Assessing the Trustworthiness of Drivers

Shengzhi Zhang and Peng Liu

The Penn State University, University Park, PA, USA
suz116@psu.edu, pliu@ist.psu.edu

Abstract. Drivers, especially third party drivers, could contain malicious code (e.g., logic bombs) or carefully designed-in vulnerabilities. Generally, it is extremely difficult for static analysis to identify these code and vulnerabilities. Without knowing the exact triggers that cause the execution/exploitation of these code/vulnerabilities, dynamic taint analysis cannot help either. In this paper, we propose a novel cross-brand comparison approach to assess the drivers in a honeypot or testing environment. Through hardware virtualization, we design and deploy diverse-drivers based replicas to compare the runtime behaviour of the drivers developed by different vendors. Whenever the malicious code is executed or vulnerability is exploited, our analysis can capture the evidence of malicious driver behaviour through comparison and difference telling. Evaluation shows that it can faithfully reveal various kernel integrity/confidentiality manipulation and resource starvation attacks launched by compromised drivers, thus to assess the trustworthiness of the evaluated drivers.

Keywords: Driver code safety, diversity, hardware virtualization

1 Introduction

Drivers, especially third party drivers, could contain malicious code (e.g., logic bombs) and/or carefully designed-in vulnerabilities. Once got executed/exploited, such compromised drivers render the attackers the opportunity of leveraging drivers' privilege to manipulate system integrity and data confidentiality. Even worse, some attackers have successfully stolen certification from benign third-party and easily obtained trust from the most cautious system engineers. For instance, *mrxcls.sys*, a driver digitally signed with a compromised Realtek certificate, may be viewed as trusted and loaded into industrial OS by system engineers. Once loaded, it injects malware *Stuxnet* into the victim OS, which in turn causes catastrophe in Siemens supervisory control and data acquisition industrial systems [2].

Fully assessing third party drivers before running them in most commodity server systems is challenging. First, static analysis of such drivers is not always possible due to the unavailability of their source code. Furthermore, carefully designed-in vulnerability or malicious code triggered by some specific logic are extremely difficult to be pinpointed during static analysis. Second, dynamic taint analysis (e.g., [37] and [11]) of driver code is generally infeasible, due to the unknown taint seed during assessment. Without an accurate reference model, tainting the entire code space of drivers can only reveal drivers' behaviour, instead of distinguishing legitimate actions from malicious ones. Last, besides promiscuous

attacks such as kernel integrity manipulation, some passive attacks, e.g., listening post, launched by compromised drivers are more difficult to be captured.

Previous research proposes to protect kernel integrity from drivers by confining the drivers' execution context, e.g., Nooks [30], Gateway [28], HUKO [35], Device Driver Reuse and Isolation [24], and Mondrix [34]. Although these systems can effectively monitor drivers' interaction with kernel functions or data, deploying such isolation approach to assess drivers would cause a large number of false positives or false negatives. For instance, both Gateway [28] and HUKO [35] rely on explicitly white-listed legitimate entry points for control transfer from drivers to OS kernel. However, such explicit and complete reference model is quite difficult to be established in practice. We observe that legitimate drivers indeed invoke kernel functions not defined in legitimate entry points occasionally, which results in false positives. Moreover, frequently invoking kernel APIs defined in legitimate entry points can also lead to Denial of Service attack due to resource starvation, which causes false negatives.

In this paper, we present a novel driver evaluation approach, Heter-device, to comprehensively assess drivers against an implicit and complete model before putting any trust on them. Heter-device relies on virtual platforms to emulate heterogeneous device (Heter-device) pairs (e.g., Intel 82540EM NIC and Realtek RTL8139) for guest operating system replicas. Each replica loads heterogeneous drivers corresponding to the devices it runs on. Heter-device approach stands on the assumption that heterogeneous drivers should not have the same exploitable vulnerability due to their separated developing processes. So they provide an implicit and complete reference model for each other when trustworthiness assessment is conducted via fine-grained auditing. Hence, by deploying Heter-device as a high-interaction honeypot, we can closely compare the divergence of two replicas when the vulnerable driver is being compromised and leveraged.

The two replicas with heterogeneous drivers are synchronized at the exported function entry points, which are declared by OS kernel and implemented by each driver. We start a fine-grained auditing of driver's execution whenever kernel calls the corresponding driver functions. During driver's execution, every jump, call or return to kernel or other kernel modules' address space are logged for verification. The logs from heterogeneous drivers are parsed and compared to check any suspicious control flow redirection, e.g., one driver jumps to a kernel segment written by itself, while the other does not exhibit such behaviour. Moreover, any modification to key kernel data by drivers is recorded and verified against the heterogeneous drivers to check if it is a legitimate modification or a malicious manipulation.

We also deal with passive attacks launched from compromised drivers, e.g., network card driver intercepts incoming/outgoing packets and redirects them to remote entities. Thus, the network outgoing packets of the two replicas are audited and compared to find mismatch. Additional amount of traffic on one replica against the other suffices an alarm of confidentiality compromise. Finally, abuse of kernel APIs, such as spin lock or kernel memory allocation requests, may cause CPU or memory starvation. Hence, any call to these resource request APIs from drivers is also verified against heterogeneous drivers. By placing the synchronization and monitoring "sensors" in Heter-device, our honeypot can faithfully reveal multiple attack vectors of compromised drivers, including kernel integrity manipulation, resource starvation, and confidentiality tampering.

We target a honeypot or testing environment; accordingly, we implement Heter-device framework based on open source QEMU [1] project for the follow-

ing reasons. First, QEMU facilitates our Heter-device architecture by providing heterogeneous device emulation options for several types of devices, e.g., sound card (sound blaster 16 or Gravis ultrasound GF1), Ethernet network card (Intel 82540EM NIC or Realtek RTL8139), video card (Cirrus Logic GD5446 Video card or Standard VGA card with Bochs VBE extensions), and etc. Furthermore, it enables our fine-grained auditing of driver's execution through binary translation blocks. Specifically, since each jump of register *eip* generates a new translation block in QEMU, we can simply monitor *eip* at the beginning of each translation block to capture the driver's execution context, rather than auditing every instruction. Last, though the overhead of QEMU is significant, providing good performance is not so critical in either honeypot or testing environment.

Our evaluation shows that Heter-device is effective in revealing multiple attack vectors of compromised drivers, e.g., kernel APIs abuse, malicious code injection, key kernel data tampering, resource starvation, and sensitive information leakage. Accordingly, typical real world use of Heter-device can be as follows: the system engineers assess drivers using Heter-device first, and then choose the trustworthy drivers [1] to run their server systems. Compared to native QEMU execution, the performance overhead incurred by auditing control flow transition and synchronization can be optimized to range from 20 % to 90 %, depending on the amount of kernel data to be audited. Heter-device driver assessment only requires drivers' binary code to run in network-oriented testing environment (i.e., honeypot), and dose not involve any modification to driver source code, compilers, or targeted operating systems.

