kernel integrity through memory writing inevitably, memory access operations are critical system events on hardware. However, memory access interception could cause significant performance overhead as it creates extremely huge number of VM exit events. To reduce the performance overhead, BehaviorKI only extracts memory access events to critical regions. The memory access operation is represented as mem_op: <*m*, *l*, *op*>, where *m* represents the name of critical memory region being monitored. *l* represents the location of region *m* in memory, and the scope of *l* is represented as $l \in$ [start_address, end_address]. These addresses are virtual addresses in the operating system. *op* represents the type of the access, such as read, write or execute.

System call based operation: System calls provide an essential interface between kernel mode functions and user programs. Many intrusion detection studies have reported that the attacks cause damage via system calls [9]. For events in operating system level, BehaviorKI intercepts system calls and parameters in them. A system call is represented as system_call: <syscall_num, arg_1, arg_2, …, arg_n >, where syscall_num is the system call number, arg_i represents the i-th argument of the system call. Rootkits usually tamper kernel integrity soon after a system call.

2) Behavior patterns

As malicious behaviors are conducted by the sequences of basic events, we use a pattern language to describe the attacking process. There are four relations between patterns: sequencing, alternation, repetition and temporal constraints. The relations between behavior patterns and their meanings are listed as follows.

pat: reg_op |mem_op |system_call represents a pattern based on basic events.



```
/*Adore-ng*/
function Adore-ng
     init_module;
     List_del (&__this_module->list);
   adore_init;
end function
function adore_init
     open /proc
     new_inode_op=proc_inode->i_op;
     orig_proc_lookup = new_inode_op->lookup;
     new_inode_op->lookup = adore_lookup;
end function
/*ava*/
process hiding
     execve ava ;
     adore_hideproc;
     adore_lookup;
fuction adore_hideproc
     close(open(buf));
end function
function adore_lookup
     current->flags |= PF_AUTH;
     if current->flags == PF_AUTH
          hide_process;
     endif
end function
```

Fig. 2. An example: Attacking process of Adore-ng

sequencing: pat1;pat2 means pat2 appears immediately after pat1. It represents sequential relation between two patterns.

alternation: pat1||pat2 means pat1 or pat2 happens at a time. It represents alternation relation between two patterns.

repetition: pat* means pat appears for zero or more than zero times. It represents repetition of one pattern.

temporal constraints: <pat1,pat2> within t means pat2 appears in t time interval after pat1. BehaviorKI also use the number of trapped events to represent t. **<pat1,pat2> within n** means that there are less than n events trapped between pat1 and pat2.

3) An example

Adore-ng[1] is a LKM-based rootkit that modifies the structure of file operations in Virtual File System (VFS) layer. VFS [33] provides an abstraction to access file system in the Linux kernel. The *inode* is a data structure in a Unix-style file system that describes a filesystem object. Operating system kernel maintains a uniform abstract interface for each file with the *inode* data structure, and the *fops* fields of *inode* define operation functions in VFS. Adore-ng replaces function pointers in *fops* with the hacked ones to hide files, processes, and ports.

When an Adore-ng module is loaded, it uses the LKM hiding technique to conceal itself from the kernel. It manipulates linked list structure module->list to hide an entry in the Loadable Kernel Module (LKM) list. Adore-ng removes the entries from the list once it injects the malicious code into the kernel memory space. The following code is frequently used

[1]*https://github.com/trimpsyw/adore-ng.*

to hide malicious LKM: *List_del(&__this_module->list)*.

Fig. 2 shows the attacking process for Adore-ng to hide processes. LKM hiding behavior is a transient attacking behavior. First, it uses system call init_module to load itself, and then deletes the list head from LKM list after a quite short time. The malicious behavior pattern is {<init_module, <module_list_head, l(module_list_head), write> > within 100 }, where 100 means the number of events trapping into VMM between module loading and module_list_head writing is less than 100. In the following, we simplify this kind of behavior patterns to be {init_module, <module_list_head, l(module_list_head), write>} for the sake of succinct. The integrity checking property is that if the LKM is deleted from the linked list, the LKM code region should also be deleted from kernel memory space.

Adore-ng attacks file operation functions in the loaded kernel module, and then uses a user space process to notify which process should be hidden. For *fops* modified behavior, the attacking target is the function pointers in operation function table of the inode data structure. The behavior pattern is {<proc_inode->fops, l(proc_inode->fops), write>}. The integrity checking activity is that the operation function table of *inode* that stores the function pointers should not be modified during runtime.

In the *task_struct* modified behavior, the attacking targets are dynamic kernel data structures in *task_struct*. The contents of *tast_struct* describe the information of the process. Because ava is a user space process to notice Adore-ng which process should be hidden, the tampered *task_struct* in Adore-ng disappears after the process terminates. The behavior pattern is {exec; open; close; <*task_struct*, l(*task_struct*), write>}, and the integrity checking target is the *flags* in *task_struct*.

B. *Kernel Invariants Extracting*

In this section, we will describe kernel integrity which means that kernel contents should not be modified in an unauthorized manner. The components in the kernel space can be divided into immutable components and mutable components. The immutable components contain kernel codes and static kernel data. The mutable components refer to dynamic kernel data including control-flow data and non-control data. BehaviorKI uses kernel data invariants [10] to describe the kernel integrity property that should be satisfied during runtime. BehaviorKI defines a set of kernel invariants by automatically analyzing kernel source codes and runtime snapshots of the operating system.

