Counteracting Data-Only Malware with Code Pointer Examination

Thomas Kittel ✉, Sebastian Vogl, Julian Kirsch, and Claudia Eckert
{kittel|vogls|kirschju|eckert}@sec.in.tum.de
Technische Universität München, Germany

Abstract. As new code-based defense technologies emerge, attackers move to data-only malware, which is capable of infecting a system without introducing any new code. To manipulate the control flow without code, data-only malware inserts a control data structure into the system, for example in the form of a ROP chain, which enables it to combine existing instructions into a new malicious program. Current systems try to hinder data-only malware by detecting the point in time when the malware starts executing. However, it has been shown that these approaches are not only performance consuming, but can also be subverted.
In this work, we introduce a new approach, Code Pointer Examination (CPE), which aims to detect data-only malware by identifying and classifying code pointers. Instead of targeting control flow changes, our approach targets the control structure of data-only malware, which mainly consists of pointers to the instruction sequences that the malware reuses. Since the control structure is comparable to the code region of traditional malware, this results in an effective detection approach that is difficult to evade. We implemented a prototype for recent Linux kernels that is capable of identifying and classifying all code pointers within the kernel. As our experiments show, our prototype is able to detect data-only malware in an efficient manner (less than 1% overhead).

Keywords: VMI; introspection; CFI; CPI; CPE; Pointer Examination; OS Integrity; Linux; kernel; data-only malware

1 Introduction

Malware is without doubt one of the biggest IT security threats of our time. This is especially true for kernel-level malware, which runs at the highest privilege level and is thus able to attack and modify any part of the system, including the operating system (OS) itself. However, even kernel-level malware has a weakness that is well-suited for its detection: in order to execute, the malware has to load its malicious instructions onto the victim's system and thereby effectively change its codebase. This makes current kernel-level malware vulnerable to code integrity-based defense mechanisms, which prevent or detect malicious changes to the code regions of the system. It is not surprising that validating the integrity of the system's code regions became a key approach to counteract malware. In the meantime, commodity OSs employ a multitude of mechanisms that protect the system's codebase (e.g. W$\oplus$X, secure boot, etc.) and researchers presented sophisticated Code Integrity Validation (CIV) frameworks that are capable of reliably and efficiently detecting malicious changes to the code regions of userspace programs [20] as well as modern OS kernels [17].

As code integrity mechanisms become more and more widespread, attackers are forced to find new ways to infect and control a system. A likely next step in malware evolution is thereby *data-only malware*, which solely uses instructions that already existed before its presence to perform its malicious computations [14]. To accomplish this, data-only malware employs code reuse techniques such as return-oriented programming (ROP) or jump-oriented programming (JOP) to combine existing instructions into new malicious programs. This approach enables the malware form to evade all existing code-based defense approaches and to persistently infect a system without changing its codebase [30]. Despite this capability and the substantial risk associated with it, there only exist a handful of countermeasures against data-only malware so far, and those can often be easily circumvented [6,9,13,24].

In this paper, we explore a new approach to the detection of data-only malware. The key idea behind this approach is to detect data-only malware based on "malicious" pointers to code regions. For simplicity we refer to them as code pointers. Similar to traditional malware, data-only malware has to control which reused instruction sequence should be executed when. To achieve this, data-only malware makes use of a control structure that contains pointers to the instructions that should be (re)used. This control structure can essentially be seen as the "code region" of the data-only program that the malware introduces. By identifying malicious code pointers in memory, we in essence aim to apply the idea of code integrity checking to the field of data-only malware by detecting malicious control data within the system. For this purpose, we introduce the concept of Code Pointer Examination (CPE).

The idea behind CPE is to identify and examine each possible code pointer in memory in order to classify it as benign or malicious. This is essentially a two-step process: In the first step, we iterate through the entire memory of the monitored machine with a byte by byte granularity in order to identify all code pointers. In the second step, we classify the identified code pointers based on heuristics. As our experiments showed, this approach results in an effective and high-performance (less than 1% overhead) detection mechanism that can detect data-only malware and is well-suited for live monitoring as well as forensic investigations.

Since the OS is the integral part of the security model that is nowadays used on most systems, we focus our work primarily on the Linux *kernel.* We chose this OS, since it is open and well documented, which makes it easier to understand and reproduce our work. However, the concepts and ideas that we present are equally applicable to userspace applications and other OSs such as Windows.

In summary we make the following contributions:

- We present CPE, a novel approach to identify and classify code pointers in 64-bit systems.
- We highlight data structures that are used for control flow decisions in modern Linux kernels and thus must be considered for control flow validation.
- We provide a prototype implementation and show that it is both effective and efficient in detecting control structures of data-only malware.

2 Background

In this section we discuss foundations required for the rest of the paper.

Protection Mechanisms. Intel provides two new protection mechanisms to make it significantly harder for an attacker to introduce malicious code or data into the kernel. The first protection mechanism is Supervisor Mode Execution Protection (SMEP). SMEP ensures that only code that is marked as *executable* and *supervisor* is executed in kernel mode. In particular, if the CPU is trying to fetch an instruction from a page that is marked as a *user* page while operating with a Current Privilege Level (CPL) that is equal to zero, SMEP will generate a protection fault. SMEP is usually used together with the No-eXecute (NX) bit, which marks a page as not executable.

The second protection mechanism is Supervisor Mode Access Prevention (SMAP). This feature can basically be seen as SMEP for data; it raises a fault if data that is marked as *user* in the page tables is accessed within the kernel. With both of these features enabled, the kernel is thus unable to access any userspace memory. In combination, this significantly reduces the amount of memory that is usable as gadget space for an attacker.

Runtime Code Validation. A key idea that our work builds upon is runtime code validation. While code-based defense mechanisms such as W$\oplus$X and secure boot ensure the integrity of code at load time, runtime code validation guarantees that all code regions of a system are coherent and valid at any point in time during its execution [17]. For this purpose, the code of the protected system is constantly monitored and the legitimacy of all observed changes is verified. As a result, any modification or extension of the existing codebase can be detected and prevented. To illustrate this, we briefly describe the runtime code validation framework presented by Kittel et al. [17], which serves as a foundation for this work.

Kittel et al. created a runtime code validation framework that is capable of reliably validating the integrity of all kernel code pages at runtime. To isolate the monitoring component from the protected system, the proposed system makes use of virtualization. Once monitoring begins, the validation framework first iterates through the page tables of the system to obtain a list of all executable supervisor pages. Since the page tables are the basis for the address translation conducted by the underlying hardware, this approach effectively enables the framework to reliably determine which memory regions are marked as executable and could thus contain instructions.

