Data-Centric OS Kernel Malware Characterization

Junghwan Rhee, *Member, IEEE*, Ryan Riley, *Member, IEEE*, Zhiqiang Lin, *Member, IEEE*, Xuxian Jiang, and Dongyan Xu, *Member, IEEE*

***Abstract*— Traditional malware detection and analysis approaches have been focusing on code-centric aspects of malicious programs, such as detection of the injection of malicious code or matching malicious code sequences. However, modern malware has been employing advanced strategies, such as reusing legitimate code or obfuscating malware code to circumvent the detection. As a new perspective to complement code-centric approaches, we propose a data-centric OS kernel malware characterization architecture that detects and characterizes malware attacks based on the properties of data objects manipulated during the attacks. This framework consists of two system components with novel features: First, a runtime kernel object mapping system which has an un-tampered view of kernel data objects resistant to manipulation by malware. This view is effective at detecting a class of malware that hides dynamic data objects. Second, this framework consists of a new kernel malware detection approach that generates malware signatures based on the data access patterns specific to malware attacks. This approach has an extended coverage that detects not only the malware with the signatures, but also the malware variants that share the attack patterns by modeling the low level data access behaviors as signatures. Our experiments against a variety of real-world kernel rootkits demonstrate the effectiveness of data-centric malware signatures.**

***Index Terms*— OS kernel malware characterization, data-centric malware analysis, virtual machine monitor.**

I. Introduction

MODERN malware use a variety of techniques to cause divergence in the attacked program's behavior and achieve the attacker's goal. Traditional malicious programs such as computer viruses, worms, and exploits have been using code injection attacks which inject malicious code into a program to perform a nefarious function. Intrusion detection approaches based on such code properties effectively detect or prevent this class of malware attacks [14], [20], [42], [43], [45], [51].

Manuscript received February 28, 2013; revised July 4, 2013 and October 8, 2013; accepted November 7, 2013. Date of publication November 20, 2013; date of current version December 17, 2013. This work was supported in part by the U.S. National Science Foundation (NSF) under Grant 1049303 and the U.S. Air Force Office of Scientific Research (AFOSR) under Contract FA9550-10-1-0099. The associate editor coordinating the review of this manuscript and approving it for publication was Prof. C.-C. Jay Kuo.

J. Rhee is with NEC Laboratories America, Princeton, NJ 08540 USA (e-mail: rhee@nec-labs.com).

R. Riley is with the Department of Computer Science and Engineering, Qatar University, Doha 2713, Qatar (e-mail: ryan.riley@qu.edu.qa).

Z. Lin is with the Department of Computer Science, the University of Texas at Dallas, Richardson, TX 75080 USA (e-mail: zhiqiang.lin@utdallas.edu).

X. Jiang is with the Department of Computer Science, North Carolina State University, Raleigh, NC 27695 USA (e-mail: jiang@cs.ncsu.edu).

D. Xu is with the Department of Computer Science and CERIAS, Purdue University, West Lafayette, IN 47907 USA (e-mail: dxu@cs.purdue.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TIFS.2013.2291964

In response to these techniques, alternate attack vectors were devised to avoid violation of code integrity and therefore elude such detection approaches. For instance, return-to-libc attacks [8], [33], return-oriented programming [6], [23], [46], and jump-oriented programming [10], [16], [17], [21], [27] reuse existing code to create malicious logic. Additionally, kernel malware can be launched via vulnerable code in program bugs [31], [49], [50], third-party kernel drivers, and memory interfaces [18] which can allow manipulation of kernel code and data using legitimate code (i.e., kernel or driver code).

In order to detect such attacks, another group of defense techniques focus on identifying malware based on behavior [3], [4], [12], [25], [26]. These approaches generate malware signatures by using a pattern of malware code sequence (e.g., instruction sequences or system call sequences) to match malware behavior. However, some malware employ techniques that obfuscate or vary the patterns of code execution. For example, code obfuscation [11], [13], [47], [53] and code emulation [48] techniques can confuse behavior-based malware detectors and hence avoid detection.

This arms-race between malware and malware detectors centers around *properties of malicious code*: injection/integrity of code or the causal sequences of malicious code patterns. While the majority of existing work focuses on the *code* malware executes, relatively little work has been done which focuses on the *data* it modifies.

Data-centric approaches require neither the detection of code injection nor malicious code patterns. Therefore they are not directly subvertible using code reuse or obfuscation techniques. However, detecting malware based on data modifications has a unique challenge that makes it distinct from code-based approaches. Unlike code, which is typically expected to be invariant, data status can be dynamic. Correspondingly, conventional integrity checking cannot be applied to data properties. In addition, monitoring data objects of an operating system (OS) kernel has additional challenges because an OS may be the lowest software layer in conventional computing environments, meaning that there is no monitoring layer below it.

In this paper, we present a novel scheme, *data-centric OS kernel malware characterization* which enables the detection and characterization of OS kernel malware based on the properties of kernel data structures. Additionally, we present a prototype called DataGene and evaluate it against a set of real world kernel malware samples. This system consists of two essential components to monitor and analyze data properties of OS kernels.

The first component is a *kernel object mapping system* that externally identifies dynamic kernel objects of the monitored OS at runtime. This component enables an external monitor

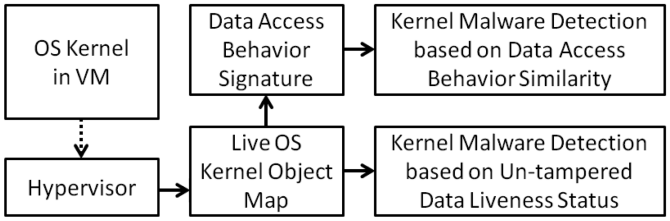


Fig. 1. Data-Centric OS Kernel Malware Characterization.

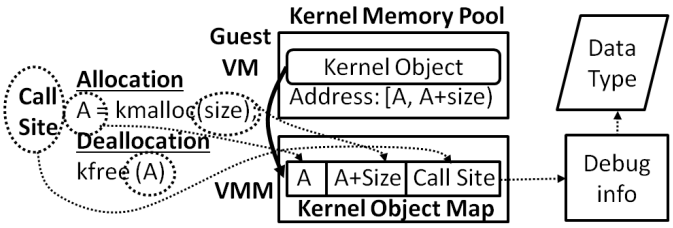


Fig. 2. Live Kernel Object Mapping System.

to recognize the access behavior to data objects. We make use of memory allocation events to build the object map. Some malware hides itself by manipulating data structures, and our experiments show that this map can reliably detect such attacks since its view is not manipulated by malware.

With this map in place, we then present a *malware characterization approach based on kernel object access patterns*. This approach can generate a signature of a malware's unique data access behavior. By matching data behavior signatures, it can detect classes of kernel malware that share common attack patterns on kernel data structures.

Contributions: The contributions of this paper are summarized as follows:

- **Reliable Detection of Kernel Object Hiding Attacks.** Kernel object hiding attacks attempt to hide data objects by manipulating pointers reaching such objects. Our kernel object mapping approach recognizes data objects based on memory allocation events, not inter-memory pointers. Therefore, such attacks do not tamper with the identification of data objects in our mapping scheme. Our experiments show that our approach successfully detects kernel data hiding rootkits that manipulate data object pointers in order to evade traditional rootkit detectors.
- **Conception of Malware Signature Based on Data Access Behavior During Attacks.** We propose a new malware signature based on the unique patterns of kernel data accesses that occur during an attack. This technique can complement code-based malware signatures.
- **Detection of Malware Variants Having Similar Data Access Patterns.** Our approach determines malware attacks by extracting and matching data access patterns specific to malware attacks. Kernel malware aiming at similar malicious features often manipulates common data structures. This mechanism can detect such malware variants having similar data access patterns.

This paper is organized as follows. In Section I, we present the problem statement. Section II introduces the approaches based on data properties. Sections III and IV present the details of those approaches. Sections V and VI present implementation and evaluation of our system. Section VIII presents related approaches in kernel malware defense and analysis. Section IX concludes this paper.

II. Data-Centric Kernel Malware Characterization

In this section, we present the overall design of data-centric kernel malware characterization. Fig. 1 illustrates our approach.

Tracking OS data allocations and uses is difficult because the OS is traditionally the lowest software layer in a conventional computer system. To overcome this challenge, we make use of virtualization technology. A guest OS runs on top of a hypervisor which transparently and efficiently captures memory related OS events to generate a kernel object map. This map is able to provide the live status of dynamic kernel objects. Many kernel rootkits are stealthy and attempt to hide themselves. Many of these attacks are implemented by manipulating data structures and making them appear dead (freed) to the OS when they are in fact alive (allocated). DataGene enables the detection of such malware based on the status of data liveness. This component is to be presented in Section III in details.

This map, which accurately identifies static and dynamic kernel data objects, enables the monitoring and analysis of kernel memory access patterns. Using this information we propose a new approach to characterize and detect kernel malware. DataGene monitors kernel memory access behavior such as reads and writes on OS kernel objects and systematically extracts memory reference patterns specific to malware attacks by comparing benign kernel execution and malicious kernel execution compromised by kernel rootkits. By matching these signatures DataGene enables the detection of kernel malware and their variants. This functionality will be presented in Section IV.

III. Live Kernel Object Mapping

DataGene uses the properties of kernel data objects for malware characterization. In this section, we introduce the allocation-driven mapping scheme which enables the creation of a live, dynamic map of kernel data object.

A. Allocation-Driven Mapping Scheme

Allocation-driven mapping is a kernel memory mapping scheme that generates a synchronous map of kernel objects by *capturing the kernel object allocation and deallocation events* of the monitored OS kernel. Fig. 2 illustrates how this scheme works. Whenever a kernel object is allocated or deallocated, the virtual machine monitor (VMM) intercedes and captures its address range and the information to derive the data type of the object subject to the event in order to update the kernel object map.

This approach does not rely on any content of the kernel memory which can potentially be manipulated by kernel malware. Therefore, the kernel object map provides an

un-tampered view of kernel memory wherein the identification of kernel data is not affected by the manipulation of memory contents by kernel malware. This tamper-resistant property is especially effective to detect sophisticated kernel attacks that directly manipulate kernel memory to hide kernel objects.

The key observation is that allocation-driven mapping captures the *liveness status* of the allocated dynamic kernel objects. For malware writers, this property makes it significantly more difficult to manipulate this view. In Section VI-B, we show how this mapping can be used to automatically detect data hiding attacks without using any knowledge specific to a kernel data structure.

There are a number of challenges in implementing a live kernel object map based on allocation-driven mapping. For example, kernel memory allocation functions do not provide a simple way to determine the type of the object being allocated.[1] One solution is to use static analysis to rewrite the kernel code to deliver the allocation types to the VMM, but this would require the construction of a new type-enabled kernel, which is not readily applicable to off-the-shelf systems. Instead, we use a technique that derives data types by using runtime context (i.e., call stack information). Specifically, this technique systematically captures code positions for memory allocation calls and translates them into data types so that OS kernels can be transparently supported without any change in the source code.