Since the immutable components in kernel cannot be modified during runtime, the invariant property for immutable components is that the hash value of their contents should be fixed. We obtain the start and end address of kernel codes and critical static data structures from System.map. Writing operations on these data structures or code regions indicate that the integrity of kernel is violated. Since writing on static regions should not happen when systems run normally, it will not put much extra performance overhead when BehaviorKI monitors the entire region of kernel codes and read-only data.

For mutable components, one characteristic is that dynamic kernel data structures are permitted to change [7]. Another one is that some dynamic kernel data structures exist transiently. An example of dynamic kernel data is the head of LKM list. It is usually modified by rootkits to hide themselves immediately after its module having been loaded. Because transient attacking behaviors may happen between two detecting intervals, the traditional passive monitors detecting memory at regular intervals cannot detect them.

Data invariants [10] are proposed as specifications of data structure integrity property that should not be violated during runtime. They can be specified either by experts [28] or automatic tools [24]. Control flow components are usually function pointers storing the addresses of kernel function. Rootkits usually change these points to their own malicious ones. The integrity property is that the contents of control flow components should point to an available location in the kernel code address space. Dynamic non-control data structures store critical information and user identification data. Recently, researchers realize that dynamic data integrity is very important to the security of computer system [21, 22]. Linked list structures, such as the process lists and LKM lists, are typical dynamic non-control data. These lists are modified when loading or unloading processes and Linux kernel modules during system runtime. One example is the list of processes, and there are two lists of processes in Linux kernel. One list is the *all-tasks* list that shows all the processes in the system. Another list is the *run-list* used for scheduling processes during execution. The invariant property of the list of processes is that tasks appear in the *run-list* should also appear in the *all-tasks* list. Another example is the LKM list. The operating system maintains a linked list data structure which stores the list of loaded LKMs. The invariant property of the LKM list is that once a LKM is deleted from the LKM list, the LKM code region should also be deleted from the memory space.

We generate kernel invariants automatically similar to Gibraltar [10]. First, we use variables in System.map as roots. The type definitions of these variables are obtained by analyzing Linux source code with CIL [30]. Second, BehaviorKI extracts variables that are reaching from these roots. For example, if the root is a *C struct*, all members in the *struct* will be extracted by BehaviorKI. By using the type definitions of their members extracted by CIL, BehaviorKI obtains the offset of each member. Then BehaviorKI can obtain the virtual address of these variables with the information in the System.map and the offsets. After translating virtual addresses to machine addresses, we can capture values of the memory variables as a snapshot. BehaviorKI periodically captures the snapshots and convert them into trace files. These trace files are the input of Daikon [24]. With the trace files and type definitions, Daikon automatically generates kernel invariants with its templates. For example, template $x = const$ checks whether the variable equals to a const. BehaviorKI selects those templates according to the system needs.

Some important structures, such as LKM list, cannot be easily extract their invariant properties form Daikon. BehaviorKI will manually obtain their invariant properties.

C. *Behavior Monitoring*

In this section, we will describe how BehaviorKI intercept events defined in the previous section. The methodology described in this paper can be applicable to any VMM platforms that support for privileged operation interceptions and memory translations. As Xen hypervisor [17] is mature and open sourced, we chose Xen to implement BehaviorKI. BehaviorKI intercepts the basic events by leveraging hardware-assisted virtualization technology. It collects the basic events and matches them with the malicious behavior patterns at hypervisor level. In this way, BehaviorKI will only launch integrity checking when malicious behavior is detected rather than detecting any basic event.

Register access operation is a privileged operation that can be intercepted by hardware virtualization technology. The guest state area in VMCS will control the access permissions of guest registers. When register accessing events happen, VM Exit will be triggered. Therefore, we can intercept register access events from VM Exit reason field in VMCS.

To intercept memory access operations, BehaviorKI needs to determine which pages should be monitored first. BehaviorKI monitors events writing to kernel immutable regions and critical dynamic data structures. The virtual addresses of important data structures like process lists are extracted from the System.map in guest kernel. With the structure information of these data structures, BehaviorKI can reconstruct kernel data structures in VMM by translating addresses from guest virtual addresses to machine addresses. Second, BehaviorKI utilizes EPT violation to intercept memory accessing events by removing the readable or writable permission of the monitored memory pages from the EPT entry. Once events are attempted to read or wrote these pages, EPT violation is triggered, and the CPU will trap into VMM to handle the EPT violation. After having intercepted the access events, EPT violation handler will recover these pages to be readable and writable to let the instruction re-executes again. Finally, we remove the read and write permission of the page again in preparation for intercepting the next access to the page immediately after this instruction. To reset the permission, BehaviorKI switches the TF (Trap Flag) on and enables trap debug in the exception bitmap in EPT violation handler. Then CPU traps into VMM in the next instruction. In the trap debug exception handler, BehaviorKI sets the permission of monitored pages to be inaccessible and remove TF flags to make the system run normally.

System call interceptions can be divided into software interrupts and fast system calls according to the different ways to implement system calls. To intercept software interrupts based system calls, BehaviorKI stores the original interrupt descriptor table entry of int 80, and then changes it to a pointer indexing to a non-executable page. To intercept fast system call, BehaviorKI sets the value of IA32_SYSENTER_EIP MSR as a non-executable address. When a guest system attempts to execute system calls, an EPT violation will be triggered. BehaviorKI obtains the system call number and parameters in the relevant registers. After tracking system calls information, BehaviorKI recover the value of EIP register back to the reserved original address. In this way, BehaviorKI simulates jumping to normal entry function of system calls.