In the next step, the monitor obtains the list of loaded kernel modules from the monitored system using virtual machine introspection (VMI). Based on this information the framework simulates the loading process of each of the modules as well as the kernel image to obtain a trusted and known-to-be-good state of the code regions that can later on be compared to the current state of the code regions. To accomplish this, the framework requires access to a trusted store that contains all modules as well as the kernel binary that are executing in the monitored system. This trusted store is implemented by storing all trusted binary files within the hypervisor.

Once the loading process has been simulated, the trusted code pages contain all load time changes that the kernel applies. However, modern kernels may also patch code regions at runtime in order to increase compatibility and performance. As a result, the trusted code pages may at this point still differ from the code pages that are currently used by the monitored system. To identify whether runtime changes have been applied, each of the trusted code pages is compared byte by byte with its counterpart in the protected system. If a difference is observed, the framework attempts to validate the changes by determining whether the change was conducted by one of runtime patching mechanisms that the kernel uses. The individual validation steps thereby heavily depend on the hardware configuration of the monitored system as well as the runtime patching mechanisms that it uses. The interested reader can find an overview of the individual runtime patching mechanisms employed by the Linux kernel in [17].

Data-only Malware. Runtime code validation frameworks effectively hinder an attacker from introducing malicious instructions into a system as this new code will be detected and prevented from execution. To be able to control a system under such circumstances, attackers must thus resort to malware forms that leave the codebase of the attacked system untouched. The only malware form that is currently known to be capable of such a feat is data-only malware, which alters the control flow of the infected system based on specially crafted data structures [14,30].

In particular, data-only malware reuses the instructions that already existed on the target system before the malware arrived to perform its malicious operations. This is achieved by applying code reuse techniques, commonly used in the field of binary exploitation, to the problem of malware creation. Well-known examples of such techniques are ROP [27], JOP [3] and ret2libc [4].

To control the execution of the system, code reuse techniques leverage a control data structure that consists of pointers to existing instruction sequences. In general one cannot reuse arbitrary instruction sequences; instead, each of the reused sequences must fulfill a particular property. For example, in the case of ROP, each reused instruction sequence must end with a `return` instruction. The property of the `return` instruction is thereby that it will load the address which currently resides on top of the stack into the instruction pointer. This enables us to control the execution of the system as follows: our first reused instruction sequence will point the stack pointer to our control data structure in memory. Since the control structure now resides on the stack, the execution of the `return` instruction at the end of each reused sequence will obtain the address of the next sequence from the control structure and initiate its execution. Consequently, the `return` instruction provides the "transition" between the individual sequences whose addresses are contained within the control structure.

While code reuse *exploits* usually only make use of a very small control data structure that simply allocates a writable and executable memory region which is then used to execute traditional shellcode, control data structures of data-only malware are in general quite large. The reason for this is that data-only *malware* solely relies on code reuse to function. Each functionality that the malware provides must be implemented by code reuse. The result are huge chains that contain hundreds of reused instruction sequences [30]. However, due to the increasing proliferation of code integrity mechanisms, attackers will likely transition to this type of malware to attack modern OS kernels.

3 Attacker Model & Assumptions

In this work we assume that the monitored system is protected by a virtualization-based runtime code integrity validation framework. In addition, we assume that an attacker has gained full access to the monitored system, which she wants to leverage to install kernel malware. While the attacker can, in principle, modify any part of the system, the code validation framework will detect some of the changes that the attacker may conduct. Most importantly, it will detect any changes to executable kernel code and will in addition enforce SMEP and SMAP from the hypervisor-level. As a result, the attacker is forced to use data-only malware to infect the kernel. In this process, the control structure that is used by the attacker must reside within kernel's memory space since SMAP is in place. We also assume that the kernel's *identity mapping* which maps the entire physical memory into kernel space is marked as usermode in the page tables. A similar approach was previously proposed by Kemerlis et al. [15], in which pages that are used by userspace applications are temporarily unmapped from the identity mapping. Finally, we assume that the data-only malware introduced into the system by the attacker is persistent, i. e. will permanently reside within the memory of the target system, as otherwise it could not be triggered by an external event. Notice that this is usually the case for malware as Petroni and Hicks [22] showed.

4 Related Work

There is a plethora of work that is concerned with verifying the integrity of software. The existing research can thereby be roughly divided into two parts. The first branch of research focuses on the integrity of the system's code regions. This led to the development of various frameworks that are capable of validating the integrity of the codebase of applications as well as the kernel code sections [12,17,20]. This work builds upon said research by assuming that the code of the monitored system cannot be modified by an attacker due to fact that it is protected by such a framework.

The second branch of research, which our work belongs to, focuses on the integrity of the kernel's data and especially the kernel's *control* data. A popular approach in this regard is Control Flow Integrity (CFI) validation, which aims to dynamically validate the target of each branch instruction [1,16]. This is accomplished by tracing and monitoring every indirect branch and the current stack pointer of the inspected machine, implementing a shadow stack, or using the performance counters of the monitored system to trace unpredicted branches [7,21,33,34]. Unfortunately, however, current approaches not only suffer from a significant performance overhead, but also rely on invalid assumptions, which makes them vulnerable to evasion attacks [6,9,13,26].

Instead of ensuring control flow integrity for the entire kernel, there also have been approaches that solely focus on the discovery of hooks, which are often used by rootkits and other malware forms to intercept events within the system [31,32]. During this process, existing approaches rely on the assumption that only persistent control data can be abused for hooking. As in the case of CFI, this assumption is invalid and can be used to circumvent existing mechanisms by targeting transient control data instead [30]. Thus, neither hook-based detection nor CFI mechanisms are currently capable of countering data-only malware.

In addition, there has been work aiming to reconstruct the kernel data structures and their interconnection on the hypervisor level in order to provide data integrity checking [5,19,11,25]. The basic idea hereby is to parse the entire kernel code to be able to reconstruct the dependencies of different data structures (points-to analysis) and to construct a map of kernel data structures. However, current approaches are so far unable to reconstruct the entire graph of kernel data structures, which allows data-only malware to evade detection by leveraging techniques such as DKSM [2].

An alternative approach, similar to the one proposed in this work, aims to scan for pointers to executable code in 32-bit userspace memory [23,28]. Unfortunately, this approach has a high number of false positives on 32-bit systems. Therefore, each detected code pointer is further analyzed using *speculative code execution*.