B. Techniques

We employ a number of techniques to implement allocation-driven mapping. First, a set of kernel functions (such as `kmalloc`) are designated as kernel memory allocation functions. If one of these functions is called, we say that an allocation event has occurred. Next, whenever this event occurs at runtime, the VMM intercedes and captures the allocated memory address range and the code location calling the memory allocation function. This code location is referred to as an *allocation call site* and we use it as a unique identifier for the allocated object's type at runtime. Finally, the source code around each allocation call site is analyzed offline to determine the type of the kernel object being allocated.

1) Runtime Kernel Object Map Generation: At runtime, the VMM captures all allocation and deallocation events by interceding whenever one of the allocation/deallocation functions is called. There are three things that need to be determined at runtime: (1) the call site, (2) the address of the object allocated or deallocated, and (3) the size of the allocated object.

To determine the call site, we use the return address of the call to the allocation function. In the instruction stream, the return address is the address of the instruction after the call instruction. The captured call site is stored in the kernel object map so that the type can be determined during offline source code analysis.

[1]Kernel level memory allocation functions are similar to user level ones. The function `kmalloc`, for example, does not take a type but a size to allocate memory.

The address and size of objects being allocated or deallocated can be derived from the arguments and return value. For an allocation function, the size is typically given as a function argument and the memory address as the return value. For a deallocation function, the address is typically given as a function argument. These values can be determined by the VMM by leveraging function call conventions. Function arguments are delivered through the stack or registers, and they are captured by inspecting these locations at the entry of memory allocation/deallocation calls. To capture the return value, we need to determine where the return value is stored and when it is stored there. Integers up to 32-bits as well as 32-bit pointers are delivered via the `EAX` register and all values that we would like to capture are either of those types. The return value is available in this register when the allocation function returns to the caller. In order to capture the return values at the correct time the VMM uses a virtual stack. When a memory allocation function is called, the return address is extracted and pushed on to this stack. When the address of the code to be executed matches the return address on the stack, the VMM intercedes and captures the return value from the `EAX` register.

2) Dynamic Data Type Inference: The object type information related to kernel memory allocation events is determined using static analysis of the kernel source code offline. First, the allocation call site of a dynamic object is mapped to the source code using debugging information found in the kernel binary. This code assigns the address of the allocated memory to a pointer variable at the left-hand side of the assignment statement. Since this variable's type can represent the type of the allocated memory, it is derived by traversing the declaration of this pointer and the definition of its type. Specifically, during the compilation of kernel source code, a parser sets the dependencies among the internal representations (IRs) of such code elements. Therefore, the type can be found by following the dependencies of the generated IRs. There are several patterns regarding how these components are related in the source code and such details are specifically described in [39].

IV. Data Behavior-Based Malware Characterization

In this section, we present how the data behavior of kernel malware is characterized and used to determine the presence of malware. The overview of this component is presented in Fig. 3, and the sub-components are as follows.

As a basic unit to represent the kernel's data behavior, DataGene generates a summary of the access patterns for all kernel objects accessed in a kernel execution instance. To identify dynamic kernel memory objects, this process takes advantage of a kernel object map (shown as The Kernel Memory Mapper in Fig. 3) described in the previous section. For each access on kernel memory in the guest OS, the VMM intercedes and records the relevant information about the kernel memory access, such as the accessing code, the accessed memory type, and the accessed offset (shown as The Data Behavior Aggregator).

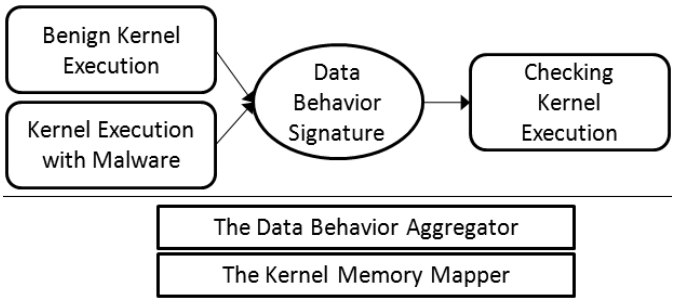


Fig. 3. Data Behavior-based Malware Characterization.

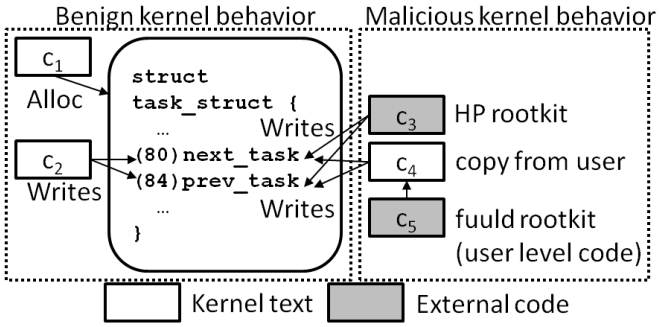


Fig. 4. An Example of Kernel Behavior.

To determine malware behavior, the memory access patterns for two kinds of kernel execution instances are generated: benign kernel runs and malicious kernel runs where kernel malware is active. By taking the difference between the two sets of memory access patterns, we estimate the data behavior specific to the kernel malware and generate its signature (Data Behavior Signature). Later, in order to detect kernel malware, the generated signatures are compared to the memory access patterns of a running instance of the OS (Checking Kernel Execution).

A. Data Behavior Profile Approach

In this section, we present basic terminologies that represent the memory access patterns of kernel execution.

Definition 1 (Data Behavior Element) *A data behavior element (DBE) represents a pattern of a memory access. It is defined as a quintuple, (c, o, m, i, f): the address of the code that accesses memory (c), the kind (read or write) of memory access (o), the kind (static or dynamic) of the accessed memory (m), the class of the accessed memory (i), and the accessed offset(s) (f) inside the memory of the class i.*

c is the address of the kernel code that reads or writes kernel memory. o represents the kind of memory access which is 0 for a memory read and 1 for a memory write.

The kind of the accessed memory, m, is 0 for a dynamic object and 1 for a static object. The class i is defined differently, depending on the memory kind. Static objects are known at compile time; therefore, we are able to assign unique numbers as their identifiers. A class of a static object can represent either a static data object or a kernel function in the kernel text. In the case of dynamic kernel objects, there are multiple memory instances for the same data type at runtime. Dynamic kernel objects allocated by the same code correspond to the data instances of the specific data type used in the allocation code. Thus, we aggregate the access patterns of dynamic kernel objects that share the allocation code. The address of this allocation code is used as a unique class for such objects.

f is an offset, or a range of offsets, accessed by the code at c. We allow a range of offsets because if this object is an array, the accessed offsets can vary for the same accessing code. Handling them as separate data behavior elements can cause a high number of elements with slightly different offsets for the same accessing code. To avoid this problem, we use a threshold (T_f) to convert a list of elements whose offsets are different (but with the same accessing code) to an element with an offset range.

Definition 2 (Kernel Execution Instance) *A kernel execution instance or a kernel run is an instance of the OS kernel execution.*

Definition 3 (Data Behavior Profile) *For a kernel execution instance r, a data behavior profile (DBP) is defined as a set of DBEs observed and it is denoted as D_r.*

A DBP represents a set of data behavior elements observed in a kernel execution instance. It is a summary of all observed kernel-mode memory access patterns in the kernel run.

Fig. 4 presents kernel code showing the examples of data behavior elements. The rounded box shows a dynamic kernel object allocated by the code c_1. This object is then accessed by the code c_2 and two fields, `next_task` (offset 80) and `prev_task` (offset 84), are written by it. Therefore, the data behavior elements for this code example are as follows.

$$(c_2, 1, 0, c_1, 80)\,, (c_2, 1, 0, c_1, 84)\,,$$

These elements are the access patterns in a benign kernel run. If kernel malware is active in this kernel, the access patterns can be extended due to the malware behavior. For instance, if kernel rootkits `hp` and `fuuld` are active as shown in the right-hand section of Fig. 4, there would be additional accesses to the `next_task` and the `prev_task` fields by the code c_3 and c_4. Consequently, the data behavior profile is extended with the additional elements as follows.

$$(c_3, 1, 0, c_1, 80)\,, (c_3, 1, 0, c_1, 84)\,,$$
$$(c_4, 1, 0, c_1, 80)\,, (c_4, 1, 0, c_1, 84)$$

Here c_3 represents the code of the `hp` rootkit, which is in the form of a kernel driver. The code integrity-based rootkit defense approach [42], [45] can determine this access as malicious based on the fact that this driver code is not in the authorized code list. In contrast, the code at c_4 is part of legitimate kernel code which is indirectly exploited to overwrite this data structure. This rootkit does not violate kernel code integrity; therefore, the approach based on code integrity cannot detect this attack behavior.

In both cases, malware behavior appears only when the malware runs. Our approach aims to capture such behavior specific to the attack in order to determine the presence of malware.

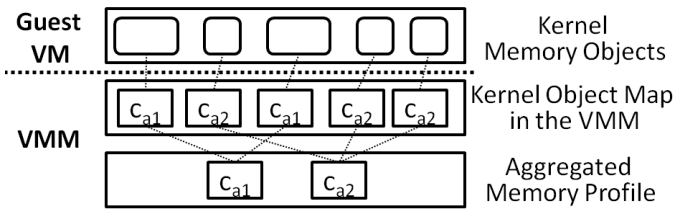


Fig. 5. Aggregating Memory Accesses on Dynamic Kernel Objects Regarding Allocation Sites c_{a1} and c_{a2}.

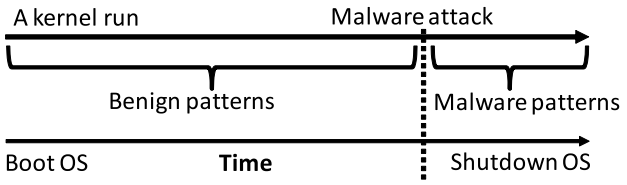


Fig. 6. Using a Single Kernel Run for Both Benign and Malware Memory Access Patterns.

In a kernel execution instance, there exist a varying number of dynamic kernel data instances. To compare the access patterns of dynamic kernel objects in different kernel runs, it is necessary to aggregate the memory accesses on such objects regarding their classes. The allocation code represents the instantiation of a data type at a specific code position. By using a memory allocation code site as the classifier of dynamic kernel objects, we aggregate the access patterns of dynamic instances of the same type.

Fig. 5 illustrates this aggregation process. When a dynamic kernel object is allocated in a guest OS kernel, the allocation code site is stored in the kernel memory map as the class information. Whenever kernel code reads or writes a dynamic kernel object, the VMM intercedes and identifies the targeted object by using its class information from the kernel object map. The memory access pattern is recorded in the aggregated memory profile.