D. *Integrity Checking*

As mentioned before, if CPU traps into VMM frequently, the performance overhead will increase. This kind of performance overhead is cause by monitoring events. Another performance bottleneck caused by integrity checking is related to large scale of address translations between guest physical addresses and machine addresses by page walking. In order to reduce performance overhead of integrity checking, BehaviorKI tries to reduce integrity checking frequency. It solves this problem by only trigger integrity checking when malicious behavior happens. Once event sequences collected by Behavior monitoring module are matched to a malicious behavior pattern, integrity checking is invoked to verify whether the system integrity is violated. As the integrity checking targets are on Linux kernel layer while BehaviorKI is deployed on the hypervisor layer, we track the data structures of the checked invariants from the operating system to the hypervisor.

BehaviorKI describes monitor rule of static kernel components as <mem_op, static_kernel_component_invariant>. The mem_op refers to <kernel_component, *l*, write>, where $l \in$ [component_start, component_end]. The component_start is the start address, and component_ end is the end address of the static component region in guest OS virtual address space. The integrity property of static kernel components is that their contents should not be modified anyway. We check the integrity of static kernel components by evaluating whether the hash value of memory region from component_start to component_end is equal to a known good one.

To check integrity of dynamic kernel data, the behavior monitoring module monitors and records sequences of basic events. Due to the fact that integrity checking mechanism deployed outside of guest kernel, it cannot easily access kernel data and kernel context like the kernel does, which is the well-known semantic gap problem [6]. With the structure information of these data structures in guest, BehaviorKI reconstructs kernel data structures in VMM by translating guest virtual addresses to machine addresses.

IV. EXPERIMENT AND EVALUATION

In this section, we first describe the experiments. Then we report the evaluation results of the integrity checking capability and performance of our BehaviorKI prototype.

A. *Experiment*

In this section, we first introduce the experiment settings. Then we describe three approaches compared in our experiments. Lastly, we describe several publicly available Linux rootkits that are used in our evaluations.

1) *Context*

Our BehaviorKI prototype has been implemented on the Xen hypervisor with hardware-assisted virtualization. All experiments were performed on Intel Core i7-4710MQ CPU with 2.5.0GHz and 8GB memory. The original Xen hypervisor is 4.4.0 version and the HVM guest OS is a 32-bit Linux 2.6.24

TABLE II. ROOTKITS USED IN OUR EVALUATION

Rootkit Name	Attacked Data Structure	Behavior Pattern	Kernel Invariant
Enyelkm	content in system call function	{{<idtr, read>; <idt, l(idt), read>}||< SYSENTER_EIP_MSR, read>};< system_call_func , l(system_call_func), write>	invariant of system call entry function (kernel code invariant)
	module->list	{init_module; <module_list_head, l(module_list_head), write> }	LKM list invariant
Adore-ng 0.56	inode->i_ops file->f_op	{<proc_inode->fops, l(proc_inode->fops), write>}	inode operation functions control component invariant
	task_struct->{flags, uid, …}	{exec; open; close; <task_struct, l(task_struct), write>}	task_struct dynamic data component invariant
	module->list	{init_module; <module_list_head, l(module_list_head), write> }	LKM list invariant
xingyiquan	system call table	<syscall_table, l(syscall_table), write>	static data invariant
	module->list	{init_module; <module_list_head, l(module_list_head), write> }	LKM list invariant

kernel. The virtualized guest was distributed with 2GB memory and 1-core configuration.

2) *Treatments*

In our experiments, we compared BehaviorKI with snapshot-based and event-triggered approaches. We implemented the snapshot-based and event-triggered approaches by ourselves since there were no publicly available tools or projects. We use following names to represent these three approaches.

- **SnapCheck** is the snapshot-based approach which conduct integrity checking at regular intervals. For snapshot-based methods, we conducted experiments at the intervals of 1ms, 10ms, 100ms and 1000ms.
- **EventCheck** is the event-triggered approach, where the trigger events contain basic events described in section Ⅲ. In the evaluation, we use a simple whitelist filter of basic events to trigger integrity checking. The events consist of CR register accessing events, all system calls events, memory accessing events considering all static regions and a dynamic region containing LKM. EventCheck500 means that 500 memory page translations are consumed in each integrity checking. While EventCheck1000, EventCheck1500, EventCheck2000 represent that the number of page translations in event-triggered methods is 1000, 1500 and 2000 respectively. **EventNoCheck** represents experiment which monitored events as EventCheck and BehaviorKI did, but EventNoCheck did not check kernel integrity.
- **BehaviorKI** is our behavior-based approach that described in previous sections. BehaviorKI-2.5k represents BehaviorKI checking 2610 invariants. BehaviorKI-25k represents BehaviorKI checking 24954 invariants.

3) *Rootkits*

Table Ⅱ shows the set of rootkits we used in our experiments, as well as violated targets, malicious behavior patterns and checking invariants. We select these three rootkits because their attacked data structures cover kernel code, kernel static data and kernel dynamic data. In addition, some of their malicious behaviors and the attacked components are transient.

Enyelkm is a kernel rootkit in the form of loadable kernel module. It hides files, directories and processes by modifying the entry function of system call instead of tampering the system call table. The code of the hacked system call is patched to redirect system call. One of the hacked system calls is *getdents* which conceals directory entries. Enyelkm modifies the system call *read* to block it return portions from files. Enyelkm also modifies the system call *kill* is to get the root privilege. When the Enyelkm module is loaded, it will use the LKM hiding technique to conceal itself from the kernel. It will manipulate linked list structure module->list to hide an entry in the LKM list. Enyelkm will remove the existing entry from the list once the malicious codes have been injected into the kernel memory space.