Finally, Szekeres et al. [29] introduced the concept of Code-Pointer Integrity (CPI), the requirement to enforce the integrity of code pointers in memory. An implementation of CPI that is based on memory splitting was then proposed by Kuznetsov et al. [18]. In their work they introduce a compile time instrumentation approach that protects control flow relevant pointers. The basic idea thereby is to separate control flow relevant pointers into a separated space in memory and to limit access to that area. Thus they split process memory into a safe region and a regular region, where the safe region is secured by the kernel and can only be accessed via memory operations that are *autogenerated and proven at compile time* [18]. However, Evans et al. [10] showed that restricting access to pointers in memory is not enough, since this separation can still be broken with the help of side channel attacks.

5 Approach

In this work we aim to detect the control data structure of persistent data-only malware. In the process, we want to achieve three main goals:

Isolation. Since the main goal of our framework is to *detect* rather than to *prevent* kernel data-only malware infections, it is crucial that the detection framework is strongly isolated from the monitored target system. This is why we will leverage virtualization as a building block for our framework.

Performance. The overhead incurred by our detection framework on the monitored system should be as small as possible. Since we use virtualization as a foundation for our framework, it is thereby of particular importance that we keep the number of Virtual Machine (VM) exists as small as possible as they will heavily impact the performance of the overall approach.

Forensic. Due to the ever increasing number of malware attacks, the investigation of incidents becomes more and more important in order to understand the approach of an successful attacker and to avoid future breaches. This is why another crucial goal of our framework is to support forensic investigations in addition to live monitoring. In this regard, its particular important that an human investigator can easily assess and analyze the situation once an anomaly is detected by our framework.

The key idea behind our approach is to detect persistent data-only malware based on its control structure. As described in Section 2, the control structure is the most important component of data-only malware that essentially defines which reused instruction sequence should be executed when. Due to this property it is comparable to the code section of traditional malware, which makes it highly suitable as a basis for a detection mechanism.

To detect the control structure in memory, we use a three-step process. In the first step, we start by checking the integrity of important control flow related kernel objects. This is done for multiple reasons. First, we can use additional contextual information about these kernel objects, and second, these objects contain a lot of code pointers by design. By validating these objects at the beginning, we can increase the performance of our approach, as the code pointer within these known objects do not need to be validated in the following steps. We refer to this step as *Kernel Object Validation*.

In the second step, we *identify* all code pointers within the kernel's memory space. Based on this information, in the third step we *classify* the identified code pointers into benign and malicious code pointers applying multiple heuristics. The combination of these latter two steps is the *Pointer Examination* phase. Figure 1 provides an overview of this process. In the following, we describe these steps in more detail. For the sake of simplicity, we thereby focus on the Intel x64_64 bit architecture and the Linux OS. However, most of what we present is equally applicable to other OSs such as Windows. While this section provides an overview of our approach, we defer a discussion of the implementation details to Section 6.

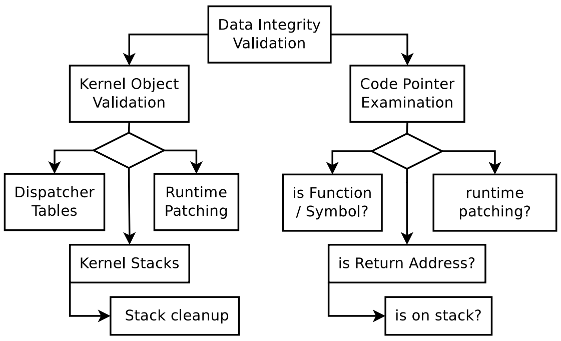

Fig. 1. Pointer classification within the proposed framework.

5.1 Control Flow Related Data Structures

We first describe control flow relevant kernel objects that we check using special semantic knowledge in the first step of our process.

Kernel Dispatcher Tables and Control Flow Registers. The most traditional control flow related data structures are the system call table and the interrupt descriptor tables. As control flow related data structures have already seen a lot of attention, we only mention this type of data structures here for sake of completeness. Our system checks every entry within these tables and ensures that it points to the correct function. This can be done by comparing the entire object to the corresponding version inside a trusted reference binary. In this step, we also validate the values of all control flow relevant registers such the model-specific registers (MSRs) and the *Debug* registers.

Tracepoints. Tracepoints are another type of data structure that is control flow relevant. An administrator can use the tracepoints feature to insert arbitrary hooks into the kernel's control flow that are executed whenever a certain point in the kernel's control flow is hit and the corresponding tracepoint is enabled. The addresses of the callback functions are stored in a list and are sequentially called by the kernel once the tracepoint is triggered. Tracepoints impose a big problem for control flow integrity validation as arbitrary function addresses can be inserted into all tracepoint locations at runtime. To counter this threat, we ensure that every hook that is installed with this mechanism calls a valid function within the Linux kernel.

Control Structures For Kernel Runtime Patching. To manage different runtime-patching mechanisms, the kernel maintains a variety of data structures. These data structures in turn contain pointers to kernel code, as they need to store the locations where kernel code should be patched at runtime. In our approach we check the integrity of the related data structures.

Kernel Stacks. Another examined type of data structure is the kernel stack of each thread in the system. We separate each kernel stack into three parts: At the very beginning of the stack, the active part of the stack is located. This part is empty if the corresponding process is currently executing in userspace. Next to the active part of the stack, old obsolete stack content is residing. On the very top of the stack, after all usable space, resides a structure called `thread_info`. It contains the thread's management information, for instance a `task_struct` pointer and the address limit of the stack.

While it is possible to validate the active part of the stack and its management structure, an attacker could use the old, currently unused stack space to hide persistent data-only malware. Therefore, this space is filled with zeros by our framework when used in live monitoring mode. Otherwise the unused stack regions are displayed to the forensic analyst for diagnosis and verification.

5.2 Pointer Identification

After we have validated control flow relevant data structures, we identify all other code pointers in memory in the second step. To identify code pointers, first of all we need to obtain a list of all executable memory regions within kernel space. For this purpose, we make use of the page tables used by the hardware. We also generate a list of all readable pages that do not contain code, as these pages contain the kernel's data. Note that using this approach we are also able to support Address Space Layout Randomization (ASLR).

Equipped with a list of all kernel code and data pages, we identify all kernel code pointers by iterating through each data page byte by byte and interpreting each 64-bit value as a potential pointer. If the potential pointer points to a code region (i.e., the 64-bit value represents an address lying within one of the code pages), we consider it to be a code pointer. While it seems that this very simple approach might produce many false positives, we like to stress that we did not observe any false positives during our experiments with various 64-bit Linux kernels. In our opinion the primary reason for this is that the 64-bit address space is much larger than the former 32-bit address space and makes it thus much more unlikely that non pointer values looking like pointers appear within memory.