The rest of this paper is organized as follows. The next section overviews Heter-device threat model. Section 3 presents the design details of Heter-device approach, focusing on Heter-device architecture, address-alias correlation, runtime synchronization and multi-aspect auditing and verification. Section 4 summarizes the implementation issues of Heter-device. In Section 5, we evaluate Heter-device by case studies and measure its performance overhead. In Section 6, we discuss the limitation and future work of Heter-device. Finally, we present related work in Section 7 and conclude in Section 8.

2 Threat Model

In this paper, we assume that the device drivers are untrusted, either with vulnerabilities that can be exploited locally or remotely, or inherently malicious. Furthermore, we only focus on the exploitations that are carefully designed and crafted by attackers. Otherwise, crashing the target system will definitely draw system engineers' attention to trace such consequence back to the root cause. Last, as the base of Heter-device, we assume that heterogeneous drivers should have different vulnerabilities or different malicious code, in terms of where and what the vulnerabilities or malicious code are. Hence, at least one driver can serve as a criterion to verify and alarm the other compromised driver's execution.

It is generally believed to be challenging to verify the behaviour of untrusted drivers in an efficient and robust way due to at least the following reasons. First, drivers in most commodity OS have exactly the same privilege as kernel and

[1] System engineers can either buy the corresponding real hardware devices or configure virtual platforms to emulate those devices.

run in the same address space as kernel. Thus, any kernel module performing auditing or monitoring tasks may be manipulated by the compromised driver. Second, the attackers may leverage the compromised driver to tamper arbitrary OS components (e.g., function pointers, file metadata, system call table, etc.), to accomplish their intrusion goals. Hence, it is also quite difficult, if not impossible, to pinpoint the comprehensive auditing points covering all possible damages/harms that could be caused by compromised drivers. Last, even if it is possible to censor drivers' execution efficiently and comprehensively, lacking a complete reference model makes the verification of drivers' behaviour challenging. Significant false positive or false negative is expected.

In this paper, we propose a novel approach, Heter-device, using driver-diversity-based replica as a complete and implicit reference model, to assess drivers in the following attack vectors:

Control Flow Manipulation. The control flow transition from driver to kernel is tampered by compromised drivers, e.g., jumping to a specific address in the middle of kernel functions, making suspicious kernel function calls to modify critical registers, and etc.

Key Kernel Data/Code Manipulation. Compromised drivers tamper with kernel code, static global variables, or key dynamic data specified by kernel developers or system engineers, e.g., system call table, interrupt descriptor table, double linked list pointers in process control block, and etc.

Confidentiality Manipulation. Compromised drivers intercept bypassing information or access sensitive files, and send them out through network to remote unknown entities. For instance, compromised NIC driver intercepts all the incoming/outgoing packets and redirects them to attackers' machine.

Resource Starvation. Compromised drivers abuse critical resources and incur denial of service, e.g., dominating CPU by locking interrupts or exhausting memory by endless allocation request.

Since we assume OS kernel is fully trusted, we don't verify the control flow transition from OS kernel to driver code, nor audit the driver's data accessed by OS kernel. Furthermore, the function parameters and stack data passed between OS kernel and drivers are not verified currently, which could be leveraged by compromised drivers to tamper kernel integrity in certain ways. For instance, when calling a certain kernel API, attackers can launch a return-oriented attack to jump to other kernel functions through carefully crafted parameters. Such attack can indeed evade the auditing of Heter-device, but we believe that currently it requires significant manual efforts of attackers[2]. Finally, since Heter-device relies on underneath virtual platform to emulate heterogeneous devices for guest OS replicas, we assume the virtual machine monitor is in the trusted computing base. Exploiting bugs in virtual machine monitor, such as [3], and then controlling the guest OS are not in the scope of this paper.

[2] The most recent work [21] fully automates the instruction sequence construction that can be used by an attacker for malicious computations. However, the side-effect of the construction time (2009 ms) and the runtime overhead (135 times slower) will cause significant divergence on the logs of the two replicas, which will be caught by Heter-device as CPU resource abuse.

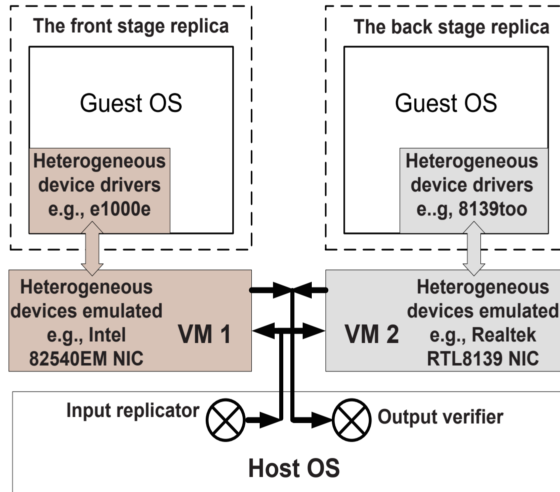

Fig. 1. Heter-device Software-based Diversity Architecture for Driver Assessment

3 Heter-device Design

In this section, we first describe the novel virtualized device diversity approach to efficiently produce driver-diversity-based OS replicas, then present Heter-device approach to evaluate drivers in multi-aspects.

3.1 Heter-device Architecture

Figure 1 shows our Heter-device architecture with the **front stage replica** and the **back stage replica** running as Guest OS atop the same Host OS. The device diversity is produced by virtual platform to emulate heterogeneous devices for the two VM replicas. The virtual platforms can run unmodified commodity OS by giving the Guest OS the illusion that it runs on top of "real" hardware. Thus, the guest OS will load corresponding drivers for the hardware devices that it regards as "real". In this way, the virtualized device diversity approach gains the same security benefits as costly hardware diversity in a much cheaper manner. For instance, software diversity approach enables two replicas to run on two separated emulated platforms, one with Intel 82540EM NIC (Network Interface Card), sound blaster 16 (sound card), Universal Host Controller Interface (USB controller) and etc., the other with Realtek RTL8139 NIC, Gravis ultrasound GF1(sound card), Intel Open Host Controller Interface (USB controller) and etc.

Our virtualized device diversity idea is inspired by both the hardware-based diversity approach and the sweeping deployment of virtual platforms (e.g., VMware, Xen, KVM, QEMU and etc.) in the production server environment. In this paper, we call the diverse devices with different models but performing the same functionality, e.g., Intel 82540EM NIC and Realtek RTL8139 NIC, as a pair of heterogeneous devices. As a result of pairs of heterogeneous devices emulated by virtual platform, the guest OS kernel of each replica will load heterogeneous

drivers correspondingly, e.g., e1000 or 8139too kernel modules[3]. Except heterogeneous drivers, the guest operating systems on the front stage VM replica and the back stage VM replica are exactly identical in terms of kernel version, installed applications and services, other loaded modules, start-up scripts, and etc.

The external input should be redirected to both the two replicas. We implement the input replication at the Host OS, totally transparent to the Guest OS replicas. Basically, every external input from network, keyboard, and mouse triggers both the device emulation modules of the two virtual machines. Since the input data from virtual disk is initialized by guest OS replicas, it does not need to be replicated. The output from the two guest OS replicas is intercepted and recorded by virtual machines. For instance, the network output traffic from each replica is audited and verified to capture any confidentiality tampering through network. During evaluation, we ensure that only the output from the front stage replica is sent out, while the output from the back stage replica is discarded, to guarantee the correctness of communication context.