Adore-ng is a LKM-based rootkit. As mentioned in previous section, Adore-ng uses the structure of file operations in Virtual File System to hide files, processes and ports. It also modifies the *module->list* structure to conceal itself as what Enyelkm does.

Xingyiquan is also a LKM-based rootkit, it hides processes, files, directories, processes, network connections, as well as adds backdoors. It hacks system call table to redirect system calls.

B. *Evaluation*

In this section, we evaluated the detection capability and performance of BehaviorKI compared with snapshot-based methods and event-triggered methods. In our evaluation, we conducted the experiments for 100 times to calculate rootkit detected percentage and the average performance overhead.

1) *Rootkit Detection Capability*

In this evaluation, we compared rootkits detection capability of BehaviorKI with snapshot-based methods and event-triggered methods. We conducted Snapshot experiments with intervals of 1ms, 10ms, 100ms and 1000ms and summarized the attacked data into three categories in Table III. The first one is static components including kernel code and static kernel data structures e.g., system call table, operations of Virtual File System (VFS). The second evaluation data in our experiments is the contents in *task_struct*. The contents in *task_struct* belong to dynamic non-control flow components. The third evaluation data is the LKM list which is also a dynamic non-control flow component.

TABLE III. ROOTKITS DETECTION RATE AGAINST 100 TRIALS OF RECURRING ATTACKS

Data Type	Snapshot 1000ms	Snapshot 100ms	Snapshot 10ms	Snapshot 1ms	EventCheck	BehaviorKI
static components	100%	100%	100%	100%	100%	100%
task_structs contents	0%	15%	88%	97%	100%	100%
LKM list	0%	0%	0%	0%	100%	100%

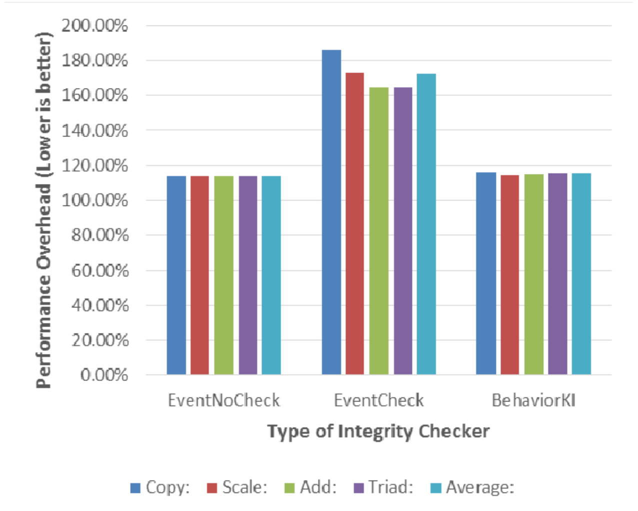


Fig. 3. Performance degradation of BehaviorKI and EventCheck comparing with system running on unmodified Xen. The performance overhead of unmodified Xen is 100%

Table III shows the results of the comparisons. Snapshot-based methods with 1ms to 1000ms intervals, EventCheck and BehaviorKI can detect 100% of static component violations. For static components, SnapCheck, EventCheck and BehaviorKI can detect all integrity violation.

In the experiments of the second evaluation data, the violated *task_struct* existed only for a short period of time. EventCheck and BehaviorKI both can detect all the violations to this kind of kernel data. However, SnapCheck can detect 97% of attacks at 1ms interval while it can only detect 15% at 100ms interval. We can see that SnapCheck missed more attacks as the interval increased.

In the third data evaluation, the lifetime of LKM list attack was much shorter than *task_struct* attack. In Table III, we can see that SnapCheck at 1ms interval still cannot detect that violation, but EventCheck and BehaviorKI can detected the violation in one hundred percent.

In conclusion, SnapCheck, EventCheck and BehaviorKI can all detect static data violations. However, SnapCheck cannot always detect transient data violations in all situations. The detection performance of SnapCheck is effected by the living duration of attacked targets and checking intervals. SnapCheck could not detect transient dynamic data attacks with quite short living duration, while EventCheck and BehaviorKI performed well in detecting this kind of attacks. All these three methods can detected violations of static components for these integrity violations persist after being violated.

TABLE IV. CHECKING FREQUENCY AND PERFORMANCE OVERHEAD OF INTEGRITY CHECKING

	Snapshot 1ms	EventCheck	BehaviorKI
Integrity checking overhead	16.12%	58.10%	1.36%
Integrity checking frequency (times/s)	1000	1558	97

2) *Runtime Performance*

In order to evaluate the effectiveness of BehaviorKI in terms of runtime performance, we compared the performance overhead of BehaviorKI with EventCheck. We evaluated the performance using the STREAM benchmark [29], which was widely used for measuring the memory bandwidth of a computer system. The evaluation was performed over four vector operations: copy, scale, add and triad. We used the average time to indicate the performance. The Copy function simply moves data from one memory location to another. The Scale function is similar to Copy, but it multiplies the value by a constant before writing to the new location. The Add function reads a value from one memory location, followed by a value from another location. It adds the values and writes the results to a third location. The Triad function combines the Scale and Add function. We run the STREAM benchmark on the guest VM. The performance of these approaches is compared with the performance of Xen that does not deploy any events monitoring or integrity checking mechanisms. The performance overhead of original unmodified Xen is 100%.