B. Characterizing Malware Data Behavior

In this section we demonstrate how we characterize the behavior of kernel malware based on data behavior profiles. We first describe the challenges and describe how we address them. Then, we describe how we generalize malware behavior in order to match similar behavior in different malware.

1) Challenges and Our Solutions: DataGene characterizes malware behavior by using dynamic kernel execution. We list several challenges caused by our foundation on dynamic analysis. We then present our solutions for these challenges.

- **Variations in the Runtime Kernel Behavior.** Generally, the difficulty in obtaining a complete set of kernel execution paths is a well-known challenge for an approach based on dynamic execution. If we focus on the data behavior in benign execution, it is in fact a problem because the runtime kernel behavior can be highly dynamic across different runs. However, we focus on the data behavior specific to malware that consistently appears only when the malware is active.
- **Irregular Access Patterns on Kernel Stacks.** Kernel stacks are kernel objects that have irregular access patterns. Whenever a kernel function is called or returns, the stack is accessed for various purposes such as return values, function arguments, and local variables. Since the kernel control flow is highly dynamic, the set of code sites that access the stack and the accessed offsets within the stack vary significantly. Also, the contents of kernel stacks are irregular at different runs. As such, a simple way to handle this problem is to exclude stacks from our analysis. The kernel memory mapper provides the identifier for kernel stacks and we solve this problem by removing the information for such dynamic objects from the analysis.
- **Varying Offsets in Arrays.** Some data structures (e.g., arrays and buffers) have a range of space, a part of which can be used at runtime. For example, the accessed offsets of a buffer can be different depending on the data contained in it. This problem is handled by using multiple instances of kernel execution. If the accessed offset of memory is different in each execution, it is not used for a malware signature because it may not be used in another run. Only the data behavior that occurs in a consistent pattern when malware is active becomes a candidate for the signature.

2) Characterizing Malicious Data Behavior: In order to reliably characterize the data behavior of kernel malware in dynamic execution, we use multiple kernel runs in the signature generation stage. Let us call a DBP for a malicious kernel run j with malware M $D_{M,j}$, and $D_{B,k}$ represents a data behavior profile for a benign kernel execution k. We apply set operations on n malicious kernel runs and m benign runs as follows. The generated signature is called a *data behavior signature* for the malware M and shown as S_M.

$$S_M = \bigcap_{j\in[1,n]} D_{M,j} - \bigcup_{k\in[1,m]} D_{B,k} \tag{1}$$

This formula represents that S_M is the set of data behavior that *consistently* appears in n malware runs, but never appears in m benign runs. The underlying observation from this formula is that kernel malware will consistently perform malicious operations during attacks. This means, we can estimate malware behavior by taking the intersection of malicious runs. Such behavior should not occur in benign runs, so we subtract the union of benign runs from the derived malware behavior.

When we use kernel execution instances to generate malware signatures, the malicious runs and benign runs can be independent. They do not need to be, however. We can use the execution period before the attack as a benign run and consider only the new patterns after the attack as the malware kernel run if we have control on the launch of malware attacks as shown in Fig. 6. This technique prunes out a significant number of benign access patterns from the malicious kernel run, hence reducing risk for potential false positives.

False positives may occur if a consistent pattern in the malicious runs is later observed in a newly tested benign run.

The cause of this problem is not unknown kernel behavior, but rather a problem of proper pruning during signature generation. By exercising a variety of workloads in multiple kernel execution instances, we expect that such potential behavior for this error can be significantly reduced.

3) Generalizing Malware Code Identity: DataGene aims at matching the variants of the rootkits whose signatures are available. For example, DataGene can be used to inspect suspicious data activity in the execution of new signed drivers (which may include hidden malicious code), the execution of an unknown driver (which may be malware or its variant), or kernel execution (where legitimate kernel code can be exploited indirectly for attacks).

In order to cover variants of malicious code, DataGene does not use specific identification of kernel drivers. When we generate or test signatures, we generalize the information specific to kernel drivers, thus allowing signatures to be tested against any driver. Specifically, when the signature for a driver-based rootkit is generated, all code sites in this malicious driver are substituted by a single anonymous code site, ε. Some rootkits allocate memory and place their code on it, and any code site in such memory is also generalized as ε. In this process, we also generalize all benign kernel modules in the same way and subtract their memory access patterns from the candidates for the signature to collect only the behavior specific to the malware.

If a piece of malware does not use a driver, but instead exploits legitimate code (e.g., the rootkits using memory devices or return-oriented rootkits) then this will result in access patterns of legitimate code that are not observed in benign runs. In addition, when we match a malware signature with the data behavior profile of a kernel run, we generalize the driver code in the tested run similarly for comparison.

4) Matching a Malware Signature With a Kernel Run: The likelihood that a malware program M is present in a tested run r is determined by deriving a set of data behavior elements in S_M which belong to the data behavior profile, D_r. This set I corresponds to the intersection of S_M and D_r[2] (i.e., $I = \{i | i \in S_M \wedge i \in D_r\}$).

V. Implementation

We have implemented DataGene in a software virtualization system and applied it to Linux based operating systems. While our approach is general enough to work with any OS that follows standard function call conventions (e.g., Linux, Windows, etc.), our prototype supports three off-the-shelf Linux OSes of different kernel versions: Fedora Core 6, Debian Sarge, and Redhat 8. For the virtual machine monitor, any software virtualization system, such as VMware Workstation [52], VirtualBox [24], and Parallels [34] can be used for implementation. We choose QEMU [5] with the KQEMU optimizer for implementation convenience.

In this section, we will discuss more details about our implementation and the challenges associated with it.

[2]The data behavior signature (S_M) is a data behavior profile (i.e., a set of data behavior elements) because it is derived by the intersection and union of data behavior profiles.

A. Live Kernel Object Map

In the kernel source code, many wrappers are used for kernel memory management, some of which are defined as macros or inline functions and others as regular functions. Macros and inline functions are resolved as the core memory function calls at compile time by a preprocessor; thus, their call sites are captured in the same way as core functions. However, in the case of regular wrapper functions, the call sites will belong to the wrapper code.

To solve this problem, we take two approaches. If a wrapper is used only a few times, we consider that the type from the wrapper can indirectly imply the type used in the wrapper's caller due to its limited use. If a wrapper is widely used in many places (e.g., `kmem_cache_alloc` – a slab allocator), we treat it as a memory allocation function. Commodity OSes, which have mature code quality, have a well defined set of memory wrapper functions that the kernel and driver code commonly use. In our experience, capturing such wrappers, in addition to the core memory functions, can cover the majority of the memory allocation and deallocation operations.

We categorize the captured functions into four classes: (1) page allocation/free functions, (2) `kmalloc/kfree` functions, (3) `kmem_cache_alloc/free` functions (slab allocators), and (4) `vmalloc/vfree` functions (contiguous memory allocators). These sets include the well defined wrapper functions as well as the core memory functions. In our prototype, we capture about 20 functions in each guest kernel. The memory functions of an OS kernel can be determined from its design specification (e.g., the Linux Kernel API), kernel source code, or tracing sample runs.

Automatic translation of a call site to a data type requires a kernel binary that is compiled with a debugging flag (e.g., `-g` to `gcc`) and whose symbols are not stripped. Modern OSes, such as Ubuntu, Fedora, and Windows, generate kernel binaries of this form. Upon distribution, typically the stripped kernel binaries are shipped; however, unstripped binaries (or symbol information in Windows) are optionally provided for kernel debugging purposes. In our experiments we found that the kernels of Debian Sarge and Redhat 8 are not compiled with this debugging flag. Therefore, we compiled the distributed source code and generated the debug-enabled kernels. These kernels share the same source code with the distributed kernels, but the offset of the compiled binary code can be slightly different due to the additional debugging information.

For static analysis we use a `gcc` [22] compiler (version 3.2.3) that we instrumented to generate IRs for the source code of the experimented kernels. We place hooks in the parser to extract the abstract syntax trees for the code elements necessary for static code analysis.

B. Data Behavior-Based Characterization

We implement the kernel object mapper and the data aggregator in the VMM. When there is a request to the VMM, a DBP is written to a file in the host OS. In order to detect kernel malware, the data behavior profile can be generated on the fly and periodically compared with the signature while the OS is running.

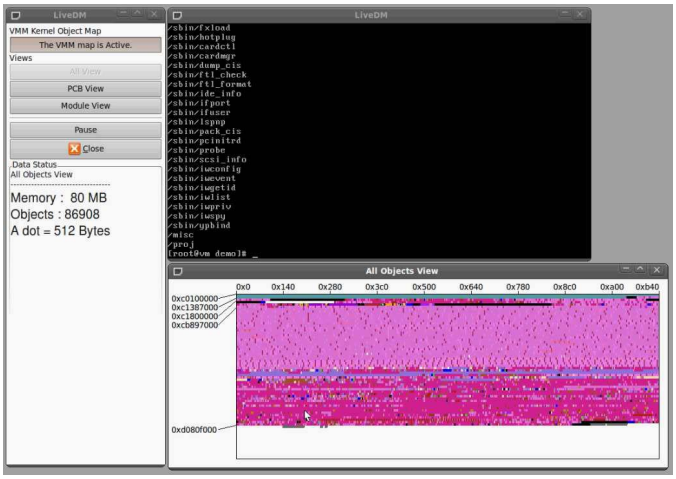


Fig. 7. A Snapshot of a Live Kernel Object Map.

During benign runs we performed various workload from daily commands to non-trivial application benchmarks. The tested workload includes kernel compilation, Apache webserver, UnixBench, nbench, mysql database, thttp webserver, `find`, `gzip`, `ssh`, `scp`, `lsmod`, `ps`, `top`, and `ls`. Some workloads were executed for several hours to allow any background administrative operation to be performed. We also used the workload of benign module loading and simple operations making use of the `/dev/kmem` device (e.g., open and close without overwriting kernel memory).

In our experiments we measured the quality of signatures, whether they trigger false positives, as we increased the number of benign runs and malicious runs used for generating malware signatures. We found with five or more sets of benign runs and malicious runs, we could generate signatures that do not cause false positives in our testing with newly generated benign runs. Therefore, in the next section we present the data of these five sets of runs. However, we believe that a large number of runs will further improve the quality of signatures.

VI. Evaluation

We have evaluated our system on a server containing a 3.2Ghz Pentium D CPU and 2GB RAM. The guest VMs being monitored are configured with 256MB RAM.

A. Live Kernel Object Map

In this section, we evaluate the functionality of live kernel object mapping with respect to the identification of kernel objects.

1) Runtime Tracking of Dynamic Kernel Objects: The live kernel object map synchronously identifies dynamic kernel objects on their allocations and deallocations. Therefore, unlike other kernel memory mapping approaches that sample memory status or traverse memory snapshots, it can continuously track changes in kernel memory state. Fig. 7 illustrates the GUI interface of our prototype. The black screen at the top shows the guest operating system. The kernel object map is illustrated below the screen. The statistics of current kernel objects are shown in the left pane.