3.2 Heter-device Approach

There exist several challenges to assess drivers based on Heter-device architecture, so we abstractly present our system design to tackle these challenges in the following.

Address-alias Correlation Through pre-configuration, both the front stage and back stage OS replicas can load root symbols (defined in *System.map* in Linux) into the same memory address. However, other dynamic kernel data may be loaded into different addresses, even if the data represents exactly the same semantics. For instance, with the same kernel version and configuration, the two OS replicas store the process descriptors of their root processes in the same address (pointed by the root symbol *init_task*). By traversing the double linked list of all the processes on the two replicas, we observe exactly the same process list, including kernel threads. However, the memory addresses storing all the other process descriptors, except the root process descriptor, do not match. Such address-alias of the same kernel data on the two replicas is prevalent and poses challenge to our auditing of kernel function calls and key kernel data accessed by evaluated drivers.

To tackle this challenge, we propose to correlate the address-alias kernel semantics of the front stage and back stage replicas. First, we need to reconstruct kernel semantics from raw physical memory of each replica respectively. Due to the challenge of reconstructing dynamic kernel data, such issue catches researchers' continuous attention recently, e.g., [22], [7], [15], [25], [14] and etc.

Heter-device efficiently integrates both the out-of-VM and in-VM approaches to comprehensively reconstruct kernel semantics. All kernel exported function pointers can be referred to from root symbol definition (*System.map* file), which is identical for both the replicas through default configuration. Some key kernel data can also be referred to in a similar way, with additional effort of recursive identification of kernel data structures. Regarding dynamic data of both kernel and drivers, we insert a fully trusted kernel module into both the front

[3] We focus our discussion on Linux operating system in this paper, but Heter-device is easily transported to other operating systems through reasonable efforts.

and back stage replicas, which notifies underneath virtual platforms about the allocation/reclaim of kernel memory and loading/unloading of kernel modules. With the reconstructed semantics, address-alias correlation recursively maps the addresses of the same kernel semantics, either function pointers or kernel data structures. Hence, it makes possible the efficient auditing and verification of heterogeneous drivers' execution.

Runtime Synchronization Running the front stage and back stage replicas at large may incur "out-of-band" comparison of heterogeneous drivers' execution on the two replicas. Though we delivered the replicated external input to the two replicas at the virtual machine monitor level simultaneously, the corresponding interrupt to CPU on each replica may not be "simultaneous". Thus the actual processing of the interrupt on the two replicas may still be "out-of-band". Researchers have proposed interrupt-redelivery approach for deterministic replay, e.g., [16], [36], and [38], which could be leveraged by Heter-device to apply the exact-replay-style synchronization. However, due to the heterogeneous driver diversity introduced by Heter-device, synchronizing such diverse replicas poses quite realistic challenges, such as different instruction execution sequences.

We observed that although the implementation of heterogeneous drivers is different, they offer the same function interfaces to OS kernel. Such layered design of most operating systems implies that OS kernel only needs to know how to invoke the device driver's methods, rather than to understand the detailed implementation of driver's methods. Figure 2 shows the interaction among NIC driver, OS kernel and other kernel modules. Besides function call returns, the control flow transition to NIC driver code must be through NIC interrupt handler or NIC driver function calls. For instance, NIC driver (Linux version) has totally 18 methods, with 8 fundamental (e.g., *open*, *stop*, *hard_start_xmit*, and etc.) and 10 optional (e.g., *poll*, *set_mac_address*, *change_mtu*, and etc.), indicating the operations that can be performed on this network card.

OS kernel declares the corresponding driver function pointers and initialize them during the loading of driver modules. As described in Figure 2, these driver functions are the only entry points for the control flow to transit from OS kernel or other kernel modules to this driver. Since OS kernel is fully trusted, it will not redirect control flow to arbitrary driver addresses except driver function call returns. More importantly, these entry points are identical for heterogeneous drivers despite different implementation details of the driver functions. Address-alias of the entry points on the two replicas can be resolved by correlating the addresses of them together. Thus, the two replicas can be synchronized by the entry points of the same device driver functions.

Hence, the auditing and verification of drivers' execution can be triggered by sensors monitoring the entry points. Specifically, when OS kernel's execution encounters an entry point, i.e., OS kernel calls a driver function, the context of current execution is recorded on the two replicas separately. Then, all the following instruction sequences of the two replicas are audited respectively, until the return to the previously logged context[4]. In particular, the entry point of the interrupt handler function deserves special attention, since nested interrupts (new

[4] Driver may also call kernel APIs during its execution. So the return to OS kernel address space does not suffice the end of driver's execution. Instead, only the return to the caller's execution context indicates the end.

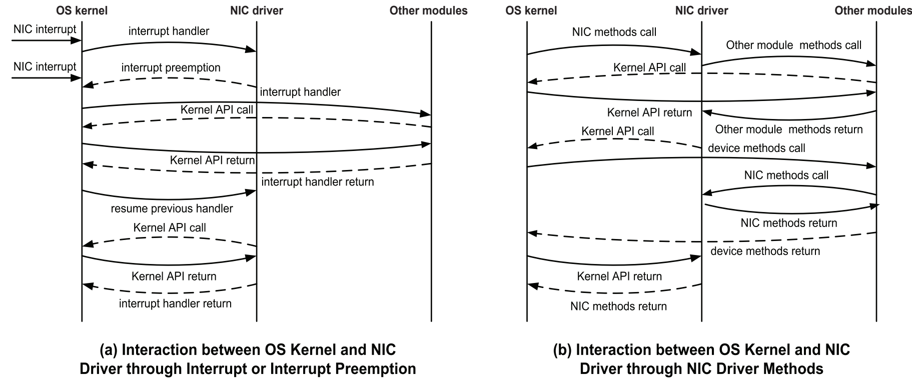

Fig. 2. NIC Driver Interaction with OS Kernel and Other Kernel Modules

interrupt comes during the processing of previous interrupt) may happen sometimes. Hence, each entry to driver's interrupt handler function is sequenced, and strictly matched to the corresponding return. In this way, the driver's execution can be identified apart from OS kernel's execution.

Kernel Integrity Mediation Compromised drivers can leverage the ultimate privilege to manipulate kernel integrity, e.g., hijacking control flow or tampering kernel data. Below, we present the approach of auditing such kernel integrity that could be tampered by compromised drivers based on Heter-device architecture.

Control Flow. For benign OS kernel and drivers, the control flow transitions among them can be well regulated by confining exported functions. For instance, Figure 2 shows that the control flow transition from OS kernel to drivers can be made by calling functions exported by drivers. Besides call return or hardware interrupt, the transition from benign drivers to OS kernel is through functions exported by OS kernel itself or other kernel modules which may call exported kernel APIs on behalf of the calling driver. Since we trust OS kernel, OS kernel only calls functions exported by drivers to transit the control flow. However, compromised drivers may directly jump to any address inside kernel or other modules to continue execution. Moreover, they can inject malicious code into OS kernel memory using DMA, and subvert function pointers on stack to the injected code, to hijack the control flow transition.