We conducted three comparisons to illustrate BehaviorKI runtime performance as follows.

a) Performance overhead compared with event-triggered methods

First, we compared the performance of BehaviorKI with EventCheck. As we mentioned before, the performance overhead of EventCheck and BehaviorKI is composed of performance overhead of events monitoring and performance overhead of integrity checking. In our approach, we only consider the decline of performance overhead caused by integrity checking. We introduced EventNoCheck to illustrate performance overhead of events monitoring. Since EventNoCheck only intercepted events but not triggered integrity checking, extra performance overhead compared with EventNoCheck was due to integrity checking. As each variable in the invariants requires a page address translation, large numbers of invariants checking will increase performance overhead of EventCheck significant-

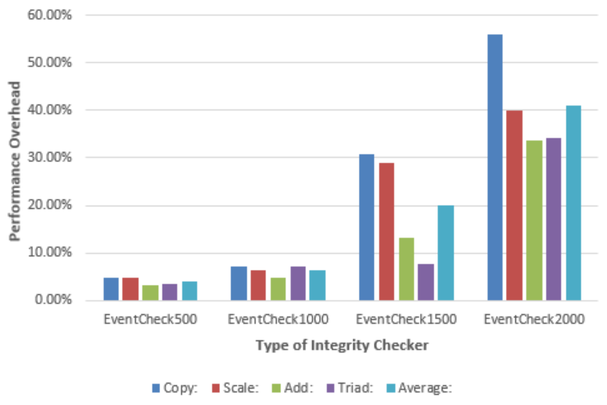


Fig. 4. Extra performance overhead of integrity checking with varied numbers of page translations compared with EventNoCheck

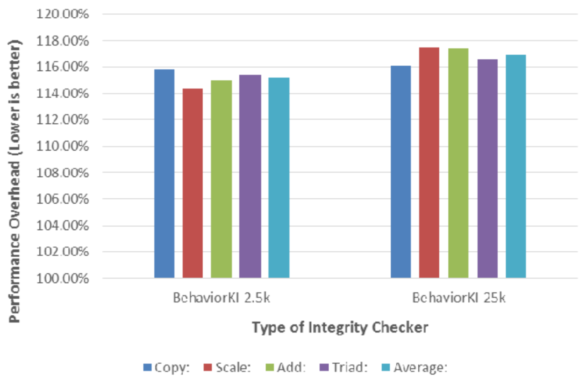


Fig. 5. Performance degradation of BehaviorKI comparing with system running on unmodified Xen. The performance overhead of unmodified Xen is 100%

ly. In this group of experiments, we automatically extracted kernel 2610 invariants with the template *x* = *const*.

The results of performance degradation for EventCheck and BehaviorKI are shown in Fig. 3. EventCheck has 71.91% performance overhead while BehaviorKI has 15.17%. Compared with EventCheck, BehaviorKI reduces 56.74% performance overhead. The results of EventNoCheck show that events monitoring increases 13.81% performance overhead. Since both EventCheck and BehaviorKI need to intercept events, extra performance overhead comparing with EventNoCheck was caused by integrity checking. Then the comparative performance overhead of integrity checking after reducing the inevitable 13.81% performance overhead is 58.10% for EventCheck and 1.36% for BehaviorKI. Comparing with EventCheck, BehaviorKI significantly declines the performance overhead of integrity checking.

To illustrate reasons of performance overhead degradation of BehaviorKI, we evaluated the integrity checking frequency and performance overhead of integrity checking. In Table IV, Snapshot-based method with 1ms intervals triggers integrity checking 1000 times per second. It takes 16.12% extra performance overhead caused by integrity checking. EventCheck triggers 1558 times of integrity checking per second, while BehaviorKI only triggers 97 per second. Compared with EventCheck, the integrity checking frequency of BehaviorKI is much lower. This is the reason that BehaviorKI can degrade performance overhead significantly compared with EventCheck. Table IV shows that when the integrity checking frequency become higher, the performance overhead of integrity checking increases.

b) Performance bottleneck of event-triggered methods with different page translation frequency

In this group of experiments, we compare the performance of EventCheck with different page translation frequency to analyze the performance bottleneck of event-triggered methods. To illustrate the relation between performance overhead and the frequency of memory page translations, we conducted experiments based on event-triggered methods with different page translation frequencies. We conducted experiments of EventCheck which need 500, 1000, 1500 and 2000 page translations.

Fig. 4 shows the comparison results about extra performance overhead of integrity checking among different numbers of page translations. The performance overhead of integrity checking is the extra performance overhead comparing with EventNoCheck. Integrity checking performance overhead of EventCheck500 is only 4.07%, and the performance overhead increases with more memory page translations in integrity checking. EventCheck involving 2000 memory page translations has 40.94% extra performance overhead, which is a high consumption. Such high consumption mainly due to the EPT walking which translates guest virtual addresses to machine addresses during memory integrity checking. Large scale of translations will increase performance overhead significantly. Since each variable in an invariant needs one page translation, event-triggered methods cannot burden large numbers of invariants in integrity checking for high performance overhead.

c) Performance overhead under large numbers of invariants

Lastly, we compared the performance of BehaviorKI with large numbers of invariants. To investigate the effectiveness of our approach with large numbers of invariants, we extracted 24954 invariants with more templates in Daikon [24]. Among these invariants, 2610 invariants were in form of *x* = *const*, while 22344 invariants contained two variables like *variableA* > *variableB*.

In Fig. 5, BehaviorKI-2.5k represents BehaviorKI with 2610 invariants, while Behavior-25k represents BehaviorKI with 24954 invariants. As each variable in invariants need one page translation, checking integrity of Behavior-25k required nearly 50000 page translations for one time. The results show that BehaviorKI cost 3.08% extra performance overhead for integrity checking with 24954 invariants. While the performance overhead of integrity checking with 2610 invariants is 1.36%. BehaviorKI did not cost much performance with large numbers of invariants.

