10.48047/jocaaa.2024.33.08.470

Real-Time Detection of Hypervisor Rootkits in Android Apps Using Machine Learning

Shubhi Mishra

M.Tech. scholar, FET, Rama University[1]

Prof(Dr.) Abhay Shukla

HoD, CSE, FET,Rama University[2]

Abstract:- The OS kernel is the beating heart of an operating system, orchestrating resource management with precision. However, it's also a prime target for attacks, particularly via kernel rootkits – malicious kernel modules that can hijack the system with elevated privileges. To counter this threat, numerous approaches have emerged, but they often fall short: many focus on user-mode rootkits, some detect only obfuscated kernel modules, and others incur significant performance overhead.

Enter VKRD, a game-changing kernel rootkit detection system leveraging hardware-assisted virtualization. By creating a transparent, efficient execution environment, VKRD exposes the runtime behavior of kernel modules, making it harder for rootkits to hide. We harnessed TF-IDF to pinpoint key runtime features, feeding them into machine learning models for robust detection. This synergy of virtualization and AI enables VKRD to thrive in cloud environments, delivering high accuracy and moderate performance costs in detecting Windows kernel rootkits.

Key Terms: OS, Kernel Rootkit, Virtualization, Machine Learning.

Introduction: Kernel rootkits are stealthy malicious modules that can infiltrate the kernel space of commodity operating systems, often with ease. The kernel's lack of robust protection and isolation mechanisms creates a fertile ground for these threats to flourish, allowing them to wield high privileges and wreak havoc – hiding processes, siphoning sensitive data, and more. This poses a significant threat to OS security, yet most existing defenses focus on user-level threats, leaving kernel rootkits largely unchecked.

The problem is twofold: not only can kernel rootkits evade traditional security measures, but they also operate with elevated privileges, making them potent adversaries. To combat this, innovative approaches are needed to detect and mitigate kernel rootkits, safeguarding the kernel space and restoring OS security.

The threat landscape is shifting: malware is increasingly leveraging kernel rootkit techniques to gain kernel-level privileges, effectively disabling OS security safeguards and masking malicious activity. ZeroAccess [32] is a prime example – this malware exploited rootkit tactics to spread its botnet, infecting over 9 million Windows systems in 2012. Fast forward to recent times, and Bitdefender's discovery of Zacinlo malware using a rootkit component for adware propagation in Windows 10 systems shows this threat is still very much alive.

Kernel rootkits are the perfect enablers for these types of attacks, allowing malware to operate under the radar and evade detection. As long as malware continues to exploit this vulnerability, robust kernel-level security measures are crucial to preventing widespread damage.

Detecting kernel rootkits is a challenging task, and existing solutions broadly fall into two camps: static and dynamic methods. Static approaches [14], [20] rely on analyzing kernel module code to extract distinguishing features, often requiring disassembly and semantic analysis. However, this breaks down when faced with obfuscated or encrypted code, making it tough to extract meaningful insights.

Static Method Limitations:
- Relies on disassembly and semantic analysis
- Struggles with obfuscated or encrypted code
- May miss runtime behavior changes

Dynamic methods [15], [33], on the other hand, focus on monitoring kernel module behavior at runtime, offering a different set of advantages and challenges. Dynamic methods tackle the obfuscation problem by executing kernel modules in a controlled environment and monitoring their runtime behavior. Typically, emulation techniques like QEMU provide this environment, but they come with limitations:

10.48047/jocaaa.2024.33.08.470

- Significant performance overhead
- Malware can detect and adapt to the emulator
- Incompatible hardware dependencies can skew behavior

These hurdles highlight the need for more robust and transparent execution environments. Hardware-assisted virtualization might be a game-changer here. To overcome these limitations, we introduce VKRD, a Virtualization-based Kernel Rootkit Detection system. Our approach builds on dynamic analysis, running target kernel modules and scrutinizing their runtime behavior. The twist? VKRD leverages virtualization to provide a transparent, efficient execution environment – a significant upgrade over emulation-based solutions.

Key Benefits:

- Transparent execution environment

- Efficient analysis with minimal overhead

- Robust against evasion techniques

By harnessing virtualization, VKRD offers a potent tool for detecting kernel rootkits.

To gain visibility into the kernel module's behavior, we employ stealth breakpoints to intercept its loading process, leveraging the power of hardware-assisted virtualization to sandbox it from the kernel space. By cleverly configuring hardware-assisted paging and VM controls, the VMM gains transparent visibility into the module's interactions with kernel memory and hardware registers. This allows us to reconstruct the module's runtime behavior, generating feature vectors that capture its essence. We harness TF-IDF (Term Frequency-Inverse Document Frequency) to pinpoint the most relevant features, feeding these vectors into supervised machine learning models to train robust classifiers. These classifiers then serve as the backbone for detecting malicious kernel modules, providing a potent tool in the fight against kernel-level threats.

We've built a VKRD prototype on top of the Xen hypervisor, with its core functionality embedded at the VMM level. This design ensures transparency, as there's no agent lurking inside the guest VM. To extract kernel module behavior from outside the VM, we harness VMI (Virtual Machine Introspection) [9], allowing us to peek into the VM's inner workings without disrupting its operation

In summary, we make the following contributions:

- We present a kernel-level sandbox for the kernel mod- ule. By conducting the kernel memory and register isolation mechanism, our system can monitor the kernel module’s run-time behavior transparently.
- We utilize the hardware assisted virtualization tech- nology for the kernel code, kernel data, and register isolation. To automatically obtain the unique signatures for kernel rootkit identification, we make use of the supervised machine learning techniques.
- We design and implement a prototype system based on the Xen hypervisor. The evaluation shows that our system can detect Windows kernel rootkits effectively with a moderate performance cost.