TABLE I
Allocation Call Sites, Derived Data Types, and the Number of Core Dynamic Kernel Objects

	Allocation Call Site	Data Type	#Objects
Task/Sig	fork.c:248	task_struct	66
	fork.c:801	sighand_struct	63
	exec.c:601	sighand_struct	1
	fork.c:819	signal_struct	66
Memory	pgtable.c:229	pgd_t	54
	fork.c:433	mm_struct	47
	fork.c:559	mm_struct	7
	fork.c:314	vm_area_struct	149
	mmap.c:923	vm_area_struct	1004
	mmap.c:1526	vm_area_struct	5
	mmap.c:1722	vm_area_struct	48
	exec.c:402	vm_area_struct	47
File System	fork.c:677	files_struct	54
	fork.c:597	fs_struct	53
	file_table.c:76	file	531
	buffer.c:3062	buffer_head	828
	block_dev.c:232	bdev_inode	5
	dcache.c:692	dentry	4203
	inode.c:112	inode	1209
	namespace.c:55	vfsmount	16
	inode.c:93	proc_inode	237
	ll_rw_blk.c:1405	request_queue_t	18
	ll_rw_blk.c:2950	io_context	10
Network	socket.c:279	socket_alloc	12
	sock.c:617	sock	3
	dst.c:125	dst_entry	5
	neighbour.c:265	neighbour	1
	tcp_ipv4.c:134	tcp_bind_bucket	4
	fib_hash.c:586	fib_node	9

2) Identifying Dynamic Kernel Objects: To demonstrate the ability to inspect the runtime status of an OS kernel, Table I presents a list of important kernel data structures captured during the execution of Debian Sarge. These data structures manage key OS status information such as process information, memory mapping of each process, and the status of file systems and the network. This information is often targeted by kernel malware and kernel bugs [31], [35]–[38], [44], [49], [50]. Kernel objects are recognized using allocation call sites shown in column **Allocation Call Site** during runtime. Using static analysis, this information is translated into the data types shown in column **Data Type** [39]. The number of the identified objects in the inspected runtime status is presented in column **#Objects**. At that time instance, the live kernel object map had identified a total of 29488 dynamic kernel objects with their data types derived from 231 allocation code positions.

In order to evaluate the accuracy of the identified kernel objects, we built a reference kernel where we modify kernel memory functions to generate a log of dynamic kernel objects and run this kernel with the live kernel object map. We observe that the dynamic objects from the log accurately match the live dynamic kernel objects captured by the live memory map. To check the type derivation accuracy, we manually translate the captured call sites to data types by traversing kernel source code as done by related approaches [9], [15]. The types derived manually match the results from our automatic static code analysis.

B. Detecting Data Hiding Malware Attacks

Existing memory map approaches [2], [9], [36], [44], [54] identify memory objects by asynchronously scanning the

TABLE II
DETECTION OF DKOM DATA HIDING ROOTKITS USING THE UN-TAMPERED VIEW OF LIVE KERNEL OBJECTS

Rootkit Name	$\|L\|$ - $\|S\|$	Manipulated Kernel Object	
		Type	Field
FL	# of hidden PCBs	task_struct	next_task, prev_task
HL	# of hidden modules	module	next
hp	# of hidden PCBs	task_struct	next_task, prev_task
LF	# of hidden PCBs	task_struct	next_task, prev_task
CL	# of hidden modules	module	next
kis	1 (rootkit self-hiding)	module	next
MH	# of hidden modules	module	next
MH1	1 (rootkit self-hiding)	module	next
AD	1 (rootkit self-hiding)	module	list.next, list.prev
EY	1 (rootkit self-hiding)	module	list.next, list.prev

FL: fuuld, HL: hide_lkm, LF: linuxfu, CL: cleaner, MH: modhide, MH1: modhide1, AD: adore-ng-2.6, EY: ENYELKM 1.1

pointers in the memory image. Therefore, they are not able to detect manipulation of objects without relying on some inconsistency of data. In this section we present a reliable hidden kernel object detector built on top of allocation mapping that does not suffer from this limitation.

Some advanced kernel rootkits hide kernel objects by simply removing all references to them from the kernel's dynamic memory. We model the behavior of this type of data hiding attack as a data anomaly in a list. If a dynamic kernel object does not appear in a kernel object list, then it is orphaned and hence an anomaly.

Allocation-driven mapping provides an *un-tampered view* of the kernel objects not affected by manipulation of the actual kernel memory content. Therefore, if a kernel object appears in the map but cannot be found by traversing the kernel memory, then that object has been hidden. More formally, for a set of dynamic kernel objects of a given data type, a live set L is the set of objects found in the kernel object map. A scanned set S is the set of kernel objects found by traversing the kernel memory as in the related approaches [2], [9], [36]. If L and S do not match, then a data anomaly will be reported.

There are two dynamic kernel data lists which are favored by rootkits as attack targets: the kernel module list and the process control block (PCB) list.[3] However, other linked list-based data structures can be similarly supported as well. The basic procedure is to generate the live set L and periodically generate and compare with the scanned set S. We tested 8 real-world rootkits and 2 of our own rootkits (`linuxfu` and `fuuld`) previously used in [29], [40], and [44]. All of these rootkits commonly hide kernel objects by directly manipulating the pointers of such objects. Our map successfully detected all of these attacks by detecting the data anomaly. The detailed results are available in Table II.

In the experiments, we focus on a specific attack mechanism – data hiding via direct kernel object manipulation (DKOM) – rather than the attack vectors of rootkits. This means that our system can still detect malware that uses a previously unknown attack vector in order to manipulate kernel data structures. For example, a large number of rootkits are based on loadable kernel module (LKM), which can be detected by code integrity approaches [42], [45] or with a kernel module signing and verification scheme. However, there exist alternate attack vectors such as `/dev/mem`, `/dev/kmem` devices, return-oriented techniques [23], [46], kernel bugs, and unproven code in third-party kernel drivers which can elude existing kernel rootkit detection and prevention approaches. We present the DKOM data hiding cases of LKM-based rootkits as part of our results because these rootkits can be easily converted to make use of these alternate attack vectors.

We also include results for two other rootkits that make use of these advanced attack techniques. `hide_lkm` and `fuuld` in Table II respectively hide kernel modules and processes without any kernel code integrity violation (via `/dev/kmem`), and existing rootkit defense approaches cannot properly detect these attacks. However, our monitor effectively detects all DKOM data hiding attacks regardless of attack vectors.

In the experiments that detect rootkit attacks, we generate and compare L and S sets every 10 seconds. When a data anomaly occurs, the check is repeated in 1 second. (The repeated check ensures that a kernel data structure was not simply in an inconsistent state during the first scan.) If the anomaly persists, then we signal that an anomaly has been detected.

With these monitoring policies, we successfully detected all tested DKOM hiding attacks without any false positives or false negatives.

So far, we have presented the detection of kernel malware which achieves its malicious functionality by hiding kernel data structures. DKOM data hiding techniques are simple to perform (i.e., isolation of data) but very challenging to detect due to non-deterministic locations and values of dynamic kernel objects. In addition to data hiding, malware can manipulate kernel data to perform a variety of other types of attacks such as privilege escalation of a backdoor process and manipulating statistics and information stored in the kernel. Due to the fact that all these attacks are performed by a manipulation of kernel data, they can be modeled in terms of kernel data access behavior. In the next section, we present the detection of a wider scope of kernel malware beyond DKOM data hiding rootkits.

C. Data Behavior-Based Malware Characterization

In this section we evaluate the effectiveness of malware characterization based on data behavior signatures as follows. First, we extract the signatures of three classic rootkits and match them with benign and malicious kernel runs. Second, we compare the signatures of all of the tested kernel rootkits to determine common data behavior across different rootkits and how such common behavior can be used to detect rootkit variants. Third, we list specific data elements that are shared by rootkit signatures, which provide an in-depth understanding of the attack operations that are common across kernel rootkits.

1) Malware Signature Generation: When a data behavior signature is generated, the information specific to the malicious

[3]A process control block (PCB) is a kernel data structure containing administrative information for a particular process. Its data type in Linux is `task_struct`.

code is largely generalized. Therefore, we hypothesize that data behavior signatures may be effective not only to detect the malware whose signature is available, but also to determine the presence of related malware. In order to validate this hypothesis, we generated the signatures of three representative, classic rootkits, and tested benign kernel runs and malicious kernel runs with 16 rootkits.

To generate malware signatures, we chose three rootkits, `adore 0.38`, `SucKIT`, and `modhide`. We chose these three for the following reasons: The `adore` rootkit has been studied in several rootkit defense approaches [35], [36], [42], [44]. This rootkit has several versions with differences in features and we chose an old version, 0.38, for the signature to evaluate its effectiveness toward newer rootkit versions (0.53 and 1.56). `SucKIT` is known for its attack vector, the `/dev/kmem` device, that avoids using a conventional driver-based mechanism [18]. Several other rootkits followed by using the same attack vector. `modhide` is a rootkit packaged with various versions of the `adore` rootkit to hide it from the list of kernel modules. We present our results for other rootkit choices in Section VI-C.3.

To generate each malware signature, we used kernel data behavior profiles (DBP) for both benign and malicious kernel execution. For benign behavior we used a diverse set of workloads including booting & shutdown, kernel compilation, apache, mysql, nbench, unixbench, and thttpd. To determine how many DBPs would be necessary for analysis, we computed the cumulative union behavior of profiles with a random order of workload. Figure 8 shows that after taking the union of seven DBPs, the kernel behavior patterns are stabilized for our workloads. This data suggests at least seven profile runs should be used to derive reliable malware signatures. This number, however, could vary depending on the dynamics of the workload. To collect stable profiles conservatively, we did not stop at seven runs. Instead, we used 15 benign runs, slightly over the twice of the number of runs that we observed the stable cumulative patterns, for our experiments.

For malicious kernel DBPs, we take the intersection of behavior to extract consistent behavior across attacks. Figure 9 shows the cumulative intersection behavior of `adore` 0.38 rootkit attacks. Since the rootkit does not vary its behavior in each attack instance, the common attack behavior is converged upon quickly even with only a few malware attack samples. In particular, the rootkits that we chose for signatures commonly show stable behavior only with two runs. Similar to our practice used for the benign case, we conservatively collected about twice this many runs. Hence, we used five malicious kernel runs to generate malware signatures.

Table III shows the summary of benign and malicious kernel execution instances (D) and the generated signatures (S). In all data behavior profiles measured, we set the threshold for aggregating offsets (T_f) as 15. Thus, we consider an object as an array if more than 15 offsets within the object are *swept over by the common code*. Typically, different data fields have corresponding sets of accessing code because most APIs access relevant data fields and do not scan the whole data object through. For some array-like data fields or strings, T_f lowers the granularity of analysis by managing a set of addresses as a range instead of many individual items. The value 15 was determined for our experiments through empirical testing.