V. DISSCUSSION

The results show that snapshot-based methods could not detect all kernel integrity violations caused by transient attacks, especially those have quite short living duration. While SnapCheck with shorter snapshot intervals increases the performance overhead heavily. EventCheck and BehaviorKI both need to intercept events, which cause CPU trapped into VMM and increase performance overhead. But both of these two methods can detect transient attacks even their living duration is quite short. By introducing BehaviorKI, it filtered unnecessary triggers to checking kernel integrity. According to our evaluation results, BehaviorKI can reduce 56.74% performance overhead with the same detection capability as EventCheck.

The results show that intercepting events and integrity checking could both increase performance overhead. Since EventCheck and BehaviorKI have to intercept the same set of events, their performance overhead caused by intercepting events is the same. This part of performance overhead is necessary for checking transient attacks. We only consider how to reduce performance overhead caused by integrity checking in this paper. Performance overhead of integrity checking is relevant to checking frequency and the memory pages translation frequency involved in integrity checking. BehaviorKI reduces checking frequency by filtering irrelevant event trapping according to malicious behavior patterns. In conclusion, BehaviorKI can detect transient attacks comparing with SnapCheck, and introduced lower performance overhead caused by integrity checking than EventCheck.

In our experiments, we automatically extracts 2610 invariants with a certain template $x = const$ to compare performance overhead with event-triggered methods. However, practicable integrity checking system needs large numbers of data invariants with more templates to describe kernel integrity. For performance overhead increases heavily, event-triggered methods cannot be used on VMM with large numbers of invariants. Fig. 5 shows that BehaviorKI can deal with large numbers of invariants with only 3.08% extra performance overhead of integrity checking.

Event-triggered integrity checking methods typically have performance overhead problem since they will trigger unnecessary kernel integrity checking on normal events. BehaviorKI can alleviate that problem by triggering integrity checking only when the event sequences match with malicious behavior patterns. Therefore, BehaviorKI has lower performance overhead on integrity checking compared to event-triggered methods. The basic events considered in BehaviorKI are listed in section Ⅲ. With more events we didn't consider in this paper, the performance overhead of event-triggered methods will increase more heavily than BehaviorKI. There are still some other basic events that we do not consider in this paper, such as I/O-related events. With more events that we did not consider in the evaluation, the performance overhead of event-triggered methods would increase more heavily than BehaviorKI.

VI. LIMITATIONS AND FUTURE WORKS

In this section, we discuss the limitations and the future works.

A. *Memory accessing monitoring*

In our implemented prototype, BehaviorKI cannot intercept memory access operations on large regions of mutable memory. This is because frequent accesses to mutable memory will cause large numbers of VM exits for EPT violation. High frequency of VMM trapping may lead to system crashing. In our future work, we plan to monitor large scale of memory accessing events, while the system can tolerate performance overhead at the same time. We will also try to reduce unnecessary interceptions of memory access operation by precisely locating the monitored regions.

B. *Behavior patterns generation*

BehaviorKI is a behavior-based approach. The more malicious behavior patterns are established, the more effective the approach will be. In our prototype, the malicious behavior patterns have been extracted and modeled based on expert's experience when analyzing historical attack processes. In the experiment, we only considered three typical rootkits to evaluate the performance of our approach. The results showed that BehaviorKI can get good performance. When including more rootkits, we believe the proposed approach can outperform snapshot-based methods and event-triggered methods too. In addition, in our future work, we will elaborate and improve our behavior pattern modeling to provide an automatic approach. The approach can mine and generate new malicious behavior patterns when unknown rootkits appear.

VII. RELATED WORK

Many researchers contribute to kernel integrity checking of the operating system. BehaviorKI was inspired by behavior model in intrusion detection systems. We briefly describe our related works as follows.

A. *Integrity Checking*

Integrity checking mechanisms are deployed on kernel space at an early stage of researches. IMA [11] is an integrity measurement system that extents TCG trust measurement concepts to dynamic executable contents from BIOS up into the application layer. IMA relies on hardware TPM (Trusted Platform Module) and Linux kernel to do integrity measurement.

As the hardware-assisted virtualization technology develops and VMM-based introspections [23] are proposed, people deploy kernel integrity checking and monitoring mechanisms on hypervisors. Para-virtualized methods have to modify kernel device drivers, while hardware-based full virtualization technologies do not need to modify the kernel which runs transparently. VMM-based approaches can be categorized by the type of protected contents. Secvisor [8] and Pioneer [18] guarantee kernel code integrity from modification and illegal execution. However, only guaranteeing code integrity cannot prevent attacks that control program behavior without modifying the code. Control-Flow Integrity (CFI) [20] is proposed to deal the issue that rootkits attack control flow of kernel without injecting malicious code. Besides, researchers realize that data integrity is also crucial to the security of computer system [21, 22]. Gibraltar [10] introduces data invariants to specify kernel data structure integrity, it automatically infers and enforces specifications of kernel data structure integrity. DADE [31] optimizes kernel data invariants generation and only check data invariants on dirty memory pages. KernelGuard [7] uses a watchpoint to monitor the location of dynamic data changing, and memoryguard is proposed to ensure the accesses to critical memory regions are caused by illegal functions.

B. *Event-triggered Monitoring*

Before event-triggered approaches are proposed, researchers use passive technologies to monitor the target system at intervals. Copilot [1] and Gbrattar [10] obtain snapshots of kernel memory to check kernel integrity violations via an external peripheral device. Petroni describes SBCFI [2] which performs periodic scans of the kernel memory. However, these methods cannot detect transient attacks and frequent monitoring will increase the performance overhead significantly.