BACKGROUND

- ***KERNEL ROOTKIT***

In typical operating systems, a clear hierarchy exists: the OS kernel and kernel modules wield kernel-mode power, while user apps stick to user mode. Kernel rootkits exploit this, residing in kernel mode and wielding the same privileges as the OS kernel. This elevated access lets them get under the radar, hiding user-level activity and evading traditional security measures.

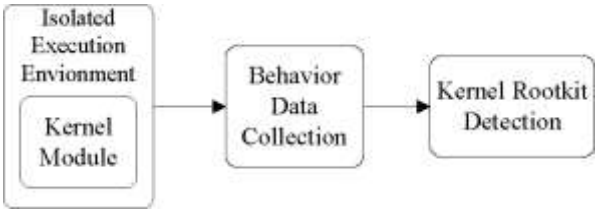

FIGURE 1: The workflow of VKRD.

10.48047/jocaaa.2024.33.08.470

With kernel-level privileges, malware can hide its tracks and shield itself from termination. To fulfill their nefarious goals, kernel rootkits often rely on invoking kernel API functions, leveraging these interfaces to manipulate system resources, intercept sensitive data, or subvert security mechanisms.

INTEL VIRTUALIZATION TECHNOLOGY: Intel VT [5] brings x86 virtualization to life with two new modes: VMX root and VMX non-root. Typically, the VMM (hypervisor) runs in VMX root mode, while guest VMs operate in VMX non-root mode. When a privileged event occurs in the VM (e.g., executing a sensitive instruction), a VM exit flips the switch to VMX root mode, handing control to the VMM. After handling the event, the VMM flips back to VMX non-root mode, restoring the VM's state. The VMM and VM states are stored in a 4-KB VMCS (Virtual Machine Control Structure) block. By tweaking VMCS settings, the VMM can selectively intercept VM operations, gaining fine-grained control over virtualized environments.

SECURITY ASSUMPTION AND THREAT MODEL

Our kernel rootkit detection approach relies on three key assumptions:

1. The OS kernel is trusted pre-rootkit execution.
2. The VMM has a relatively small, trusted TCB.
3. Kernel modules (benign or malicious) load via standard kernel functions.

Threat Model:
- Attackers load malicious kernel modules into the guest VM's kernel space.
- These modules can access memory and CPU registers with elevated privileges.
- However, they're unable to escape the VM and compromise the trusted analyzer VM

I. OVERVIEW OF OUR APPROACH

The goal of VKRD is to build an online system that can detect kernel rootkits dynamically. The key idea for our detection approach is based on an observation: most of kernel rootkits' run-time behaviors differ significantly from the behaviors of benign kernel modules (e.g., device drivers). To analyze the kernel module's behavior, we leverage a hardware assisted virtualization approach. In general, the working process of VKRD is shown in Fig. 1.

First, we exploit the hardware assisted paging mechanism to isolate the target kernel module from the OS kernel. Then, our system can dynamically intercept the interaction between

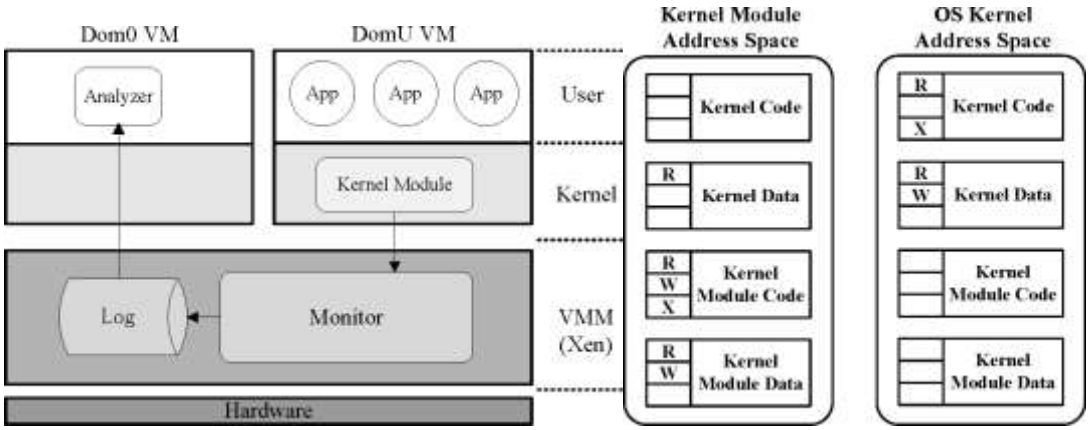

FIGURE 2: The VKRD architecture.

the kernel module and OS kernel. Moreover, to trap the hardware register access operations of the kernel module, we need to well configure the VM control structure inside VMM.

Once the behavior of a kernel module has been extracted, the subsequent step is to determine whether the observed behavior is malicious. Conventional detection methods rely on predefined behavior specifications; however, creating these specifications requires substantial manual effort and expert knowledge. In contrast to these

10.48047/jocaaa.2024.33.08.470

traditional approaches, this work leverages hardware-assisted virtualization to capture a comprehensive set of features from the run-time behavior of kernel modules and subsequently derive key attributes for training a detection model. The extracted features are categorized into three groups: code access operations, data access operations, and register access operations. The selection of these run-time features is motivated by an in-depth analysis of collected kernel rootkits. Empirical observations indicate that most kernel rootkits execute kernel functions, manipulate kernel code or data, and access privileged hardware registers to carry out malicious activities. Utilizing these discriminative features, supervised machine learning techniques are employed to effectively detect kernel rootkits.