5.3 Pointer Classification

After we have found a pointer, we classify it based on its destination address in order to decide whether it is malicious or benign. In a legitimate kernel there are multiple targets which a pointer is *allowed* to point to. In the following, we list those valid targets and describe how we are able to determine to which category the pointer belongs to.

Function Pointers. One important type of kernel code pointers are *function pointers*, which are frequently used within the kernel. To determine whether a code pointer is a function pointer, we make use of symbol information that is extracted from a trusted reference binary of the monitored kernel. Amongst these symbols are all functions that the kernel provides. We leverage the symbol list to verify whether a code pointer points to a function or not. In the former case, we consider the pointer to be benign. Otherwise, we continue with the classification process in order to determine whether the code pointer belongs to one of the other categories discussed below. Note that this implies that our approach might still be vulnerable to data-only malware that solely makes use of return-to-function (ret2libc).

Return Addresses. Another important type of code pointers are *return addresses*. In contrast to a function pointer, which must point to the beginning of a function, a return address can point to any instruction within a function that is preceded by a call instruction. To identify whether a code pointer is a return address, we leverage multiple heuristics. Note that most of the return addresses are located on a stack which is already checked during the Kernel Object Validation phase.

Pointers Related to Runtime Patching. A third type of pointer destinations are addresses that are stored by the kernel and point to a location where dynamic code patching is performed. While most of these pointers are contained within special objects that are checked in the Kernel Object Validation step as previously described, there are still some exceptions that must be considered separately.

Unknown Pointer Destinations. Any code pointer pointing into executable code which can not be classified into one of the above categories is considered to be malicious.

As we intend to identify kernel level data-only malware with our approach and we assume that the malware is persistently stored in memory, we propose to execute CPE in regular intervals.

6 Implementation

After describing the general idea of our approach, we cover the details of our implementation in this section. The code pointer examination framework presented in this work is based on our kernel code integrity framework [17]. This framework provides multiple advantages for our implementation:

First, it keeps track of all kernel and module code sections and ensures their integrity during runtime. In addition, it keeps track of all functions and symbols that are available inside the monitored kernel, as it already resembles the Linux loading process. This ensures that the information about the monitored kernel is binding by its nature, that is, it reflects the actual state of the monitored system. In our implementation we can use this database as a ground truth to classify kernel code pointers.

Secondly, the underlying framework keeps track of all dynamic runtime code patching that is conducted by the Linux kernel. We use this information to identify and validate data structures that are related to kernel runtime patching.

Third, our approach is usable for multiple hypervisors, while most of the features can also be used to analyze memory dumps in a forensic scenario. Currently tests have been conducted with both KVM as well as XEN.

6.1 Kernel Object Validation

Before we scan the kernel's memory for pointers, we check the integrity of important kernel data structures. This allows to minimize the parts of kernel data that may contain arbitrary function pointers or other pointers into executable kernel code. The validation of those structures leverages semantic information about the kernel that was generated by the underlying code validation framework or manually collected while analyzing the kernel. In the following, we only list a couple of examples to illustrate the requirement of this step.

First, we validate various dispatcher tables and the kernel's read-only data segments. These locations usually contain a lot of kernel code pointers, whereas the target of each pointer is well defined. The validation is performed by comparing these objects to the trusted reference versions of the binaries that are loaded by the underlying validation framework.

Next, we validate kernel data structures used for runtime patching. These are for example: Jump Labels (`__start___jump_table`), SMP Locks (`__smp_locks`), Mcount Locations (`__start_mcount_loc`), and Ftrace Events (`__start_ftrace_events`). To validate these structures we semantically compare them to the data extracted from trusted reference binaries by the underlying framework. In addition to these runtime patching control data structures, there also exist data structures in the kernel that are used to actually conduct the runtime patch. For clarification, we discuss one example for legitimate kernel code pointers related to self-patching: the kernel variables `bp_int3_handler` and `bp_int3_addr`.

To understand why these pointers are required, we explain how runtime patching takes place in the Linux kernel. If the kernel patches a multibyte instruction in the kernel, it can not simply change the code in question. The kernel's code would be in an inconsistent state for a short period of time, which might lead to a kernel crash. Thus, the kernel implements a special synchronization method. It first replaces the first byte of the change with an *int3* instruction. As a result, every CPU trying to execute this instruction will be trapped. Then the rest of the space is filled with the new content. As a last step, the kernel replaces the first byte and notifies all waiting CPUs. During this process the address containing the *int3* instruction is saved in the variable `bp_int3_addr`.

This enables the *int3* interrupt handler upon invocation to determine whether the interrupt originates from the patched memory location or not. While the interrupt handler will simply process the interrupt normally in the latter case, it will in the former case invoke a specific handler whose address is stored within the variable `bp_int3_handler`. In the case of a patched jump label, for example, the *handler variable* will point to the instruction directly after the patched byte sequence, which effectively turns the sequence into a *NOP* sequence during the patching process. Since both of the `bp_int3` variables are not reset after patching is complete, they always point to the last patched location and the last handler respectively. To solve this issue, our framework checks whether the current value of the `bp_int3_addr` points to a self patching location and if the handler address matches the type of patching conducted.

Finally, we iterate through all pages that contain a stack. Each process running in a system owns its own kernel stack that is used once the application issues a system call. To gather the addresses of all stacks from the monitored host, we iterate through the list of running threads (`init_task.tasks`) and extract their corresponding stacks. In case the process is not currently executing within the kernel, the current stack pointer is also saved within that structure. Ideally the process is currently not executing in kernel space in which case its stack must be *empty*. Otherwise we must validate the contents of the stack.

In order to validate a stack we use the following approach: For each *return address* found on the stack, we save the addresses of two functions. First, we save the address of the function that the return address is pointing to (`retFunc`). In addition, we also extract the address of the target, of the call instruction preceding the *return address* (`callAddr`). This is possible, since in most cases, the destination of the call is directly encoded in the instruction, or a memory address is referenced in the instruction that can in turn be read from the introspected guest system's memory.

This information is then used to validate the next return address that is found on the stack. In particular, the `callAddr` of the next frame needs to match the `retFunc` of the previous stack frame, as the previous function must have called the function, that the return address is pointing to.