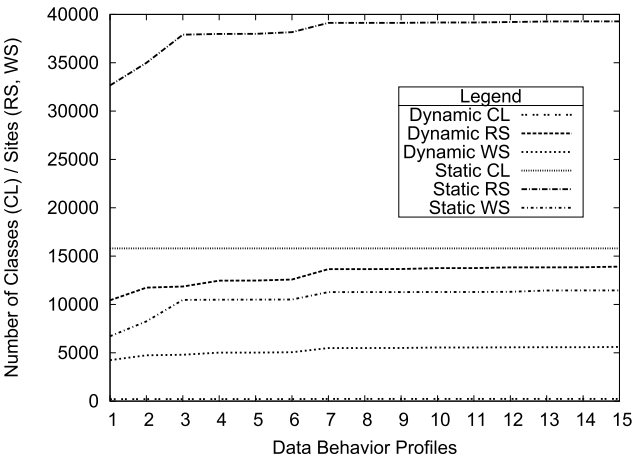


Fig. 8. Cumulative Union Characteristics of Benign DBPs. CL: Classes, RS: Read Sites, WS: Write Sites.

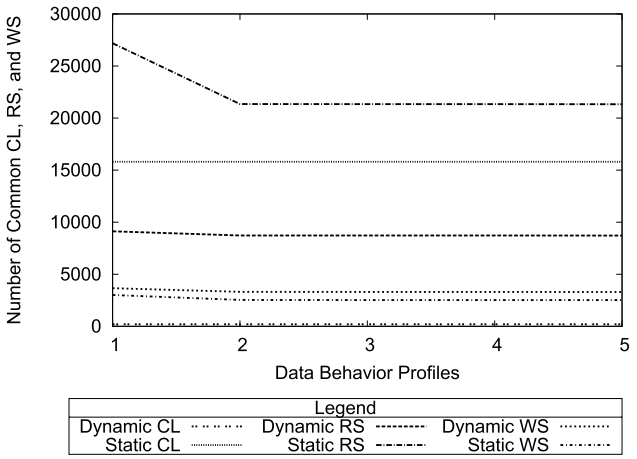


Fig. 9. Cumulative Intersection Characteristics of DBPs for `adore` 0.38 Rootkit Attacks.

TABLE III
DETAILS OF BENIGN AND MALICIOUS KERNEL DBPS (D) AND SIGNATURES (S). CL: # OF CLASSES, RS: # OF READ SITES, WS: # OF WRITE SITES

	D/ S	Dynamic Objects			Static Objects		
		CL	RS	WS	CL	RS	WS
Benign runs	$\bigcup D$	221	13918	5608	15800	39283	11449
`adore 0.38`	$\bigcap D$	193	8716	3296	15800	21333	2518
	S	2	1	2	1	1	1
`SucKIT`	$\bigcap D$	193	8720	3303	15800	22564	2515
	S	5	13	8	1192	1212	6
`modhide`	$\bigcap D$	192	8608	3276	15800	21306	2517
	S	1	0	1	0	0	0

Table IV presents the details of our three sample rootkits. The data behavior signatures of the `adore`, `SucKIT`, and `modhide` rootkits have 35, 12010, and 1 data behavior elements (DBEs), respectively. `SucKIT` has a significantly high number of elements because it scans kernel memory to collect information about the attack targets (e.g., the

TABLE IV
DETAILS OF THE ROOTKIT SIGNATURES. CL: # OF CLASSES, RD: # OF READ DBES, WD: # OF WRITE DBES

Rootkit Name	Dynamic Objects			Static Objects			Total
	CL	RD	WD	CL	RD	WD	DBE
adore 0.38	2	5	14	1	8	7	35
SucKIT	5	29	12	1192	11963	6	12010
modhide	1	0	1	0	0	0	1

system-call table), and this behavior is observed as reading numerous static objects with a variety of offsets. The modhide rootkit simply manipulates the kernel module list; thus, it has only one element.

2) False Positive Analysis: To evaluate the false positives of the generated signatures, we compared the signatures with new benign kernel execution instances. In these extra, benign kernel runs we ran additional workloads not included during our initial signature generation phase in order to ensure more code paths and data operations were executed than previously. In this experiment, no false positive cases were found, which confirms that our signature generation procedure captures a reasonably close set of the data behavior specific to the kernel rootkits and that the tested runs did not contain any data behavior that appears in the signatures.

3) Detecting Rootkits Using Data Behavior Signatures: Malicious kernel runs were next tested by using three signatures to determine any running malware based on the similarity of the data access patterns between the compared signature and the kernel run. We tested a total of 80 kernel runs of 16 rootkits having a variety of targets and attack vectors. For instance, seven rootkits (fuuld, hide_lkm, hp, linuxfu, cleaner, modhide, and modhide1) directly manipulate kernel objects (DKOM [7]). Four rootkits (fuuld, hide_lkm, SucKIT, and superkit) manipulate kernel memory by using the /dev/kmem memory device, among which two rootkits (fuuld and hide_lkm) directly manipulate only kernel data and do not violate kernel code integrity. Therefore, they are not detected by code integrity-based defense systems [42], [45].

For this testing we use a slightly different set of rootkits than the DKOM hiding rootkits evaluated in Section VI-B (Table II). Among these, two rootkits, adore-ng-2.6 and ENYELKM 1.1, are not included in this evaluation due to the fact that they require a specific OS platform that is supported by the live kernel object map, but not by the system as a whole. Incompatibilities such as this are not uncommon in rootkit defense research, and we have parallel work [41] that is meant to address this issue for future research in the area. We would like to note that, at a fundamental level, these two rootkits have behavior similar to other rootkits which were tested, and there is no reason to believe that they would be more difficult to detect.

Table V presents the number of matched data behavior elements between signatures and kernel runs with rootkits (I). Two left-hand columns show the information about signatures: the name (M) of the rootkit used for the signature and the size of the signature ($|S_M|$). The remaining 16 columns present the number of data behavior elements common in the compared signature (based on the rootkit in the row heading) and the kernel run (where the rootkit in the column heading is active).

We consider a tested run to include malware if it contains a DBE that matches a known malware signature. In our experiments, all kernel runs with rootkits share elements with one or more signatures (shown in the row at the bottom of the table), leading to the detection of all 16 kernel rootkits.

One potential question regarding malware signatures would be the selection of kernel rootkits for signatures. To understand which signatures would be effective on which rootkits, we performed a more comprehensive set of experiments using different rootkits for signatures. We first generated the rootkit signatures of all 16 kernel rootkits using five malicious kernel runs and 15 benign kernel runs. Then we applied them to the kernel runs (different sets from the ones used for signature generation) contaminated by 16 kernel rootkits.

The comparison result is presented in Table VI. When the rootkits in the signature and the tested run are matched, the entire signature is matched (# matched DBE = $|S_M|$, the numbers are shown in italics). The bottom row shows that given a rootkit in the column heading, how many rootkit signatures other than its own signature can detect the rootkit. This number varies from 2 to 10 depending on how many similar rootkits exist in the set of our experiments. On average more than six rootkit signatures are able to detect a given rootkit.

4) Similarities Among Data Behavior Signatures: In this section we quantitatively analyze the similarities in data behavior across rootkits by generating and comparing the signatures of the tested rootkits.

We calculated the similarities among signatures by comparing the signatures of 16 kernel rootkits with one another. Our experiments reveal that each rootkit shares its data behavior with 2∼10 other rootkits (more than six rootkits on average) which is consistent with the results of the cross comparison in the previous section.

The rootkits show similar data behavior not only among close variants, (e.g., different versions of adore) but also across rootkits having different attack mechanisms. For example, the /dev/kmem based SucKIT shows similarities with driver-based rootkits such as knark and kis, despite the fact that they are not derived from one another.

The strong similarities of data behavior across rootkits are visualized in Fig. 10. The family of adore rootkits are strongly related in general. The adore-ng 1.56 is connected to other versions with less strong connections, thick dashed arrows, because in newer adore versions, the internal attack vector is substantially changed to use dynamic objects instead of static objects. A group of rootkits using the /dev/kmem memory device (i.e., SucKIT, hide_lkm, fuuld, and superkit) have a strong relationship to one another. SucKIT and superkit are especially connected by using thick solid arrows because they share a majority of data behavior. Some rootkits have relationships with different kinds of rootkits. For example, the kis rootkit is connected to other driver-based rootkits such as the adore rootkits and the knark rootkit, but it is also closely related to /dev/kmem based rootkits such as the SucKIT.

TABLE V

THE NUMBER OF MATCHED DATA BEHAVIOR ELEMENTS BETWEEN THREE ROOTKIT SIGNATURES AND THE KERNEL RUNS WITH 16 KERNEL ROOTKITS (AVERAGE OF 5 RUNS)

Signature (S_M)		# of matched DBEs between S_M and the kernel runs with the rootkits shown below ($\|I\|$).															
M	$\|S_M\|$	AD1	AD2	AD3	FL	HL	SK	ST	hp	kbdv3	knark	LF	Rial	CL	kis	MH	MH1
AD1	35	35	30	14	0	0	2	2	2	5	20	3	4	0	2	0	0
SK	12010	2	1	1	16	16	12010	11983	0	0	1	0	0	0	16	0	0
MH	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1
Detected		√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√
# of effective S_M		2	2	2	1	1	2	2	1	1	2	1	1	1	2	1	1

AD1: adore 0.38, AD2: adore 0.53, AD3: adore-ng 1.56, FL: fuuld, HL: hide_lkm, SK: SucKIT, ST: superkit, LF: linuxfu, CL: cleaner, MH: modhide, MH1: modhide1

TABLE VI

THE NUMBER OF COMMON DATA BEHAVIOR ELEMENTS BETWEEN 16 ROOTKIT SIGNATURES AND THE KERNEL RUNS WITH 16 KERNEL ROOTKITS (AVERAGE OF 5 RUNS)