We developed VKRD, a prototype system built on the Xen virtualization platform [1], to validate the proposed approach. The overall system architecture is depicted in Fig. 2. Prior to the execution of the target kernel module, hardware-assisted virtualization is employed to isolate the module from the kernel address space. This isolation enables the Monitor component operating within the Virtual Machine Monitor (VMM) layer to intercept any memory or hardware register access attempts made by the kernel module.

I. DESIGN AND IMPLEMENTATION

To determine whether these intercepted operations are malicious, a detection component, referred to as the *Analyzer*, is deployed in the user space of the trusted virtual machine (Dom0). Through the use of direct memory mapping techniques [29], the Analyzer gains access to the operation logs stored in the VMM space. A timer mechanism is configured within the VMM to trigger log analysis by notifying the Analyzer at predefined intervals. Subsequently, the Analyzer evaluates the recorded access operations using a trained detection model. If the observed behavior deviates from the established normal patterns, the system flags the activity as potentially malicious.

MEMORY ISOLATION VIA EPT

A. MEMORY ISOLATION

In commodity operating systems such as Windows and Linux, kernel modules execute with the same privilege level as the operating system kernel and share a common memory space within the kernel region. Consequently, isolating the execution of a kernel module and monitoring its run-time behavior becomes a challenging task. To overcome this issue, we employ hardware-assisted paging mechanisms to establish an isolated execution environment for the target kernel module. Specifically, Intel's Extended Page Table (EPT) technology [12] is utilized, which introduces an additional layer of memory address translation. Similar to conventional page tables, each EPT entry contains memory page mappings along with associated permission bits. By carefully configuring these permission settings, the Virtual Machine Monitor (VMM) can prevent the kernel module from directly accessing memory regions outside its designated code and data sections.

Before carrying out the memory isolation, we need to first know the memory area of the kernel module. Since this memory area is dynamically allocated by the kernel memory allocator, it cannot be pre-determined. To deal with this issue, we need to hijack the kernel module loading code so that the dynamic memory area information can be obtained. For this purpose, we make use of the stealth breakpoint technique [7] to hook the kernel function MmLoadSystemImage. To lo- cate this function, we apply a signature-based method [23] to scan the kernel memory. Then, we set a software breakpoint on the entry of this function. After that, the VMM can inter- cept the invocation of the function MmLoadSystemImage. If the kernel module to be loaded is the target, our system needs to change the function's return address to an invalid address so that the VMM can trap the return operation.

After the kernel function *MmLoadSystemImage* returns—signifying that the kernel module has been successfully loaded—the Virtual Machine Monitor (VMM) extracts the kernel module descriptor from the stack. This descriptor provides critical information, including the module's base address and memory size. Utilizing these details, the VMM enforces memory isolation for the kernel module.

By configuring distinct memory access permissions within the Extended Page Table (EPT), the VMM partitions the original kernel address space into two separate regions. As illustrated in Fig. 3, the isolated kernel module is prevented from accessing the kernel code and data segments of the operating system's kernel address space. Due to this enforced isolation, any attempt by the kernel module to access protected kernel memory triggers an EPT violation, resulting in a virtual machine exit (VM Exit). This event enables the VMM to intercept, record, and analyze the access attempt for security monitoring and anomaly detection.

10.48047/jocaaa.2024.33.08.470

To allow the kernel module to continue executing code access operations (such as invoking kernel functions), the Virtual Machine Monitor (VMM) must switch the active address space. A straightforward approach would be to traverse the Extended Page Table (EPT) entries and modify the memory access permissions accordingly. However, this method incurs considerable performance overhead. To overcome this limitation, we employ two separate EPTs that share identical memory mappings but differ in their access permissions. This design enables the VMM to switch between EPTs simply by updating the EPT base pointer, eliminating the need to repeatedly reset permission bits. Furthermore, Intel's Virtual Processor Identifier (VPID) technology prevents unnecessary Translation Lookaside Buffer (TLB) flushes during EPT switching, thereby significantly improving performance.

In addition, when the kernel module attempts to access kernel data (for example, updating a kernel buffer), the enforced memory isolation triggers a VM exit. To allow this operation to proceed, a conventional approach [27] involves temporarily restoring memory access permissions and enabling single-step execution. This temporary relaxation allows the data access to complete successfully. Due to the single-step mechanism, the VMM regains control immediately after the instruction executes, at which point it reinstates the original memory permissions and disables single-step mode. Consequently, this traditional method introduces two VM exits for every kernel data access operation.

To minimize the number of additional VM exits, we employ an emulation-based technique to handle common data access operations. Specifically, the Virtual Machine Monitor (VMM) first retrieves and disassembles the data access instruction. If the instruction semantics correspond to a common operation, the VMM instantiates a CPU emulator to interpret and execute the instruction. The resulting state changes are then written back to the guest CPU context (i.e., VMCS). Subsequently, the VMM advances the guest program counter to bypass the emulated instruction and resumes execution of the kernel module.

In contrast, if the data access instruction is not categorized as a common operation, the traditional single-step execution approach is adopted to facilitate kernel data access.

To minimize the number of additional VM exits, we employ an emulation-based technique to handle common data access operations. Specifically, the Virtual Machine Monitor (VMM) first retrieves and disassembles the data access instruction. If the instruction semantics correspond to a common operation, the VMM instantiates a CPU emulator to interpret and execute the instruction. The resulting state changes are then written back to the guest CPU context (i.e., VMCS). Subsequently, the VMM advances the guest program counter to bypass the emulated instruction and resumes execution of the kernel module.

In contrast, if the data access instruction is not categorized as a common operation, the traditional single-step execution approach is adopted to facilitate kernel data access.