Existing researches, e.g., Gateway [28] and HUKO [35], prevent such control flow integrity manipulation by isolating address spaces of OS kernel and drivers. Legitimate entry points for execution transition from drivers to OS kernel are explicitly listed, e.g., system root symbols in Linux *System.map*). However, we observe that legitimate drivers may invoke some kernel APIs not defined in their legitimate entry point set. Rather, invoking kernel APIs frequently in their legitimate entry point set can also lead to denial of service attack through resource starvation. Furthermore, compromised drivers may inject malicious code into their own stack or heap to launch attack without violating control flow transition policies.

The runtime synchronization facilitates the fine-grained auditing of drivers' execution, from the call to driver's function to the corresponding return to caller. During the auditing of driver's execution, every jump or call out of driver's code address space is logged. These calls of kernel APIs or other kernel modules should be verified against the two replicas. Since the implementation of the heterogeneous drivers is different, strict verification of the sequence of their OS kernel API calls would always fail. However, to provide the same functionality, during assessment we observed a set of specific kernel APIs is frequently called by heterogeneous drivers. For instance, the kernel API calls made by NIC converge at irq locking/unlocking and memory allocation/deallocation.

We expect system engineers to manually analyze and verify the logged kernel API calls made by heterogeneous drivers within each specific driver function. Based on our experience, most system engineers (even those not quite familiar with OS kernel) have a sense of which set of kernel APIs are relevant in a specific driver function based on our cross-checking reference model. In addition, it is also relatively easy for them to capture some outlier kernel API calls through the pairwise comparison for further verification. For example, one volunteer system engineer at the first glance, pointed out that *rtl8139_open* makes a *kernel_thread* call, while *e1000_open* does not. Though such behaviour is finally verified as benign, we believe that such significant variance deserves further verification.

Besides the outlier kernel APIs, the kernel APIs that are called with an extremely high frequency deserve further verification as well. For instance, endlessly calling resource request kernel APIs will cause resource starvation as discussed in Section 3.2. Repeatedly calling *prepare_to_wait* kernel API will put all the runnable processes into sleep. Such malicious behaviour can be easily captured through the comparison of the amount of each kernel API calls within a specific driver function against heterogeneous drivers. Furthermore, benign drivers typically will only write data, rather than code, into their own stack/heap, or DMA-mapped kernel memory. Thus, any jump/call to the address within driver's stack/heap, or DMA-mapped kernel memory is strictly verified to check if such behaviour is general for the heterogeneous drivers to provide desired functionality. If not, further verification should be given to the driver that exhibited such suspicious behaviour.

Data Integrity. Drivers have the privilege to modify any kernel control-relevant or control-non-relevant data. Such modification by drivers should be strictly verified, rather than forbidden, since some of the modification might be legitimate to provide some desired functionality. For instance, previous Linux kernel does not export *set_current_state* API for drivers to change the state of a certain process. Instead, drivers have to directly set the state of current running process by *current->state = TASK_INTERRUPTIBLE*. However, compromised drivers may take advantage of those exported kernel APIs to hijack control flow, e.g., manipulating kernel control-relevant data, such as system call table, IDT, etc. A particular process can also be hidden by tampering kernel control-non-relevant data, such as pointers of double linked list processes.

Hence, we propose to verify the modification to key kernel data by heterogeneous drivers to capture any malicious manipulation. Beginning from the call to driver functions, copy-on-write is triggered on the memory storing the key kernel data on each replica, until the corresponding return to caller. Hence, the modification to key kernel data can be accounted to the corresponding drivers. However, driver's execution may be disrupted by a preempted interrupt, which is handled by OS kernel and corresponding interrupt handler. The modifications

during the preempted interrupt handling should be accounted to the driver that implements the interrupt handler function. These modification logs of each driver function are verified against heterogeneous drivers for any malicious manipulation.

We observe that most kernel data integrity manipulation is accompanied with control flow hijacking, which can be identified as discussed in control flow manipulation. Regarding pure data integrity manipulation, kernel developers or security engineers can provide a list of critical kernel data based on empirical experiences or referring to kernel critical data profiling [32]. Generally, when the amount of key kernel data to be verified becomes large, the runtime overhead and the false positive of Heter-device verification will become significant. Hence, we propose to select a subset of kernel integrity critical data as the verification candidate, e.g., system call table, IDT, critical function pointers in process descriptor, double linked list pointers, and etc.

Confidentiality and Resource Consumption Passive attacks, such as confidentiality tampering or resource abuse, are challenging to be identified or detected. Unfortunately, compromised drivers can leverage their ultimate privilege to maliciously intercept any data flow or repeatedly request any critical system resource, thus tampering confidentiality or crashing the system. Below, we discuss the methodology of relying on Heter-device architecture to capture such passive attacks during driver assessment, and present the framework we integrated into our approach.

Confidentiality. Compromised drivers can intercept bypassing data, call transmission (*hard_start_xmit* in Linux) function in network card driver, and send out to remote machines. Through control flow auditing and verification of heterogeneous drivers, the additional call to network card driver functions can be captured and identified as suspicious as discussed in Section 3.2. However, if network card driver itself is compromised, such data interception can be done totally within its own execution context, without any call to functions in other kernel modules. Although data flow auditing and verification of heterogeneous drivers can be added to capture the data interception, it would incur significant runtime overhead.

In this paper, we only focus on confidentiality leakage through network, that is, the intercepted data is transmitted to remote machines through network interface card. We assume that servers' running environment has strict physical access restrictions. Thus, it is out of our scope that the compromised drivers intercept the data and write it to local disk, which is then fetched through local access. Based on our assumption, we monitor and verify the network output of the front stage and back stage replicas for any confidentiality leakage. When deploying Heter-device architecture, OS kernel and service applications on the two replicas are identical, and the incoming packets to the emulated network cards are exactly replicated. So the output from the two replicas should be kept in rhythm unless anomaly happens. Hence, the output from the two replicas are matched with the combination of receiver's IP and packet sequence number. The additional traffic for information leakage from the compromised replica can be captured and alarmed.

Resource Consumption. Compromised drivers can launch various resource abuse attacks, and even cause denial of service due to resource starvation. Acquiring/releasing interrupt lock, allocating/freeing memory and etc., are benign

operations for most drivers to provide desired functionality. However, such legitimate operations may be leveraged by compromised drivers to launch CPU or memory starvation attacks. Certainly, it is infeasible to restrict these kernel APIs from drivers, because benign drivers may not work or malfunction. Heter-device captures such resource abuse attack by strictly auditing and verifying the resource request kernel APIs issued by heterogeneous drivers. Although different implementation of heterogeneous drivers may cause variance in system resource consumption, we believe significant variance must indicate suspicious driver, at least inefficient implementation of the driver. System engineers can easily set up a threshold of such variance to alarm resource abuse. Based on our experience, such variance threshold can be set from 5 % to 15 % based on various context for reasonable false negative and false positive.