Signature (S_M)		# of matched DBEs between S_M and the kernel runs with the rootkits shown below ($\|I\|$).															
M	$\|S_M\|$	AD1	AD2	AD3	FL	HL	SK	ST	hp	kbdv3	knark	LF	Rial	CL	kis	MH	MH1
AD1	35	*35*	30	14	0	0	2	2	2	5	20	3	4	0	2	0	0
AD2	46	30	*46*	24	0	0	1	1	2	5	19	2	4	0	2	0	0
AD3	97	14	24	*97*	0	0	1	1	2	4	9	6	0	2	2	0	0
FL	19	0	0	0	*19*	13	16	16	0	0	0	0	0	0	0	0	0
HL	3406	0	0	0	13	*3406*	13	13	0	0	0	0	0	0	0	0	0
SK	12010	2	1	1	16	13	*12010*	11983	0	0	1	0	0	0	16	0	0
ST	11998	2	1	1	16	13	11983	*11998*	0	0	1	0	0	0	1	0	0
hp	17	2	2	2	0	0	0	0	*17*	0	1	5	0	0	1	0	0
kbdv3	16	5	5	4	0	0	0	0	0	*16*	4	0	0	0	0	0	0
knark	67	20	19	9	0	0	1	1	1	4	*67*	1	4	0	2	0	0
LF	24	3	2	6	0	0	0	0	5	0	1	*24*	0	0	1	0	0
Rial	46	4	4	0	0	0	0	0	0	0	4	0	*46*	0	0	0	2
CL	3	0	0	2	0	0	0	0	0	0	0	0	0	*3*	0	1	1
kis	31203	2	2	2	0	0	16	1	1	0	2	1	0	0	*31203*	0	2
MH	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	*1*	1
MH1	6	0	0	0	0	0	0	0	0	0	0	0	2	1	2	1	*6*
# of effective S_M		10	10	10	3	3	8	8	6	4	10	6	4	3	9	2	4

AD1: adore 0.38, AD2: adore 0.53, AD3: adore-ng 1.56, FL: fuuld, HL: hide_lkm, SK: SucKIT, ST: superkit, LF: linuxfu, CL: cleaner, MH: modhide, MH1: modhide1

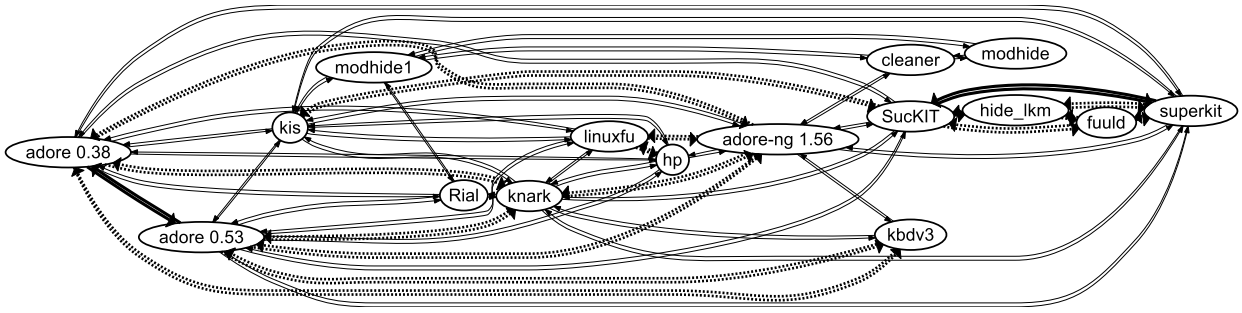


Fig. 10. Similarities Among the Data Behavior of Rootkits. Types of Arrows ($|I|$: # of Matched Elements): Thin Solid ($0 < |I| < 5$), Thick Dashed ($5 <= |I| < 25$), and Thick Solid ($|I| >= 25$).

In summary, the data behavior is not only common in the family of rootkits or similar kinds, but also is available across different kinds of rootkits. The signatures of these related rootkits can be interchangeably used to detect one another.

5) Extracting Common Data Behavior Elements: In this section we demonstrate the details of common rootkit attacks which are systematically extracted based on similarities in rootkit data behaviors. The data behavior elements (DBEs) from the signatures of all experimented rootkits are ranked with the order of the appearance in rootkits' signatures (N). The top DBEs are presented in Table VII after being classified into several categories.

The first three columns present the information regarding rootkits which share data behavior elements. The number N and the names of rootkits whose signatures share a DBE are listed. A short description of the DBE is provided in the next column.

The next five columns present the contents of the DBEs: the accessing code (c); the kind of memory access (o) such as a read (R) or a write (W); the kind of accessed memory

TABLE VII
TOP COMMON DATA BEHAVIOR ELEMENTS AMONG THE SIGNATURES OF 16 ROOTKITS

Rootkits			Accessing code		Accessed data			
N	Rootkits with common behavior	Rootkit behavior	Code (c)	o	m	Data class (i)	Field,Offset (f)
7	AD1, AD2, AD3, hp, knark, LF, kis	Reading a process's ID	ε	R	D	task_struct	pid
6	AD1, AD2, AD3, SK, ST, knark	Reading a process's flag	ε	R	D	task_struct	flags
5	AD1, AD2, AD3, kbdv3, knark	Privilege escalation	ε	W	D	task_struct	uid, euid, gid, egid
5	AD1, AD2, AD3, hp, LF	Listing processes	ε	R	D	task_struct	next_task
4	AD1, SK, ST, kis	Setting an address space	ε	W	D	task_struct	addr_limit
4	AD1, AD2, AD3, knark	Privilege escalation	ε	W	D	task_struct	suid, fsuid, fsgid
3	AD1, AD2, AD3	Privilege escalation	ε	W	D	task_struct	cap_effective
3	AD1, AD2, AD3	Privilege escalation	ε	W	D	task_struct	cap_inheritable
3	AD1, AD2, AD3	Privilege escalation	ε	W	D	task_struct	cap_permitted
3	AD1, AD2, kbdv3	Reading a user's ID	ε	R	D	task_struct	uid
3	AD1, AD3, LF	Reading a process' name	ε	R	D	task_struct	comm
2	hp, LF	Hiding a process	ε	W	D	task_struct	next_task, prev_task
4	FL, HL, SK, ST	Manipulation via /dev/kmem	read_kmem, write_kmem	R,W	D	file	f_pos
4	FL, HL, SK, ST	Manipulation via /dev/kmem	memory_lseek	W	D	file	f_pos
3	FL, SK, ST	Manipulation via /dev/kmem	do_write_mem	R,W	D	file	f_pos
3	CL, MH, MH1	Hiding a kernel module	ε	W	D	module	next
2	kis, MH1	Hiding a kernel module	ε	W	S	module_list	0
4	AD1, AD2, knark, Rial	Hijacking a system call	ε	W	S	sys_call_table	# 141
3	AD1, AD2, knark	Hijacking a system call	ε	W	S	sys_call_table	# 2,37,120,220
3	AD1, AD2, Rial	Hijacking a system call	ε	W	S	sys_call_table	# 6
2	Rial, MH1	Hijacking a system call	ε	W	S	sys_call_table	# 5
2	knark, Rial	Hijacking a system call	ε	W	S	sys_call_table	# 3
2	SK, ST	Hijacking a system call	ε	W	S	sys_call_table	# 59
2	SK, ST	Hijacking a system call	__generic_copy_from_user	W	S	sys_call_table	# 59
2	AD1, AD2	Hijacking a system call	ε	W	S	sys_call_table	# 39
2	AD2, AD3	Hijacking a hook	ε	W	S	proc_root_inode_operations	lookup

(m) such as a dynamic object (D) or a static object (S); the accessed memory's class (i), which is converted to a data type for dynamic data or a variable name for static data; and the accessed offset(s) (f). The offset is converted to a field name if it corresponds to a specific field. If the accessed object is the system-call table, a system-call number (#) is presented by dividing the offset by the size of a pointer.

a) Attacks on process control blocks (PCBs): The first category at the top of Table VII lists the data behavior that targets a process control block. This is a core data structure that maintains administrative information about processes. Therefore, it is a major target of rootkits. Table VII shows that seven rootkits read the process ID numbers in PCBs during attacks. Several rootkits, such as the family of `adore` rootkits, the `kbdv3` rootkit, and the `knark` rootkit, provide a back-door that permits the root privilege to an ordinary user (privilege escalation). The `hp` and `linuxfu` rootkits manipulate the pointers connecting PCBs. This behavior is for hiding PCBs from the view of OS.

b) Attacks using /dev/kmem: The second category shows the rootkit behavior that manipulates kernel memory by using a memory device (e.g., `/dev/kmem`). This device allows a user program to read and write kernel memory like a file putting the kernel integrity at risk. The kernel runs compromised by `fuuld`, `hide_lkm`, `SucKIT`, and `superkit` rootkits commonly show specific data behavior that the memory related kernel functions access `file` objects.

c) Attacks on the kernel module list: The next category lists rootkit attacks on the kernel module list. The `next` pointer field of `module` objects are written by the `cleaner`, `modhide`, and `modhide1` rootkits. The `module` objects constitute the list of kernel modules and they are connected by this `next` pointer. The rootkit attacks that hide a module appear as direct manipulation of this field.

d) Attacks on static kernel objects: The last category is the manipulation of static kernel objects. Several rootkits hijack system-calls by replacing system-call table entries with the addresses of malicious functions. This behavior is captured by the manipulation of the system-call table by several code sites, depending on the attack vector. In the case of driver-based rootkits, such behavior is captured as access by the generalized rootkit code, ε. The rootkits based on memory devices (e.g., `/dev/kmem`) use legitimate kernel code for manipulation (e.g., `__generic_copy_from_user`).

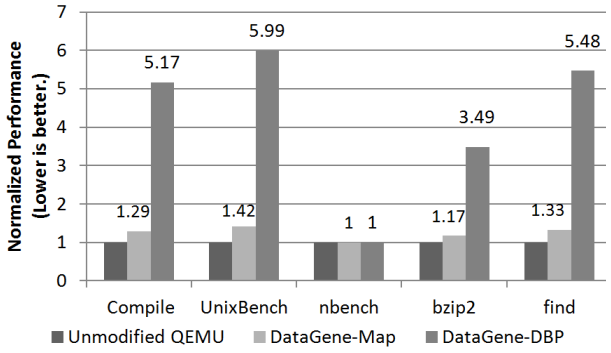


Fig. 11. Performance Comparison of QEMU and DataGene (DataGene-Map: Kernel Object Map, and DataGene-DBP: Data Behavior Profile).

D. Performance Evaluation

Since DataGene primarily targets non-production environments such as malware analysis honeypots, performance is not a primary concern. Still, we would like to provide a general idea of the cost of data-centric malware characterization.

We evaluated the performance of DataGene compared to unmodified QEMU. We performed five benchmarks : compiling the kernel source code, nbench, `bzip2`, the `find` utility, and UnixBench.

Fig. 11 presents the performance overhead of unmodified QEMU, DataGene with the live kernel object map (DataGene-Map), and DataGene with data behavior

profile support (DataGene-DBP). All performance numbers are normalized to the result of unmodified QEMU and a lower number represents a faster execution.

In DataGene-Map, the VMM only intercedes when the kernel executes kernel memory allocation and deallocation code. Therefore it has a $1 \sim 1.42$x overhead. DataGene-DBP intercedes on every kernel mode memory access to generate a data behavior profile which is the summary of all kernel mode memory access patterns. Therefore full DataGene has a higher performance overhead of $1 \sim 5.99$x.