Active monitor of kernel data has been proposed to address transient attacks, they use event-triggered techniques to do this. KernelGuard [7] uses VMM-based memory access monitor to prevent malicious memory accesses on protected kernel data actively. OSck [5] utilizes event-triggered method to monitor static regions of the kernel. HyperTap [4] defines four kinds of events to trigger memory monitor based on hardware-assistant virtualization. KI-Mon [3] proposes a hardware-assisted event-triggered monitoring platform for mutable kernel object. It utilizes a whitelist filter to eliminate unnecessary software involvement in value verification. However these approaches use events to trigger memory monitoring without considering its context and basic events do not have enough sematic to trigger integrity checking. In this way it still contains lots of irrelevant events that lead to unnecessary kernel integrity checking and will still have performance problem. BehaviorKI proposed in this paper tried to consider event context and filter irrelevant trigger events.

C. *Malware Behavior Model*

In the researches of intrusion detection system, large numbers of works use behavior models to characterize benign behaviors and malware behaviors. A general method is to use system call sequences and their arguments to characterize the features of malicious software and benign software separately [6, 9, 25]. Some researchers use graph-based models of system calls to describe the system behavior [27]. Recently, Rhee et al [26] use memory access patterns to characterize the behaviors and this method does not rely on temporal control information. Zhixing et al [32] use machine learning to analyze virtual memory access. However, malware detections with intrusion detection technology have great false positives. In our work, we introduce malware behavior model in intrusion detection to describe malicious behavior in integrity checking system. By using malicious behavior to trigger kernel integrity checking, BehaviorKI eliminated unnecessary events that trigger kernel integrity checking.

VIII. Conclusion

In this paper, we introduced BehaviorKI which is a novel behavior-triggered kernel integrity checking technology. BehaviorKI uses behavior patterns to identify malicious behavior. When malicious behavior is detected, BehaviorKI will trigger integrity checking of kernel invariants. BehaviorKI can reduce integrity checking performance overhead by filter unnecessary triggering events. Our experiments showed that BehaviorKI was able to detect all integrity violations, while snapshot-based methods missed some integrity violations on dynamic data structures. BehaviorKI outperformed event-triggered methods by reducing 56.74% performance overhead with 2610 invariants and can deal with large numbers of invariants checking.

Acknowledgment

We would like to thank the anonymous reviewers for their insightful comments. This work is supported by National Natural Science Foundation of China under Grant No. 61432001, and Core Electronic Devices, High-end Generic Chips and Fundamental Software Program under Grant No. 2014ZX01029101-002.

References

[1] N. L. Petroni, Jr., T. Fraser, J. Molina and W. A. Arbaugh, “Copilot - a coprocessor-based kernel runtime integrity monitor,” In Proceedings of the 13th USENIX Security Symposium, Berkeley, CA, 2004, pp. 179-194.

[2] N. L. Petroni, Jr. and M. Hicks, “Automated detection of persistent kernel control-flow attacks,” In Proceedings of the 14th ACM conference on Computer and communications security (CCS '07), New York, NY, USA, 2007, pp. 103-115.

[3] H. Lee, H. Moon, D. Jang, K. Kim, J. Lee, Y. Paek and Brent ByungHoon Kang, “KI-Mon: a hardware-assisted event-triggered monitoring platform for mutable kernel object,” In Proceedings of the 22nd USENIX conference on Security (SEC'13), Berkeley, CA, USA, 2013, pp. 511-526.

[4] C. Pham, Z. Estrada, P. Cao, Z. Kalbarczyk and R. K. Iyer, “Reliability and Security Monitoring of Virtual Machines Using Hardware Architectural Invariants,” 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, Atlanta, GA, 2014, pp. 13-24.

[5] O. S. Hofmann, A. M. Dunn, S. Kim, I. Roy and E. Witchel, “Ensuring operating system kernel integrity with OSck,” In Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems (ASPLOS XVI), New York, NY, USA, 2009, pp. 279-290.

[6] B. Jain, M. B. Baig, D. Zhang, D. E. Porter and R. Sion, “SoK: Introspections on Trust and the Semantic Gap,” 2014 IEEE Symposium on Security and Privacy, San Jose, CA, 2014, pp. 605-620.

[7] J. Rhee, R. Riley, D. Xu and X. Jiang, “Defeating Dynamic Data Kernel Rootkit Attacks via VMM-Based Guest-Transparent Monitoring,” Availability, Reliability and Security, 2009. ARES '09. International Conference on, Fukuoka, 2009, pp. 74-81.

[8] A. Seshadri, M. Luk, N. Qu and A. Perrig, “SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes,” In Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles (SOSP '07), New York, NY, USA, 2007, pp. 335-350.

[9] F. Maggi, M. Matteucci and S. Zanero, “Detecting Intrusions through System Call Sequence and Argument Analysis,” in IEEE Transactions on Dependable and Secure Computing, vol. 7, no. 4, pp. 381-395, Oct.-Dec. 2010.

[10] A. Baliga, V. Ganapathy and L. Iftode, “Automatic Inference and Enforcement of Kernel Data Structure Invariants,” Computer Security Applications Conference, 2008. ACSAC 2008. Annual, Anaheim, CA, 2008, pp. 77-86.

[11] R. Sailer, X. Zhang, T. Jaeger and L. van Doorn, “Design and implementation of a TCG-based integrity measurement architecture,” In Proceedings of the 13th conference on USENIX Security Symposium, Berkeley, CA, USA, 2004, pp. 223-238.