4 Implementation

In this section, we present the implementation of Heter-device framework. We begin with Heter-device architecture deployment based on QEMU open source project, followed by address-alias correlation for the heterogeneous drivers based replicas. At last, we describe the implementation of the fine-grained mediation of heterogeneous drivers' execution.

4.1 Heter-device Deployment

Software Diversity Architecture Instead of deploying the replicas on costly real heterogeneous devices platforms, we implement our Heter-device architecture using QEMU, with one virtual machine as the front stage replica, and the other one as the back stage replica. We configure the virtual machine (QEMU) to emulate heterogeneous devices for the two replicas, i.e., one with Realtek RTL8139 NIC and Gravis ultrasound GF1, the other with Intel 82540EM NIC and sound blaster 16. Although other heterogeneous devices options are also available, e.g., USB, video card and etc., we believe heterogeneous network cards and sound cards are sufficient to demonstrate the proof-of-concept of Heter-device.

The disk image file is replicated for the two replicas to ensure the same guest operating system (with kernel version 2.6.15), service applications, configurations, startup scripts, and etc. Moreover, the two replicas are configured with the same amount of memory and networking model. Hence, the only difference between the two replicas is heterogeneous drivers, which interact with underneath heterogeneous devices. During the assessment of heterogeneous drivers, Heter-device serves as "honeypot" to trigger either the inherently malicious drivers or remote exploitation to drivers' vulnerabilities.

Input Replication and Output Verification To implement the external input replication to the two replicas, we insert a small piece of replication code into the host operating system kernel. Whenever there is any external input, i.e., keyboard, mouse, network packet, to the front stage replica, the inserted code on host OS kernel replicates the input and notifies both the two replicas for incoming events. Since the virtual machine we use (QEMU) behaves as a user process on host OS, the notification can be done either by signal or bit

masking based on the context. In contrast, the network output from each replica is logged by the emulated network card of each virtual machine. On host OS, we implement a verification process examining the logs from the two replicas. In particular, it extracts the destination IP, sequence number information from each packet and matches the corresponding packets from the two replicas. A threshold of the amount of unmatched packets can be pre-determined, to alarm any confidentiality leakage.

Synchronization of Replicas We rely on both virtual machine and guest OS support to synchronize the front stage and back stage replicas at the granularity of driver function calls. We assume that host OS runs on multi-core hardware platform, thus each virtual machine is configured to run on a dedicated core for maximum CPU capacity. We craft a trusted kernel module to monitor the loading of heterogeneous drivers on each replica. For instance, it audits the procedure of initializing functions declared by OS kernel, e.g., the interrupt handler function and other functions registered by the driver. The memory addresses of these functions are sent to underneath virtual machine through a secure channel. Only the functions implemented by both heterogeneous drivers are selected as synchronization points for the two replicas. During runtime, the value of register *eip* is monitored on both replicas to capture the synchronization points, as discussed in Section 4.3.

4.2 Address-alias Correlation

We focus our implementation on Linux OS and start from a set of root symbols in *System.map* file. Since operating systems on the front stage and back stage replicas are with the same kernel version and configuration, the symbols and their addresses in *System.map* are exactly the same. Then we apply a *CIL* module [27] on the source code of guest OS kernel to automatically extract type definitions of kernel data structures. Finally, beginning from each correlated root symbol, we recursively reconstruct memory semantics based on type definitions of kernel data structures, and correlate the same data structure with address-alias.

In particular, in order to correlate *task_struct* of each process, we start from the data structure *CPUState* defined by QEMU to emulate the processor for virtual machine. All the CPU registers can be referred to through the instance of *CPUState*: *env*. From the register *tr*, we locate the kernel stack of the currently running process. At the bottom of kernel stack resides the *thread_info* structure, which includes a pointer to the *task_struct* of the corresponding process. By traversing the double linked list processes through the *task* pointers on the two replicas, we can obtain all the process descriptors and correlate their addresses together.

4.3 Fine-Grained Driver Execution Mediation

QEMU is a binary translation based virtual machine, which facilitates fine-grained auditing of guest OS execution. Instead of instruction-by-instruction translation, QEMU implements translation block to improve performance. Specifically, QEMU generates host code from a piece of guest code without control flow redirection or static CPU state modification. Thus, for each translation block, guest OS executes without QEMU intervention unless interrupt occurs. At the

end of each translation block, QEMU takes over the control and prepares for the next translation block.

The translation block mechanism provides a perfect mediation approach for drivers' execution. We can audit the program counter at the beginning of each translation block, which represents a control flow redirection, including *return*, *jump*, or *call*, etc. In this way, the entry into or the leave from driver's code section can be recorded efficiently without monitoring every executed instruction. However, QEMU also implements translation block chaining for performance boost. In particular, every time when a translation block returns, QEMU tries to chain it to previous block, thus saving the overhead of context switch to QEMU emulation manager. The translation block chaining indeed poses challenges for our control flow redirection mediation, since QEMU emulation manager may miss some redirections, i.e., some transitions between OS kernel and drivers.

In order to tackle this challenge, we trace down through the translation block chain whenever QEMU emulation manager begins a new translation block. QEMU defines *TranslationBlock* data structure for each translation block, where we can locate the program counters of this block and the next one along the chain. Hence, we can traverse the chain till the end to audit and record the program counter at the each control flow redirection. However, key kernel data cannot be recorded in this way since detailed execution context has not been established yet during the pre-traversing of translation block chain. In order to preserve the performance and key kernel data integrity, we mark the memory regions storing those key kernel data as non-writeable. Any attempt to write to the memory will be trapped to QEMU manager, and validated against the heterogeneous drivers. Currently, we assume that key kernel data always involves some static code, data, critical function pointers, and etc.

5 Evaluation

In this section, we present experimental results on Heter-device framework in three aspects. First, we present the comparison results on OS kernel APIs called by different functions of heterogeneous drivers. Second, we show the effectiveness of Heter-device in capturing compromised drivers by two case studies. Last, we evaluate the performance overhead incurred by Heter-device approach. The host OS is Ubuntu 10.10 with kernel version 2.6.35, and both of the two guest operating systems are installed with Fedora 5 (kernel version 2.6.15). We choose qemu-0.12.5 as the virtual machine monitor emulating two virtual platforms: one with Realtek RTL8139 NIC and Gravis ultrasound GF1, the other with Intel 82540EM NIC and sound blaster 16.

5.1 Profiling Heterogeneous Drivers

First, we load Heterogeneous NIC drivers *e1000* and *8139too* on the two replicas running Intel 82540EM NIC and Realtek RTL8139 NIC respectively. Our trusted kernel module monitors *alloc_netdev* function to trace the newly allocated *net_device* structure for the network card. Then the function pointers in *net_device*, such as *open*, *stop*, *hard_start_xmit*, etc., are audited during the initialization of NIC drivers to obtain the addresses of these driver functions. The functions implemented by both heterogeneous drivers are correlated as the synchronization entries of the two replicas. We start to audit the control flow transition between OS kernel and NIC driver since the booting of the two replicas.