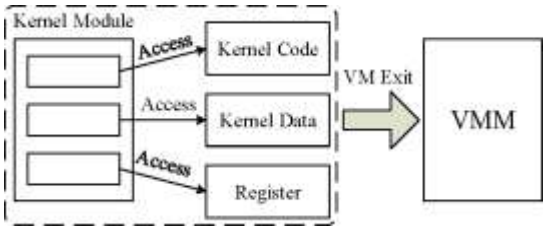

FIGURE 4: Trap kernel module access operations via VMM

Kernel rootkits often manipulate critical system registers for malicious purposes, such as modifying the IDT register to perform IDT hooking. The specific Virtual Machine Control Structure (VMCS) configuration used in our system is presented in Table 1.

To monitor write operations to the SYSENTER_EIP Model-Specific Register (MSR), the Virtual Machine Monitor (VMM) first enables MSR bitmaps by setting the 28th bit in the Processor-Based VM-Execution Controls field. Subsequently, the write bitmap is configured accordingly. Similarly, to monitor write accesses to descriptor registers (IDTR, GDTR, LDTR, and TR), the VMM activates secondary processor-based controls by setting the 31st bit in the same control field and then enables the corresponding control options. In addition, the VMM configures the guest/host mask and read-shadow fields to trap write operations to the write-protect (WP) bit of the CR0 control register.

10.48047/jocaaa.2024.33.08.470

Once the VMCS is properly configured, any access to these sensitive registers triggers a VM exit. By examining the VM exit reason and disassembling the faulting instruction, the VMM can accurately determine the type of register access being performed. To verify whether the access originates from the target kernel module, the VMM checks if the faulting instruction address falls within the module's code memory region. If it does not, the access is attributed to another kernel component.

After analyzing the access attempt, execution must be allowed to continue. Similar to the handling of code and data access operations, the VMM employs a combination of instruction emulation and single-step execution techniques to safely resume the register access operation.

A. FEATURE EXTRACTION

Thanks to the hardware assisted virtualization technology, the VMM can extract the kernel module's run-time behavior. To determine whether this behavior is malicious or not, we need to first extract a set of features from it. To overcome the semantic gap problem [4] in the virtualization environment, we leverage the virtual machine introspection technique [9] to reconstruct the operation semantic from the VM Exit events.

As shown in Fig. 4, when a kernel module tries to access the kernel memory (or a hardware register), the operation will result in a VM Exit due to the memory (or register) isolation. Then, the VMM first checks whether the VM Exit event is caused by our isolation mechanism. If not, the VMM will deal with this event by its default handler. If it is, the

TABLE 1: VMCS configuration of registers

Monitored Registers	VMCS Setting	VM Exit Reason
SYSENTER_EIP MSR	Setting CPU_BASED_VM_EXEC_CONTROL[28] bit for enabling MSR bitmaps; Setting the write bitmap for SYSENTER_EIP MSR	32
IDTR/GDTR/LDTR /TR	Setting CPU_BASED_VM_EXEC_CONTROL[31] bit to activate secondary controls; Setting the descriptor-table exiting filed of Secondary Processor-Based VM-Execution Controls	46/47
CR0.WP	Setting guest/host mask field to 0x00010000; Setting read shadow field to 0x00010000	28

VMM will exam the VM_EXIT_REASON bit of the VMCS structure to identify the VM Exit reason. If the VM Exit reason is a MSR write, it shows the kernel module attempts to modify the MSR. On the other hand, if the reason is an EPT violation, it indicates the kernel module tries to access the kernel memory. Next, the VMM will inspect the specific access type.

Due to the memory isolation, there will be two different EPT violations: EPT execute and EPT write. For the first one, it shows that the kernel module tries to invoke the kernel function (or execute the code in the kernel data memory). To locate the specific kernel function, we utilize the debugging symbols to resolve the function name. Specifically, the Win- dows kernel symbol file (i.e., ntoskrnl.pdb) is downloaded from Microsoft server. Then, we utilize the pdbparse tool to parse the symbol file and extract the RVAs (Relative Virtual Addresses) and function names of kernel APIs. Next, we employ the memory introspection to traverse the kernel module list to locate the kernel base address. By adding the kernel base address and RVAs, we can determine the kernel API names from the EPT execute violation addresses.

Since the kernel APIs are the key bridges between the ker- nel module and the OS kernel, they should be very important features for discriminating between benign and malicious kernel modules. However, based on our observations, we find some kernel APIs do not seem to be the discriminators because they appear in both benign and malicious kernel modules. To address this problem, we utilize the TF-IDF (Term Frequency-Inverse Document Frequency) method for the kernel API selection. This method has been widely used in the field of information retrieval to identify important words. In our study, the Term Frequency (TF) is calculated as follows:

$$TF(i, j) = \frac{n(i, j)}{|k(j)|}$$

where TF(i, j) denotes the frequency of the API i in the kernel module k(j), n(i, j) refers to the number of occurrences of a kernel API i in the kernel module k(j), and $|k(j)|$ represents the total number of all the APIs in the kernel module k(j). On the other hand, the Inverse Document Frequency (IDF) is
calculated as follows:

$$IDF(i) = \log 2 \frac{|K|}{n(i)}$$

where IDF(i) describes whether the kernel API i is rare in all the dataset, $|K|$ represents the total number of all the kernel modules in the dataset, and n(i) refers to the number of the kernel modules where the kernel API i appears.

10.48047/jocaaa.2024.33.08.470

By multiplying TF with IDF, we can get the TF-IDF as follows:

$$TF-IDF\ (i, j) = TF\ (i, j) \times IDF\ (i)$$

where TF-IDF(i,j) describes the importance of the kernel API i in the kernel module j.