[12] Intel I. and IA-32 Architectures Software Developer’s Manual [J]. Volume 3A: System Programming Guide, Part, 64, 1.

[13] J. Gandhi, A. Basu, M. D. Hill and M. M. Swift, "Efficient Memory Virtualization: Reducing Dimensionality of Nested Page Walks," 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture, Cambridge, 2014, pp. 178-189.

[14] Intel Corporation, Intel Xeon Processor E7 V2 Family Technical Overview, https://software.intel.com/en-us/articles/intel-xeon-processor-e7-v2-family-technical-overview.

[15] H. Moon, H. Lee, J. Lee, K. Kim, Y. Paek and B. B. Kang, “Vigilare: toward snoop-based kernel integrity monitor,” In Proceedings of the 2012 ACM conference on Computer and communications security (CCS '12), New York, NY, USA, 2012, pp. 28-37.

[16] A. Vasudevan, S. Chaki, L. Jia, J. McCune, J. Newsome and A. Datta, “Design, Implementation and Verification of an eXtensible and Modular Hypervisor Framework,” Security and Privacy (SP), 2013 IEEE Symposium on, Berkeley, CA, 2013, pp. 430-444.

[17] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt and A. Warfield, “Xen and the art of virtualization,” In Proceedings of the nineteenth ACM symposium on Operating systems principles (SOSP '03), New York, NY, USA, 2003, pp. 164-177.

[18] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. V. Doorn and P. Khosla, “Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems,” In Proceedings of the twentieth ACM symposium on Operating systems principles (SOSP '05), New York, NY, USA, 2005, pp. 1-16.

[19] T. Shinagawa, H. Eiraku, K. Tanimoto, K. Omote, S. Hasegawa, T. Horie, M. Hirano, K. Kourai, Y. Oyama, E. Kawai, K. Kono, S. Chiba, Y. Shinjo and K. Kato, “BitVisor: a thin hypervisor for enforcing i/o device security,” In Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments (VEE '09), New York, NY, USA, 2009, pp.121-130.

[20] M. Abadi, M. Budiu, Ú. Erlingsson and Ja. Ligatti, “Control-flow integrity,” In Proceedings of the 12th ACM conference on Computer and communications security (CCS '05), New York, NY, USA, 2005, pp. 340-353.

[21] A. Baliga, P. Kamat and L. Iftode, “Lurking in the Shadows: Identifying Systemic Threats to Kernel Data,” 2007 IEEE Symposium on Security and Privacy (SP '07), Berkeley, CA, 2007, pp. 246-251.

[22] S. Chen, J. Xu, E. C. Sezer, P. Gauriar and R. K. Iyer, “Non-control-data attacks are realistic threats,” In Proceedings of the 14th conference on USENIX Security Symposium, Berkeley, CA, USA, 2005.

[23] T. Garfinkel and M. Rosenblum, “A Virtual Machine Introspection Based Architecture for Intrusion Detection,” In Proceedings of the 10th Symposium on Network and Distributed System Security (NDSS '03), 2003, pp. 191-206.

[24] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz and C. Xiao, “The Daikon system for dynamic detection of likely invariants,” Sci. Comput. Program. 69, 1-3, pp. 35-45, 2007.

[25] H. Shimada and T. Nakajima, “Automatically Generating External OS Kernel Integrity Checkers for Detecting Hidden Rootkits,” Ubiquitous Intelligence and Computing, 2014 IEEE 11th Intl Conf on and IEEE 11th Intl Conf on and Autonomic and Trusted Computing, and IEEE 14th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UTC-ATC-ScalCom), Bali, 2014, pp. 441-448.

[26] J. Rhee, Z. Lin and D. Xu, “Characterizing kernel malware behavior with kernel data access patterns,” In Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS '11), New York, NY, USA, 2011, pp. 207-216.

[27] C. Kolbitsch, P. Milani Comparetti, C. Kruegel, E. Kirda, X. Zhou and X. Wang, “Effective and efficient malware detection at the end host,” In Proceedings of the 18th conference on USENIX security symposium (SSYM'09), Berkeley, CA, USA, 2009, pp. 351-366.

[28] N. L. Petroni, Jr., T. Fraser, A. Walters and W. A. Arbaugh, “An architecture for specification-based detection of semantic integrity violations in kernel dynamic data,” In Proceedings of the 15th conference on USENIX Security Symposium (USENIX-SS'06), Berkeley, CA, USA, pp. 289-304.

[29] J. D. McCalpin, “Memory bandwidth and machine balance in current high performance computers,” IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter, 1995, pp. 19-25.

[30] G. C. Necula, S. McPeak, S. P. Rahul and W. Weimeret, “CIL: Intermediate language and tools for analysis and transformation of C programs,” International Conference on Compiler Construction, Springer, Heidelberg, Berlin, 2002, pp. 209-265.

[31] H. Yi, Y. Cho, Y. Paek and K. Ko “DADE: a fast data anomaly detection engine for kernel integrity monitoring,” The Journal of Supercomputing, 2017, pp. 1-26.

[32] Z. Xu, S. Ray, P. Subramanyan and S. Malik, “Malware detection using machine learning based analysis of virtual memory access patterns,” Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017, Lausanne, 2017, pp. 169-174.

[33] R. Love, Linux Kernel Development. Novell Press, 2005.

[34] Intel Virtualization Technology Processor Virtualization Extensions and Intel Trusted execution Technology, https://software.intel.com/sites/default/files/m/0/2/1/b/b/1024-Virtualization.pdf

[35] B. Kauer, “OSLO: Improving the Security of Trusted Computing,” USENIX Security Symposium, 2007, pp. 229-237.