Then we trigger a set of user commands (e.g., ssh, sftp, ping, and etc.) and applications (e.g., Firefox, Filezilla, and etc.), which involve network card operations, to invoke the interaction between OS kernel and NIC driver. Simultaneously, the kernel API calls issued by each synchronized function of heterogeneous drivers are profiled.

Table 1 shows our profiling results of heterogeneous drivers *e1000* and *8139too*. Although implemented by different teams, the same functions of heterogeneous drivers typically invoke a similar set of kernel APIs. In particular, we find that *8139too* calls *kernel_thread* kernel API in *open* function. Thus, we monitor the forked kernel thread and observe the following kernel APIs invoked during the thread's lifetime: *daemonize*, *allow_signal*, *interruptible_sleep_on_timeout*, *refrigerator*, *flush_signal*, *rtnl_lock_interruptible*, *rtnl_unlock*, and *complete_and_exit*. Table 1 also indicates that previous works will generate lots of false positive when referring to exported functions in *System.map* as trusted entries from drivers to OS kernel[5].

5.2 Case Study 1: Kernel Integrity Manipulation

We refer to the implementation of *adore-ng* kernel rootkit, and integrate its malicious code into the function *snd_gf1_stop_voice* of *gus* (for Gravis ultrasound GF1) driver. When users try to turn off the audio, the injected code gets executed to replace the functions of *readdir*, *lookup*, and *get_info* with its own implementation to hide files, processes and ports. The newly generated driver *gus* is recompiled and loaded into OS kernel. In contrast, driver *sb16* (for sound blaster 16) remains unchanged.

During the assessment of drivers *gus* and *sb16*, we simulate user's command to turn off the audio, which is replicated to both replicas. The modification of those static kernel data (function pointers) by driver *gus* is observed and alarmed, while driver *sb16* does not exhibit such behaviour. Then we clear this alarm, let the two replicas run forward, and issue process and file listing commands. We observe that the control flow transition from OS kernel to driver *gus* code section through unrecognised entry. Afterwards, driver *gus* calls kernel APIs, i.e., *readdir*, *lookup*, and *get_info*, from its execution context. In contrast, driver *sb16* on the other replica is not involved in the process and file listing procedures.

5.3 Case Study 2: Resource Abuse and Confidentiality Tampering

With the kernel privilege of compromised driver, attackers can launch resource starvation attack to reduce the productivity of the victim systems, or tamper confidentiality by intercepting bypassing data. We simulate resource abuse by inserting malicious code into the source code of RTL8139 NIC driver. In particular, after *spin_lock* is called in function *rtl8139_interrupt*, repeated call of *alloc_skb* is issued until kernel memory is overwhelmed. Then the driver is recompiled and loaded into OS kernel as *8139too* module. We repeat a subset of the user commands and applications in Section 5.1. During the assessment, after the synchronization of *interrupt_handler* function entry, *e1000_intr* quickly returns. However, *rtl8139_interrupt* continues running with lots of *alloc_skb* calls

[5] Similar profiling has been performed on heterogeneous sound card drivers *gus* (for Gravis ultrasound GF1) and *sb16* (for sound blaster 16). The profiling results are excluded due to page restriction.

Table 1. Kernel APIs called by different functions in *e1000* and *8139too*. For each synchronization function, the upper box contains the invoked kernel APIs defined in *System.map* file of guest OS, while the lower box includes the indirected invoked kernel APIs that are called by drivers through the following procedure. Drivers call some other *extern* kernel functions (not defined in *System.map* file) by including some *.h* files, and these functions in turn invoke the indirected kernel APIs identified by us.

Synch. Entry	Kernel APIs by *e1000*	Kernel APIs by *8139too*
*open	request_irq, mod_timer, kmalloc, pci_clear_mwi, vmalloc_node	netif_carrier_on, netif_carrier_off, request_irq, spin_unlock_irqrestore, spin_lock_irqsave, kernel_thread
	dma_alloc_coherent, __alloc_skb, __aloc_pages	dma_alloc_coherent
*stop	free_irq, netif_carrier_off, mmset, vfree, kfree	free_irq, wait_for_completion, kill_proc, spin_lock_irqsave, spin_unlock_irqrestore
	netpoll_trap, dma_free_coherent, lock_timer_base, list_del, __kfree_skb, local_irq_save, local_irq_restore	netpoll_trap, dma_free_coherent
*interrupt_handler	spin_lock, spin_unlock, eth_type_trans, netif_rx	spin_lock, spin_unlock
	__alloc_skb, netpoll_trap, kree_skb local_irq_save, local_irq_restore	netpoll_trap, local_irq_save, local_irq_restore
*tx_timeout	schedule_work	spin_lock, spin_lock_irqsave, spin_unlock, spin_unlock_irqrestore
	spin_lock, spin_lock_irqsave, __wake_up	
*do_ioctl	request_irq, spin_unlock_irqrestore free_irq, spin_lock_irqsave, netif_carrier_off, mod_timer	spin_lock_irq, spin_unlock_irq
	lock_timer_base, list_del, __kfree_skb, local_irq_save, local_irq_restore netpoll_trap	capable
*hard_start_xmit	spin_trylock, spin_unlock_irqrestore, pskb_pull_tail, pskb_expand_head	spin_lock_irq, spin_unlock_irq
	local_irq_save, netpoll_trap, local_irq_restore	__kfree_skb, netpoll_trap
*poll	spin_lock, spin_unlock, disable_irq enable_irq, netif_carrier_ok	spin_lock, spin_unlock, netif_receive_skb
	local_irq_save, __kfree_skb, local_irq_restore	local_irq_disable, __alloc_skb local_irq_enable, list_del, local_irq_save, local_irq_restore
*set_multicast_list		spin_lock_irqsave, spin_unlock_irqrestore
*get_stats	spin_lock_irqsave, spin_unlock_irqrestore	spin_lock_irqsave, spin_unlock_irqrestore

Table 2. Runtime Performance of Different Benchmarks

Benchmark	Key Kernel Data	Whole Kernel
LMbench	*1.2021*	*1.2444*
Apache Benchmark	*1.4420*	*2.8273*
Interbench	*1.3356*	*1.4160*
Kernel Decompression	*1.1663*	*1.2262*

recorded. Our verification alarms such anomaly immediately with a pre-defined difference threshold (200 in our experiment) reached.

Furthermore, we also simulate confidentiality tampering attack by injecting malicious code into the packet transmission function *e1000_xmit_frame* of e1000 NIC driver. The newly compiled *e1000* module will intercept all the outgoing packets and redirect them to a remote machine. During the assessment, we replicate *Apache http* servers on both the two replicas, and simulate continuous client requests to them on another machine. The verification on the front-tier proxy matches the output packets from the two replicas. An alarm is signalled when the amount of unmatched packets from the replica with *e1000* module reaches the pre-defined threshold (20 in our experiment) in two minutes.