After computing the TF-IDF values, we can evaluate the importance of the kernel APIs in benign and malicious kernel modules. According to the importance, the Top-N kernel APIs are selected to generate the feature vectors. Given one sample, we set the given value to be the number of times that the important API appears in this sample. If the important API does not appear, the corresponding given value will be set to zero.

For the EPT write violation, it indicates the kernel module tries to modify the kernel data (or code). If the kernel data to be accessed is global, we can directly obtain the semantic by looking up debugging symbols. However, if the kernel module tries to access the dynamic kernel data, the semantic may not be directly obtained considering that the memory address of dynamic data cannot be pre-determined.

To address this problem, we leverage the memory traver- sal technique [3], [6] to gather the semantic information. The basic idea of this technique is to follow the global kernel object pointers recursively to identify dynamic data. For example, by accessing the FS register, we can obtain the Kernel Processor Control Region (KPCR) structure. By further traversing one pointer field of this structure, we can get the Kernel Processor Control Block (KPRCB) structure. Since this structure contains the current thread structure (i.e., KTHREAD), it can help us calculate the address of process descriptor (i.e., EPROCESS).

It is worth noting that the VMM needs to filter some unexpected events that are not related to the kernel module's behavior. For example, the OS kernel may preempt the kernel module's execution for higher priority interrupts. To deal with this issue, the VMM should check whether the faulting address belongs to an interrupt handler routine in the IDT table. If it is, the VMM will mark this VM Exit event as unexpected.

After analyzing the VM Exit events, we can extract the semantic features as follows:

1) Code access operations: The set includes the important kernel API invocation, invoking the undocumented kernel functions that are not exported by the OS kernel, and executing the code in the kernel data regions (e.g., code packing).

2) Data access operations: These features mainly focus on the write operations to the kernel memory area. These operations include 1) modification to the kernel code (e.g.,

Operation Type	Operation Targets
Kernel API invocations	ZwSetSystemInformation, ZwMapViewOfSection, ZwAllocateVirtualMemory, ZwOpenSection, ZwOpenProcess, ZwUnmapViewOfSection, ZwWriteVirtualMemory, ZwWriteFile, ZwCreateFile, ZwCreateThread, ZwQueryObject, MmGetSystemRoutineAddress, MmProbeAndLockPages, KeGetCurrentIrql, KeQueryInterruptTime, KeQueryPriorityThread, KeQuerySystemTime, KeInsertQueueApc, KeInsertQueueDpc, KeInitializeApc, KeInitializeDpc, KeRaiseIrqlToDpcLevel, KeDelayExecutionThread, KeAttachProcess, KeStackAttachProcess, Ke386IoSetAccessProcess, Ke386SetIoAccessMap, KeInitializeEvent, KeWaitForSingleObject, KeInitializeMutex, MmAllocatePagesForMdl, KeReleaseMutex
Code write operations	ntoskrnl.exe, HAL.dll, win32k.sys, ndis.sys, tcpip.sys, ntfs.sys
Data write operations	EPROCESS, SSDT, IDT, GDT, LDT
Register access operations	IDTR, GDTR, LDTR, TR, SYSENTER_EIP, CR0

TABLE 2: Run-time features for kernel modules

install an in-line hook to ntoskrnl.exe), 2) modification to the existing legitimate kernel module's code (e.g., Tcpip.sys), 3) modification to the kernel control data (e.g., system service descriptor table), and 4) modification to the kernel non- control data (e.g., process descriptor).

3) Register access operations: These features are mainly related to the write operations to the important hardware registers that will have direct (or indirect) effect on the kernel execution. For example, some malicious kernel modules may utilize the load IDT instruction (i.e., LIDT) to change the base address of the interrupt descriptor table (IDT), and redirect the hardware interrupt handling to the malicious one. All these semantic features are associated with the kernel module's run-time behavior, and each of them is either a binary flag or a

10.48047/jocaaa.2024.33.08.470

counter of the number. The binary flag indicates whether the related attribute is present or not, while the counter refers to the number of certain events. To normal- ize the counter features, we apply Min-Max normalization method so that the counter values are adjusted to 0∼1. Table 2 shows the main run-time features in detail.

B. MACHINE LEARNING-BASED DETECTION

With the set of run-time behavior features of the kernel modules, we apply the popular machine learning techniques for kernel rootkit detection. In our experiments, we leverage 4 machine learning algorithms: SVM, Decision Tree, Random Forest, and KNN.

1) SVM

The basic idea of the SVM (Support Vector Machine) is to select a small number of critical boundary points (i.e., support vectors) from each class. Based on the support vec- tors, the SVM learns a hyperplane that can separate two classes with maximum margin, which correspond to vectors of positive samples and negative samples. To classify an unknown sample, we just need to compute whether its vector belongs to the one specific side of the hyperplane.

2) Decision Tree

The basic idea of the Decision Tree is to use the tree structure for making a decision. In general, the tree structure contains one root node, some leaf nodes and some internal nodes.

The leaf nodes represent the classification results while other nodes refer to tests on feature properties. The decision tree has a good interpretation for the classification results.

3) Random Forest

To improve the generalization ability of decision tree, the Random Forest combines multiple decision trees and em- ploys the voting mechanism for the final decision. It uses the bootstrapping method to randomly obtain the samples for training each decision tree. By merging all the decision results of decision trees, the overall result will be improved.

4) K Nearest Neighbor

K Nearest Neighbor (KNN) is a distance-based machine learning algorithm. The key idea of this algorithm is to find k nearest neighbors for one sample in the feature space. Based on the majority vote of these neighbors, the category of this sample is determined. Thanks to the simplicity, KNN does not need to make any assumption on the dataset distribution.