Kernel compilation, UnixBench, and `find` intensively use system resources such as file systems, pipes, and processes. Such activities invoke kernel services such as system calls and page fault handling which indirectly triggers kernel-level memory activities, which causes a overhead greater than 5x. The nbench benchmark involves only user-level CPU workload. Both DataGene-Map and DataGene-DBP do not have additional overhead for this case. The `bzip2` benchmark involves both file system access and user-level computation. Therefore it causes a lower overhead compared to kernel compilation, UnixBench, and `find`.

VII. Discussion

Since DataGene operates in the VMM beneath the hardware interface, we assume that kernel malware cannot directly access DataGene code or data. However, it can exhibit potentially obfuscating behavior to confuse the view seen by DataGene. Here we describe several scenarios in which malware can affect DataGene and our counter-strategies to detect them.

First, malware can implement its own custom memory allocators to bypass DataGene observation. This attack behavior can be detected based on the observation that any memory allocator must use internal kernel data structures to manage memory regions or its memory may be accidentally re-allocated by the legitimate memory allocator. Therefore, we can detect unverified memory allocations by comparing the resource usage described in the kernel data structures with the amount of memory being tracked by DataGene. Any deviance may indicate the presence of a custom memory allocator.

In a different attack strategy, malware could manipulate valid kernel control flow and jump into the body of a memory allocator without entering the function from the beginning. This behavior can be detected by extending DataGene to verify that the function was entered properly. For example, the VMM can set a flag when a memory allocation function is entered and verify the flag before the function returns by interceding before the return instruction(s) of the function. If the flag was not set prior to the check, the VMM detects a suspicious memory allocation.

DataGene is a signature-based approach that detects known and unknown rootkits based on kernel data access patterns similar to the signatures of previously analyzed rootkits. If a rootkit's attack behavior is not similar to any behavior in existing signatures or it does not involve kernel data accesses, such malware is out of coverage of DataGene since such behavior does not match the DataGene's signature.

Many existing rootkits that share common attack goals often exhibit similar data access patterns because essentially these malicious programs generate a false view by manipulating legitimate kernel data structures relevant to the goals. Our approach can detect rootkits by focusing on the common attack targets described in the malware signatures even though such rootkits have different functionalities.

Obfuscating data access patterns involves comparatively more sophistication than code obfuscation because malware is required to use alternate legal code to access kernel data beyond the diversification of a malware's own code patterns. Such attack attempts can be detected by employing defense approaches related to control flow integrity [1].

DataGene is mainly designed for kernel malware analysis where a potential attack sample is analyzed to determine whether it is malware based on its data behavior. In such an analysis/classification environment with controlled configurations, it is possible to produce no false alarms as presented in our experiments. However, if this technique is further aimed towards a production environment where a wider diversity of workload could be generated, false alarms may occur due to the fact that our technique is founded on dynamic execution.

Broadly, DataGene can be categorized as a behavior-based approach due to its use of memory access behavior. However, this approach is clearly distinguished from traditional behavior-based methods. Traditional code behavior-based approaches use code sequences as patterns. Since code execution follows a program control flow specified in the program semantics, this approach is intuitively understandable. Unlike the program control flow; however, data accesses are not a single continuous flow. From the data point of view, the accesses from various code can be interleaved making a sequence not stable as a consistent pattern for a behavior signature. DataGene solves this problem by using a different aspect of program behavior. Instead of simply using the code to create malware signatures, we model data accesses with two entities: the subject (the accessing code) and the object (the accessed data). This allows us to determine the patterns of relationship between subjects and objects, and hence provides more robust signatures.

Regarding DataGene's effectiveness when compared to code behavior-based approaches, there are more constraints a malware author must consider when designing an evasion technique. For example, one evasion technique for a standard code behavior-based approach would be to find a functionally similar code sequence from the existing code and use that instead of including your own code. Return-oriented and jump-oriented programming would be such examples. In contrast, data access behavior has multiple dimensions to consider: accessing code, specific field of data, and the source of data (allocation). First of all, regarding the accessing code, our approach has an advantage since DataGene normalizes accessing code to detect malware variants as shown in Section IV. Second, specific fields being accessed should be preserved for the data object to be valid so that legitimate code can also properly use them. Third, using a custom allocator could be a feasible attack, but such an unknown memory allocation would be trackable by the OS as previously

discussed. By checking the allocation code of data objects in kernel data structures, foreign objects could be detected.

Sections VI-B and VI-C, for instance, present `hide_lkm` and `fuuld` which could not be detected by existing code-based approaches because they perform attacks on data by utilizing legitimate code. These rootkits highlight the unique detection capability of the data-centric malware defense approach.

VIII. Related Work

DataGene introduces a new approach that generates the signature of kernel malware by using their unique data access patterns. There are several approaches related to DataGene in the area of malware analysis and detection.

Malware Defense Based on Code Behavior. There has been a variety of approaches which characterizes malware behavior by using its control flow (e.g., instruction sequences and system-call graphs) [3], [4], [12], [25], [26], and such approaches can face the following challenges.

First, malware can obfuscate its execution to elude the code behavior-based malware analyzers. Several papers describe obfuscating techniques such as dead code insertion, code transformation, and instruction substitution [11], [13], [47], [53], and new techniques also have been introduced [47]. Most such techniques focus on the control dependency. Approaches characterizing malware behavior using its control flow can face an arms-race with anti-analysis schemes such as these obfuscation techniques.

Second, malware control flow can vary at runtime and the detection mechanism using malware code behavior should be able to handle such variations. In [3], the authors describe several cases where the system-call trace can be inconsistent, such as the expiration of timeout and the delivery of signals. Their system handles this problem by using a flexible matching algorithm.

Compared to these approaches, DataGene uses a more general characteristic, the pattern of kernel memory accesses, to characterize malware behavior. Because this approach avoids using control dependency in malware behavior, it can be tolerant to obfuscation techniques and variations in the malware's control flow. Moreover, it has an advantage that it can match common behavior across malware.

Kernel Malware Defense Based on Code Integrity. Another approach for malware defense is based on code integrity [42], [45]. This approach allows only authorized kernel code to execute: the kernel text and white listed kernel modules. This approach is effective in preventing driver-based kernel rootkits (i.e., kernel modules in Linux) that introduce their own code. However, some advanced rootkits operate without explicit malicious code by using techniques such as kernel memory devices (e.g., `/dev/kmem`) or return-oriented programming [23]; and this approach cannot handle such cases. DataGene uses unique data access patterns of kernel rootkits regardless of their attack vectors. Thus it can handle these challenging rootkits based on their unique data behavior.

This approach also determines benign or malicious driver code based on policies (e.g., a white list and code-signing [30]). Such policies often are not based on systematic examination of code behavior, rather they are based on trusting the OS developers or vendors. This kind of classification of code does not guarantee safety from undesired effects. For instance, as seen in Sony's rootkit incident [32], the code from third party vendors may include potentially malicious code.

Kernel Rootkit Profilers. Kernel rootkit profilers [44], [54] provide a variety of aspects of rootkit behavior by tracking the memory access targets of malware code or examining user space impact. The profiling result of these approaches is specific to the analyzed malware. In contrast, DataGene uses the generalized memory access patterns of malware and explores common characteristics across multiple rootkits. Therefore, it has the potential to detect rootkit variants or unknown rootkits that are similar in data behavior to current rootkits.

These profilers can be used as a component of DataGene in place of the kernel object mapper. Such an implementation can have the following limitations, however. First, some rootkits have attack mechanisms (e.g., using registers) that are resistant to these rootkit profilers as shown in [40]. Second, these profilers rely on code integrity-based approach [42], [45] to recognize malware code. Thus, the scope of malware to be analyzed is limited to the rootkits that violate kernel code integrity.

Signatures Based on Data Structures. Laika [15] can detect malware by determining data structures and classifying their unique patterns for malware. This approach is effective for user space malware (e.g., botnet programs), which have their own memory space. However, kernel malware code and data resides in kernel memory together with legitimate kernel code and data. In addition, kernel malware mainly targets legitimate kernel data and uses very little of its own data. Therefore, kernel malware may have a relatively weaker set of data information to determine the malware's characteristics compared to malware based on a user process.

Several approaches [19], [28] can detect kernel data structures based on data invariant properties such as data values and pointer connections. However, if a data structure is simple, such as a string buffer that can have arbitrary values without any pointers, these signature approaches cannot be applied. In comparison, DataGene does not have any restrictions on the coverage of kernel data structures.

IX. Conclusion

In this paper, we present DataGene, a new OS malware characterization system based on data-centric properties. The system works by building a live kernel object map which can reliably detect data hiding rootkit attacks due to its untampered view of kernel objects. The map is then used in combination with a monitoring agent to track memory access patterns on kernel data objects. Based on these access patterns, we propose a new malware signature approach using consistent patterns specific to malware attacks. We demonstrate this scheme is not only effective at detecting previously evaluated rootkits, but also their variants which often share similar

memory access patterns. Our evaluation on real world rootkits shows that data-centric malware characterization is highly effective. It could be an effective solution that complements code-centric approaches in the kernel malware defense.

REFERENCES

[1] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity: Principles, implementations, and applications," in *Proc. 12th ACM Conf. CCS*, Nov. 2005, pp. 1–4.

[2] A. Baliga, V. Ganapathy, and L. Iftode, "Automatic inference and enforcement of kernel data structure invariants," in *Proc. 24th ACSAC*, Dec. 2008, pp. 77–86.

[3] D. Balzarotti, M. Cova, C. Karlberger, C. Kruegel, E. Kirda, and G. Vigna, "Efficient detection of split personalities in malware," in *Proc. 17th Annu. NDSS*, Feb. 2010, pp. 1–17.

[4] U. Bayer, P. Milani Comparetti, C. Hlauscheck, C. Kruegel, and E. Kirda, "Scalable, behavior-based malware clustering," in *Proc. 16th Symp. NDSS*, Feb. 2009, pp. 1–26.

[5] F. Bellard, "QEMU: A fast and portable dynamic translator," in *Proc. USENIX Annu. Tech. Conf.*, Mar. 2005, pp. 41–46.

[6] E. Buchanan, R. Roemer, H. Shacham, and S. Savage, "When good instructions go bad: Generalizing return-oriented programming to RISC," in *Proc. 15th ACM Conf. CCS*, Oct. 2008, pp. 27–38.

[7] J. Butler. (2012, Dec. 12). *DKOM (Direct Kernel Object Manipulation)* [Online]. Available: http://www.blackhat.com/presentations/win-usa-04/bh-win-04-butler.pdf

[8] (2010). *Bypassing Non-Executable-Stack During Exploitation Using Return-to-Libc* [Online]. Available: http://www.citeulike.org/user/rvermeulen/author/C0ntex

[9] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang, "Mapping kernel objects to enable systematic integrity checking," in *Proc. 16th ACM Conf. CCS*, Nov. 2009, pp. 555–565.