5.4 Performance Evaluation

The runtime overhead of Heter-device highly depends on the amount of key kernel data that needs to be verified. Table 2 shows the performance (the ratio of Heter-device execution and QEMU native execution) of Heter-device architecture based on several benchmarks. By key kernel data protection, we only verify static key kernel data, including system call table, IDT, root symbols in *System.map* files. In contrast, by whole kernel protection, the entire kernel address space is verified by Heter-device during driver assessment. During each round of evaluation, both the heterogeneous NIC and sound card drivers are verified for fine-grained control flow transition.

We use *LMbench* to evaluate the pipe bandwidth, and also evaluate the time consumed to decompress Linux kernel *3.0* as shown in Table 2. Since both of them involve little interaction with either NIC or sound card, the pipe bandwidth and CPU capacity are mostly retained. We run *Apache Benchmark* to evaluate the network performance, and *Interbench* to evaluate the audio performance. Table 2 demonstrates that network throughput drops more significantly than audio performance. We think the main reason is that NIC drivers interact more frequently with OS kernel during packet transmission than sound card drivers do during audio playing.

6 Discussion and Future Work

In this section, we discuss limitations and future work of Heter-device. First, although Heter-device architecture is totally compatible with most existing fault tolerant systems, deploying Heter-device approach as a real-time compromised driver detection system requires performance boost. Due to performance overhead (largely incurred by QEMU), Heter-device is currently deployed as honeypot to assess drivers before they are put into use in server systems. Hence,

it should be our future work to facilitate hardware support, such as Intel VT or EPT techniques into our Heter-device architecture to feasibly detect compromised drivers in responsive server environment.

Second, currently QEMU supports limited number of emulated devices. Most existing device emulation modules in virtual machines such as virtualbox, Xen and KVM are all based on QEMU. Hence, assessing other drivers, e.g., keyboard, mouse and etc., is impossible right now on Heter-device architecture. Aware of our design, attackers can craft malicious drivers only for those devices that have not been emulated by QEMU. We suggest that system engineers consider the devices that can be emulated by QEMU right now, thus facilitating the driver assessment of Heter-device architecture. As the renaissance of virtualization, we hope that such device emulation options will bloom in the near future, making Heter-device more general and practical.

Third, there exist some counter-attacks to Heter-device architecture. As a honeypot or testing approach, Heter-device cannot be claimed to be able to capture any malicious code or vulnerability of drivers. There is always the possibility that the malicious code is not detected because the environment or the workload did not trigger it. Furthermore, transparently translating instructions [13] and faithfully emulating hardware devices are challenging tasks. For instance, QEMU can be detected by various methods as discussed in [20]. Aware of our design, attackers can craft malicious code that first examines whether it runs on emulated platforms. If so, the malicious code will "keep silent" to avoid being detected or profiled. Otherwise, it will compromise the victim system. Hence, the compromised drivers with "split-personality" can generally evade the auditing and verification of Heter-device.

Last, existing Heter-device approach involves manual intervention during the driver assessment. For instance, key kernel data to be recorded and verified should be provided by system engineers in advance, though we also offer a candidate list. Moreover, the verification procedure (i.e., control flow transition verification) requires system engineers to investigate the variance to reduce false positive. Furthermore, such manual inspection can also help to determine which driver is compromised, since the two replicas serve as reference model for each other rather than always treating one as golden standard. Our future work is set to comprehensively profile the driver's behaviour, thus improving the automation by providing more general key kernel data verification list and enforcing more detailed verification policies.

7 Related Work

In this section, we will first briefly review the state-of-art diversity techniques, and then discuss prior approaches protecting kernel integrity or reliability from driver faults or bugs.

Diversity approach. Software diversity approach for intrusion detection has been studied in several works, such as COTS [31], Behavioural Distance [19], Diversified Process Replica [9], Detection of Split Personalities [8] and DRASP [39]. COTS Diversity [31] and DRASP[39] applies N-version programming into web servers to verify their interactions with the environment for any anomaly, e.g., HTTP responses from those web servers. Generally, [8] and [39] are ideal to detect anomaly targeting web servers, especially for most promiscuous attacks. But for other attacks, such as denial of service attack, or resource abuse, comparing network packets cannot work, since such attacks does not involve additional

network packet transmission. On the other hand, comparing network packets is not always possible. For instance, if the payload is encrypted through IPsec, comparing the payload is meaningless.

Behavioural Distance [19] and Detection of Split Personalities [8] aim to detect intrusion or anomaly by comparing the system call sequences made by diverse applications. Diversified Process Replica [9] proposes non-overlapping processes address spaces to defeat memory error exploits. Although all the above approaches are effective in detecting compromised applications, the response or system call comparison schemes cannot be applied to diverse drivers as in Heter-device.

The seminal work N-variant [12] proposes address space partition and instruction set tag diversities to detect divergences caused by intrusions. Although such approaches are quite effective in detecting code injection related attacks, other types of exploitation, such as direct kernel object manipulation, kernel APIs abuse and confidentiality tampering can evade N-variant's auditing. Heter-device proposes heterogeneous device based diversity design, and specially focuses on the auditing and verification of drivers to cover multiple attack vectors that could be leveraged by compromised drivers.

Isolation based protection. Isolation-based approach continues drawing researchers' attention to protect OS kernel from buggy drivers for years. Nooks ([30] and [29]) pioneers the driver isolation approach to protect OS reliability from driver failures. Mondrix [34] integrates hardware support to isolate kernel modules by memory protection domains. Virtual machine technique has also been applied to isolate OS from buggy drivers. For instance, [24] and [17] isolate drivers by running a subset of untrusted drivers in a separated OS/VM domain, thus achieving both driver reuse and isolation.

Moreover, efficient address space isolation approaches have been proposed to protect kernel integrity [35] or monitor kernel APIs issued by untrusted kernel extensions [28]. Neither HUKO [35] and Gateway [28] is not comprehensive to assess drivers, because it doesnt cover kernel data integrity, confidentiality manipulation and resource starvation attacks as Heter-device. Instead of address space isolation, Heter-device audits drivers' execution at a finer granularity: instruction sequences. Furthermore, besides kernel control flow or data integrity protection which are the primary focus of previous approaches, Heter-device also proposes feasible approach to capture confidentiality tampering and resource abuse attacks launched by malicious drivers.

User mode driver based protection. User mode device driver framework has been proposed recently to de-grant driver code's privilege, thus ensuring the kernel's integrity ([4], [5], [23]). However, they either suffer from significant performance degradation ([6], [26]), or require complete rewriting of driver code and modifications to OS kernel ([5], [23]). Concerning the performance issue, Microdrivers [18] present a novel approach to split driver code into both user mode and kernel mode execution, with only performance-sensitive code remaining in the kernel. Based on Microdriver, RPC monitor [10] protects kernel from vulnerable driver by mediating all data transfers from untrusted user-mode execution to kernel-mode execution to preserve kernel integrity. In addition, the reference validation mechanism can be integrated into Microkernels (e.g., Nexus [33]) to effectively audit driver's behaviour against safety specification. In general, Heter-device requires no rewriting of existing drivers, and is applicable to most commodity operating systems.