II. EVALUATION

To test the capabilities of VKRD, we conduct perform a se- ries of experiments on a Dell PowerEdge T310 Workstation, which is equipped with a 2.4 G Intel Xeon X3430 CPU and 4 GB memory. We implement our prototype based on the Xen hypervisor (3.4.3 version). We use Fedora 12 (linux-2.6.31) on Dom0 VM and Windows XP on DomU VM.

A. EFFECTIVENESS

To evaluate the effectiveness of our detection system, we need to first train our machine learning classifiers. For this purpose, we collect 192 kernel rootkits (from Rootkit.com, VirusShare and VX Heaven) and 208 legitimate kernel mod- ules (from Windows OS and our lab) as the training data. In general, these rootkits employ different types of kernel techniques, including inline hooking, IDT hooking, SSDT hooking, GDT/LDT Hooking, IRP Hooking, kernel filter, and DKOM. The main malicious activities of these rootkits include modifying kernel code, modifying the key drivers, hiding a process, and injecting the malicious code into the user process. To obtain the corresponding feature set, we run each kernel rootkit/module in the training set by using

TABLE 3: The classification accuracy of different classifiers

Outcome	SVM	Decision Tree	Random Forest	KNN
True Positive Rate	91.59%(207/226)	93.36%(211/226)	95.13%(215/226)	90.71%(205/226)
False Positive Rate	3.77%(10/265)	3.40%(9/265)	1.89%(5/265)	7.17%(19/265)
True Negative Rate	96.23%(255/265)	96.60%(256/265)	98.11%(260/265)	92.83%(246/265)
False Negative Rate	8.41%(19/226)	6.63%(15/226)	4.87%(11/226)	9.29%(21/226)

10.48047/jocaaa.2024.33.08.470

Overall Accuracy	94.09%(462/491)	95.11%(467/491)	96.74%(475/491)	91.85%(451/491)

the virtualization isolation. Before carrying out the run-time training for the machine learning classifiers, the clean virtual- ized environment files (e.g., OS image) are saved first. Then, a kernel module is executed for 5 minutes, ensuring that its run-time behavior is recorded by the underlying VMM. After that, the VMM will restore the clean virtualized environment files and run the next kernel module.

In order to achieve good performance for our classifiers, we have used a 10-fold cross-validation for the training data. After the classifiers are well trained, we apply them for unknown kernel rootkit detection. To do so, we collect additional 226 kernel rootkits and 265 legitimate kernel mod- ules as the test data. Table 3 shows the detection results of different classifiers. The true positive rate represents the rate of correctly identified malicious kernel modules. The false positive rate refers to the rate of wrongly identified benign kernel modules when the classifier detects benign kernel modules as kernel rootkits. The true negative rate represents the rate of correctly identified benign kernel modules. The false negative rate refers to the rate of wrongly identified malicious kernel modules when the classifier fails to detect kernel rootkits.

Compared with the KNN and SVM classifiers, the Deci- sion Tree and Random Forest classifiers have better detection results, and their overall detection accuracy can reach up to 95.11% and 96.74%. Moreover, the false positive rates and false negative rates of our trained classifiers are relatively small. For the common false positive, the main reason is that some benign kernel modules need to modify the impor- tant kernel memory and registers (or invoke sensitive kernel API functions) for their legitimate purpose. To analyze the common false negative, we manually check the target rootkit samples and find that there are mainly two causes: 1) some of them use unexpected ways to inject the malicious code into the OS kernel (e.g., utilizing their own code loader). 2) some of them do not exhibit the malicious behavior unless some specific events happen (e.g., the ioctl system call is invoked).

PERFORMANCE OVERHEAD

To assess the performance overhead introduced by our protection mechanism, we conduct a series of benchmark evaluations. First, a micro-benchmark is executed to measure performance degradation at the kernel function call level. Subsequently, to analyze the application-level overhead, we run a set of application benchmarks while isolating the relevant kernel modules.

For the micro-benchmark, we develop a kernel module to measure the execution time of the *ExAllocatePool* function. Specifically, we utilize the *KeQueryPerformanceCounter* API to obtain high-resolution timestamps immediately before and after invoking the function to allocate 500 bytes of nonpaged pool memory. In the native system, *ExAllocatePool* completes the allocation in approximately 4 microseconds. However, under our detection system, the function requires about 6 microseconds to complete the operation, indicating a modest performance overhead.

To evaluate application-level performance overhead, we conduct a series of application benchmarks. For each benchmark, we isolate a corresponding kernel module from the kernel space. To determine the memory region of the kernel module, we employ virtual machine introspection techniques to retrieve the module descriptor, which contains the module's base address and size information.

In the first benchmark, we target the TCP/IP driver module (*Tcpip.sys*). For this evaluation, an Apache web server is deployed on the target system, and the *ApacheBench* tool is used to measure the server's data transfer rate and response time.

In the second benchmark, the target kernel module is the NTFS file system driver (*Ntfs.sys*). To perform this test, we use the *WinRAR* utility to decompress a standard Linux kernel source package (*linux-2.6.24.tar.gz*) and record the total decompression time.

In the third benchmark, we evaluate the performance overhead introduced by isolating the disk driver (*Disk.sys*). For this test, we use the *xcopy* utility to transfer 57,695 KB of data from one directory to another. By recording the start and end timestamps, we calculate the total execution time required for the copy operation.

In the final benchmark, the target kernel module is a custom driver (*crypt_file.sys*) designed for file encryption. To conduct this test, we measure the time required for the driver to encrypt a 762 MB file using the Advanced Encryption Standard (AES) algorithm. For comparison, all experiments are also performed in the original virtual

10.48047/jocaaa.2024.33.08.470

machine environment to quantify the additional overhead introduced by our mechanism.