Since it is not possible to extract all call targets using the method described above, we use an additional mechanism to extract all possible targets of indirect calls: we monitor the execution of the test systems in a secure environment and activate the processor's Last Branch Register (LBR) mechanism in order to extract the call and the target address of every indirect branch instruction executed by the system's CPU. Using this mechanism we generated a whitelist of targets for each call for which the target address is generated during runtime. This list is then also used by our stack validation component. With this we were, in our experiments, able to validate most of the kernel stacks within our test system. While this mechanism is not perfect yet, it certainly reduces the attack surface further.

The entire problem arises because the stack is currently not designed to be verifiable even under normal circumstances. However, the kernel developers currently discuss an enhancement to the code that would make stack validation more reliable, which could, once implemented, be used to improve our current approach and would allow removal of the whitelist.[1]

6.2 Code Pointer Examination

After we have checked important data structures, we scan through the rest of kernel data memory to find pointers to executable kernel code. This is achieved using the following steps: We first extract the memory regions of executable kernel code sections in the monitored virtual machine using the page tables structure. As a second step, we extract the data pages of the monitored guest

[1] `https://lkml.org/lkml/2015/5/18/545`

system. For this purpose, we obtain all pages that are marked as supervisor and not executable in the page tables. These pages contain the data memory of the kernel and therefore all pointers that are accessible from within the Linux kernel. Note that the information we use for our analysis is binding, since it is derived from either the hardware or the trusted kernel reference binaries.

Having obtained the code and data pages, we iterate through the extracted pages in a byte by byte manner. We interpret each eight byte value (independently of its alignment) as a pointer and check whether it points into one of the memory locations that was identified as containing kernel code. If we found a pointer that points to executable kernel memory we first check if its destination is contained in the list of valid functions.

In case the pointer does not point to a valid function, we check if the pointer is a return address. There are currently multiple approaches used in our framework to identify a return address. First and foremost, a return address must point to an instruction within a function that is preceded by `call` instruction. Consequently, our initial check consists of validating whether the instruction it points to is actually contained within the function.

For this purpose, we disassemble the function the pointer allegedly points to from the beginning and verify that the value of the pointer points to a disassembled instruction and not somewhere in between instructions. In such a case, we additionally ensure a `call` instruction resides before the instruction the pointer points to. If any of these conditions fail, we consider the code pointer not to be a valid return address and continue to the next category.

Most of the return addresses used within the kernel are stored within one of the kernel stacks. However, there exist a few functions within the kernel that save the return address of the current function to be able to identify the current caller of that function. This was first introduced as a debug feature to print the address of the calling function to the user in case of an error. However, in the meantime this feature is also used for other purposes such as timers. For example, the struct `hrtimer` contains a pointer `start_site` that points to the instruction after the call instruction that started the timer.

With such a feature in place and used by the kernel it is hard to differentiate between legitimate return addresses and specially crafted control structures for code reuse techniques. To limit this problem we created a whitelist of all calls to functions that contain the problematic instruction and only allow return addresses in the kernel's data segment if they point to one of the functions in question.

If the pointer does not point to a valid function or a return address, the pointer is considered as malicious and a human investigator is notified. At this point the system also enriches the error message with the name of the function or symbol the pointer is pointing into.

7 Evaluation

In this section, we evaluate our approach using the prototype implementation described in Section 6. In order to determine whether our framework is able to achieve the goals set in Section 5, we first determine its performance characteristics, before we evaluate its effectiveness against data-only malware in both live monitoring as well as forensic applications. We follow this with an in-depth discussion of the security aspects of our system.

7.1 Experiments

Our host system consisted of an AMD Phenom II X4 945 CPU with 13 GB of RAM running Linux kernel version 3.16 (Debian Jessie). As guest systems we used two different VMs running Linux 3.8 as well as Linux 3.16. Each VM had access to two virtual CPUs and 1 GB of RAM. In these experiments, we used XEN as the underlying hypervisor.

Performance and False Positives. First of all, we evaluated the performance of our system as well as its susceptibility to false positives. For this purpose, we used the *Phoronix-Test-Suite* to run a set of Benchmarks on our system. In detail, we ran the *pts/kernel* test suite. We conducted these benchmark three times on each test kernel. During the first set of tests, we disabled all external monitoring to obtain a baseline of the normal system performance. In the second test set, we enabled the code validation component to be able to differentiate between the overhead of our framework and the code validation system. Finally, we enabled both the code validation component as well as our new pointer validation module in order to identify the additional overhead that our system incurs. During the tests, the integrity validation component was executed in a loop, if enabled, to stress the guest system as much as possible. The results of the benchmarks of each set of experiments as well as the overall performance degradation are shown in Table 1 for Linux 3.8 and in Table 2 for Linux 3.16.

While evaluating the Linux 3.8 kernel, the kernel contained 80 code pages and 426 data pages. One complete *Code Integrity Validation* was completed in 255.8 ms, while in the experiment with Code Integrity Validation *and* Pointer Examination enabled, one iteration took 567.58 ms (that is 341.78 ms for CPE). The Linux 3.16 kernel that was used during our evaluation contained 408 code pages and 986 data pages. The Code Integrity Validation alone took 639.8 ms per iteration, while the combined CIV and Pointer Examination took 962.0 ms per iteration (that is 322.2 ms for CPE). Note that these values are mean values. This shows that it takes less than 1 ms on average to check the integrity of one page.

As one can see the performance overhead that our framework incurs is very small. In fact, the use of the underlying Code Validation Component incurs a larger overhead than our CPE framework. The performance impact of our system is for the most benchmarks well under one percent. The main reason for this is that our framework, in contrast to many other VMI-based approaches, uses passive monitoring of the guest system whenever applicable. As a result,