[10] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie, "DROP: Detecting return-oriented programming malicious code," in *Proc. 5th ICISS*, Dec. 2009, pp. 163–177.

[11] M. Christodorescu and S. Jha, "Static analysis of executables to detect malicious patterns," in *Proc. 12th USENIX Sec. Symp.*, Aug. 2003, pp. 169–186.

[12] M. Christodorescu, C. Kruegel, and S. Jha, "Mining specifications of malicious behavior," in *Proc. 6th Joint Meeting ESEC/FSE*, Sep. 2007, pp. 1–10.

[13] C. Collberg, C. Thomborson, and D. Low, "Manufacturing cheap, resilient, and stealthy opaque constructs," in *Proc. 25th ACM SIGPLAN-SIGACT Symp. POPL*, Jan. 1998, pp. 184–196.

[14] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, *et al.*, "StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *Proc. 7th USENIX Sec. Conf.*, Jan. 1998, pp. 63–78.

[15] A. Cozzie, F. Stratton, H. Xue, and S. T. King, "Digging for data structures," in *Proc. 8th USENIX Symp. OSDI*, 2008, pp. 1–16.

[16] L. Davi, A.-R. Sadeghi, and M. Winandy, "Dynamic integrity measurement and attestation: Towards defense against return-oriented programming attacks," in *Proc. ACM Workshop STC*, 2009, pp. 49–54.

[17] L. Davi, A.-R. Sadeghi, and M. Winandy, "ROPdefender: A detection tool to defend against return-oriented programming attacks," Syst. Sec. Lab., Tech. Univ. Darmstadt, Darmstadt, Germany, Tech. Rep. HGI-TR-2010-001, 2010.

[18] (2001, Dec. 28). *Linux on-the-Fly Kernel Patching Without LKM* [Online]. Available: http://www.phrack.com/issues.html?issue=58&id=7

[19] B. Dolan-Gavitt, A. Srivastava, P. Traynor, and J. Giffin, "Robust signatures for kernel data structures," in *Proc. 16th ACM Conf. CCS*, 2009, pp. 1–12.

[20] H. Etoh. (2011, May). *GCC Extension for Protecting Applications From Stack-Smashing Attacks* [Online]. Available: http://www.trl.ibm.com/projects/security/ssp/

[21] A. Francillon, D. Perito, and C. Castelluccia, "Defending embedded systems against control flow attacks," in *Proc. 1st ACM Workshop Secure Execution Untrusted Code*, Nov. 2009, pp. 19–26.

[22] Free Software Foundation, Boston, MA, USA. (2013). *The GNU Compiler Collection* [Online]. Available: http://gcc.gnu.org/

[23] R. Hund, T. Holz, and F. C. Freiling, "Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms," in *Proc. 18th USENIX Sec. Symp.*, 2009, pp. 383–398.

[24] Innotek, Singapore. (2011, May). *Virtualbox* [Online]. Available: http://www.virtualbox.org/

[25] C. Kolbitsch, P. Milani Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang, "Effective and efficient malware detection at the end host," in *Proc. 18th USENIX Sec. Symp.*, Aug. 2009, pp. 351–366.

[26] C. Kruegel, W. Robertson, and G. Vigna, "Detecting kernel-level rootkits through binary analysis," in *Proc. 20th ACSAC*, Dec. 2004, pp. 91–100.

[27] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram, "Defeating return-oriented rootkits with 'return-less' kernels," in *Proc. 5th ACM Eur. Conf. Comput. Syst.*, Apr. 2010, pp. 1–14.

[28] Z. Lin, J. Rhee, X. Zhang, D. Xu, and X. Jiang, "SigGraph: Brute force scanning of kernel data structure instances using graph-based signatures," in *Proc. 18th Annu. NDSS*, Feb. 2011, pp. 1–18.

[29] Z. Lin, R. D. Riley, and D. Xu, "Polymorphing software by randomizing data structure layout," in *Proc. 6th Int. Conf. DIMVA*, May 2009, pp. 107–126.

[30] Microsoft, Redmond, WA, USA. (2007, Mar. 21). *Driver Signing Requirements for Windows* [Online]. Available: http://www.microsoft.com/whdc/driver/install/drvsign/default.mspx

[31] MITRE Corporation, Bedford, MA, USA. (2013, Sep. 5). *Common Vulnerabilities and Exposures* [Online]. Available: http://cve.mitre.org/

[32] D. K. Mulligan and A. K. Perzanowski. (2007). The magnificence of the disaster: Reconstructing the Sony BMG rootkit incident. *22 Berkeley Tech. L.J. 1157* [Online]. Available: http://scholarship.law.berkeley.edu/facpubs/2130/

[33] Nergal, "The advanced return-into-lib(c) exploits: PaX case study," *Phrack*, vol. 11, no. 58, article 4, Dec. 2001.

[34] Parallels, Inc., Renton, WA, USA. (2013). *Parallels* [Online]. Available: http://www.parallels.com/

[35] N. L. Petroni, T. Fraser, J. Molina, and W. A. Arbaugh, "Copilot—A coprocessor-based kernel runtime integrity monitor," in *Proc. 13th USENIX Sec. Symp.*, Aug. 2004, pp. 179–194.

[36] N. L. Petroni and M. Hicks, "Automated detection of persistent kernel control-flow attacks," in *Proc. 14th ACM Conf. CCS*, Oct. 2007, pp. 103–115.

[37] N. L. Petroni, A. Walters, T. Fraser, and W. A. Arbaugh, "FATKit: A framework for the extraction and analysis of digital forensic data from volatile system memory," *Digit. Invest. J.*, vol. 3, no. 4, pp. 197–210, Dec. 2006.

[38] N. L. Petroni, T. Fraser, A. Walters, and W. A. Arbaugh, "An architecture for specification-based detection of semantic integrity violations in kernel dynamic data," in *Proc. 15th Conf. USENIX Sec. Symp.*, 2006, pp. 289–304.

[39] J. Rhee, R. Riley, D. Xu, and X. Jiang, "Kernel malware analysis with un-tampered and temporal views of dynamic kernel memory," in *Proc. 13th Int. Symp. RAID*, Sep. 2010, pp. 178–197.

[40] J. Rhee and D. Xu, "LiveDM: Temporal mapping of dynamic kernel memory for dynamic kernel malware analysis and debugging," CERIAS, West Lafayette, IN, USA, Tech. Rep. 2010-02, 2010.

[41] R. Riley, "A framework for prototyping and testing data-only rootkit attacks," *Comput. Sec.*, vol. 37, pp. 62–71, Sep. 2013.

[42] R. Riley, X. Jiang, and D. Xu, "Guest-transparent prevention of kernel rootkits with VMM-based memory shadowing," in *Proc. 11th Int. Symp. RAID*, 2008, pp. 1–20.

[43] R. Riley, X. Jiang, and D. Xu, "An architectural approach to preventing code injection attacks," *IEEE Trans. Dependable Secure Comput.*, vol. 7, no. 4, pp. 351–365, Dec. 2009.

[44] R. Riley, X. Jiang, and D. Xu, "Multi-aspect profiling of kernel rootkit behavior," in *Proc. 4th ACM Eur. Conf. Comput. Syst.*, Apr. 2009, pp. 47–60.

[45] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes," in *Proc. 21st SOSP*, Oct. 2007, pp. 1–17.

[46] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proc. 14th ACM Conf. CCS*, 2007, pp. 1–30.

[47] M. Sharif, A. Lanzi, J. Giffin, and W. Lee, "Impeding malware analysis using conditional code obfuscation," in *Proc. 15th Annu. NDSS*, 2008, pp. 65–88.

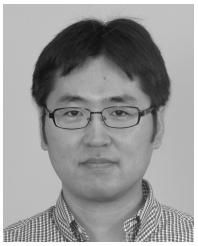
[48] M. Sharif, A. Lanzi, J. Giffin, and W. Lee, "Automatic reverse engineering of malware emulators," in *Proc. 30th IEEE Symp. Sec. Privacy*, Mar. 2009, pp. 1–16.

[49] (2006). *The Month of Kernel Bugs (MoKB) Archive* [Online]. Available: http://projects.info-pull.com/mokb/

[50] US-CERT, Washington, DC, USA. (2013). *US-CERT Vulnerability Notes Database* [Online]. Available: http://www.kb.cert.org/vuls/

[51] (2011, May). *Stack Shield: A 'Stack Smashing' Technique Protection Tool for Linux* [Online]. Available: http://www.angelfire.com/sk/stackshield/info.html

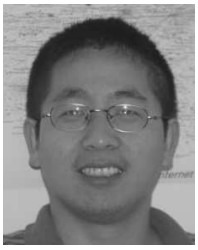
[52] (2013, Sep.). *VMware Workstation: Run Multiple OS, Linux, Windows 8 & More* [Online]. Available: http://www.vmware.com/products/workstation/
[53] C. Wang, J. Hill, J. C. Knight, and J. W. Davidson, "Protection of software-based survivability mechanisms," in *Proc. Int. Conf. DSN*, Jul. 2001, pp. 193–202.
[54] C. Xuan, J. A. Copeland, and R. A. Beyah, "Toward revealing kernel malware behavior in virtual execution environments," in *Proc. 12th Int. Symp. RAID*, 2009, pp. 304–325.



Junghwan Rhee (M'11) received the B.S. degree from Korea University, the master's degree from the University of Texas at Austin, and the Ph.D. degree in computer science from Purdue University in 2011. He is a Researcher at NEC Laboratories America, Princeton, NJ, USA. His research interests include malware analysis, system security, software debugging, and cloud computing.

Ryan Riley (M'13) received the B.S. degree in computer engineering and the Ph.D. degree in computer science in 2009 from Purdue University. He is an Assistant Professor of computer science with Qatar University, Doha. His current research interests include virtualization technologies, malware, and operating system security.

Zhiqiang Lin (M'12) is an Assistant Professor with the Computer Science Department, University of Texas at Dallas. He received the Ph.D. degree from Purdue University in 2011. His current research focuses on system and software security with an emphasis on binary code reverse engineering, vulnerability discovery, malicious code analysis, and OS kernel protection.



Xuxian Jiang is an Associate Professor with the Computer Science Department and a Core Member of the Cyber Defense Laboratory, North Carolina State University. He received the Ph.D. degree in computer science from Purdue University in 2006. His research interests are mainly in smartphones, hypervisors, and malware defense.

Dongyan Xu (M'03) received the B.S. degree from Zhongshan (Sun Yat-Sen) University in 1994 and the Ph.D. degree in computer science from the University of Illinois at Urbana-Champaign in 2001. He is a Professor of computer science with Purdue University. His current research interests include virtualization technologies, computer malware defense, and cloud computing. He is a recipient of the U.S. National Science Foundation CAREER Award.