Table 4 presents the results of the three application benchmarks. The findings indicate that the performance overhead introduced by VKRD is moderate. Overall, the overhead largely depends on how frequently a kernel module accesses kernel resources. Applications that require frequent access to kernel memory or registers tend to experience slightly higher performance costs. In contrast, when such access operations are less frequent, the additional overhead remains minimal.

To further compare performance with recent emulation-based approaches, we also conduct benchmark tests using DECAF's monitoring framework [11], which employs the QEMU emulator for dynamic analysis.

Benchmark	Target kernel module	Original performance	VKRD performance	DECAF performance
ApacheBench transfer rate	Tcpip.sys	2351Kb/s	2068Kb/s	1507Kb/s
ApacheBench response time	Tcpip.sys	1421ms	1632ms	2103ms
File decompression	Ntfs.sys	331517ms	415286ms	451869ms
File copy rate	Disk.sys	3817Kb/s	3382Kb/s	2295Kb/s
File encryption time	crypt_file.sys	61.26s	67.73s	78.32s

Table 4: Application-level performance of VKRD and DECAF

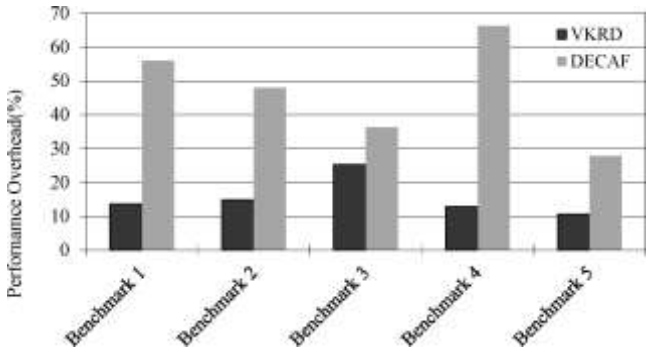

FIGURE 5: Performance overhead comparison

DECAF performs dynamic executable code analysis by default, primarily focusing on capturing system-wide behavioral information of user processes. To monitor kernel module behavior, we develop a lightweight plugin for DECAF. For simplicity, this plugin traces only key kernel API invocations associated with the kernel modules.

Due to its reliance on software emulation, DECAF introduces substantial performance overhead. Specifically, it incurs more than a 66% performance penalty in the fourth benchmark (i.e., file copy). As illustrated in Figure 5, our system demonstrates significantly lower performance overhead compared to DECAF, highlighting its efficiency.

DISCUSSION AND LIMITATIONS

Some advanced kernel rootkits may exploit VM-aware techniques [2] to detect the presence of a hardware-assisted virtualization environment and subsequently conceal their malicious activities. As a result, such rootkits could potentially bypass our detection mechanism. One possible approach to mitigate this issue is to employ bare-metal dynamic analysis methods [13] to detect VM-aware operations, which are typically abnormal for legitimate kernel modules.

Additionally, certain sophisticated rootkits may rely on specific events to trigger their execution. Similar to prior dynamic analysis techniques, our method may not effectively detect this class of kernel rootkits.

10.48047/jocaaa.2024.33.08.470

The memory isolation mechanism in our system may also introduce notable performance overhead when the target kernel module frequently interacts with the OS kernel. To alleviate this overhead, emerging hardware features [25] could be leveraged to enable efficient context switching without involving the VMM. However, implementing this solution may require modifications to the target kernel module.

RELATED WORK

Static Methods. Kruegel et al. [14] propose a static analysis approach for detecting kernel rootkits, employing symbolic execution to identify improper kernel memory accesses. Atefeh et al. [20] combine static analysis with the C5.0 decision tree algorithm to classify and detect kernel rootkits. *RootkitDet* [35] first performs static analysis on rootkit sample code to extract characteristic patterns, which are then used to detect typical rootkits in cloud environments. Despite their effectiveness in identifying certain threats, static methods are inherently limited when dealing with obfuscated kernel rootkits. For instance, Moser et al. [19] demonstrate a binary obfuscation scheme designed to evade static analysis tools. In practice, both benign and malicious kernel modules often employ code obfuscation techniques to hinder static inspection, reducing the reliability of purely static approaches.

Emulation-Based Methods. Emulation-based approaches aim to dynamically observe kernel behavior for rootkit detection. *Limbo* [31] uses a custom emulator to extract behavioral features of kernel rootkits, while *dAnubis* [21] applies a generic emulation-based framework to analyze kernel driver behavior. *K-Tracer* [15] employs taint analysis to capture malicious kernel activity, and *Rkprofiler* [33] monitors control flow transitions between trusted and untrusted code to reveal comprehensive kernel module behavior. Rhee et al. [24] introduce a data-centric approach to OS kernel malware characterization, detecting rootkits by identifying abnormal kernel data manipulations. *MrKIP* [30] combines emulation with Markov chain modeling to recognize malware based on kernel function invocation patterns. However, because most emulation-based techniques simulate execution rather than run it natively, they incur significant performance overhead. Furthermore, sophisticated rootkits can exploit anti-emulation techniques [8] to bypass detection, limiting the robustness of these approaches.

Virtualization-Based Methods. Leveraging hardware-assisted virtualization has emerged as a promising alternative for kernel security. *MemoryMon-RWX* [27] is a hypervisor-based tool for logging and controlling memory access operations, offering potential for rootkit detection. *DRAKVUF* [17] utilizes modern hardware virtualization extensions to perform dynamic malware analysis, capturing behavior in both user- and kernel-space. *MOSKG* [34] applies virtualization to safeguard critical kernel data, while Grimm et al. [10] propose a rootkit defense mechanism that leverages hypervisor-level isolation for enhanced protection. These virtualization-based approaches benefit from running in an isolated environment, which mitigates tampering by kernel rootkits and reduces the risk of evasion through anti-emulation techniques. Nevertheless, integrating these solutions often requires careful handling of performance trade-offs and compatibility with existing OS kernels.