Test (Unit)	**w/o**	**CIV (%)**	**CIV & CPE (%)**
FS-Mark (Files/s)	32.57	30.10 (8.21%)	31.73 (2.65%)
Dbench (MB/s)	69.84	66.53 (4.98%)	71.54 (−2.38%)
Timed MAFFT Alignment (s)	20.63	20.70 (0.34%)	20.63 (0.00%)
Gcrypt Library (ms)	2857	2853 (−0.14%)	2837 (−0.70%)
John The Ripper (Real C/S)	1689	1689 (0.00%)	1688 (0.06%)
H.264 Video Encoding (FPS)	35.38	35.23 (0.43%)	35.31 (0.20%)
GraphicsMagick 1 (Iter/min)	95	95 (0.00%)	95 (0.00%)
GraphicsMagick 2 (Iter/min)	58	58 (0.00%)	58 (0.00%)
Himeno Benchmark (MFLOPS)	593.59	585.73 (1.34%)	586.24 (1.25%)
7-Zip Compression (MIPS)	4715	4702 (0.28%)	4706 (0.19%)
C-Ray - Total Time (s)	130.96	131.00 (0.03%)	130.99 (0.02%)
Parallel BZIP2 Compression (s)	36.35	36.58 (0.63%)	36.47 (0.33%)
Smallpt (s)	445	445 (0.00%)	446 (0.22%)
LZMA Compression (s)	234.50	236.39 (0.81%)	236.12 (0.69%)
dcraw (s)	124.24	124.38 (0.11%)	124.35 (0.09%)
LAME MP3 Encoding (s)	25.20	25.19 (−0.04%)	25.19 (−0.04%)
Ffmpeg (s)	27.00	27.02 (0.07%)	26.82 (−0.67%)
GnuPG (s)	15.34	14.98 (−2.35%)	14.94 (−2.61%)
Open FMM Nero2D (s)	1137.17	1148.95 (1.04%)	1144.94 (0.68%)
OpenSSL (Signs/s)	173.70	173.73 (−0.02%)	173.80 (−0.06%)
PostgreSQL pgbench (Trans/s)	115.11	114.69 (0.37%)	115.21 (−0.09%)
Apache Benchmark (Requests/s)	10585.45	10481.21 (0.99%)	10506.23 (0.75%)

Table 1. Results of the Phoronix Test Suite for Linux 3.8.

Test (Unit)	**w/o**	**CIV (%)**	**CIV & CPE (%)**
FS-Mark (Files/s)	30.90	31.37 (−1.50%)	31.67 (−2.43%)
Dbench (MB/s)	61.42	60.76 (1.09%)	61.04 (0.62%)
Timed MAFFT Alignment (s)	20.74	20.79 (0.24%)	20.75 (0.05%)
Gcrypt Library (ms)	3747.00	3740 (−0.19%)	3733 (−0.37%)
John The Ripper (Real C/S)	1693.00	1693 (0.00%)	1692 (0.06%)
H.264 Video Encoding (FPS)	34.60	34.32 (0.82%)	34.35 (0.73%)
Himeno Benchmark (MFLOPS)	598.71	582.78 (2.73%)	585.78 (2.21%)
7-Zip Compression (MIPS)	4850.00	4805 (0.94%)	4730 (2.54%)
C-Ray - Total Time (s)	89.80	89.81 (0.01%)	89.80 (0.00%)
Parallel BZIP2 Compression (s)	31.25	31.41 (0.51%)	31.37 (0.38%)
Smallpt (s)	407.00	407 (0.00%)	407 (0.00%)
LZMA Compression (s)	236.62	241.49 (2.06%)	242.17 (2.35%)
dcraw (s)	117.54	117.47 (−0.06%)	117.29 (−0.21%)
LAME MP3 Encoding (s)	23.39	23.41 (0.09%)	23.40 (0.04%)
GnuPG (s)	13.72	13.65 (−0.51%)	13.98 (1.90%)
OpenSSL (Signs/s)	173.63	173.37 (0.15%)	173.57 (0.03%)
Apache Benchmark (Requests/s)	9504.78	9156.01 (3.81%)	9383.66 (1.29%)

Table 2. Results of the Phoronix Test Suite for Linux 3.16.

the guest system can execute through most of the validation process without being interrupted by the hypervisor, which drastically reduces the performance overhead of the monitoring. Only for the FSMark benchmark a performance degradation of about 2.65 percent is noticed on Linux 3.8. This degradation can not be seen in the results of the benchmark on Linux 3.16. While using the guest system with monitoring enabled, we did not observe any noticeable overhead from within the guest system. This clearly shows that our framework can achieve the performance goal set in Section 5 and is, from a performance point of view, well suited for real world applications. Sometimes the results even showed that the tests were better with our pointer examination framework enabled than without our framework. We argue that this may be due to the fact that the performance impact of our system is much smaller than the impact of other standard software within the tested Debian system that also influenced the result.

At the same time we did not observe any false positives during our experiments. That is, when enabled, our system could classify all of the pointers it encountered during the validation process using the heuristics we described in Section 5. However, note that we can, due to the design of our system, not rule out false positives entirely. We perform a more detailed discussion about the possibility of encountering false positives in Section 7.2.

Malware Detection. Having evaluated the performance of our system and touched upon its susceptibility to false positives, we continued to evaluate the effectiveness of our framework against data-only malware. For this purpose, we infected our test VMs with the persistent data-only rootkit presented by Vogl et al. [30]. We chose this rootkit, since it is, to the best of our knowledge, the only persistent data-only malware available to date.

While our framework did not detect any malicious code pointers during the performance experiments, our system immediately identified the various malicious control structures used by the rootkit. In particular, our system identified the modified `sysenter` MSR and the modified system call table entries for the `read` and the `getdents` system call during the prevalidation step and thus classified the system as malicious. As these hooks are also found by other systems, we then removed these obvious manipulations manually and once more validated the system state. While the prevalidation step yielded no results in this case, the pointer validation found all of the malicious code pointers in memory. This proves that our framework can be very effective against data-only malware even if the malware avoids the manipulation of key data structures such as the system call table.

Finally, to evaluate the usefulness of our framework in forensic applications, we conducted an experiment where we randomly installed the rootkit on the test VMs while we periodically took snapshots of the guest systems. Our system detected all of the infected snapshots reliably.

7.2 Discussion

In this section, we provide a detailed discussion of the security of our system.

False Positives. Although we did not encounter false positives throughout our experiments, we cannot rule out false positives entirely, since our system relies on heuristics to identify code pointers. However, we like to stress that we consider the likelihood of encountering false positives in our system to be quite small on a 64-bit architecture. To encounter a false positive with our system, we essentially would need to find a value in kernel space that contains the address of a kernel code section even though it is not a pointer. Since the virtual address space on a 64-bit system has a size of $1.8 * 10^{19}$ bytes and the kernel code section typically only has a size of 15 megabytes at maximum, the chance of encountering such a rare case, if all values in memory were uniformly distributed would be merely $8.5 * 10^{-11}\%$. And that is only the case if the kernel is not optimized as the kernel code section even becomes smaller in this case. However, we admit that this is only the case if the kernel is mapped to a random location within the address space and not directly to the beginning or the end of the address space. In other words, we consider a 64-bit address space to be sufficiently large that the chance of arbitrary data looking like a pointer by chance are small at best. Consequently, we assume that false positives are not a big issue in most scenarios. In case of false positives, one could further analyze the detected pointers using speculative code execution as proposed by Polychronakis [23]. Note that an attacker could also introduce benign data into the system that will be identified as code pointers by our system. We argue that this kind of tampering with our system should still be identified as malicious.

ret2libc. When searching for malicious pointers in memory, we currently do not penalize pointers that point to function entry points. As a consequence, our system is at the moment unable to detect data-only malware that solely makes use of entire kernel functions to perform its malicious computations. While this is certainly a weakness of our approach, it is important to note that this is a very common limitation that almost all existing defense mechanisms against code reuse attacks face [8,26]. In fact, to the best of our knowledge, the detection of ret2libc attacks still remains an open research problem.