8 Conclusion

In this paper, we present a novel diversity based honeypot, Heter-device, to assess the trustworthiness of drivers from multiple aspects, including kernel integrity manipulation, resource starvation and confidentiality tampering. Heter-device relies on virtual platforms to emulate heterogeneous devices for guest operating systems, and correspondingly produce driver-diverse replicas. The diverse replicas are deployed as honeypot to audit and verify the heterogeneous drivers' execution by placing synchronization and monitoring "sensors". We also propose automatic address-alias correlation, a subset of kernel data for default integrity protection, and a set of policies to defeat resource abuse and confidentiality tampering. The case studies show that Heter-device can capture various kernel integrity manipulation, resource starvation, and confidentiality tampering launched from compromised drivers, thus delivering the trustworthy drivers after assessment.

Acknowledgment

This work was supported by AFOSR FA9550-07-1-0527 (MURI), ARO W911NF-09-1-0525 (MURI), NSF CNS-0905131, NSF CNS-0916469, and ARO W911NF 1210055.

References

1. QEMU, open source processor emulator, http://wiki.qemu.org/Main_Page.
2. Stuxnet, http://en.wikipedia.org/wiki/Stuxnet.
3. Vulnerability Summary for CVE-2008-1943: Buffer overflow in the backend of XenSource Xen Para Virtualized Frame Buffer, http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-1943.
4. Windriver cross platform device driver development, Technical report, Jungo Corporation, 2002. http://www.jungo.com/windriver.html.
5. Architecture of the user-mode driver framework, Version 1.0, Microsoft, 2007.
6. Francois A.: Give a process to your drivers. EurOpen, 1991.
7. Arati B., Vinod G., Liviu I.: Automatic Inference and Enforcement of Kernel Data Structure Invariants. 24th ACSAC, 2008.
8. Davide B., Marco C., Christoph K., Christopher K., Engin K., Giovanni V.: Efficient Detection of Split Personalities in Malware. NDSS, 2010.
9. Danilo B., Lorenzo C., Andrea L.: Diversified Process Replicae for Defeating Memory Error Exploits. IEEE International Performance, Computing, and Communications Conference, 2007.
10. Shakeel B., Vinod G., Michael M. S., Chih-Cheng C.: Protecting Commodity Operating System Kernels from Vulnerable Device Drivers. ACSAC, 2009.
11. Jim C., Ben P., Tal G., Kevin C., Mendel R.: Understanding data lifetime via whole system simulation. USENIX Security Symposium, 2004.
12. Benjamin C., David E., Adrian F., Jonathan R., Wei H., Jack D., John K., Anh N., Jason H.: N-variant systems: A secretless framework for security through diversity. USENIX Security Symposium, 2006.
13. Artem D., Paul R., Monirul S., Wenke L.: Ether: malware analysis via hardware virtualization extensions. 15th ACM CCS, 2008.
14. Brendan D., Tim L., Michael Z., Jonathon G., Wenke L.: Virtuoso: Narrowing the Semantic Gap in Virtual Machine Introspection. IEEE Security and Privacy Symposium, 2011.

15. Brendan D., Abhinav S., Patrick T., Jonathon G.: Robust signatures for kernel data structures. 16th ACM CCS, 2009.
16. George W. D., Samuel T. K., Sukru C., Murtaza A. B., Peter M. C.: Revirt: enabling intrusion analysis through virtual-machine logging and replay. OSDI, 2002.
17. Ulfar E., Tom R., Ted W.: Virtual Environments for Unreliable Extensions. Technical Report MSR-TR-2005-82, Microsoft Research, 2005.
18. Vinod G., Matthew J. R., Arini B., Michael M. S., Somesh J.: The design and implementation of microdrivers. 13th ASPLOS, 2008.
19. Debin G., Michael K. R., Dawn S.: Behavioral Distance for Intrusion Detection. RAID, 2005.
20. Tal G., Keith A., Andrew W., Jason F.: Compatibility is not transparency: VMM detection myths and realities. 11th USENIX HotOS, 2007.
21. Ralf H., Thorsten H., Felix C. F.: Return oriented rootkits: Bypassing kernel code integrity protection mechanisms. USENIX security symposium, 2009.
22. Xuxian J., Xinyuan W., Dongyan X.: Stealthy Malware Detection Through VMM-Based 'Out-of-the-Box' Semantic View Reconstruction. 14th ACM CCS, 2007.
23. Ben L., Peter C., Nicholas F., Stefan G., Charles G., Luke M., Daniel P., Yueting S., Kevin E., Gernot H.: User-level device drivers: Achieved performance. Journal of Computer Science and Technology, vol. 5, pp. 654–664, 2005.
24. Joshua L., Volkmar U., Jan S., Stefan G.: Unmodified device driver reuse and improved system dependability via virtual machines. 6th OSDI, 2004.
25. Zhiqiang L., Junghwan R., Xiangyu Z., Dongyan X., Xuxian J.: SigGraph: Brute Force Scanning of Kernel Data Structure Instances Using Graph-based Signatures. 18th NDSS, 2011.
26. Kevin T. Van M.: The Fluke device driver framework. Master's thesis, University of Utah, 1999.
27. George C. N., Scott M., Shree P. R., Westley W.: CIL: Intermediate language and tools for analysis and transformation of C programs. International Conference on Compiler Construction, 2002
28. Abhinav S., Jonathon G.: Efficient Monitoring of Untrusted Kernel-Mode Executio. 18th NDSS, 2011.
29. Michael M. S., Muthukaruppan A., Brian N. B., Henry M. L.: Recovering Device Drivers. 6th OSDI, 2004.
30. Michael M. S., Brian N. B., Henry M. L.: Improving the reliability of commodity operating systems. 19th SOSP, 2003.
31. Eric T., Frederic M., Ludovic M.: COTS diversity based intrusion detection and application to web servers. RAID, 2005.
32. Zhi W., Xuxian J., Weidong C., Xinyuan W.: Countering Persistent Kernel Rootkits Through Systematic Hook Discovery. RAID, 2008.
33. Dan W., Patrick R., Kevin W., Emin Gn S., Fred B. S.: Device Driver Safety Through a Reference Validation Mechanism. 8th OSDI, 2008.
34. Emmett W., Krste A.: Memory isolation for Linux using Mondriaan memory protection. 12th SOSP, 2005.
35. Xi X., Donghai T., Peng L.: Practical Protection of Kernel Integrity for Commodity OS from Untrusted Extensions. 18th NDSS, 2011.
36. Min X., Vyacheslav M., Jeffrey S., Ganesh V. Boris W.: Retrace: Collecting execution trace with virtual machine deterministic replay. 3rd MoBS, 2007.
37. Heng Y., Dawn S., Manuel E., Christopher K., Engin K.: Panorama: capturing system-wide information flow for malware detection and analysis. 14th ACM CCS, 2007.
38. Shengzhi Z., Xiaoqi J., Peng L., Jiwu J.: Cross-Layer Comprehensive Intrusion Harm Analysis for Production Workload Server Systems. 26th ACSAC, 2010.
39. Shengzhi Z., Peng L.: Letting Applications Operate through Attacks Launched from Compromised Drivers. AsiaCCS, 2012.