We propose a **virtualization-based protection mechanism** to secure vulnerable kernel modules. Unlike existing approaches that primarily focus on general OS kernel protection or kernel malware analysis, our method is specifically designed to inspect whether a target kernel module exhibits malicious behavior. This distinction allows for more targeted monitoring and protection at the module level.

Hardware-Based Methods. To enhance monitoring performance, several hardware-assisted solutions have been proposed. *Copilot* [22] introduces a coprocessor-based approach that periodically collects memory snapshots and verifies the integrity of the kernel's static regions. Hyungon Moon et al. [18] utilize specialized *Snooper* hardware for snoop-based kernel monitoring, while Hojoon Lee et al. [16] propose a hardware-assisted, event-triggered monitoring mechanism for the kernel. Although these methods offer improved performance and precise monitoring, they rely on specialized hardware, which limits their applicability in general production environments.

To overcome this limitation, Singh et al. [26] leverage commodity hardware features, specifically Hardware Performance Counters (HPCs), for kernel rootkit detection. While this approach can operate on standard hardware, it is restricted in scope and primarily effective against DKOM (Direct Kernel Object Manipulation) rootkits, leaving other types of kernel attacks unaddressed.

Our virtualization-based mechanism addresses these limitations by providing a flexible, hardware-agnostic framework that can monitor kernel modules efficiently while minimizing performance overhead, making it more practical for real-world deployment.

10.48047/jocaaa.2024.33.08.470

CONCLUSION

In this paper, we presented **VKRD**, a virtualization-based system for detecting kernel rootkits. The operation of VKRD can be broadly divided into two key steps. First, we leverage hardware-assisted virtualization to transparently isolate the target kernel module from the rest of the kernel space, ensuring that its execution can be monitored without affecting normal system operations. Second, we utilize the underlying Virtual Machine Monitor (VMM) to capture the runtime behavior of the isolated kernel module and generate detailed feature vectors representing its activity.

To identify the most relevant features for classifier training—including SVM, Decision Tree, Random Forest, and K-Nearest Neighbors (KNN)—we employ the TF-IDF method, which effectively highlights significant behavioral patterns while filtering out redundant information.

Our experimental evaluation demonstrates that VKRD can accurately detect kernel rootkits with a moderate performance overhead, striking a balance between security and system efficiency. By combining virtualization-based isolation with feature-driven behavioral analysis, VKRD offers a practical and effective framework for kernel-level threat detection. Future work may explore extending VKRD to handle more sophisticated anti-virtualization rootkits, as well as optimizing feature selection and classifier models to further reduce performance impact while improving detection accuracy.

REFERENCES

[1] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles, pages 164–177, New York, NY, 2003. ACM.

[2] Michael Brengel, Michael Backes, and Christian Rossow. Detecting hardware -assisted virtualization. In Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, pages 207–227, 2016.

[3] Martim Carbone, Weidong Cui, Long Lu, Wenke Lee, Marcus Peinado, and Xuxian Jiang. Mapping kernel objects to enable systematic integrity checking. In Proceedings of the 16th ACM Conference on Computer and Communications Security, CCS '09, pages 555–565, 2009.

[4] P. M. Chen and B. D. Noble. When virtual is better than real [operating system relocation to virtual machines]. In Proceedings Eighth Workshop on Hot Topics in Operating Systems, pages 133–138, May 2001.

[5] Intel Corporation. Intel 64 and ia-32 architectures software developer's manuals (volume 3, charpter 24). https://software.intel.com/sites/default/files/managed/a4/60/325384-sdm-vol-3abcd.pdf.

[6] Weidong Cui, Marcus Peinado, Zhilei Xu, and Ellick Chan. Tracking rootkit footprints with a practical memory analysis system. In Presented as part of the 21st USENIX Security Symposium (USENIX Security 12), pages 601–615. USENIX, 2012.

[7] Aristide Fattori, Roberto Paleari, Lorenzo Martignoni, and Mattia Monga. Dynamic and transparent analysis of commodity production systems. In Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 417–426, 2010.

[8] Peter Ferrie. Attacks on virtual machine emulators. Technical report, Symantec, 2006.

[9] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In Proceedings of the 10th Annual Network and Distributed System Security Symposium (NDSS), 2003.

[10] Jonathan Grimm, Irfan Ahmed, Vassil Roussev, Manish Bhatt, and Man- Pyo Hong. Automatic mitigation of kernel rootkits in cloud environments. In Brent ByungHoon Kang and Taesoo Kim, editors, Information Security Applications, pages 137–149, Cham, 2018. Springer International Publish- ing.

[11] A. Henderson, L. K. Yan, X. Hu, A. Prakash, H. Yin, and S. McCamant. Decaf: A platform-neutral whole-system dynamic binary analysis plat- form. IEEE Transactions on Software Engineering, 43(2):164–184, Feb 2017.

[12] Intel. Intel 64 and ia-32 architectures software developer's manuals. http://www.intel.com/Assets/PDF/manual/253669.pdf.

[13] Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. Barecloud: Bare-metal analysis-based evasive malware detection. In Proceedings of the 23rd USENIX Security Symposium, August 2014.

[14] Christopher Kruegel, William Robertson, and Giovanni Vigna. Detecting kernel-level rootkits through binary analysis. In ACSAC '04: Proceedings of the 20th Annual Computer Security Applications Conference, pages 91–100, Washington, DC, USA, 2004. IEEE Computer Society.