In addition, while ret2libc is a powerful technique that is very difficult to detect, we argue that it is actually quite difficult to design pure data-only malware that solely relies on entire functions to run on a 64-bit architecture. The main reason for this is that in contrast to 32-bit systems, function arguments in Linux and Windows are no longer passed on the stack on a 64-bit architecture, but are provided in registers instead. As a consequence, to create 64-bit ret2libc data-only malware, an attacker must actually have access to “loader” functions that allow her to load arbitrary function arguments into the registers that the calling conventions dictate. Otherwise, without access to loader functions, the attacker is unable to pass arguments to any of the functions she wants to invoke, which significantly restricts her capability to perform attacks.

It goes without saying that such loader functions are probably rare if they exist at all. A possible approach to further reduce the attack surface could thus be to analyze the kernel code for such loader functions. If they should exist, one can then monitor the identified functions during execution to detect their use in ret2libc attacks. We plan to investigate this idea in more depth in future work.

Return Addresses. If an attacker requires gadgets in addition to entire functions to execute her persistent data-only malware (e.g. to load function arguments into registers), she can only use a gadget that is directly following a call instruction. The only location that she can place the required control structure to without being detected is the kernel stack of a process. Should a code pointer that points inside a function appear anywhere else within the kernel memory, it will be classified and identified as malicious by our system. In addition, due to the fact that our system enforces SMAP from the hypervisor, the control structure cannot be placed in userspace if it should be executable from kernelspace. This only leaves a kernel stack for kernel data-only malware. But even here the attacker faces various constraints. First of all, she can only make use of gadgets that appear legitimately in the code and that are preceded by a call instruction, since all other pointers into a function would be classified as malicious. Secondly, as the kernel stack where the control structure resides may also be used by the process it belongs to, the attacker must ensure that her persistent control structure is not overwritten by accident. While this is not necessarily an issue for data-only exploits, this is crucial in the case of persistent data-only malware as the persistent control structure of the malware must never be changed uncontrollably. Otherwise, if the control structure would be modified in an unforeseen way, it is very likely that the malware will fail to execute the next time it is invoked. This is comparable to changing the code region of traditional malware. This is also why our system zeroes all data that belongs to a memory page that is part of the kernel stack, but currently resides at a lower address than the stack pointer points to as a final defense layer. Since this data should be unused in a legitimate scenario, zeroing it will not affect the normal system behavior. However, in the case of persistent data-only malware, this approach may destroy the persistent control structure of the malware, which will thwart any future execution. This will be case if the malware is currently executing while our system performs the validation. Since an attacker cannot predict when validations occur as our system resides on the hypervisor-level, this makes it difficult for her to stay unnoticed in the long run.

As a further enhancement one could set the kernel stacks of processes that are currently not executing to not readable within the page tables. This could for example be done during the process switch. As a result, the attacker would only be able to use her control structure when the process on whose kernel stack the structure resides is currently executing. This raises the bar if the attacker wants to hook the execution of all processes instead of just one, which is generally the case.

Taking all this into account we argue that while our system cannot eliminate persistent data-only malware entirely, it significantly reduces the attack surface. In future work, we plan to further enhance our detection by developing novel techniques to validate the legitimacy of a kernel stack that are also applicable in forensic scenarios. In addition we plan to investigate the applicability of our approach to userspace applications or an Android environment.

8 Conclusion

In this paper, we have proposed *Code Pointer Examination*, an approach that aims to detect data-only malware by identifying and classifying pointers to executable memory. To prove the validity and practicability of our approach, we employed it to examine all pointers to executable kernel memory in recent Linux kernels. In the process, we discussed important control flow relevant data structures and mechanisms within the Linux kernel and highlighted the problems that must be solved to be able to validate kernel control data reliably. Our experiments show that the prototype, which we implemented based on the discussed ideas, is effective in detecting data-only malware, while only incurring a very small performance overhead (less than 1% in most of the benchmarks). In combination, with code integrity validation, we thus provide the first comprehensive approach to kernel integrity validation. While our framework still exhibits a small attack surface, we argue that it considerably raises the bar for attackers and thus provides a new pillar in the defense against data-only malware.

Acknowledgments

We thank the anonymous reviewers for their insightful comments. This work was supported by the Bavarian State Ministry of Education, Science and the Arts as part of the FORSEC research association.

References

1. Abadi, M., Budiu, M., Erlingsson, U., and Ligatti, J. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security* (New York, NY, USA, 2005), CCS '05, ACM, pp. 340–353.
2. Bahram, S., Jiang, X., Wang, Z., Grace, M., Li, J., Srinivasan, D., Rhee, J., and Xu, D. Dksm: Subverting virtual machine introspection for fun and profit. In *Proceedings of the 29th IEEE International Symposium on Reliable Distributed Systems (SRDS 2010)* (New Delhi, India, October 2010).
3. Bletsch, T., Jiang, X., Freeh, V. W., and Liang, Z. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security* (New York, NY, USA, 2011), ASIACCS '11, ACM, pp. 30–40.
4. C0ntex. Bypassing non-executable-stack during exploitation using return-to-libc.
5. Carbone, M., Cui, W., Lu, L., Lee, W., Peinado, M., and Jiang, X. Mapping kernel objects to enable systematic integrity checking. In *Proceedings of the 16th ACM conference on Computer and Communications Security (CCS'09)* (2009), ACM, pp. 555–565.
6. Carlini, N., and Wagner, D. Rop is still dangerous: Breaking modern defenses. In *23rd USENIX Security Symposium (USENIX Security 14)* (San Diego, CA, Aug. 2014), USENIX Association, pp. 385–399.
7. Cheng, Y., Zhou, Z., Yu, M., Ding, X., and Deng, R. H. Ropecker: A generic and practical approach for defending against ROP attacks. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014* (2014).
8. Davi, L., Liebchen, C., Sadeghi, A.-R., Snow, K. Z., and Monrose, F. Isomeron: Code randomization resilient to (just-in-time) return-oriented programming. *Proc. 22nd Network and Distributed Systems Security Sym.(NDSS)* (2015).

9. Davi, L., Sadeghi, A.-R., Lehmann, D., and Monrose, F. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *23rd USENIX Security Symposium (USENIX Security 14)* (San Diego, CA, Aug. 2014), USENIX Association, pp. 401–416.
10. Evans, I., Fingeret, S., González, J., Otgonbaatar, U., Tang, T., Shrobe, H., Sidiroglou-Douskos, S., Rinard, M., and Okhravi, H. Missing the point (er): On the effectiveness of code pointer integrity.
11. Feng, Q., Prakash, A., Yin, H., and Lin, Z. Mace: High-coverage and robust memory analysis for commodity operating systems. In *Proceedings of the 30th Annual Computer Security Applications Conference* (New York, NY, USA, 2014), ACSAC '14, ACM, pp. 196–205.
12. Gilbert, B., Kemmerer, R. A., Kruegel, C., and Vigna, G. Dymo: Tracking dynamic code identity. In *RAID* (2011), R. Sommer, D. Balzarotti, and G. Maier, Eds., vol. 6961 of *Lecture Notes in Computer Science*, Springer, pp. 21–40.
13. Göktaş, E., Athanasopoulos, E., Polychronakis, M., Bos, H., and Portokalidis, G. Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *23rd USENIX Security Symposium (USENIX Security 14)* (San Diego, CA, Aug. 2014), USENIX Association, pp. 417–432.
14. Hund, R., Holz, T., and Freiling, F. C. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In *Proceedings of 18th USENIX Security Symposium* (2009).
15. Kemerlis, V. P., Polychronakis, M., and Keromytis, A. D. ret2dir: Rethinking kernel isolation. In *23rd USENIX Security Symposium* (Aug. 2014), USENIX Association.
16. Kemerlis, V. P., Portokalidis, G., and Keromytis, A. D. kguard: Lightweight kernel protection against return-to-user attacks. In *Proceedings of the 21st USENIX Conference on Security Symposium* (Berkeley, CA, USA, 2012), Security'12, USENIX Association.
17. Kittel, T., Vogl, S., Lengyel, T. K., Pfoh, J., and Eckert, C. Code validation for modern os kernels. In *Workshop on Malware Memory Forensics (MMF)* (Dec. 2014).
18. Kuznetsov, V., Szekeres, L., Payer, M., Candea, G., Sekar, R., and Song, D. Code-pointer integrity. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Broomfield, CO, Oct. 2014), USENIX Association, pp. 147–163.
19. Lin, Z., Rhee, J., Zhang, X., Xu, D., and Jiang, X. Siggraph: Brute force scanning of kernel data structure instances using graph-based signatures. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)* (2011), IEEE.
20. Litty, L., Lagar-Cavilla, H. A., and Lie, D. Hypervisor support for identifying covertly executing binaries. In *Proceedings of the 17th Usenix Security Symposium* (Berkeley, CA, USA, 2008), USENIX Association, pp. 243–258.
21. Pappas, V., Polychronakis, M., and Keromytis, A. D. Transparent rop exploit mitigation using indirect branch tracing. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)* (Washington, D.C., 2013), USENIX, pp. 447–462.
22. Petroni, Jr., N. L., and Hicks, M. Automated detection of persistent kernel control-flow attacks. In *Proceedings of the 14th ACM conference on Computer and communications security* (New York, NY, USA, 2007), CCS '07, ACM.

23. Polychronakis, M., and Keromytis, A. D. Rop payload detection using speculative code execution. In *Malicious and Unwanted Software (MALWARE), 2011 6th International Conference on* (2011), IEEE, pp. 58–65.
24. Sadeghi, A.-R., Davi, L., and Larsen, P. Securing legacy software against real-world code-reuse exploits: Utopia, alchemy, or possible future? - keynote -. In *10th ACM Symposium on Information, Computer and Communications Security (ASIACCS 2015)* (Apr. 2015). Keynote.
25. Schneider, C., Pfoh, J., and Eckert, C. Bridging the semantic gap through static code analysis. In *Proceedings of EuroSec'12, 5th European Workshop on System Security* (Apr. 2012), ACM Press.
26. Schuster, F., Tendyck, T., Liebchen, C., Davi, L., Sadeghi, A.-R., and Holz, T. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In *36th IEEE Symposium on Security and Privacy (Oakland)* (May 2015).
27. Shacham, H. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security* (New York, NY, USA, 2007), CCS '07, ACM, pp. 552–561.
28. Stancill, B., Snow, K. Z., Otterness, N., Monrose, F., Davi, L., and Sadeghi, A.-R. Check my profile: Leveraging static analysis for fast and accurate detection of rop gadgets. In *16th Research in Attacks, Intrusions and Defenses (RAID) Symposium* (Oct. 2013).
29. Szekeres, L., Payer, M., Wei, T., and Song, D. Sok: Eternal war in memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2013), SP '13, IEEE Computer Society, pp. 48–62.
30. Vogl, S., Pfoh, J., Kittel, T., and Eckert, C. Persistent data-only malware: Function hooks without code. In *Proceedings of the 21th Annual Network & Distributed System Security Symposium (NDSS)* (Feb. 2014).
31. Wang, Z., Jiang, X., Cui, W., and Ning, P. Countering kernel rootkits with lightweight hook protection. In *Proceedings of the 16th ACM conference on Computer and communications security* (New York, NY, USA, 2009), CCS '09, ACM, pp. 545–554.
32. Wang, Z., Jiang, X., Cui, W., and Wang, X. Countering persistent kernel rootkits through systematic hook discovery. In *Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection* (Berlin, Heidelberg, 2008), RAID '08, Springer-Verlag, pp. 21–38.
33. Xia, Y., Liu, Y., Chen, H., and Zang, B. Cfimon: Detecting violation of control flow integrity using performance counters. In *Proceedings of the 2012 42Nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (Washington, DC, USA, 2012), DSN '12, IEEE Computer Society, pp. 1–12.
34. Zhang, C., Wei, T., Chen, Z., Duan, L., Szekeres, L., McCamant, S., Song, D., and Zou, W. Practical control flow integrity and randomization for binary executables. In *Security and Privacy (SP), 2013 IEEE Symposium on* (2013), IEEE, pp. 559–573.